

ARIES

AUTONOMOUS ROBOTIC INSPECTION EXPERIMENTAL SYSTEM

TECHNICAL APPENDICES

Period of Performance: 30 September 1992 through 31 July 1998

by

Robert O. Pettus, Project Manager & Principal Investigator
Jerry L. Hudgins, Senior Investigator
David N. Rocheleau, Senior Investigator
Robert J. Schalkoff, Senior Investigator
Paul McCarty, Graduate Research Assistant
Edward A. Hamilton, Associate Director, SCUREF

August 1998

Work performed under Contract
FETC DE-AC21-92MC29115 -99

Submitted by

**SOUTH CAROLINA UNIVERSITIES
RESEARCH AND EDUCATION FOUNDATION**

Strom Thurmond Institute
Clemson, South Carolina 29634-5701

TABLE OF CONTENTS

| | |
|--------------------------------|-------------------|
| COMPUTERS AND CONTROLS | APPENDIX A |
| COMPUTER VISION SYSTEM | APPENDIX B |
| ELECTRICAL SYSTEMS | APPENDIX C |
| MECHANICAL SYSTEMS | APPENDIX D |
| ROBOTIC VEHICLE SYSTEMS | APPENDIX E |
| FIELD TRIALS | APPENDIX F |

Appendix A

ARIES: An Intelligent Inspection and Survey Robot

COMPUTERS AND CONTROLS

Department of Electrical & Computer Engineering
University of South Carolina

Table Of Contents

| | |
|--|-----------|
| A.1. INTRODUCTION | 1 |
| <hr/> | |
| A.1.1 ARIES OFF-BOARD SOFTWARE INSTALLATION | 1 |
| A.1.1.1 INSTALLATION | 1 |
| A.1.1.2 ENVIRONMENT | 1 |
| A.1.1.3 RECOMPILING THE SOFTWARE | 2 |
| A.1.2 RUNNING THE SITE MANAGER | 2 |
| | |
| A.2. MAIN INTERFACE | 3 |
| <hr/> | |
| A.2.1 INTERFACE LAYOUT | 3 |
| A.2.2 MAP NAVIGATION | 3 |
| A.2.2.1 PANNING | 3 |
| A.2.2.2 ZOOMING | 3 |
| A.2.3 VIEWING OPTIONS | 4 |
| | |
| A.3. SITE CREATION | 5 |
| <hr/> | |
| A.3.1 CREATING A NEW SITE | 5 |
| A.3.2 NAMING THE BUILDING AND THE SITE | 5 |
| A.3.3 SPECIFYING BUILDING MODELS | 6 |
| A.3.4 CREATING ROBOTS | 6 |
| A.3.4.1 ADDING ROBOTS | 6 |
| A.3.4.2 SETTING THE CURRENT ROBOT | 6 |
| A.3.4.3 CHANGING A ROBOT’S COLOR | 7 |
| A.3.4.4 DELETING ROBOTS | 7 |
| A.3.5 PATH CREATION | 7 |
| A.3.5.1 TERMINOLOGY | 8 |
| A.3.5.2 MAP SECTION | 8 |
| A.3.5.3 DOCK MODIFICATION | 9 |
| A.3.5.4 LIST SECTION | 10 |
| A.3.5.5 THE CREATION PROCESS | 10 |
| A.3.6 PATH MODIFICATION | 10 |
| A.3.6.1 CLASSIFICATION | 11 |
| A.3.6.2 DIRECTION | 11 |
| A.3.6.3 INSPECTION TYPE | 12 |
| A.3.6.4 TARGET ASSIGNMENT | 13 |
| A.3.7 COMPLETING THE SITE | 13 |
| | |
| A.4. MISSION ASSIGNMENT | 14 |
| <hr/> | |
| A.4.1 PATH ASSIGNMENT | 14 |
| A.4.2 MISSION GENERATION | 15 |

A.5. REAL-TIME MISSION DATA **16**

| | | |
|--------------|---|-----------|
| A.5.1 | ROBOT DISPLAY | 16 |
| A.5.2 | NAVIGATION INFORMATION | 16 |
| A.5.3 | MONITOR | 17 |
| A.5.3.1 | INVOCATION | 17 |
| A.5.3.2 | INTERFACE LAYOUT | 18 |
| A.5.3.2.1 | Main Interface | 18 |
| A.5.3.2.1.1 | Pulldown Menu | 18 |
| A.5.3.2.1.2 | Active Indicators List | 18 |
| A.5.3.2.1.3 | Action Buttons | 19 |
| A.5.3.2.2 | Indicator Dialog | 20 |
| A.5.3.2.2.1 | Database Browser | 20 |
| A.5.3.2.2.2 | Defining Variables Outside the Database | 20 |
| A.5.3.2.2.3 | Indicator Types | 21 |
| A.5.3.2.2.4 | Action Buttons | 21 |
| A.5.3.2.3 | Monitoring Variables | 21 |
| A.5.3.2.3.1 | ARIES High Level Status | 22 |
| A.5.3.2.3.2 | Indicator Display | 22 |

A.6. MISSION REPORTS **24**

| | | |
|--------------|----------------------------|-----------|
| A.6.1 | MISSION INFORMATION | 24 |
| A.6.2 | AISLE INFORMATION | 25 |
| A.6.3 | STATUS FILTER | 25 |
| A.6.4 | SUCCESS GRAPH | 25 |
| A.6.5 | VIEW HISTORY | 25 |

A.7. 3D SITE INSPECTION **27**

| | | |
|--------------|-----------------------------------|-----------|
| A.7.1 | VIEWING OPTIONS | 27 |
| A.7.1.1 | RENDERING | 28 |
| A.7.1.2 | PERSPECTIVE | 29 |
| A.7.1.3 | VIEW | 29 |
| A.7.1.4 | OPTIONS | 30 |
| A.7.2 | DRUM CULLING AND RENDERING | 30 |
| A.7.2.1 | DETAIL | 30 |
| A.7.2.2 | PROXIMITY | 31 |
| A.7.2.3 | LEVEL | 31 |
| A.7.2.4 | COLOR CODING | 32 |
| A.7.3 | DRUM SELECTION | 32 |

A.8. DATABASE MANAGEMENT **33**

| | | |
|--------------|-------------------|-----------|
| A.8.1 | INVOCATION | 33 |
| A.8.1.1 | FROM SITE MANAGER | 33 |

| | | |
|--------------|---------------------------------|-----------|
| A.8.1.2 | FROM IRIX COMMAND PROMPT | 33 |
| A.8.2 | BROWSER INTERFACE LAYOUT | 34 |
| A.8.2.1 | MAIN INTERFACE | 34 |
| A.8.2.1.1 | Pulldown Menu | 34 |
| A.8.2.1.2 | Entries List | 34 |
| A.8.2.2 | SEARCH/SORT DIALOG | 35 |
| A.8.2.2.1 | Searching | 35 |
| A.8.2.2.2 | Sorting | 36 |
| A.8.2.3 | DATA SET MANIPULATION | 36 |
| A.8.3 | RECORD EDITOR | 36 |
| A.8.3.1 | FIELD TYPES | 36 |
| A.8.3.2 | IMAGES | 36 |
| A.8.3.2.1 | Image Fields | 37 |
| A.8.3.2.2 | gviewer Utility | 37 |
| A.8.4 | DATABASE UTILITIES | 37 |
| A.8.4.1 | DBTOOL | 37 |
| A.8.4.2 | DBUTIL | 38 |

List of Figures

| | |
|---|----|
| Figure 1 (Main interface) | 3 |
| Figure 2 (Building name dialog) | 5 |
| Figure 3 (Robot name dialog) | 6 |
| Figure 4 (Robots section) | 7 |
| Figure 5 (Color Dialog) | 7 |
| Figure 6 (Map section) | 8 |
| Figure 7 (Dock Dialog) | 9 |
| Figure 8 (List section) | 10 |
| Figure 9 (Path Dialog) | 11 |
| Figure 10 (Path Assembler dialog) | 12 |
| Figure 11 (Path assignment) | 14 |
| Figure 12 (Mission Dialog) | 15 |
| Figure 13 (2D and 3D robot display) | 16 |
| Figure 14 (Navigation diagnostics) | 17 |
| Figure 15 (Monitor interface) | 18 |
| Figure 16 (Active list filled) | 19 |
| Figure 17 (Indicator dialog) | 20 |
| Figure 18 (Write dialog) | 21 |
| Figure 19 (ARIES status display) | 22 |
| Figure 20 (Indicator display) | 22 |
| Figure 21 (Alternative grid arrangement) | 23 |
| Figure 22 (Maximum columns and rows) | 23 |
| Figure 23 (Mission Report dialog) | 24 |
| Figure 24 (Highlighted aisle) | 25 |
| Figure 25 (History Dialog) | 25 |
| Figure 26 (3D Tour) | 27 |
| Figure 27 (3D Tour menu hierarchy) | 28 |
| Figure 28 (Points, Wire frame, Flat shaded, and Smooth shaded scenes) | 28 |
| Figure 29 (Drum culling and rendering options) | 30 |
| Figure 30 (Low (left) and high (right) detail drums) | 31 |
| Figure 31 (High (left) and low (right) proximity scenes) | 31 |
| Figure 32 (Record editor via 3D tour) | 33 |
| Figure 33 (ariesddb main interface) | 34 |
| Figure 34 (Record field controls) | 34 |
| Figure 35 (Search dialog) | 35 |
| Figure 36 (Search results) | 35 |
| Figure 37 (Record editor) | 36 |
| Figure 38 (gviewer utility) | 37 |

A.1. Introduction

The Site Manager is the mechanism by which users of the Autonomous Robotic Inspection Experimental System (ARIES) govern the inspection of a site. The interface allows users to associate each building with a site, multiple inspection vehicles (these vehicles are referred to as robots in the interface) to each building, and graphically determine the paths that each robot is to follow. Additionally, the Site Manager provides a wide range of real-time feedback from each robot during an inspection mission. This data ranges from a graphical display (2D and 3D) of the robot's location and the position of its cameras to detailed graphs displaying any on-board variable desired. Mission reports are generated by each robot at the end of its mission. These reports detail the status of every inspected aisle and provide a second by second account of the robot's whereabouts and camera positions for that mission. The Site Manager also provides a method for inspecting the drum database associated with each storage facility. Each drum in the database is shown in its correct location on both the 2D and 3D maps of the building. Users can gain access to detailed drum information simply by "clicking" on a drum. Through its wide range of functions, the Site Manager provides all of the tools necessary for the management of the ARIES system in an easy to use graphical user interface.

This manual is arranged in the typical order that a user would operate the Site Manager. Section A.2 describes the layout of the main interface. Section A.3 describes the creation of a site. Section A.4 describes mission creation and execution. A description of real-time robot monitoring follows in section A.5. Mission reports are discussed in section A.6. Section A.7 details the use of the 3D-site inspection interface. The final section illustrates the use of database management and viewing interfaces.

A.1.1 ARIES Off-board Software Installation

A.1.1.1 Installation

The ARIES off-board software distribution is contained in a single file called ARIES.tar.Z. Copy this file to the desired destination directory and execute the following pair of commands:

```
uncompress ARIES.tar.Z
tar -xvf ARIES.tar
```

This will create a directory called ARIES, under which the entire ARIES distribution is stored.

A.1.1.2 Environment

In order for the ARIES software to execute correctly, a number of steps must be taken:

1. The environment variable ARIES_ROOT_DIR must be set to the installation destination directory. If the archive was extracted in /usr/local/, then ARIES_ROOT_DIR must be set to /usr/local/ARIES/.
2. ARIES_ROOT_DIR/platforms/IRIX/bin must be added to the execution path of any user who is to run the ARIES software.
3. All of the files in ARIES_ROOT_DIR/src/apps/resource must be copied into /usr/lib/X11/app-defaults.

4. The ARIES software has been tested under IRIX 5.3 and 6.2. However, it is directly dependent on one library that is not usually installed with the IRIX OS. libGLC.a (Silicon Graphics' OpenGL Character Rendering Library) must be in /usr/lib.

A.1.1.3 Recompiling the Software

All of the off-board software source code is contained in ARIES_ROOT_DIR/src under the directories apps/ and libraries/. Each of the applications is stored in its own directory and contains its own Imakefile. The first line of these Imakefiles contains a macro revealing the location of the ARIES software distribution. If the software is to be recompiled, this macro must be reassigned to the correct value (the value of ARIES_ROOT_DIR).

A.1.2 Running the Site Manager

The Site Manager executable should already be in your PATH environment variable. Check with your system administrator if it is not. The following statement gives the usage of the Site Manager executable:

```
site [options] [site file]
```

where options can be

| | |
|--------------|--|
| -vi | use the vi editor for code modification. |
| -dead | use only dead reckoning in automatically generated code. |
| -r <integer> | number of milliseconds between updates. |
| -noserver | disable communications (monitor, display). |
| -john | use user defined origin in code generation. |
| -notex | disable texture mapping for inventor files. |
| -movie | save frames for animation (3D Tour). |
| -h | this help screen. |

A.2. Main Interface

A.2.1 Interface Layout

Figure 1 depicts the main interface of the Site Manager. It might be useful to use this screen shot as reference while reading the rest of the user's guide. Most of the terminology labeled on this figure is used throughout the manual.

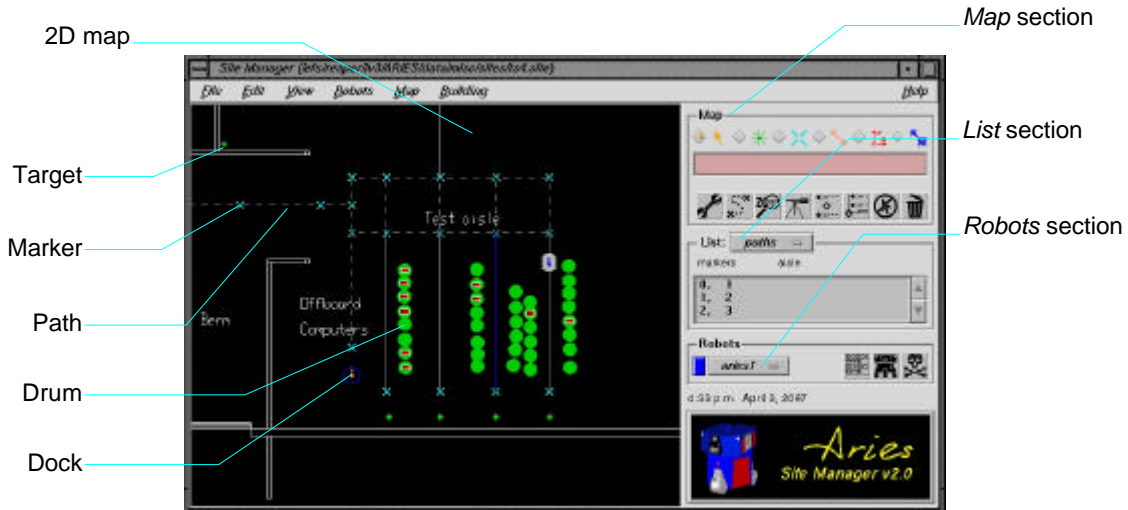


Figure 1 (Main interface)

A.2.2 Map Navigation

The following two sections describe the two methods of map navigation used in the Site Manager. These tools provide an easy method for detailed viewing of any section of the map.

A.2.2.1 Panning

Panning is simply the act of "sliding" the map to the desired view. Press and hold the middle mouse button. The map will translate with the cursor as long as the middle mouse button remains pressed. Drag the map to the desired location and release the button.

A.2.2.2 Zooming

Zooming is accomplished by drawing a zoom box with the right mouse button. Press and hold the right mouse button. Drag out a zoom box by moving the cursor away from the point where the right button was originally pressed. A white box is drawn using the original point and the cursor position as its corners.

When the right button is released, the boundary of the box will become the boundary of the new view. This method can be used any number of times in succession to progressively zoom in on a desired region. The *Map→Zoom previous* selection will make the previous view the current view. Additionally, *Map→Zoom extents* will cause the entire building to be visible on the map.

A.2.3 Viewing Options

The *View* pull down menu provides a method for controlling what is drawn on the map. The following list details the options available via the *View* menu. Turning on only those items of interest is an excellent method of reducing the complexity in any given area of the map.

- *Targets.* Determines if LIDAR targets will be drawn.
- *Markers.* Determines if navigation markers will be drawn.
- *Docks.* Determines if docks will be drawn.
- *Paths.* Determines if paths will be drawn.
- *Animation.* When checked, all zoom operations are animated over a short period to indicate a changing map view.
- *Drums.* The options in this menu determine which, if any, of the drum levels are drawn.

A.3. Site Creation

A site is defined to be a storage facility with at least one building in which drums are stored. Sites can have as many buildings as desired. Each building must have a two dimensional, AutoCAD description of the building's layout. Each building may optionally have a three dimensional, Inventor description of the building's structure. Each building must be assigned at least one robot. Multiple robots can be assigned as needed. Associated with each building is a set of paths for the robots to travel. This section describes how this information is entered into the Site Manager.

A.3.1 Creating a New Site

Site information is stored in a site file. These files are analogous to the commonly used document files associated with word processors. Site files can be opened, saved, and saved as another filename. Additionally, new sites can be created. All of these operations are performed through the *File* pull down menu. If the Site Manager is invoked with no command line arguments, a new site is automatically created and is ready for editing.

A.3.2 Naming the Building and the Site

A name must be given for both the building and the site so that it can be identified by the ARIES system. The site name can be entered/changed via the *Edit*→*Site name* menu selection. Similarly, the building name can be entered/changed via the *Edit*→*Building name* selection. Figure 2 represents the dialog box used to enter each of these names.

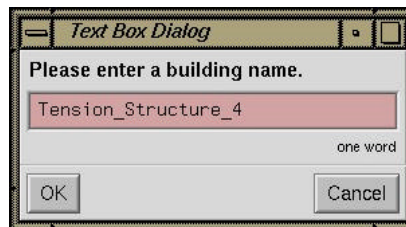


Figure 2 (Building name dialog)

Each site file is associated with only one building. A separate site file must be maintained for each building at a site. Buildings within the same site should be given unique building names but should all use the same site name. This naming convention, while not critical to the ARIES system, will organize all site data in the same place in the ARIES file structure. This organization will help users to identify mission reports with the appropriate buildings and sites.

A.3.3 Specifying Building Models

Every building must be assigned an AutoCAD model that describes its layout. AutoCAD models used in the Site Manager must be stored in the version 12 DXF format. Selecting *Map→Change DXF* pops up a file selector that allows users to select the appropriate model. The *Map→DXF units* option lets users specify the units used in the DXF file (inches, feet, centimeters, or meters). Optionally, a 3D Inventor v1.0 file can be assigned to the building for use in the 3D Tour. Use the *Map→Change IV* and *Map→IV units* selections in the same manner described above to assign the Inventor file.

A.3.4 Creating Robots

Multiple robots may be associated with each building. Each robot is represented by both a unique name and a unique color. The name of the robot must match the name of the computer board installed inside of the robot. Site Manager uses this robot name to identify it during mission specification, associate mission reports with the correct robot, and establish a communication link with the robot during mission execution. It is crucial that this name is correct; therefore it should only be entered or changed by someone with intimate knowledge of the system. The color of the robot is only used to identify it within the Site Manager and may be changed freely.

A.3.4.1 Adding Robots

Robot's are added via the *Robots→Add...* menu selection. Enter the name of the robot in the dialog box shown in Figure 3. Ensure that this name matches the name of the computer board installed in the inspection vehicle.

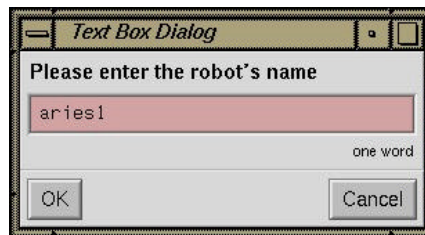


Figure 3 (Robot name dialog)

Repeat this process for each robot assigned to the building. As each new robot is added, it is assigned a random, unique color. Section A.3.4.3 describes how this color can be changed.

A.3.4.2 Setting the Current Robot

The *Robots* section of the work region (Figure 4) provides access to most of the robot manipulation functions. The option menu on the left-hand side of the section allows users to select which robot is considered the current robot. Operations such as aisle assignment, mission generation, and real-time monitoring are all performed on the current robot. Simply click on the option menu to get a list of all of the robots associated with the building. Choose the robot for which you want subsequent actions to modify or use.



Figure 4 (Robots section)

A.3.4.3 Changing a Robot's Color

The colored push button next to the current robot option menu indicates the color that represents the current robot within the site manager. Pressing this button activates the standard color selector shown in Figure 5. Use this dialog to change the color of the current robot.

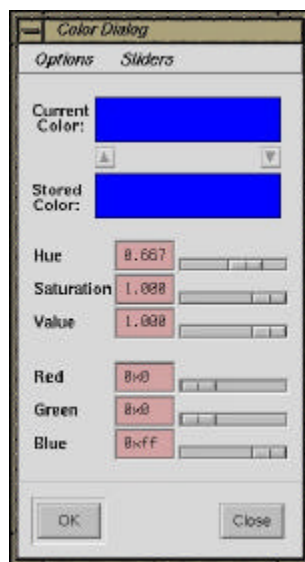


Figure 5 (Color Dialog)

A.3.4.4 Deleting Robots

Select *Robots*→*Delete current* to delete the current robot. Additionally, the skull and crossbones icon in the *Robots* section will delete the current robot.

A.3.5 Path Creation

Path creation and modification is by far the most complex action performed by users of the Site Manager. In a nutshell, the process of path creation entails marking the places on the map which define navigation points, connecting these points to create paths and drum aisles, specifying the location of navigational aids, and instructing the Site Manager to create the actual assembly programs the robots will use to navigate and inspect the building.

A.3.5.1 Terminology

The following list describes the terminology that must be understood for path creation.

- **Marker** - Markers are simply locations on the floor placed by the user to indicate points where the robot will have to go. For example, markers are used to define the start and end of a drum aisle. Markers are denoted as cyan X's on the map of the building.
- **Path** - A path is defined as the straight line between two markers. When two markers are connected, two paths are created; the first path goes from marker 1 to marker 2, and the second path goes from marker 2 to marker 1. It is important to understand this concept for the cases where detailed editing of path programs is necessary. Paths are denoted as dashed lines or solid lines (solid lines indicate drum aisles) on the map.
- **Aisle** - Aisles are simply paths that cause the robot to traverse an aisle of drums. Only these paths will cause the robot to search for and inspect drums. Aisles are represented as solid lines drawn in the color of the robot that is assigned to them (see Section A.4 - Mission assignment).
- **Dock** - Docks represent the location of a docking station. These are represented by peach circles with an arrow representing the direction of the docking beacon. Each dock is circled in the color of the robot assigned to dock with it.
- **Target** - LIDAR Targets are the mechanism by which the robot corrects its position in environments when sonar alone is not adequate. Targets are represented as small green circles surrounded by radiating lines.
- **Origin** - The origin represents the coordinates (0.0, 0.0). It is represented by a red X on the map. All coordinates entered by the user are relative to this origin.

A.3.5.2 Map Section

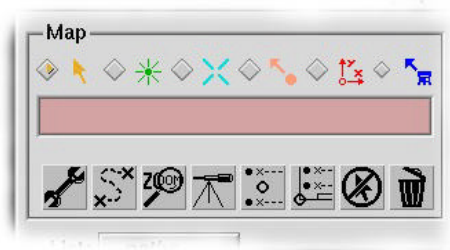


Figure 6 (*Map* section)

The *Map* section (Figure 6) of the work region is crucial to path creation. The toggle buttons across the top of the *Map* section control what action will be taken when the user clicks on the map or types in a coordinate in the input text box. The cursor shown over the map always reflects which button is currently selected. The following list describes what actions each button represents.

- **Yellow Arrow** - The arrow allows users to select items on the map by clicking on them. Selected items appear in yellow and are highlighted in the *List* section. Additionally, the arrow allows users to connect markers to form paths.
- **Green Target** - Each mouse click or coordinate entered in the text box will result in the creation of a new target.
- **Cyan Marker** - Each mouse click or coordinate entered in the text box will result in the creation of a new marker.

- **Peach Dock** - Each mouse click or coordinate entered in the text box will result in the creation of a new dock. In addition to coordinates, a third value may be entered in the text box to indicate the direction of the docking beacon (direction should be given in "bdegrees," or binary degrees).
- **Red Origin** - Each mouse click or coordinate entered in the text box will result in a new location for the origin.
- **Robot** - This button allows the user to assign aisles to specific robots for inspection. This process is detailed in section A.4.1.

Below the text box is a series of icons. The following list describes the action performed by each icon.

- **Wrench** - This icon brings up the modification dialogs for any items selected. Docks and paths each require more information that can be entered on the main interface. Once a dock or a path is selected, click on this icon to bring up its modification dialog.
- **Path** - This icon will create a path between two selected markers.
- **Magnifying glass** - This icon zooms out one level. Zooming in is accomplished by drawing a zoom rectangle over the map. Press the right mouse button and drag out a box. When the button is released, the view will zoom in to the rectangle. Users can zoom in as many times as they wish. This icon will back out to previous views one at a time.
- **Telescope** - This icon zooms out to the building's extents (zoom all).
- **Center origin** - This icon places the user's origin at the center of the building.
- **Original origin** - This icon places the user's origin at the origin of the building model.
- **Deselect** - This icon deselects all items.
- **Trash can** - This icon deletes all selected items.

A.3.5.3 Dock Modification

In addition to a dock's position, the azimuth, dock number and marker offset from of the dock must be entered. Select a dock and click on the wrench icon in the *Map* section (or select *Edit*→*Modify selected*). This action invokes the *Dock Dialog* shown in Figure 7.

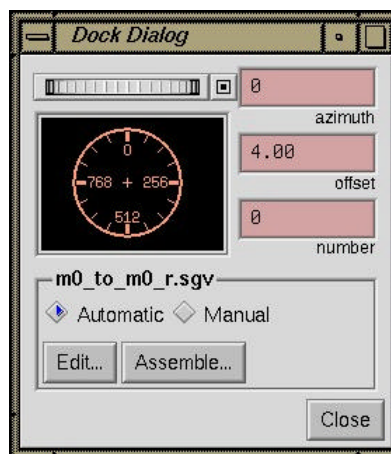


Figure 7 (Dock Dialog)

Turn the azimuth thumb wheel or enter a value in the text box in order to specify the direction of the docking beacon. The azimuth must be specified in bdegrees (binary degrees) which are defined according to the chart on the dialog. The arrow on the dock symbol on the map will point in the direction of the specified azimuth. The distance entered in the offset text box determines how far away the robot will be when it docks. Each docking station is given a number from 0 to 31 when it is installed. Find out the number assigned to the dock and enter it in the number text box. The number and azimuth must be correct so that the robot can be properly referenced. The final section of the dialog allows users to manually edit and assemble the docking program. See section A.3.6 for a complete description on manual program modification.

A.3.5.4 List Section



Figure 8 (List section)

The *List* section (Figure 8) of the work region provides users with a detailed list of markers, docks, targets, and paths. The option menu at the top of the section can be used to determine which list is seen. Clicking on a list item will result in it being selected as if it were clicked on the map. The list provides an excellent method for viewing detailed information about each item on the map.

A.3.5.5 The Creation Process

Paths are typically entered by first measuring the building to determine the positions of the navigation markers and navigational aids. The navigation markers are entered by clicking the marker button in the *Map* section and entering the coordinates of the measured markers. The navigational aids are then entered using the appropriate operator in the *Map* section. Paths are created in one of two ways. First, enter multiple selection mode under *Edit*→*Multiple selection*. Select exactly two markers and press the *Connect Markers* icon in the *Map* section (or select *Edit*→*Dis/Connect markers*). A shortcut to this method is to use the arrow cursor to select a marker. Without releasing the left mouse button, drag the cursor to the next marker. A dashed line should follow your cursor from the first marker. Release the mouse button near the desired marker and the path will be created. Continue connecting markers until all of the paths have been created.

A.3.6 Path Modification

Path creation is only the first step in completing the path entering process. Paths have many attributes that can be set via the *Path Dialog* (Figure 9). To access the path dialog, select a path (use the yellow arrow cursor or the path list in the *List* section) and click on the wrench icon in the *Map* section (or select *Edit→Modify selected*). Notice that the Path dialog allows parameter editing in both of the directions associated with the connection of two markers. Make sure that the selected path direction matches that of the path for which you wish to change the attributes.

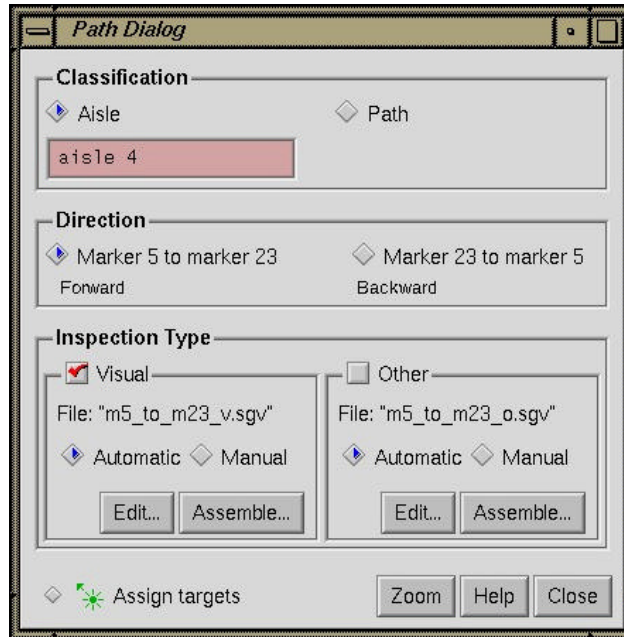


Figure 9 (Path Dialog)

The following sections describe the functions of each of the *Path Dialog* sections.

A.3.6.1 Classification

The classification section allows users to specify whether the selected path is a drum aisle or a navigation path. The classification of a path is independent of the direction of the path. In other words, if a path is designated as an aisle, both directions will be treated as aisles. This treatment is necessary because of the drum navigation routines. Even if drums do not exist on one side of an aisle, the robot must operate cautiously while operating near the drums that do exist. Selecting the aisle classification will ensure that the robot will navigate correctly in the vicinity of drums. The text box provides a method for users to give each aisle a name. For example, a naming system for each aisle may already exist in the storage facility. Users can enter the same names here for use in the Site Manager.

A.3.6.2 Direction

Because each path actually represents a forward and backward path, the user must specify which direction will be modified by subsequent path modification operations. Listed with each direction are the marker

numbers where the path starts and ends. Marker numbers can be viewed in the marker list in the *List* section. A yellow arrow appears on the map indicating the currently selected direction of travel for the selected path.

A.3.6.3 Inspection Type

The inspection type section is only available for paths designated as aisles. Two types of inspections are possible: visual and other. Visual inspections cause the robot to stop at each drum column, take and analyze several pictures of each drum, and record the results in the drum database. Other type inspections simply cause the robot to traverse the aisle in the selected direction. Other type inspections are useful for test runs as well as cases where only one side of an aisle has drums on it; use a visual inspection for the direction that has drums and an other inspection for the direction that does not.

Within each inspection type is a section which allows expert users to manually edit the programs generated by the Site Manager (before the programs can be edited, the *Building*→*Update paths* selection described in section A.3.7 must be executed). The file name of the SGV file generated is given in quotes so that it can be identified within the ARIES file structure. Pressing the *Edit* push button invokes a standard text editor on the SGV file associated with the path. Users can edit the path program in this editor. If the program is edited manually, the *Manual* toggle must be selected so that the Site Manager will not write over the user's file the next time the paths are updated. If the file should ever need to be rewritten by the Site Manager, select the *Automatic* toggle button. Manually edited paths must also be manually assembled. The *Assemble* push button will invoke the *Path Assembler* (PASM) dialog shown in Figure 10.

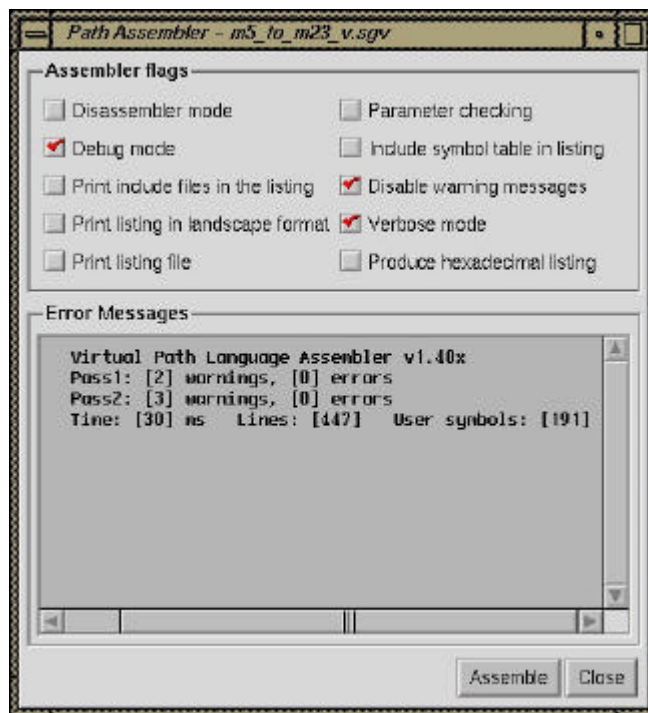


Figure 10 (*Path Assembler* dialog)

Click on each toggle button next to any option desired during the assembly process. Typical options include *Debug mode* and *Verbose mode* to aid in debugging errant programs. Press the *Assemble* button to invoke PASM on the file with the selected options. Any error messages will appear in the *Error Messages* text box. It may be useful to keep both the *Path Assembler* dialog and the text editor open until the program assembles correctly. It is important that all manually edited paths have assembled correctly or a “No Path” error will be generated when any missions requiring that path are generated.

A.3.6.4 Target Assignment

The green target/arrow icon allows users to specify which targets will be used for the currently selected path and direction in the automatically generated paths. Clicking this icon will change the cursor to the green target/arrow. Use this cursor to graphically select which targets (zero to three targets can be selected). Targets currently selected for use on this path are highlighted by a yellow, double arrow. Selected targets can be deselected by clicking on them again. Target assignment is the most crucial aspect of path modification. Targets should be selected so that the LIDAR system has a good view of all the targets selected for most or all of the length of the path. Target assignment is usually a trial and error process until a combination is found that allows robots to navigate the path with minimal error. Don't forget to assign targets for both directions of each path.

A.3.7 Completing the Site

After all of the path information is correct, the Site Manager must be told to write the assembly programs necessary for robot navigation. Select the *Update paths* entry from the *Building* pull down menu. There will be a slight delay while the programs are written and assembled. The user will be notified when the path writing process is complete. It is important to repeat this step any time new paths are added or old ones are modified.

The final step in site creation is saving your work. Select either *Save* or *Save as...* from the *File* pull down menu. Site files should be saved with a .site extension so that they will be easily recognized the next time Site Manager is executed.

A.4. Mission Assignment

Mission assignment is the most common task executed by users of the Site Manager. Fortunately, mission assignment, unlike site creation, can be broken down into two easy steps.

A.4.1 Path Assignment

Each aisle in the building must be assigned to a robot before it can be inspected. Path assignment is accomplished by selecting the robot toggle button from the *Map* section. The toggle button is represented by a robot cursor drawn in the color of the current robot. Use the cursor to click on any aisle(s) that the current robot will be responsible for inspecting. All aisles selected will be drawn in the color of the robot assigned to inspect them. Each aisle can only be assigned one robot for inspection. In addition, clicking on any dock will cause that dock to become the current robot's home dock. A circle drawn in the robot's color around a dock indicates the home dock of the current robot. The sample map image in Figure 11 depicts one aisle assigned to a blue robot and two unassigned aisles.

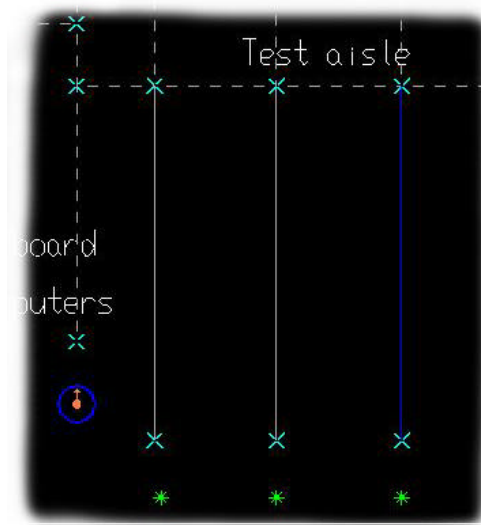


Figure 11 (Path assignment)

Use the technique described above to associate all of the robots in each building to the appropriate set of aisles. The current robot can be changed via the *option menu of the Robot section* at any time during the path assignment process. Each time path assignments are made remember to save the site. Path assignments are stored along with the rest of the site information.

A.4.2 Mission Generation

Once path assignments are complete, only one step remains before inspection missions can be executed. Select a robot for the next mission by making it the current robot. Next, activate the *Mission Dialog* shown in Figure 12 by selecting *Building*→*Mission generation*.

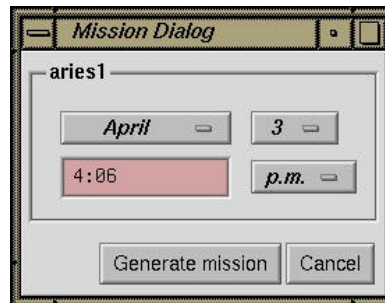


Figure 12 (*Mission Dialog*)

The current time and date will appear in the *Mission Dialog*. Set the time and date that the robot should begin its inspection mission by using the option menus and time text box. If the robot receives the mission later than the specified time, it will leave immediately. Therefore, if the robot should leave as soon as it is activated, simply leave the current time on the dialog. Once a time is selected, press the *Generate mission* push button. The robot will inspect each of the aisles that it has been assigned.

A.5. Real-time Mission Data

The Site Manager is capable of displaying a wide range of data gathered from the robot while a mission is being executed. It is very important that the name of each robot in the site matches the name of the machine installed in the corresponding ARIES vehicle; this name is used to establish a communication link between the Site Manager and the robot.

A.5.1 Robot Display

The current robot can be displayed by selecting the *Display current* option under the *Robots* pull down menu. Figure 13 depicts how the robot will be displayed on both the 2D and 3D maps (the 3D map is discussed in section A.7). The position and orientation of the robot will be accurately represented on both maps. Additionally, the 3D map displays the position and orientation of the camera and mast systems. Using the Site Manager as a display mechanism provides a method for the user to monitor the vehicle's progress without following the robot around or using a video surveillance system.

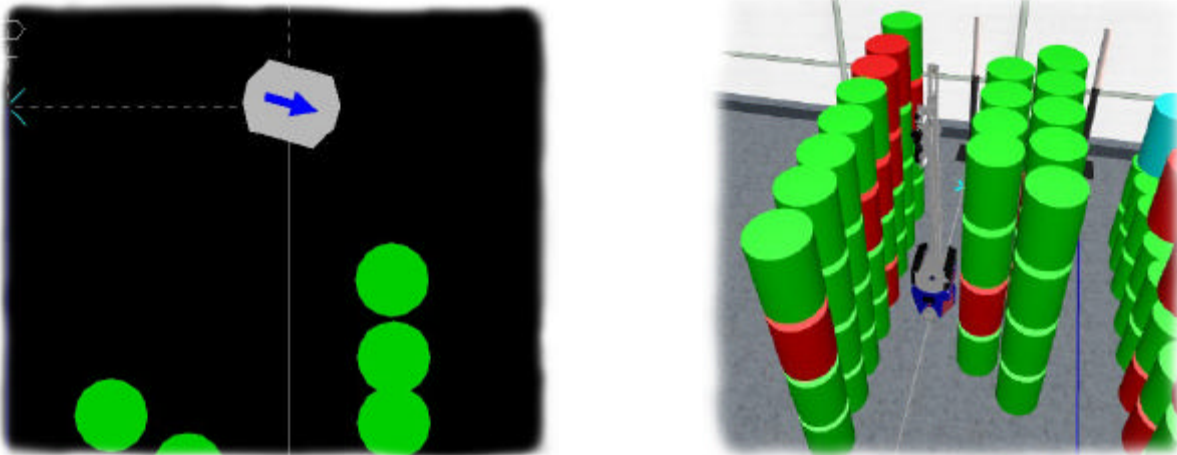


Figure 13 (2D and 3D robot display)

A.5.2 Navigation Information

If a robot is currently being displayed as described in the previous section, additional navigational information can be gathered from the robot. *Select Robots*→*Navigation diagnostics*. Figure 14 shows a typical screen from the Site Manager when both display and navigation information have been selected.

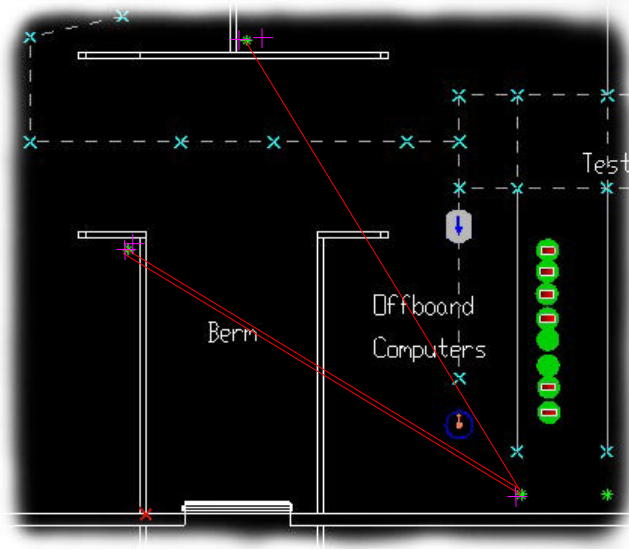


Figure 14 (Navigation diagnostics)

The solid red lines indicate that the robot has made a navigational correction based on a positive LIDAR fit. The line is drawn between what the robot thinks is the center of the two targets used to make the correction. This information is useful for troubleshooting in conditions where the robot is consistently veering from its intended path. In errant situations, these lines will get progressively farther away from the actual location of the targets. The small “+” signs indicate one of several things based on the following color code:

- Magenta - LIDAR target
- Green - drum center
- Cyan - wall corner

A.5.3 Monitor

Monitor is a separate program that can be invoked via the Site Manager interface, using *Robot→Monitor current*. Monitor was designed to help keep the user informed as to the status of the ARIES vehicle while it is inspecting a site. It allows the user to monitor specified variables on-board ARIES in real-time, displaying them via a variety of indicators using OpenGL graphics.

Controls are provided for specifying which variables are to be monitored, including several variable databases containing over 1,000 preset entries. The user may also add user-defined variables to the database, monitor higher level states, and write values to any of the ARIES computers.

A.5.3.1 Invocation

Monitor will most often be invoked from the Site Manager application. It will automatically begin monitoring the currently selected vehicle.

Monitor is also designed to be invoked directly from an IRIX command prompt. Typing “monitor” will give the following usage:

ARIES Monitoring Program v3.0

```
Usage: monitor [options] <robot name>
      -h                               display this help message
      -f <.mon file>                   preload monitor file
      -r <update rate (ms)>            indicator update rate
      -aa                              antialias lines
```

- <robot name> corresponds to the name of the on-board computer connected to the radio Ethernet bridge. Use the name “none” to run monitor in a diagnostic mode that allows the program to function without a connection to an actual vehicle.
- ‘-h’ will give the above usage statement.
- ‘-f <.mon file>’ allows the user to pass a file containing a particular saved Monitor configuration. Monitor requires that the file be specified with the correct path.
- ‘-r <update rate (ms)>’ sets the update frequency of the Monitor indicator, in milliseconds. This value defaults to 500 ms. Care should be taken not to set the value too high. 33 ms will give a 30 frames/sec update speed, far and away faster than the data can currently be polled from the vehicle.
- ‘-aa’ enables OpenGL antialiasing. This smoothes lines over, removing their pixilated appearance. It is an expensive process, and should only be used on machines that support this feature through hardware.

A.5.3.2 Interface Layout

A.5.3.2.1 Main Interface

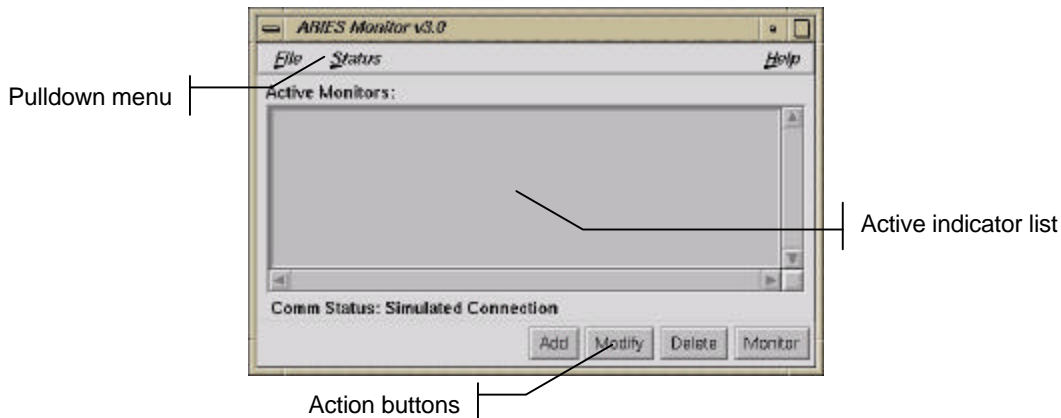


Figure 15 (Monitor interface)

When Monitor is invoked, it is in the state shown in Figure 15, and starts with no monitors defined.

A.5.3.2.1.1 Pull down Menu

The pull down menu contains two panes, File and Status. File contains controls for loading and saving the current list of indicators. It will automatically come up in the directory defined specifically for holding Monitor files within the ARIES installation. When launched from Site Manager, the exit option is disabled. To close Monitor simply double click in the upper left corner.

A.5.3.2.1.2 Active Indicators List

While Figure 15 does not show any active indicators, it will contain all of the defined indicators for the monitoring session. Figure 16 illustrates what a typical list may contain.

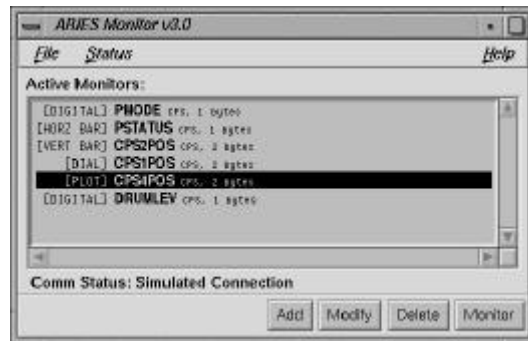


Figure 16 (Active list filled)

Each list entry is formatted as [Indicator Type] [Variable Name] [ARIES Computer] [Size of variable in bytes].

A.5.3.2.1.3 Action Buttons

Press *Add* to invoke the indicator dialog. This will let the user define new indicators to be added to the active list. If an indicator is selected in the list, *Modify* will invoke the indicator dialog with the controls set for that particular indicator. *Delete* will remove the indicator from the list.

Monitor initiates communication between the application and the ARIES on-board computer. It will invoke the indicator displays and update them at a preset frequency.

A.5.3.2.2 Indicator Dialog

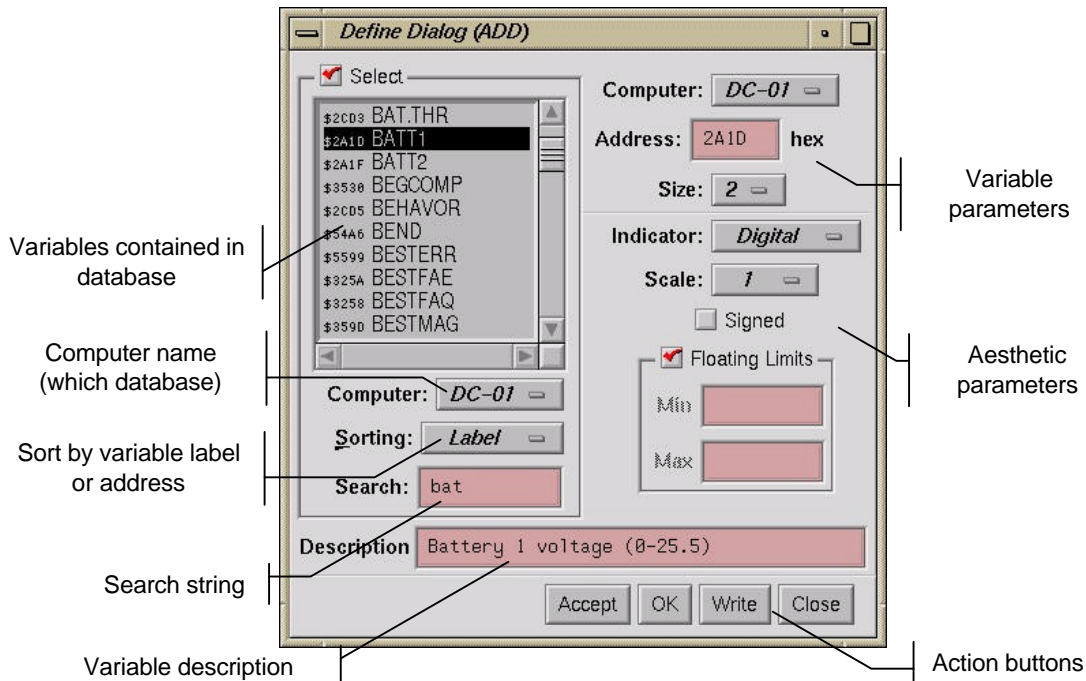


Figure 17 (Indicator dialog)

The indicator dialog is used to define and modify individual indicator attributes. Figure 17 shows the layout of the dialog when a particular variable from the database is chosen.

A.5.3.2.2.1 Database Browser

The left section of the dialog allows the user to browse the databases for the ARIES vehicle. The current database is displayed in the top list, listed by address and variable name. The *Computer* option menu changes which database is active in the list. There is a variable database for each of the computers on-board ARIES.

To help browse the database more efficiently, controls are provided for searching and sorting the current database. The *Sorting* option menu allows the user to choose between sorting the database entries by name or by address. As the user types in the *Search* text box, the list is dynamically searched for entries that match the letters in the box.

A.5.3.2.2.2 Defining Variables Outside the Database

When a variable is to be monitored that is not predefined in any of the databases, the control located in the upper right corner can be used to set its essential parameters. A variable's *Computer*, *Address*, and *Size* define it completely.


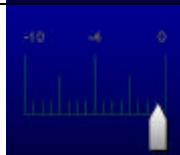
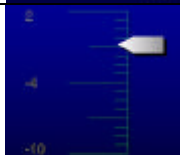

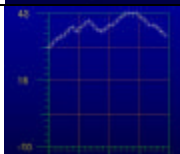
| Name | Display |
|----------------------|---|
| Digital |  |
| Horizontal Bar Graph |  |
| Vertical Bar Graph |  |
| Needle and Dial |  |
| Plot vs. Time |  |

Table 1 (Indicator displays)

A.5.3.2.2.3 Indicator Types

The section in the middle-right of the dialog contains aesthetic parameters of the database. These are not necessary for the variable definition, but allow for more control over how the variable will be indicated. In addition to scale, sign, and limit controls is the “Indicator:” option menu. It allows the user to choose among five different indicator forms:

A.5.3.2.2.4 Action Buttons

Among the action buttons is *Write*. For the variable currently defined in the dialog, a value can be written to that address. Monitor will take into account both the sign and size of the variable when it is written. Figure 18 shows the layout of the write dialog.

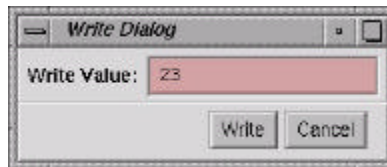


Figure 18 (Write dialog)

A.5.3.2.3 Monitoring Variables

The action button *Monitor* from the main interface initiates communication between the application and the ARIES on-board computer. It will invoke the indicator displays and update them at a preset frequency.

A.5.3.2.3.1 ARIES High Level Status

Aside from detailed variables from assorted locations on the on-board computers, Monitor also provides high-level status information about the ARIES vehicle (shown in Figure 19). This is the “executive summary” status, determined from a series of variables from different computers.

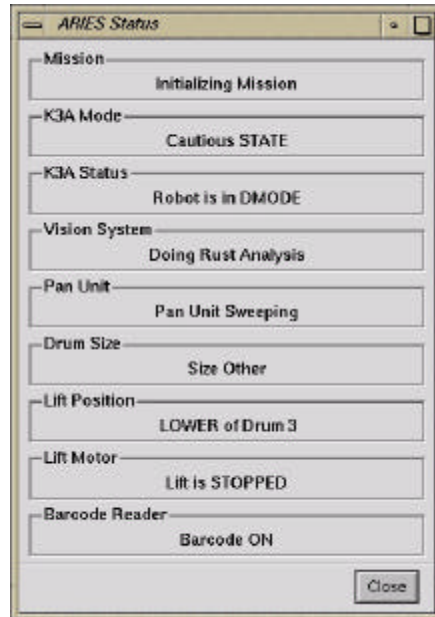


Figure 19 (ARIES status display)

A.5.3.2.3.2 Indicator Display

Figure 20 illustrates the layout of the indicator display. The indicators are arranged in a configurable grid. Action buttons *More Cols* and *More Rows* are provided to modify the arrangement of the grid. Figure 21 shows a better arrangement of the indicators, one that takes advantage of the shape of the window.

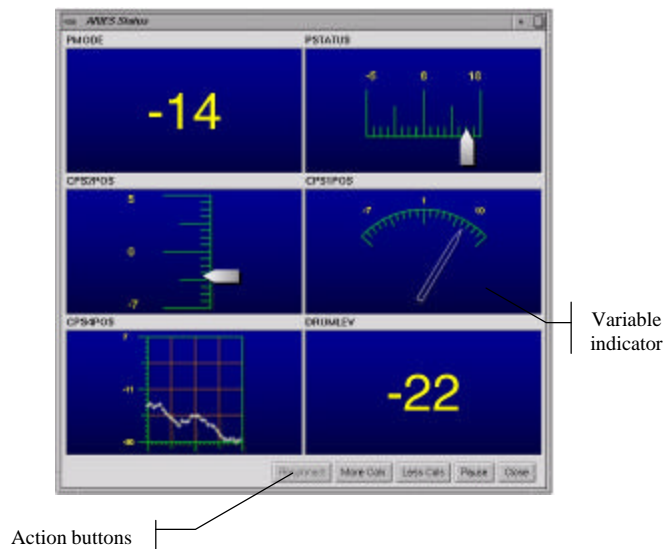


Figure 20 (Indicator display)

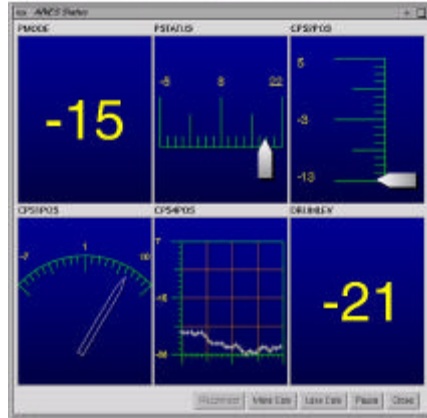


Figure 21 (Alternative grid arrangement)

Figure 22 further demonstrates the effects of the *More Cols* and *More Rows* action buttons. Resizing the indicator window also helps the indicators cover the maximum amount of space. Among the control buttons is *Pause* which temporarily discontinues communication with the ARIES vehicle.

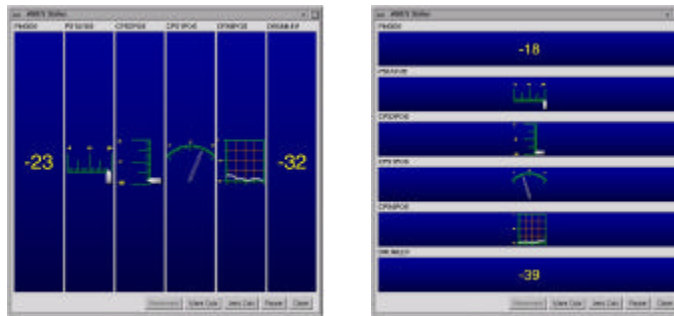


Figure 22 (Maximum columns and rows)

A.6. Mission Reports

Each robot generates a mission report at the conclusion of an inspection. The status of each aisle inspected as well as a second by second history of the robot's activities are included in the report. The latest mission report from the current robot can be viewed by selecting *Building*→*View mission report*. The dialog shown in Figure 23 allows users to view the contents of the mission report.

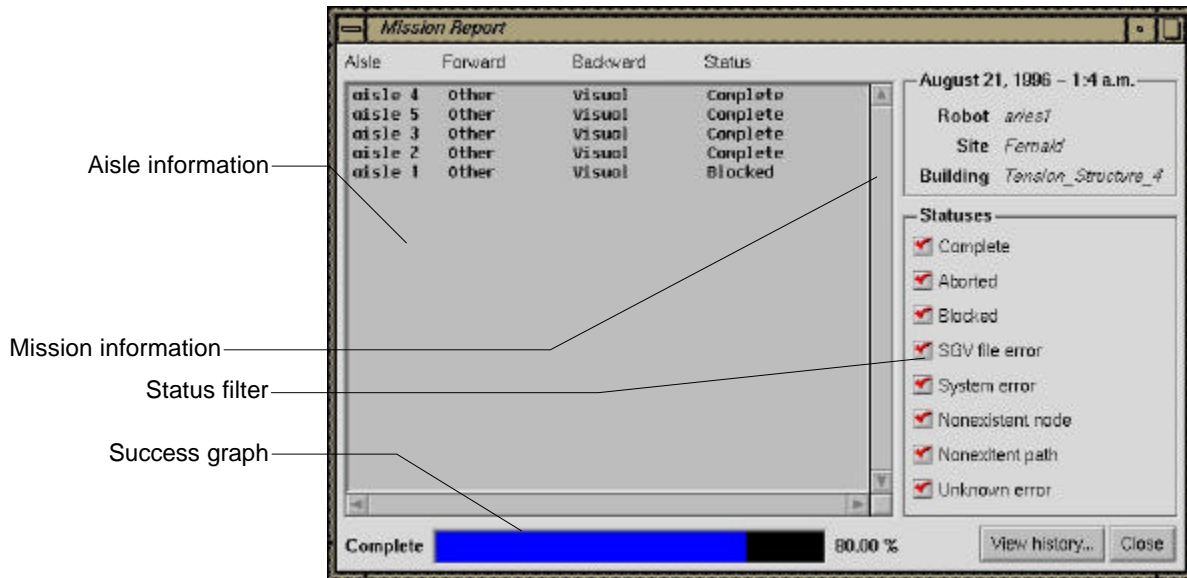


Figure 23 (*Mission Report* dialog)

A.6.1 Mission Information

The mission information section provides high-level mission information. The time and date the mission started, the building inspected, and the inspection vehicle are all listed in this section.

A.6.2 Aisle Information

The aisle information area lists each aisle that was supposed to be inspected on the mission. Aisles are listed in the order in which they were inspected. Aisles are identified by either the user-defined name or the aisle number. Other information includes the inspection type for both entering and exiting the aisle and the status of the robot after inspecting the aisle. The eight possible statuses are listed in the *Statuses* section. Note that the status refers only to whether or not the aisle was inspected, not the state of the drums on the aisle. Clicking on any aisle in the list causes it to be highlighted by a white box in the map region (Figure 24). If the selected aisle was not completed, a white “X” will appear where the error occurred. This marker is useful for determining the location of aisle blockages or sections of an aisle that are too narrow to navigate.

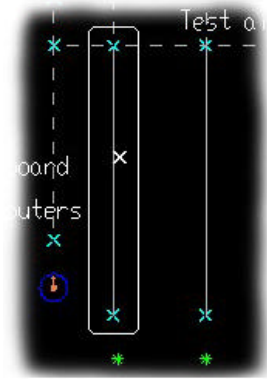


Figure 24
(Highlighted aisle)

A.6.3 Status Filter

The toggle buttons in the *Statuses* section can be used to filter out what is shown in the aisle list. Only those aisles whose status matches a checked toggle button will be listed. It is often useful to list all statuses except those that are “complete”. This filter will provide a list of only those aisles where an error occurred.

A.6.4 Success Graph

The success graph provides a quick method for viewing the overall success of a mission. The graph indicates the percentage of the aisles which received a “complete” status.

A.6.5 View History

A detailed account of the robot's activities are stored with each mission report. This history can be viewed on both the 2D and 3D maps. Viewing the history is like watching a movie of the inspection mission. Pressing the *View history* push button pops up the dialog shown in Figure 25.

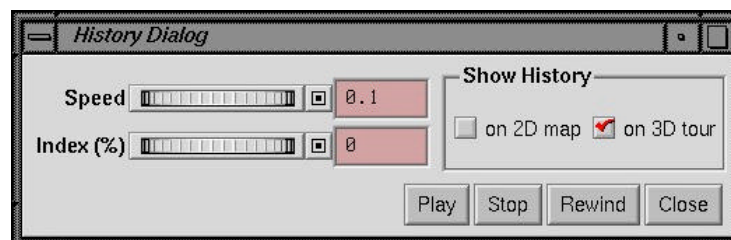


Figure 25 (*History Dialog*)

The toggle buttons in the *Show History* section control which maps the history will be shown on. Use the speed thumb wheel to control how quickly the animation is played. The index thumb wheel allows the user to control precisely which part of the animation is played. The Play, Stop, and Rewind buttons act just like their counterparts on a VCR. Press Play to start the animation, Stop to stop it, and Rewind to return to the beginning. Closing the dialog causes the history animation to disappear from both maps.

A.7. 3D Site Inspection

The *3D Tour* provides a method for visualizing a site that simple 2D maps cannot furnish. The use of 3D Tour requires a 3D AutoCAD (or Open Inventor) file of the site. This file is typically specified during Site creation (see A.3.3). The *3D Tour* allows users to monitor not only the position of the robot, but also the movements of its cameras. The entire drum database can viewed at once as opposed to one level at a time on a 2D map. Each component of a site from docks to targets to the actual building are rendered as life-like 3D objects as opposed to being represented by symbols on a 2D map. To launch the *3D Tour* (Figure 26) select *Building*→*3D Tour*.

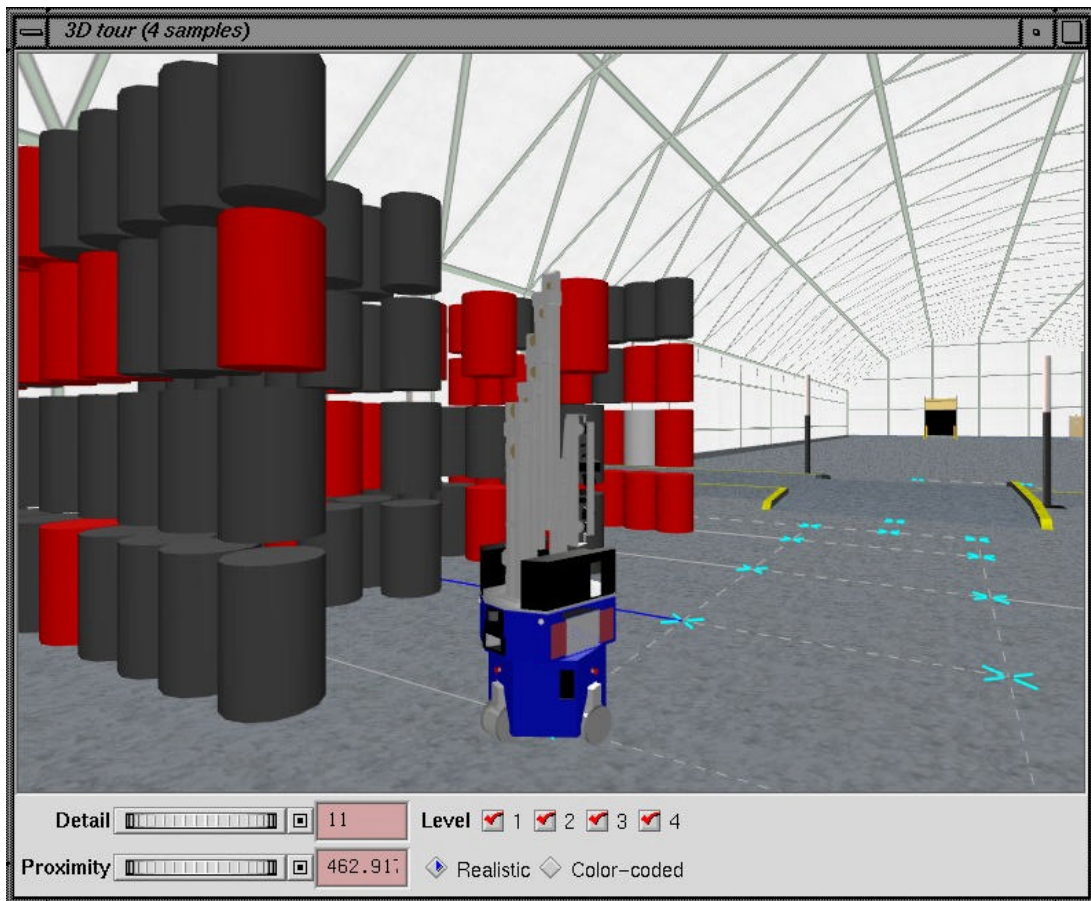


Figure 26 (*3D Tour*)

A.7.1 Viewing Options

The viewing options are accessed via a pop up menu. Press the right mouse button anywhere over the 3D window to pop up the menu hierarchy shown in Figure 27.

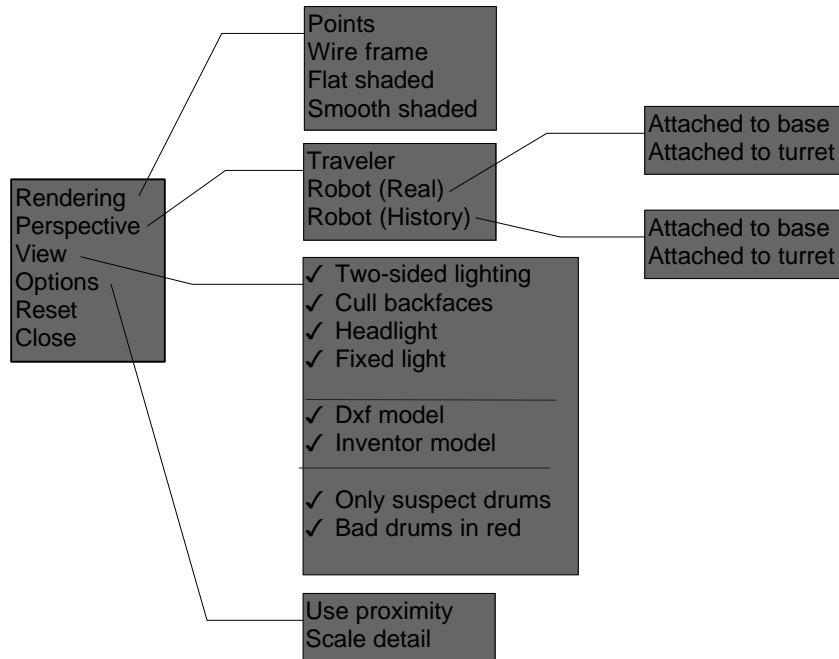


Figure 27 (3D Tour menu hierarchy)

A.7.1.1 Rendering

Four different rendering options are available. *Points* simply renders each vertex in the 3D scene as a point draws no connecting lines or faces. This rendering mode is the fastest but is by far the lowest quality. *Wire frame* rendering is slightly slower but connects the adjacent vertices of each object with lines providing a better view of the scene. *Flat shaded* rendering draws each item as a series of connected polygons. *Smooth shaded* rendering goes one step further by using shading to help approximate curved surfaces and lighting effects. Although this rendering technique is the slowest, it provides the highest quality images.

Figure 28 shows examples of the same scene rendered using each of the four rendering techniques.

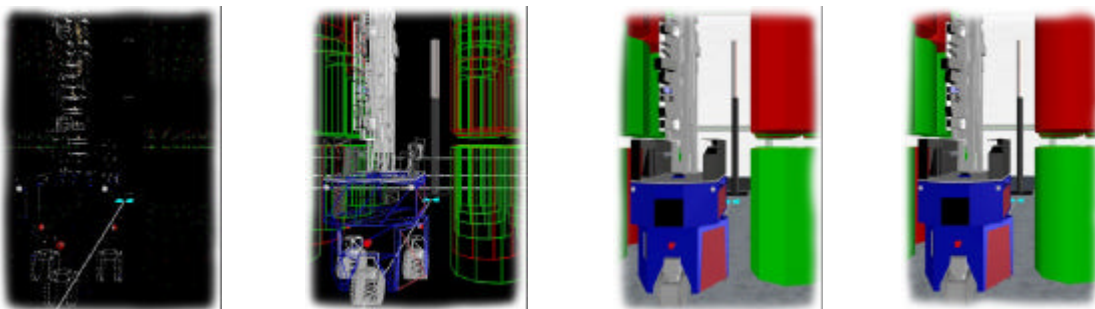


Figure 28 (Points, Wire frame, Flat shaded, and Smooth shaded scenes)

A.7.1.2 Perspective

Several perspectives are provided for viewing and navigating the 3D scene. The following list describes each perspective and how mouse movements are used to navigate within it. The navigation methods will require a little practice for those users unaccustomed to 3D interfaces.

- *Traveler* - This perspective acts as though the user were a traveler in the scene. Pressing only the left mouse button and moving the mouse in a vertical manner causes the traveler to walk forward and backward. Horizontal motion with the left button pressed rotates the traveler left and right. The middle mouse button in combination with vertical motion raises and lowers the traveler's head.
- *Robot (Real)* - This perspective attaches the user to the robot being displayed (This option is only valid if a robot is currently being displayed). The user can attach to the base (fixed orientation) or the turret (revolving orientation) of the robot. As the robot moves around the building, the user will travel with it. Pressing the left mouse button in combination with vertical movement increases and decreases the distance between the user and the robot. Horizontal motion with the left button pressed changes the angle at which the user is attached. Pressing the middle button allows users to modify not only the angle of attachment with respect to the 'z' axis of the robot (up), but also with the 'x' axis of the robot. Pressing both the left and middle mouse buttons in combination with vertical motion causes the user to look up and down based on the point of attachment.
- *Robot (History)* - If a history is being played, this perspective will attach the user to the history robot. Navigation in this mode works identically to that of the *Robot (Real)* perspective.

The Site Manager will remember the latest navigation settings for each perspective. Therefore, if the user switches from one perspective to another, the navigation settings will be restored the next time the perspective is chosen. This feature allows users to jump from one perspective to another without having to readjust the viewing position. The *Reset* button near the bottom of the menu will reset the current perspective to its default values. *Close* closes the *3D Tour*.

A.7.1.3 View

The *View* menu provides a list of rendering options that can be toggled on and off by the user. The following list details each entry in the menu.

- *Two-sided lighting* - Two-sided lighting should be turned off for well organized 3D models. If some faces appear very dark while others adjacent to it are lit correctly, turn this option on.
- *Cull backfaces* - Polygons that face away from the user do not need to be drawn. Turning this option on will greatly increase the frame rate of the scene. If some of the polygons in the scene disappear, turn this option off.
- *Headlight* - Turning this option on will light the scene as if the user were wearing a miner's hat.
- *Fixed light* - This option turns on the fixed light source that is automatically placed at the origin of the building.

- *DXF model* - When this option is selected the 2D map will be superimposed over the floor of the 3D scene. This option is particularly useful when the text of the 2D map is used to label different sections of a large building;
- *Inventor model* - This option determines whether the 3D model of the building is drawn. Turning this option off for complex scenes may increase the response of the system.
- *Only suspect drums* - If selected, this option causes only those drums that are bad to be drawn. Doing so provides a quick method for locating all of the bad drums in a building.
- *Bad drums in red* - If selected, all bad drums will be drawn in red. Otherwise, the bad drums will be drawn in their actual or coded color, depending on the selected mode.

A.7.1.4 Options

Both of the selections in the options menu address drum rendering. The first selection, Use proximity, controls whether or not the proximity field is used to cull drums (drum culling is discussed in detail in the following section). The second selection, Scale detail, applies only when the proximity field is in use. This selection causes the level of detail of each drum to be based on its distance from the viewer. Drums farther away will be drawn in lower detail than those close to the observer. Activating detail scaling should result in a dramatic increase in frame rate.

A.7.2 Drum Culling and Rendering

Drum culling is the process by which some of the drums in a building are not drawn based on either the observer's position or the user's selection. Drum culling is necessary due to the large number of drums (up to 12,000) that can be stored in a typical building. The frame rate on a typical graphics workstation would become so low as to make the *3D Tour* unusable if all of the drums in a building were drawn. Figure 29 shows the drum culling and rendering options available at the bottom of the *3D Tour* window.

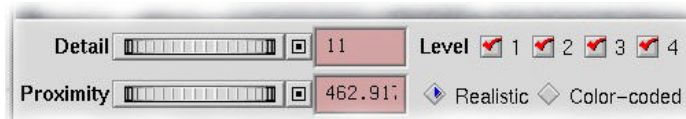


Figure 29 (Drum culling and rendering options)

A.7.2.1 Detail

The *Detail* thumb wheel and text box control how much detail is used to render each drum. Because curved surfaces, such as cylinders, can not be rendered exactly by computers, they must be approximated by a series of connected polygons. Changing the detail level simply changes the number of polygons used to approximate each drum. The higher the number, the better the drums will look and the longer it will take to render each drum. Try to find a setting where both the drum quality and frame rate are acceptable. Figure 30 shows the same scene rendered with low and high detail drums.

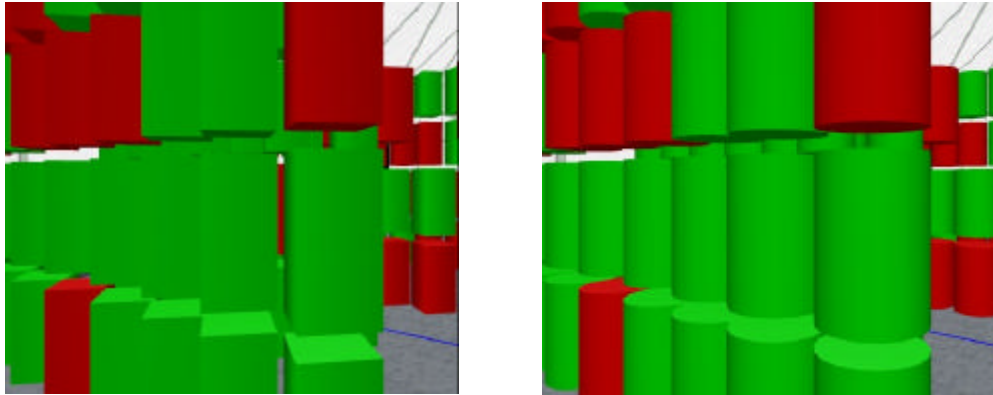


Figure 30 (Low (left) and high (right) detail drums)

A.7.2.2 Proximity

Changing the detail level of the drums is often not sufficient for achieving an acceptable frame rate. The *Proximity* thumb wheel and text box control the area around the observer in which drums are rendered. If the *Use proximity* option is selected, only those drums within the specified distance from the observer will be rendered. Using the proximity distance to cull drums allows users to see drums in their immediate area while allowing Site Manager to increase the frame rate of the *3D Tour* by not rendering unnecessary drums. Additionally, if the *Scale detail* option is selected, drums are drawn with decreasing detail as their distance from the observer increases.

Figure 31 shows two images of the same scene. The left image was rendered with a very high proximity; therefore, all of the drums are rendered and are rendered with the same level of detail. The right image used a proximity value of twenty and was rendered with the *Scale detail* option on. Notice that the drums are drawn with successively fewer polygons as they get farther away from the observer.

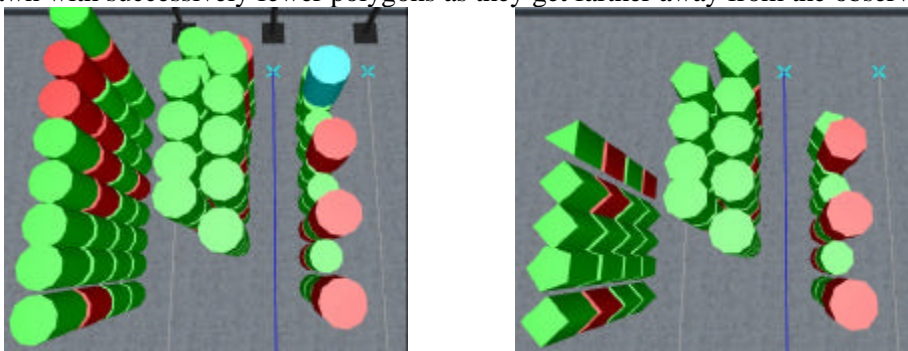


Figure 31 (High (left) and low (right) proximity scenes)

A.7.2.3 Level

The *Level* toggle buttons allow the user to control which drum levels will be drawn. Use these toggle buttons to reduce the number of drums drawn or to focus on a particular drum level.

A.7.2.4 Color Coding

Drums can be rendered in either their actual colors or using a color coding scheme. If *Realistic* colors are selected, the drums are rendered in the color stored in the corresponding database record. If *Color-coded* is selected, drums are rendered using the following color code.

- Bad drums - Red
- 55 gallon drums - Green
- 85 gallon drums - Cyan
- 110 gallon drums - Yellow

A.7.3 Drum Selection

One of the greatest advantages of the *3D Tour* is the ability to graphically view the drum database. Because each drum is located in the correct position within the building, the *3D Tour* allows users to locate drums in the database in the same manner that they would locate drums in the building. Bad drums can quickly be recognized by the user and quickly found in the actual building by those responsible for replacing them. The drum database record for each drum can be accessed by clicking on any drum from any perspective with the left mouse button. The database record viewer described in the next section is popped up and the drum is outlined by a yellow highlight box.

A.8. Database Management

Separate databases are kept for the most recent ARIES vehicle mission and for the entire facility defined in the Site Manager. The databases are stored in a proprietary binary format and special tools are needed to retrieve the data. Select *Building*→*Read building's drum database* or *Building*→*Read robot's drum database* to bring the desired database into the Site Manager.

A.8.1 Invocation

These databases can be accessed through Site Manager or from a stand alone program called 'ariesddb' (ARIES Drum Database Browser).

A.8.1.1 From Site Manager

'ariesddb' can be invoked from the *Building*→*Database browser* entry in the Site Manager. More conveniently, individual drum records can be accessed from the 3D tour or the 2D map. Simply click on any of the drums, and a drum database record editor will appear, allowing the user to scroll through all of the field values for that drum.

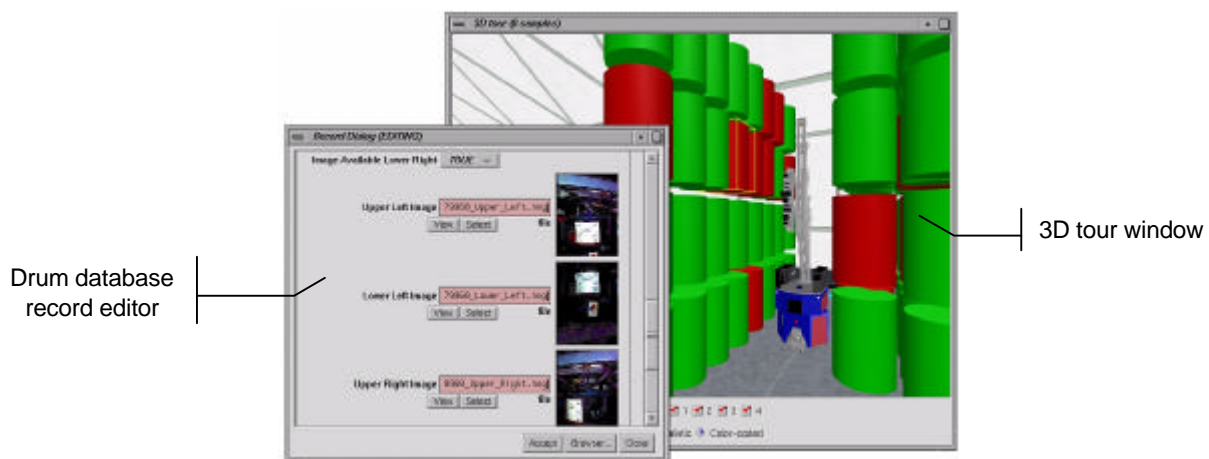


Figure 32 (Record editor via 3D tour)

A.8.1.2 From IRIX Command Prompt

ariesddb is also designed to be invoked directly from an IRIX command prompt. Typing "ariesddb" will give the following usage:

```
ARIES Database v2.0  
Usage: ariesddb [.db file]
```

- <.db> corresponds to the name of the ARIES drum database to be browsed.

A.8.2 Browser Interface Layout

A.8.2.1 Main Interface

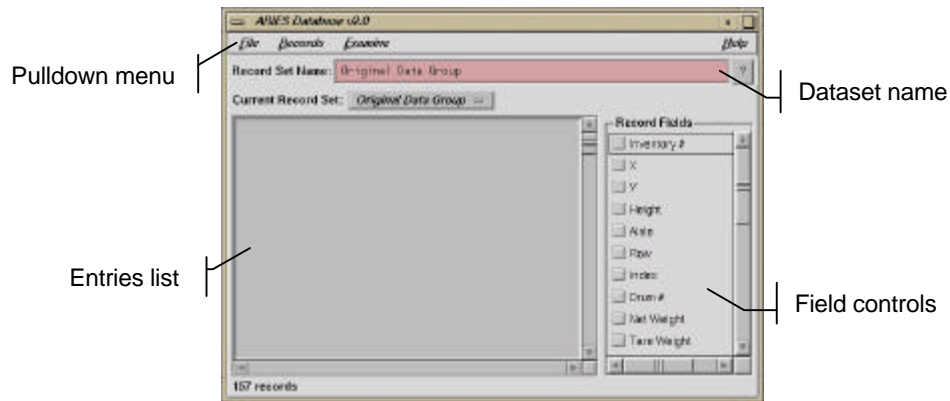


Figure 33 (ariesddb main interface)

When ariesddb is invoked, it is in the state shown in Figure 33, and starts with no records in the entries list.

A.8.2.1.1 Pull down Menu

The pull down menu contains three panes: *File*, *Records*, and *Examine*. *File* contains controls for loading and saving the current database. It will automatically come up in the directory defined specifically for holding ARIES database files for a particular site or vehicle. When launched from Site Manager, the exit option is disabled. To close ariesddb simply double click in the upper left corner.

Records contains entries for adding, editing, and deleting records in the database. *Examine* will pop up the search dialog in either search or sort mode.

A.8.2.1.2 Entries List

While Figure 33 does not show any record entries, it will contain some or all of the fields and their corresponding values for all of the records in the database. The controls on the right side of the interface control what fields will be shown for each record.

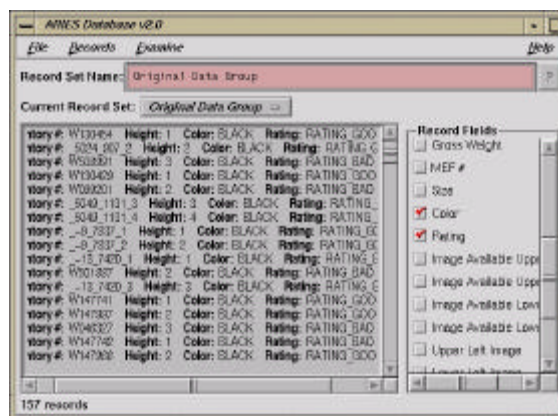


Figure 34 (Record field controls)

Each record is listed (shown in Figure 34) with the field name first, following by that field's value for that record.

A.8.2.2 Search/Sort Dialog

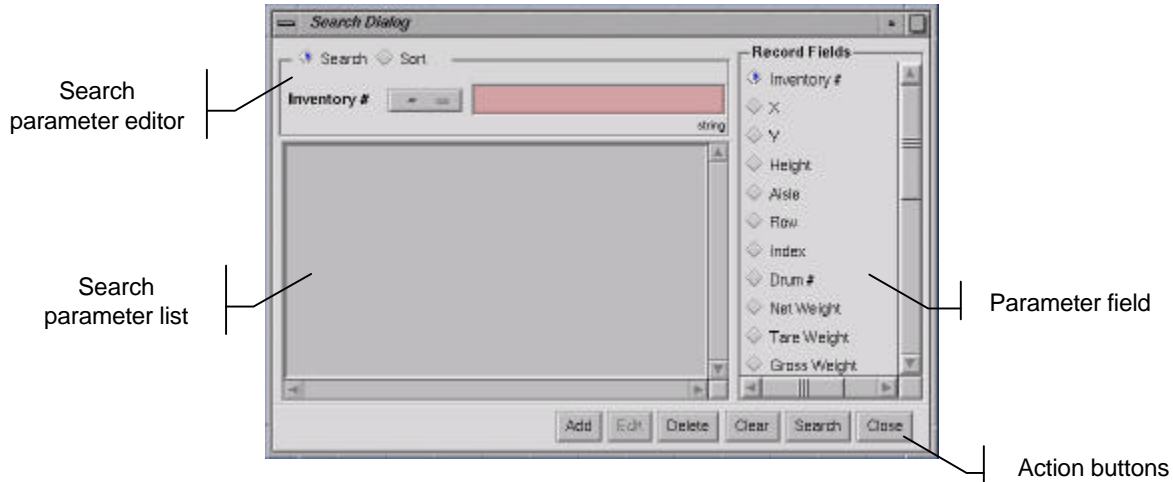


Figure 35 (Search dialog)

A.8.2.2.1 Searching

Searching the database for records meeting certain criteria is accomplished through the search dialog (Figure 33). How to search for something is best demonstrated by example:

Suppose that we are interested in finding all of the drums that have a HEIGHT greater than 2, COLOR of black, and a known RATING. Click HEIGHT from the right set of radio buttons, and enter the top left section of the dialog. Notice that it says HEIGHT and gives an editor type based on the type of field selected. Enter “2” and select ‘>’ from the list of comparison operators. Go to the set of action buttons and click ‘Add’. This particular search field now appears in the search fields list. Add the other two search fields by repeating this procedure for “COLOR = black” and “RATING != unknown.”

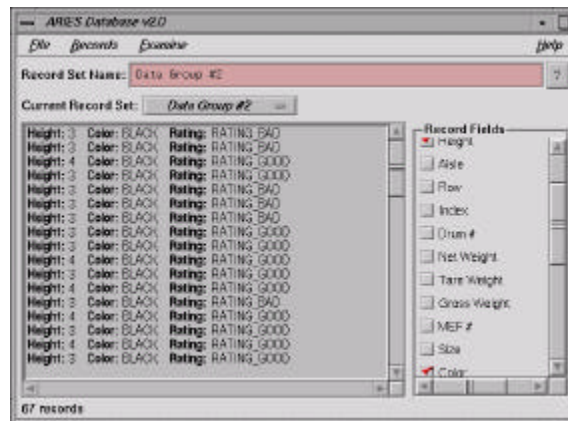


Figure 36 (Search results)

Figure 13 shows the main interface after a search has been completed. The user is able to view any field from the records, not just the ones used in the search.

A.8.2.2.2 Sorting

In the search parameter section of the search dialog, a radio box contains buttons for both search and sort. At any point in adding new search criteria to the search list, a sort may be added. Simply select a field and choose if the records should be sorted in ascending or descending order.

A.8.2.3 Data Set Manipulation

Notice in Figure 36 that data set indicator indicates “Data Group #2.” Each time the database is searched or sorted, a new data set is generated. This allows the user to move freely among all of the data sets, including the original. As the browser moves to new data sets, the 3D tour in Site Manager will show that data set.

A.8.3 Record Editor

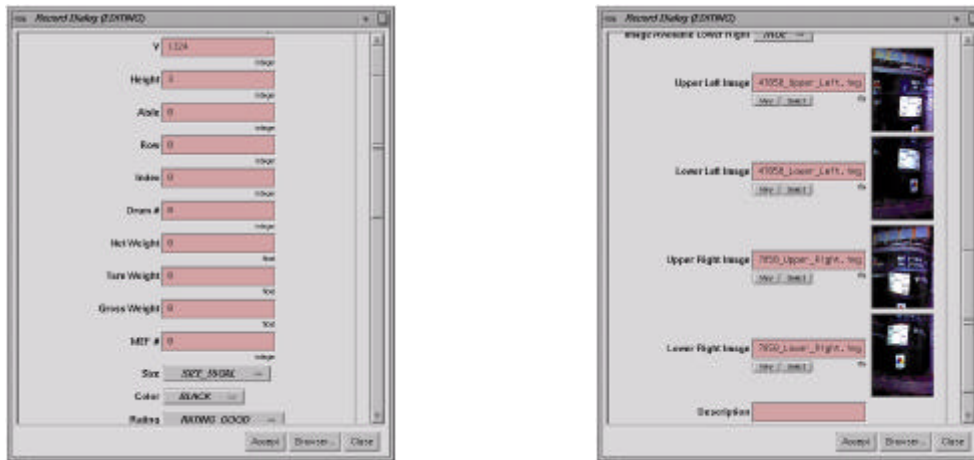


Figure 37 (Record editor)

The record editor, shown in Figure 37 allows the user to view and edit the information contained in all fields of the record.

A.8.3.1 Field Types

Field types supported by ARIES include:

| Field Type | Field Editor |
|-----------------|-----------------------------------|
| Integer | Text box |
| Char | Text box |
| Float | Text box |
| Enumeration | Option menu |
| File | Text box + file selector controls |
| Time | Text box |
| Character Array | Text box |

Table 2 (Editor field types)

A.8.3.2 Images

Images are treated differently than the other fields. While all other fields are generic, image fields are reserved for ARIES image files specifically.

A.8.3.2.1 Image Fields

A window is provided beside the text box containing the filename of the image. This allows a small preview of the image file read by the ARIES vehicle. Controls are provided for reassigning this field via a file selector, as well as viewing it more closely. Pressing the “View” button will activate the *gviewer* utility.

A.8.3.2.2 *gviewer* Utility

gviewer, shown in Figure 38, allows for a closer view of the image. Once up, the user can use the up and down arrow keys to zoom or shrink the image. The escape key will exit the application.

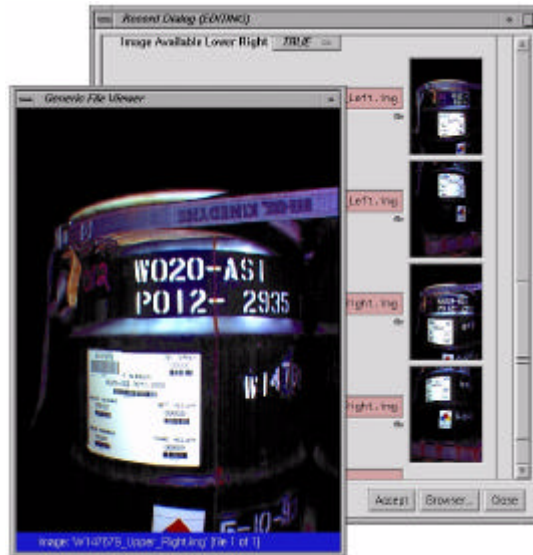


Figure 38 (*gviewer* utility)

gviewer is also a stand-alone IRIX application. It can be invoked on any *img* file created by the ARIES vision software. When invoked on multiple *img* files, the left and right arrow keys will move through the images.

A.8.4 Database Utilities

Because the ARIES database format is in binary, files cannot be edited or viewed with standard text editors. These utilities were added for easier manipulation of the ARIES drum databases.

A.8.4.1 *dbtool*

When invoked from the IRIX command prompt, *dbtool* gives the following usage:

```
ARIES Drum Database Tool v0.x

Usage: dbtool [options] <mode>
where [options] are:
    -I    case insensitive field matching
    -v    verbose mode

where <mode> is:
-merge  (merge one database with another)
args: <new db> <out db> <field name>
* the merger will automatically extract the
  name of the database to merge with from
```

the header of the <new db>.

```
-rename (rename a field in a database)
      args: <in db> <out db> <old field> <new field>

-convert (convert a database to a new template)
      args: <in db> <out db> <template>
```

- ‘-I’ will make *dbtool* case insensitive when scanning for field names.
- ‘-V’ will make *dbtool* report each step it is taking when executing a command.
- ‘-merge’ merges a database brought back from an ARIES vehicle with the master database on the off-board computers.
- ‘-rename’ copies the input database to a new database, with one field name changed.
- ‘-convert’ copies one database to another given a new database template. Any fields that match from the input database to the given template will be copied.

A.8.4.2 *dbutil*

When invoked from an IRIX command prompt, *dbutil* gives the following usage:

```
Drum Database Utility v1.0

Usage: dbutil <option>
      -gen          <filename> <count> <desc>
      -header      <dbfile> <header string>
      -read        <filename>
      -print       print data
      -search      <dbfile> <fieldname> <value>
      -sort        <dbfile> <fieldname>
      -stats       <filename>
      -subset      <filename>
```

- ‘-gen’ will generate a new database file with random record field values.
- ‘-header’ changes the user header value of the database. ARIES uses this field to store information about the site and mission the database was created in.
- ‘-read’ will read the entire contents of the database. Often used to time the speed of the database reader.
- ‘-print’ cause *dbutil* to print the records as it operates in them.
- ‘-search’ performs a simple search of a database for a single field value.
- ‘-sort’ performs a simple sort of a database for a single field value.
- ‘-stats’ prints low-level information about the internal organization of the database file. An example output:

```
      database size:      98304 bytes
      block size:        512 bytes
      total blocks:      192 blocks
      record table size: 790 bytes
      delete table size: 5 bytes
      format table size: 771 bytes
      total records:     157
      empty records:     0
smallest record size:    112 bytes
average record size:    440 bytes
largest record size:    522 bytes
      deleted blocks:    0
```

total efficiency: 70.27 %
overhead space: 3.36 %
unused space: 26.37 %

- '-subset' is reserved for ARIES software developers.

Program Flow: Outline

1. `umask()` [`io.h`] and [`sys/stat.h`] and [`sys/types.h`] (`_umask()`)
2. `XUInitErrorLog()` [`xu_error.c`]
3. `InitRandom()` [`mon_main.cpp`]
4. `Getenv()` [`stdlib.h`]
5. `sprintf()` [`stdio.h`]
6. `strcat()` [`string.h`]
7. `XUProgError()` [`error.c`]
8. `ParseCommandLine()` [`sm_main.cpp`], [`adb_main.cpp`], [`dbt_main.cpp`], [`dbu_main.cpp`], [`mon_main.cpp`]
9. `Gethostname()` [`winsoc2.h`]
10. `InitParameters()` [`sm_main.cpp`]
11. `XtVaAppInitialize()`
12. `sprintf()`
13. `XtVaSetValues()`
14. `XtDisplay()`
15. `XtScreen()`
16. `DefaultGC()` [`sm_xwin.cpp`]
17. `DefaultColormap()`
18. `InitX()`
19. `ChangeRobotMenu()` [`sm_robots.cpp`]
20. `SetUnits()` [`sm_utility.cpp`]
21. `FillItemList()` [`sm_items.cpp`]
22. `sprintf()`
23. `XULabelSetString()` [`xu_utility.cpp`]
24. `NewMessage()` [`sm_utility.cpp`]
25. `XtAppMainLoop()`

Program Flow: Ring One Detail

1. ***umask()*** [*io.h*] and [*sys/stat.h*] and [*sys/types.h*] (***_umask()***)
2. ***XUInitErrorLog()*** [*xu_error.c*]
 - 2.1. *fopen()*
 - 2.2. *XUSystemError()* [*xu_error.c*]
 - 2.3. *fchmod()*
 - 2.4. *fileno()*
 - 2.5. *XUProgError()* [*error.c*]
3. ***InitRandom()*** [*mon_main.cpp*]
 - 3.1. *time()*
 - 3.2. *srand()*
4. ***Getenv()*** [*stdlib.h*]
5. ***sprintf()*** [*stdio.h*]
6. ***strcat()*** [*string.h*]
7. ***XUProgError()*** [*error.c*]
 - 7.1. *fprintf()*
 - 7.2. *va_start()*
 - 7.3. *vsprintf()*
 - 7.4. *time()*
 - 7.5. *localtime()*
 - 7.6. *strftime()*
 - 7.7. *fflush()*
 - 7.8. *va_end()*
 - 7.9. *sprintf()*
 - 7.10. *FatalErrorMessage()* [*xu_error.c*]
 - 7.11. *ErrorMessage()* [*xu_error.c*]
 - 7.12. *Strcmp()*
 - 7.13. *exit()*
8. ***ParseCommandLine()*** [*sm_main.cpp*], [*adb_main.cpp*], [*dbt_main.cpp*], [*dbu_main.cpp*], [*mon_main.cpp*]
 - 8.1. *sprintf()*
 - 8.2. *strcmp()*
 - 8.3. *fprintf()*
 - 8.4. *GiveHelp()* [*sm_main.cpp*]
 - 8.5. *atoi()*
9. ***Gethostname()*** [*winsock2.h*]
10. ***InitParameters()*** [*sm_main.cpp*]
 - 10.1. *ParseWarehouseFile()* [*sm_parser.cpp*]
 - 10.2. *InitNullHouse()* [*sm_house.cpp*]
11. ***XtVaAppInitialize()***
12. ***sprintf()***
13. ***XtVaSetValues()***
14. ***XtDisplay()***
15. ***XtScreen()***

16. DefaultGC()

17. DefaultColormap()

18. InitX()

- 18.1. GetColors() [sm_xwin.cpp]
- 18.2. XmCreateForm()
- 18.3. CreateMenuBar() [sm_xwin.cpp]
- 18.4. XmCreateForm()
- 18.5. XtVaSetValues()
- 18.6. OGLXCreateColorEditor()
- 18.7. CreateMap()
- 18.8. CreateToggleIcons()
- 18.9. CreateLogo()
- 18.10. XmCreateLabel()
- 18.11. XtVaSetValues()
- 18.12. XULabelSetString() [xu_utility.cpp]
- 18.13. XtManageChild()
- 18.14. CreateRobotArea() [sm_xwin.cpp]
- 18.15. CreateItemListArea() [sm_xwin.cpp]
- 18.16. CreateDockShell() [sm_docks.cpp]
- 18.17. CreatePathShell() [sm_xpath.cpp]
- 18.18. CreateMissionShell() [sm_md1.cpp]
- 18.19. CreatePasmShell() [sm_xpa.cpp]
- 18.20. CreateRDLShell() [am_rdl.cpp]
- 18.21. CreateTourShell() [sm_tour.cpp]
- 18.22. CreateHistoryShell() [sm_history.cpp]
- 18.23. InitBrowserState() [sm_db_int.cpp] [db_init.cpp] [adb_init.cpp]
- 18.24. XtManageChild()
- 18.25. XtRealizeWidget()
- 18.26. CreateCursors() [sm_xwin.cpp]
- 18.27. InitMap() [sm_map.cpp]
- 18.28. GetCurrentTime() [sm_utility.cpp]

19. ChangeRobotMenu() [sm_robots.cpp]

- 19.1. XtUnmanageChild()
- 19.2. XtDestroyWidget()
- 19.3. LLRetrieve()
- 19.4. // XfreeColors()
- 19.5. MakeNullMenu() [sm_robots.cpp]
- 19.6. XmCreatePushButton()
- 19.7. XtVaSetValues()
- 19.8. XtAddCallback()
- 19.9. XmCreatePulldownMenu()
- 19.10. XmCreateOptionsMenu()
- 19.11. InstallHelpText() [sm_utility.cpp]
- 19.12. XtMalloc()
- 19.13. XallocColor()
- 19.14. XmCreateToggleButton()
- 19.15. XtManageChild()
- 19.16. ChangeRobotToggle() [sm_robots.cpp]

20. SetUnits() [sm_utility.cpp]

- 20.1. XtVaSetValues()
- 20.2. ResetLights() [sm_3Ddraw.cpp]

21. FillItemList() [sm_items.cpp]

- 21.1. XmListDeleteAllItems()
- 21.2. sprintf()
- 21.3. sprintf()
- 21.4. PadString() [sm_utility.cpp]
- 21.5. PadString() [sm_utility.cpp]
- 21.6. sprintf()
- 21.7. XmStringCreateSimple()
- 21.8. XmListAddItem()
- 21.9. XmListSelectPos()
- 21.10. XULabelSetString() [xu_utility.cpp]

22. sprintf()

23. XULabelSetString() [xu_utility.cpp]

- 23.1. XmStringCreateSimple()
- 23.2. XtVaSetValues()
- 23.3. free()

24. NewMessage() [sm_utility.cpp]

- 24.1. XUProgError() [error.c]

25. XtAppMainLoop()

Program Flow: Ring Two Detail

1. *umask()* [*io.h*] and [*sys/stat.h*] and [*sys/types.h*] (*_umask()*)

2. *XUInitErrorLog()* [*xu_error.c*]

2.1. *fopen()*

2.2. *XUSystemError()* [*xu_error.c*]

- 2.2.1. *fprintf()*
- 2.2.2. *exit()*
- 2.2.3. *va_start()*
- 2.2.4. *vsprintf()*
- 2.2.5. *strcat()*
- 2.2.6. *strcat()*
- 2.2.7. *time()*
- 2.2.8. *localtime()*
- 2.2.9. *strftime()*
- 2.2.10. *fprintf()*
- 2.2.11. *fflush()*
- 2.2.12. *va_end()*
- 2.2.13. *sprintf()*
- 2.2.14. *FatalErrorMessage()*
- 2.2.15. *ErrorMessage()*
- 2.2.16. *fprintf()*
- 2.2.17. *exit()*

2.3. *fchmod()*

2.4. *fileno()*

2.5. *XUProgError()* [*error.c*]

- 2.5.1. *fprintf()*
- 2.5.2. *va_start()*
- 2.5.3. *vsprintf()*
- 2.5.4. *time()*
- 2.5.5. *localtime()*
- 2.5.6. *strftime()*
- 2.5.7. *fflush()*
- 2.5.8. *va_end()*
- 2.5.9. *sprintf()*
- 2.5.10. *FatalErrorMessage()*
- 2.5.11. *ErrorMessage()*
- 2.5.12. *strcmp*
- 2.5.13. *exit()*

3. *InitRandom()* [*mon_main.cpp*]

3.1. *time()*

3.2. *srand()*

4. *Getenv()* [*stdlib.h*]

5. *sprintf()* [*stdio.h*]

6. *strcat()* [*string.h*]

7. *XUProgError()* [*error.c*]

7.1. fprintf()

7.2. va_start()

7.3. vsprintf()

7.4. time()

7.5. localtime()

7.6. strftime()

7.7. fflush()

7.8. va_end()

7.9. sprintf()

7.10. FatalErrorMessage() [xu_error.c]

7.10.1. XmCreateWarningDialog()

7.10.2. XtVaSetValues()

7.10.3. XtUnmanageChild()

7.10.4. XmMessageBoxGetChild()

7.10.5. XtAddCallback()

7.10.6. XmStringCreateLtoR()

7.10.7. XmStringFree()

7.10.8. XtManageChild()

7.11. ErrorMessage() [xu_error.c]

7.11.1. XmCreateWarningDialog()

7.11.2. XtVaSetValues()

7.11.3. XtUnmanageChild()

7.11.4. XmMessageBoxGetChild()

7.11.5. XmStringCreateLtoR()

7.11.6. XmStringFree()

7.11.7. XtManageChild()

7.12. Strcmp()

7.13. exit()

8. ParseCommandLine() [sm_main.cpp], [adb_main.cpp], [dbt_main.cpp], [dbu_main.cpp], [mon_main.cpp]

8.1. sprintf()

8.2. strcmp()

8.3. fprintf()

8.4. GiveHelp() [sm_main.cpp]

8.4.1. printf()

8.4.2. exit()

8.5. atoi()

9. Gethostname() [winsock2.h]

10. InitParameters() [sm_main.cpp]

10.1. ParseWarehouseFile() [sm_parser.cpp]

10.1.1. OpenFile()

10.1.2. XUmalloc()

10.1.3. LLCreateList()

10.1.4. NextToken()

10.1.5. ParseSite()

- 10.1.6. CleanUp()
- 10.1.7. ParseBuilding()
- 10.1.8. ParseAcad()
- 10.1.9. ParseIv()
- 10.1.10. ParseRobots()
- 10.1.11. ParseOrigin()
- 10.1.12. ParseTargets()
- 10.1.13. ParseMarkers()
- 10.1.14. ParseDocks()
- 10.1.15. ParsePaths()
- 10.1.16. ReportParseError()

10.2. InitNullHouse() [sm_house.cpp]

- 10.2.1. XUmalloc()
- 10.2.2. LLCreateList()

11. XtVaAppInitialize()

12. sprintf()

13. XtVaSetValues()

14. XtDisplay()

15. XtScreen()

16. DefaultGC() [sm_xwin.cpp]

17. DefaultColormap()

18. InitX()

18.1. GetColors() [sm_xwin.cpp]

- 18.1.1. XWhitePixelOfScreen()
- 18.1.2. XtScreen()
- 18.1.3. XBlackPixelOfScreen()
- 18.1.4. XtVaGetValues()
- 18.1.5. XtVaSetValues()
- 18.1.6. XmCreatePushButton()
- 18.1.7. XQueryColor()
- 18.1.8. XAllocNamedColor()
- 18.1.9. XAllocColor()
- 18.1.10. XtDestroyWidget()

18.2. XmCreateForm()

18.3. CreateMenuBar() [sm_xwin.cpp]

- 18.3.1. XmStringCreateSimple()
- 18.3.2. XmVaCreateSimpleMenuBar()
- 18.3.3. XtNameToWidget()
- 18.3.4. XtVaSetValues()
- 18.3.5. XmStringFree()
- 18.3.6. CreateFilePulldown()
- 18.3.7. CreateEditPulldown()
- 18.3.8. CreateViewPulldown()
- 18.3.9. CreateRobotsPulldown()
- 18.3.10. CreateMapPulldown()
- 18.3.11. CreateBuildingPulldown()
- 18.3.12. CreateHelpPulldown()
- 18.3.13. XtVaSetValues()
- 18.3.14. XtManageChild()

- 18.4. *XmCreateForm()***
- 18.5. *XtVaSetValues()***
- 18.6. *OGLXCreateColorEditor()***
- 18.7. *CreateMap()***
- 18.8. *CreateToggleIcons()***
- 18.9. *CreateLogo()***
- 18.10. *XmCreateLabel()***
- 18.11. *XtVaSetValues()***
- 18.12. *XULabelSetString()* [*xu_utility.cpp*]**
 - 18.12.1. *XmStringCreateSimple()*
 - 18.12.2. *XtVaSetValues()*

- 18.13. *XtManageChild()***
- 18.14. *CreateRobotArea()* [*sm_xwin.cpp*]**
 - 18.14.1. *XmCreateFrame()*
 - 18.14.2. *XmCreateLabel()*
 - 18.14.3. *XtVaSetValues()*
 - 18.14.4. *XmCreateForm()*
 - 18.14.5. *XmCreateRowColumn()*
 - 18.14.6. *XtVaSetValues()*
 - 18.14.7. *XtMalloc()*
 - 18.14.8. *XCreatePixmapFromBitmapData()*
 - 18.14.9. *XtVaCreateManagedWidget()*
 - 18.14.10. *XtAddCallback()*
 - 18.14.11. *InstallHelpText()*
 - 18.14.12. *CreateRobotOption()*
 - 18.14.13. *XtManageChild()*

- 18.15. *CreateItemListArea()* [*sm_xwin.cpp*]**
 - 18.15.1. *XmCreateFrame()*
 - 18.15.2. *XUCreateOptionMenu()*
 - 18.15.3. *XtVaSetValues()*
 - 18.15.4. *InstallHelpText()*
 - 18.15.5. *XmCreateForm()*
 - 18.15.6. *XmCreateLabel()*
 - 18.15.7. *XmCreateScrolledList()*
 - 18.15.8. *XtAddCallback()*
 - 18.15.9. *XULabelSetString()*
 - 18.15.10. *XtManageChild()*

- 18.16. *CreateDockShell()* [*sm_docks.cpp*]**
 - 18.16.1. *XtCreatePopupShell()*
 - 18.16.2. *XUAddCloseProtocol()*
 - 18.16.3. *XUTurnOffResize()*
 - 18.16.4. *XmCreateForm()*
 - 18.16.5. *XUCreateThumbWheel()*
 - 18.16.6. *XUGetThumbWheelWidgets()*
 - 18.16.7. *XUWidgetInForm()*
 - 18.16.8. *XtVaSetValues()*
 - 18.16.9. *XmCreateLabel()*
 - 18.16.10. *XtManageChild()*

- 18.16.11. XmCreateText()
- 18.16.12. XtAddCallback()
- 18.16.13. XmCreateFrame()
- 18.16.14. XCreatePixmapFromBitmapData()
- 18.16.15. XCreatePixmapFromBitmapData()
- 18.16.16. XtVaCreateManagedWidget()
- 18.16.17. XmCreatePushButton()
- 18.16.18. XmCreateRadioBox()
- 18.16.19. XmCreateToggleButton()
- 18.16.20. XUWidgetInForm()
- 18.16.21. XmCreateRowColumn()

18.17. CreatePathShell() [sm_xpath.cpp]

- 18.17.1. XtCreatePopupShell()
- 18.17.2. XUAddCloseProtocol()
- 18.17.3. XmCreateForm()
- 18.17.4. CreateBottomArea()
- 18.17.5. XmCreateRowColumn()
- 18.17.6. XtVaSetValues()
- 18.17.7. XtMalloc()
- 18.17.8. XtNumber()
- 18.17.9. XmCreateFrame()
- 18.17.10. XmCreateLabel()
- 18.17.11. XtManageChild()
- 18.17.12. XmCreateRadioBox()
- 18.17.13. XmCreateToggleButton()
- 18.17.14. XtAddCallback()
- 18.17.15. XmCreateRowColumn()
- 18.17.16. XmCreateText()
- 18.17.17. CreateInspectionArea()

18.18. CreateMissionShell() [sm_mdl.cpp]

- 18.18.1. XtCreatePopupShell()
- 18.18.2. XUAddCloseProtocol()
- 18.18.3. XmCreateForm()
- 18.18.4. XmCreateFrame()
- 18.18.5. XmCreateLabel()
- 18.18.6. XtVaSetValues()
- 18.18.7. XUCreateOptionMenu()
- 18.18.8. XUOptionMenuButtons()
- 18.18.9. XmCreateText()
- 18.18.10. XmCreateRowColumn()
- 18.18.11. XmCreatePushButton()
- 18.18.12. XtAddCallback()
- 18.18.13. XtManageChild()

18.19. CreatePasmShell() [sm_xpa.cpp]

- 18.19.1. XtCreatePopupShell()
- 18.19.2. XUAddCloseProtocol()
- 18.19.3. XmCreateForm()
- 18.19.4. XmCreateFrame()
- 18.19.5. XmCreateLabel()
- 18.19.6. XtVaSetValues()
- 18.19.7. XmCreateRowColumn()
- 18.19.8. XtMalloc()

- 18.19.9. XtNumber()
- 18.19.10. XmCreateToggleButton()
- 18.19.11. XtManageChild()
- 18.19.12. XmCreatePushButton()
- 18.19.13. XtAddCallback()
- 18.19.14. XmCreateScrolledText()

18.20. CreateRDLShell() [am_rdl.cpp]

- 18.20.1. XtCreatePopupShell()
- 18.20.2. XUAddCloseProtocol()
- 18.20.3. XmCreateForm()
- 18.20.4. XmCreateFrame()
- 18.20.5. XmCreateLabel()
- 18.20.6. XtVaSetValues()
- 18.20.7. XtManageChild()
- 18.20.8. XULabelSetString()
- 18.20.9. CreateBarGraph()
- 18.20.10. XmCreateScrolledList()
- 18.20.11. XtAddCallback()
- 18.20.12. XmCreateRowColumn()
- 18.20.13. XtMalloc()
- 18.20.14. XtNumber()
- 18.20.15. XmCreatePushButton()
- 18.20.16. XmCreateToggleButton()

18.21. CreateTourShell() [sm_tour.cpp]

- 18.21.1. XtCreatePopupShell()
- 18.21.2. XUAddCloseProtocol()
- 18.21.3. XmCreateForm()
- 18.21.4. CreateControls()
- 18.21.5. XmCreateFrame()
- 18.21.6. XtManageChild()
- 18.21.7. XUWidgetInForm()
- 18.21.8. XUProgError()
- 18.21.9. // use ChoosePixelFormat()
- 18.21.10. glXChooseVisual()
- 18.21.11. sprintf()
- 18.21.12. XtVaSetValues()
- 18.21.13. XtVaCreateManagedWidget()
- 18.21.14. XtAddCallback()
- 18.21.15. GlXCreateContext()
- 18.21.16. fprintf()
- 18.21.17. exit()
- 18.21.18. AddPopupMenu()
- 18.21.19. ResetTraveller()
- 18.21.20. ResetRealBase()
- 18.21.21. ResetRealTurret()
- 18.21.22. ResetHistoryBase()
- 18.21.23. ResetHistoryTurret()

18.22. CreateHistoryShell() [sm_history.cpp]

- 18.22.1. XtCreatePopupShell()
- 18.22.2. XUAddCloseProtocol()
- 18.22.3. XmCreateForm()
- 18.22.4. XmCreateRowColumn()

- 18.22.5. XUCreateThumbWheel()
- 18.22.6. XtManageChild()
- 18.22.7. XmCreateFrame()
- 18.22.8. XmCreateLabel()
- 18.22.9. XtVaSetValues()
- 18.22.10. XmCreateToggleButton()
- 18.22.11. XtVaSetValues()
- 18.22.12. XtNumber()
- 18.22.13. XmCreatePushButton()
- 18.22.14. XtAddCallback()
- 18.22.15. XUWidgetInForm()
- 18.22.16. XUGetThumbWheelWidgets()

18.23. *InitBrowserState() [sm_db_int.cpp] [db_init.cpp] [adb_init.cpp]*

- 18.23.1. GUBuildHSItoRGBTable()
- 18.23.2. fprintf()
- 18.23.3. exit()
- 18.23.4. LLCreateList()
- 18.23.5. LLNewAppend()
- 18.23.6. strepy()
- 18.23.7. DBInitX()

18.24. *XtManageChild()*

18.25. *XtRealizeWidget()*

18.26. *CreateCursors() [sm_xwin.cpp]*

- 18.26.1. XCreatePixmapCursor()
- 18.26.2. XCreateBitmapFromData()

18.27. *InitMap() [sm_map.cpp]*

- 18.27.1. XtVaGetValues()
- 18.27.2. GLwDrawingAreaMakeCurrent()
- 18.27.3. LLCreateList()
- 18.27.4. LLNewAppend()
- 18.27.5. CreateDisplayLists()
- 18.27.6. glClearColor()
- 18.27.7. LLRetrieve()
- 18.27.8. GUCopyPoint2D()
- 18.27.9. SetAspectRatio()
- 18.27.10. //glBlendFunc()
- 18.27.11. //glHint()
- 18.27.12. //glEnable()

18.28. *GetCurrentTime() [sm_utility.cpp]*

- 18.28.1. time()
- 18.28.2. localtime(0)
- 18.28.3. ClearHelpText()
- 18.28.4. XtAppAddTimeOut()

19. *ChangeRobotMenu() [sm_robots.cpp]*

19.1. *XtUnmanageChild()*

19.2. *XtDestroyWidget()*

19.3. *LLRetrieve()*

19.4. *//XfreeColors()*

19.5. *MakeNullMenu()* [*sm_robots.cpp*]

- 19.5.1. *XmCreatePushButton()*
- 19.5.2. *XtVaSetValues()*
- 19.5.3. *XtAddCallback()*
- 19.5.4. *XmCreatePulldownMenu()*
- 19.5.5. *XmCreateOptionsMenu()*
- 19.5.6. *XtMalloc()*
- 19.5.7. *XmCreateToggleButton()*
- 19.5.8. *XtManageChild()*
- 19.5.9. *XtManageChild()*

19.6. *XmCreatePushButton()*

19.7. *XtVaSetValues()*

19.8. *XtAddCallback()*

19.9. *XmCreatePulldownMenu()*

19.10. *XmCreateOptionsMenu()*

19.11. *InstallHelpText()* [*sm_utility.cpp*]

- 19.11.1. *XtAddEventHandler()*

19.12. *XtMalloc()*

19.13. *XallocColor()*

19.14. *XmCreateToggleButton()*

19.15. *XtManageChild()*

19.16. *ChangeRobotToggle()* [*sm_robots.cpp*]

- 19.16.1. *LLRetrieve()*
- 19.16.2. *XtVaGetValues()*
- 19.16.3. *XCreatePixmapFromBitmapData()*
- 19.16.4. *XtVaSetValues()*
- 19.16.5. *XmDestroyPixmap()*
- 19.16.6. *XCreatePixmapCursor()*
- 19.16.7. *UpdateCommunication()*

20. *SetUnits()* [*sm_utility.cpp*]

20.1. *XtVaSetValues()*

20.2. *ResetLights()* [*sm_3Ddraw.cpp*]

21. *FillItemList()* [*sm_items.cpp*]

21.1. *XmListDeleteAllItems()*

21.2. *sprintf()*

21.3. *sprintf()*

21.4. *PadString()* [*sm_utility.cpp*]

21.5. *PadString()* [*sm_utility.cpp*]

21.6. *sprintf()*

21.7. *XmStringCreateSimple()*

21.8. *XmListAddItem()*

21.9. *XmListSelectPos()*

21.10. *XULabelSetString()* [*xu_utility.cpp*]

- 21.10.1. *XmStringCreateSimple()*
- 21.10.2. *XtVaSetValues()*
- 21.10.3. *free()*

22. *sprintf()*

23. *XULabelSetString()* [*xu_utility.cpp*]

23.1. *XmStringCreateSimple()*

23.2. *XtVaSetValues()*

23.3. *free()*

24. *NewMessage()* [*sm_utility.cpp*]

24.1. *XUProgError()* [*error.c*]

24.1.1. *fprintf()*

24.1.2. *va_start()*

24.1.3. *vsprintf()*

24.1.4. *time()*

24.1.5. *localtime()*

24.1.6. *strftime()*

24.1.7. *fflush()*

24.1.8. *va_end()*

24.1.9. *sprintf()*

24.1.10. *FatalErrorMessage()*

24.1.11. *ErrorMessage()*

24.1.12. *strcmp*

24.1.13. *exit()*

25. *XtAppMainLoop()*

Header File Structure

```
<stdio.h> [\h]
<stdlib.h>
<math.h>
<time.h>
<ctype.h>
<float.h>
<signal.h>
<time.h>
// <times.h>
<sys\types.h>
<sys\stat.h>
// <unistd.h>
<stdarg.h>
<string.h>
<limits.h>

// <X11/Intrinsic.h>
// <X11/Shell.h>
// <X11/StringDefs.h>
// <X11/cursorfont.h>
// <X11/Xutil.h>

// <Xm/Xm.h>
// <Xm/DrawingA.h>
// <Xm/Text.h>
// <Xm/Label.h>
// <Xm/Form.h>
// <Xm/Frame.h>
// <Xm/PushButton.h>
// <Xm/RowColumn.h>
// <Xm/BulletinB.h>
// <Xm/Separator.h>
// <Xm/ToggleB.h>
// <Xm/ToggleBG.h>
// <Xm/ArrowB.h>
// <Xm/Scale.h>
// <Xm/DrawnB.h>
// <Xm/CascadeBG.h>
// <Xm/CascadeB.h>
// <Xm/Protocols.h>
// <Xm/AtomMgr.h>
// <Xm/List.h>
// <Xm/SelectioB.h>

// <X11/GLw/GLwMDrawA.h>
<windows.h>
<GL/gl.h>
// <GL/glx.h>
<GL/glu.h>

// <Sgm/ThumbWheel.h>

"..\\..\\libraries\\include\\ACAD.h"
```

```
"..\libraries\include\XU.h"  
"..\libraries\include\GU.h"  
"..\libraries\include\IGLX.h"  
"..\libraries\include\OGLX.h"  
"..\libraries\include\OGLIv.h"  
"..\libraries\include\OGLAlive.h"  
"..\libraries\include\LL.h"
```

main()

[sm_main.cpp]

Calls:

- umask(0);
- XUInitErrorLog()[Xu_error.c]
- getenv()
- sprintf()
- strcat()
- XUProgError()
- ParseCommandLine()
- gethostname()
- strcmp()
- InitParameters()
- XtVaAppInitialize()
- XtVaSetValues()
- XtDisplay()
- XtScreen()
- DefaultScreen()
- DefaultGC()
- DefaultColormap()
- InitX()
- ChangeRobotMenu()
- SetUnits()
- FillItemList()
- XULastFilename()
- XULabelSetString()
- NewMessage()
- XtAppMainLoop()

Variables (global & static local):

- XmNtitle
- Dxfw
- Ivw

Purpose:

Sets up parameters, generates errors if program not started properly,

Warnings:

XUInitErrorLog ()

[Xu_error.c]

Calls:

- fopen()
- XUSystemError() [error.c]
- fchmod()
- fileno()
- XUProgError() [error.c]

Variables (global & static local):

- stderr
- TopLevel
- logfile

Purpose:

Generates error if logfile cannot be opened with append permissions. Also starts the XUProgError function

Warnings:

XUProgError ()

[Xu_error.c]

Calls:

- fprintf()
- exit()
- va_start()
- time()
- localtime()
- strftime()
- fflush()
- va_end()
- sprintf()
- FatalErrorMessage()
- ErrorMessage()
- strcmp()

Variables (global & static local):

- stderr
- errlog
- StatusTable[]

Purpose:

Determines type of error and generates the correct error message.

Warnings:

ParseCommandLine ()

Calls:

- sprintf()
- strcmp()
- fprintf()
- GiveHelp()
- atoi()

Variables (global & static local):

- info
- SiteFile
- lead
- stderr
- UseTex

Purpose:

Examines command line parameters and generates any help information if needed. Also sets up internal information based on command line parameters.

Warnings:

InitParameters ()

[sm_main.cpp]

Calls:

- ParseWarehouseFile()
- InitNullHouse()

Variables (global & static local):

- SiteFile
- house
- New
- info

Purpose:

Starts the InitNullHouse function. Also sets up all the parameters for the info class.

Warnings:

InitX ()

[sm_xwin.cpp]

Calls:

- GetColors()
- XmCreateForm()
- CreateMenuBar()
- XmCreateForm()
- XtVaSetValues()
- OGLXCreateColorEditor()
- CreateMap()
- CreateToggleIcons()
- CreateLogo()
- XmCreateLabel()
- XtVaSetValues()
- XULabelSetString()
- XtManageChild()
- CreateRobotArea()
- CreateItemListArea()
- CreateDockShell()
- CreatePathShell()
- CreateMissionShell()
- CreatePasmShell()
- CreateRDLShell()
- CreateTourShell()
- CreateHistoryShell()
- InitBrowserState()
- XtManageChild()
- XtRealizeWidget()
- CreateCursors()
- InitMap()
- GetCurrentTime()

Variables (global & static local):

- TopLevel
- RightForm
- XmNrightAttachment
- XmNbottomAttachment
- XmNtopAttachment
- XmNtopWidget
- XmATTACH_FORM
- XmATTACH_WIDGET
- MenuBar
- ColorShell
- HelpLabel

Purpose:

Sets up all window objects (menus, toolbars, etc.). Sets up program parameters (warehouse, robot, paths, etc.)

Warnings:

ChangeRobotMenu ()

[sm_robots.cpp]

Calls:

```
XtUnmanageChild ()
XtDestroyWidget ()
//XFreeColors()
LLRetrieve()
MakeNullMenu()
XmCreatePushButton()
XtVaSetValues()
XtAddCallback()
XmCreatePulldownMenu()
XmCreateOptionMenu()
InstallHelpText()
XtMalloc()
XAllocColor()
XmCreateToggleButton()
XtManageChild()
ChangeRobotToggle()
```

Variables (global & static local):

```
info
RobotBs[ ]
house
RobotOption
RobotMenu
ColorB
XmNleftAttachment
XmATTACH_FORM
XmNy
XmNleftOffset
XmNwidth
XmNheight
XmNactivateCallback
RobotModCB
RobotForm
XmNradioBehavior
display
Cmap
XmNindicatorType
RobotOptCB
XmNbackground
XmNmenuHistory
```

Purpose:

Alters the robot menu. Determine which robot has been selected. Updates all necessary parameters (color, buttons, etc.) for the robot.

Warnings:

XFreeColors is commented!

SetUnits ()

[sm_utility.cpp]

Calls:

- XtVaSetValues()
- ResetLights()

Variables (global & static local):

- house
- DXFUnits[]
- IVUnits[]
- XmNset (I think this is an X-windows parameter)

Purpose:

Determines which measurement units (length) to use depending on the type of file opened. Seems to set everything to feet though.

Warnings:

FillItemList ()

[sm_items. cpp]

Calls:

- XmListDeleteAllItems ()
- LLRetrieve()
- sprintf()
- PadString()
- XmStringCreateSimple()
- XmListAddItem()
- XmListSelectPos()
- XULabelSetString()

Variables (global & static local):

- ItemList
- house
- ItemLab

Purpose:

Determines what the user is trying to place (TARGET, MARKER, DOCK, and PATH) and places it into the list.

Warnings:

No default value for switch statement!!!

***XULastFilename ()**

[xu_utility.cpp]

Calls:

strlen ()

Variables (global & static local):

Purpose:

Truncates a full path to extract only the last filename.

Warnings:

XULabelSetString ()

[xu_utility.cpp]

Calls:

XmStringCreateSimple ()

XtVaSetValues()

free()

Variables (global & static local):

XmNlabelString

Purpose:

Simple convenience function to set the string of a label widget.

Warnings:

NewMessage ()

[sm_utility. cpp]

Calls:

XUProgError ()

Variables (global & static local):

TopLevel

NOTE_ERROR

Purpose:

Generates an error message

Warnings:

XUSystemError ()

[xu_error.cpp]

Calls:

- fprintf ()
- exit()
- va_start()
- vsprintf()
- strcat()
- strerror()
- time()
- localtime()
- strftime()
- fflush()
- va_end()
- sprintf()
- FatalErrorMessage()
- ErrorMessage()

Variables (global & static local):

- stderr
- errno
- errlog
- StatusTable[]
- TopLevelUpFlag

Purpose:

Generates the appropriate system error desired by the context of the call.

Warnings:

FatalErrorMessage ()

[XU_error.cpp]

Calls:

- XmCreateWarningDialog ()
- XtVaSetValues()
- XtUnmanageChild()
- XmMessageBoxGetChild()
- XtAddCallback()
- XmStringCreateLtoR()
- XmStringFree()
- XtManageChild()

Variables (global & static local):

- DialogPixmap
- XmNsymbolPixmap
- XmNalignment
- XmALIGNMENT_BEGINNING

XmNokCallback
ExitCallback
ErrorString
XmFONTLIST_DEFAULT_TAG
XmNmessageString

Purpose:

 Pops up a standard dialog box displaying a message with an OK button to make it go away.

Warnings:

ErrorMessage ()

[xu_error.cpp]

Calls:

 XmCreateWarningDialog ()
 XtVaSetValues()
 XtUnmanageChild()
 XmMessageBoxGetChild()
 XtVaSetValues()
 XmStringCreateLtoR()
 XmStringFree()
 XtManageChild()

Variables (global & static local):

 DialogPixmap
 XmNsymbolPixmap

Purpose:

 Pops up a standard dialog box displaying a message with an OK button to make it go away.

Warnings:

 Only differs from FatalErrorMessage by NOT calling XtAddCallBack and XtManageChild.

GiveHelp ()

[sm_main.cpp]

Calls:

 printf ()
 exit()

Variables (global & static local):

Purpose:

 Generates a listing of all command line parameters and what they correspond to.

Warnings:

ParseWarehouseFile ()

[sm_parser.cpp]

Calls:

- OpenFile ()
- XUmalloc()
- LLCreateList()
- NextToken()
- ParseSite()
- ParseBuilding()
- ParseAcad()
- ParseIv()
- ParseRobots()
- ParseOrigin()
- ParseTargets()
- ParseMarkers()
- ParseDocks()
- ParsePaths()
- CleanUp()
- ReportParseError()
- CloseFile()

Variables (global & static local):

- ware
- CurrentToken

Purpose:

Goes through a file and sets-up sites, building, robot information. Returns NULL if errors occur.

Warnings:

No default on switch statement.

InitNullHouse ()

[sm_house.cpp]

Calls:

- XUmalloc ()
- LLCreateList()

Variables (global & static local):

- house
- SiteFile[]

Purpose:

Sets the house (struct???) to default positions.

Warnings:

GetColors ()

[sm_xwin.cpp]

Calls:

- XWhitePixelOfScreen ()
- XBlackPixelOfScreen()
- XtVaGetValues()
- XtVaSetValues()
- XmCreatePushButton()
- XQueryColor()
- XAllocColor()
- XAllocNamedColor()
- XtDestroyWidget()

Variables (global & static local):

- TopLevel
- wp
- bp
- BackgroundColor
- backp
- display
- Cmap
- frontp
- mp
- tp
- dp
- mdp
- np
- op

Purpose:

Sets the colors of the screen to values allowed.

Warnings:

CreateMenuBar ()

[sm_xwin.cpp]

Calls:

- XmStringCreateSimple ()
- XmVaCreateSimpleMenuBar()
- XtNameToWidget()
- XtVaSetValues()
- XmStringFree()
- CreateFilePulldown()
- CreateEditPulldown()
- CreateViewPulldown()
- CreateRobotsPulldown()
- CreateMapPulldown()
- CreateBuildingPulldown()
- CreateHelpPulldown()
- XtVaSetValues()

XtManageChild()

Variables (global & static local):

MenuBar

Purpose:

Creates a menu with File, Edit, View, Robots, Map, Building, Help pulldowns.

Warnings:

OpenGLXCreateColorEditor ()

[oglx_xcolor.cpp]

Calls:

XmStringCreateSimple ()
SgOglCreateColorChooserDialog()
XtVaSetValues()
XtAddCallback()
XmStringFree()
XtVaGetValues()
XtUnmanageChild()

Variables (global & static local):

Purpose:

Creates a dialog box with OK, Cancel, and a color editor.

Warnings:

CreateMap ()

[sm_xwin.cpp]

Calls:

XmCreateFrame ()
XtManageChild()
XtVaSetValues()
XUProgError()
glXChooseVisual()
XtVaCreateManagedWidget()
XtAddCallback()
XtAddEventHandler()
XtVaSetValues()
glXCreateContext()
fprintf()
exit()

Variables (global & static local):

TopLevel
attribs[]
glxmap
glxcontext

Purpose:

Creates the map in which the site will be created.

Warnings:

CreateToggleIcons ()

[sm_xwin.cpp]

Calls:

- XmCreateFrame ()
- XmCreateLabel()
- XtVaSetValues()
- XmCreateForm()
- XtMalloc()
- XmCreateRadioBox()
- XCreatePixmapFromBitmapData()
- DefaultRootWindow()
- DefaultDepthOfScreen()
- XtVaCreateManagedWidget()
- InstallHelpText()
- XtAddCallback()
- CreateMapIcons()
- XmCreateText()
- XtAddCallback()
- XtManageChild()

Variables (global & static local):

- MapFrame
- ctoggles []
- RobTog
- InputText
- InputLab

Purpose:

Creates Menu Icons and updates them based on selection.

Warnings:

CreateLogo ()

[sm_xwin.cpp]

Calls:

- XmCreateBulletinBoard ()
- LoadGIF()
- XCreatePixmapFromBitmapData()
- DefaultDepthOfScreen()
- XtVaCreateManagedWidget()
- XCreatePixmap()
- RootWindow()
- DefaultScreen()
- DefaultDepthOfScreen()
- XPutImage()
- CreateProcessedLogos()
- //XUSetDialogPixmap()

XtManageChild()

Variables (global & static local):

LogoBB

Purpose:

Loads a GIF image, if error sets logo to default. If load is correct, updates logo with loaded image.

Warnings:

XUSetDialogPixmap is commented.

CreateRobotArea ()

[sm_xwin.cpp]

Calls:

XmCreateFrame ()
XmCreateLabel()
XtVaSetValues()
XmCreateForm()
XmCreateRowColumn()
XtMalloc()
XCreatePixmapFromBitmapData()
DefaultDepthOfScreen()
XtVaCreateManagedWidget()
XtAddCallback()
XtNumber()
InstallHelpText()
CreateRobotOption()
XtManageChild()

Variables (global & static local):

RobotFrame
RobotForm

Purpose:

Allows you to change a robots name, add/delete a new robot.

Warnings:

CreteItemListArea ()

[sm_xwin.cpp]

Calls:

XmCreateFrame ()
XUCreateOptionsMenu()
XtVaSetValues()
InstallHelpText()
XtVaSetValues()
XmCreateForm()
XmCreateLabel()
XmCreateScrolledList()

XtAddCallback()
XULabelSetString()
XtManageChild()

Variables (global & static local):

ItemLab
ItemList

Purpose:

Sets up a list corresponding to the button pressed ("targets", "markers", "docks", "paths")

Warnings:

CreateDockShell ()

[sm_docks.cpp]

Calls:

XtCreatePopupShell ()
XUAddCloseProtocol()
XUTurnOffResize()
XmCreateForm()
XUCreateThumbWheel()
XUGetThumbWheelWidgets()
XUWidgetInForm()
XtVaSetValues()
XmCreateLabel()
XtManageChild()
XmCreateText()
XtAddCallback()
XmCreateFrame()
XCreatePixmapFromBitmapData()
DefaultDepthOfScreen()
XtVaCreateManagedWidget()
XmCreatePushButton()
XmCreateRadioBox()
XUWidgetInForm()
XmCreateRowColumn()
XtParent()

Variables (global & static local):

DockShell
AzWheel
wheel
box
offset
DockCB
number
SGVLabel
AutoTog
ManTog

Purpose:

Creates a shell that is responsible for determining the dock type, and position.

Warnings:

CreatePathShell ()

[sm_xpath.cpp]

Calls:

- XtCreatePopupShell ()
- XUAddCloseProtocol()
- XmCreateForm()
- CreateBottomArea()
- XmCreateRowColumn()
- XtVaSetValues()
- XtMalloc()
- XtNumber()
- XmCreateFrame()
- XmCreateLabel()
- XtManageChild()
- XmCreateRadioBox()
- XtAddCallback()
- XmCreateText()
- CreateInspectionArea()

Variables (global & static local):

- PathShell
- frames[]
- toggles[]

Purpose:

Creates a shell that is responsible for path placement (classification and placement).

Warnings:

CreateMissionShell ()

[sm_md1.cpp]

Calls:

- XtCreatePopupShell()
- XUAddCloseProtocol()
- XmCreateForm()
- XmCreateFrame()
- XmCreateLabel()
- XtVaSetValues()
- XUCreateOptionsMenu()
- XUOptionsMenuButtons()
- XmCreateText()
- XmCreateRowColumn()
- XmCreatePushButton()
- XtAddCallback()
- XtManageChild()

Variables (global & static local):

- MissionShell
- MissionFrame
- robotlab
- month
- MonthWs
- day
- DayWs
- timebox
- meridian
- MeridianWs

Purpose:

Sets up mission parameters based on month, day, time.

Warnings:

CreatePasmShell ()

[sm_xpa.cpp]

Calls:

- XtCreatePopupShell ()
- XUAddCloseProtocol()
- XmCreateForm()
- XmCreateFrame()
- XmCreateLabel()
- XtVaSetValues()
- XmCreateRowColumn()
- XtMalloc()
- XtNumber()
- XmCreateToggleButton()
- XtManageChild()
- XmCreatePushButton()
- XtAddCallback()
- XmCreateScrolledText()

Variables (global & static local):

- PasmShell
- FlagTogs
- ErrorText

Purpose:

Creates buttons used for pasm assembly.

Warnings:

The following is commented from one of the XtVaSetValues function calls

```
/*  
    XmNcolumns,          80,  
*/
```

CreateRDLSHELL ()

[sm_rdl.cpp]

Calls:

- XtCreatePopupShell ()
- XUAddCloseProtocol()
- XmCreateForm()
- XmCreateFrame()
- XmCreateLabel()
- XtVaSetValues()
- XtManageChild()
- XULabelSetString()
- CreateBarGraph()
- XmCreateScrolledList()
- XtAddCallback()
- XmCreateRowColumn()
- XtMalloc()
- XtNumber()
- XmCreatePushButton()
- XmCreateToggleButton()

Variables (global & static local):

- RDLShell
- date
- Robot
- site
- build
- ReportList

Purpose:

COME BACK TO THIS ONE !!! Creates a

Warnings:

Errors if necessary

CreateTourShell ()

[sm_tour.cpp]

Calls:

- XtCreatePopupShell ()
- XUAddCloseProtocol()
- XmCreateForm()
- CreateControls()
- XmCreateFrame()
- XtManageChild()
- XUWidgetInForm()
- XUProgError()
- glXChooseVisual()
- DefaultScreen()
- sprintf()
- XtVaSetValues()
- XtVaCreateManagedWidget()
- XtAddCallback()
- glXCreateContext()

fprintf()
exit()
ResetTraveller()
ResetRealBase()
ResetRealTurret()
ResetHistoryBase()
ResetHistoryTurret()

Variables (global & static local):

TourShell
attribs[]
glxtour
TourContext

Purpose:

Sets up the 3D Tour app. Resets 3D Robot values and traveller's orientation.

Warnings:

CreateHistoryShell ()

[sm_history.cpp]

Calls:

XtCreatePopupShell()
XUAddCloseProtocol()
XmCreateForm()
XmCreateRowColumn()
XmCreateToggleButton
XUCreateThumbWheel()
XtManageChild()
XmCreateFrame()
XmCreateLabel()
XtVaSetValues()
XUWidgetInForm()
XUGetThumbWheelWidgets()

Variables (global & static local):

HistoryShell
SpeedWheel
IndexWheel
MapTog
TourTog

Purpose:

Initializes the history dialog and sets up other history parameters.

Warnings:

InitBrowserState ()

[sm_db_init.cpp]

Calls:

```
GUBuildHSItoRGBTable ()  
fprintf()  
exit()  
LLCreateList()  
LLNewAppend()  
#ifdef SITE_MANAGER DBInitX( )
```

Variables (global & static local):

```
BrowserState
```

Purpose:

Initializes the BrowserState parameters and calls functions for color values, linklists, etc.

Warnings:

DBInitX() only works if SITE_MANAGER is defined as true.

CreateCursors ()

[sm_xwin.cpp]

Calls:

```
XCreatePixmapCursor()  
XCreateBitmapFromData()
```

Variables (global & static local):

```
cselect  
ctarget  
cmarker  
cdock  
corigin  
crobot  
cchoose
```

Purpose:

Sets the color and other attributes of the cursors used throughout the program.

Warnings:

InitMap ()

[sm_map.cpp]

Calls:

```
XtVaGetValues ()  
GLwDrawingAreaMakeCurrent()  
LLCreateList()  
LLNewAppend()  
CreateDisplayLists()  
LLRetrieve()  
GUCopyPoint2D ()  
SetAspectRatio()
```

```
//glBlendFunc()
//glHint()
//glEnable();
```

Variables (global & static local):

```
MapWidth
MapHeight
ViewList
view
info
border
```

Purpose:

Sets up map parameters, and creates display list on first run.

Warnings:

GetCurrentTime ()

[sm_utility.cpp]

Calls:

```
time()
localtime()
ClearHelpText()
XtAppAddTimeOut()
```

Variables (global & static local):

```
MyTime
```

Purpose:

Gets the current time from the system clock.

Warnings:

Uses GetCurrentTime as a parameter for the XtAppAddTimeOut() function. Recursive function with no stop??

***LLRetrieve ()**

[ll_lists.cpp]

Calls:

```
fprintf()
LLQuickReadTraverse()
```

Variables (global & static local):

Purpose:

Returns the data specified.

Warnings:

Even though function is listed as void returns node->data.

MakeNullMenu ()

[File.cpp]

Calls:

- XmCreatePushButton ()
- XtVaSetValues()
- XtAddCallback()
- XmCreatePulldownMenu()
- XmCreateOptionsMenu()
- XmCreateToggleButton()
- XtMalloc()
- XtManageChild()

Variables (global & static local):

- ColorB
- RobotMenu
- RobotOption
- RobotBs[]
- info

Purpose:

Sets up the robot menu so there is no entries.

Warnings:

InstallHelpText ()

[sm_utility.cpp]

Calls:

- XtAddEventHandler()

Variables (global & static local):

Purpose:

Sets up handlers for the help text dialogs.

Warnings:

ChangeRobotToggle ()

[sm_robots.cpp]

Calls:

- LLRetrieve ()
- XtVaGetValues()
- XCreatePixmapFromBitmapData()
- XtVaSetValues()
- XmDestroyPixmap()
- XCreatePixmapCursor()
- XCreateBitmapFromData()
- UpdateCommunication()

Variables (global & static local):

- crobot

info

Purpose:

Sets the robot button's toggle parameter based on connection to server.

Warnings:

The ConnectedToServer if section sets the info.display parameter to true and then false. I'm not sure how this updates the toggle button.

ResetLights ()

[sm_3Ddraw.cpp]

Calls:

Variables (global & static local):

light_position

Purpose:

Sets the OpenGL lights positions to the X and Y coordinates of the origin. Z value is set to 6.

Warnings:

There is a section that is commented out that moves the lights according to the scale used.

PadString ()

[sm_utility.cpp]

Calls:

Variables (global & static local):

Purpose:

This function adds empty spaces " " to a string of characters. The length of the spaces depends on the second parameter passed to the function.

Warnings:

Although it is an integer, found is set to false. It is probably defined as an integer somewhere else though.

OpenFile ()

[sm_scanner.cpp]

Calls:

PushFileContext()
LLCreateList()
PushFileContext()
strcpy()
fopen()
fprintf()
PopFileContext()
XUmalloc()
NextLine()

Variables (global & static local):

- FileList
- CurrentFilename
- line
- TokenBuffer
- LineNum

Purpose:

Opens the file for reading, initializes the line counter to 1, and reads in the first buffer full.

Warnings:

Errors if necessary

***XUmalloc ()**

[xu_utility.cpp]

Calls:

- malloc()
- fprintf()
- exit()

Variables (global & static local):

Purpose:

Gets memory from the OS for data.

Warnings:

LLCreateList ()

[ll_lists.cpp]

Calls:

- malloc()
- fprintf()

Variables (global & static local):

Purpose:

Creates an empty Linked List.

Warnings:

NextToken ()

[sm_scanner.cpp]

Calls:

- Eof()
- getch()
- inspect()
- ProcessComment()
- ProcessFilename()
- isspace()
- isalpha()
- isdigit()
- ProcessLiteral()

Variables (global & static local):

- TokenBuffer[]
- LineNum
- TabOffset

Purpose:

Parses a string and returns the next token.

Warnings:

ParseSite ()

[sm_parser.cpp] and [sm_rdlparser.cpp]

Calls:

- Match()
- XUstrdup()

Variables (global & static local):

- SiteSym
- AssignOp
- ware
- TokenBuffer
- StringSym
- SemiColon

Purpose:

COME BACK TO THIS!!!!

Warnings:

There are two listings for this function in the site manager directory. Which one will get picked by the compiler to perform the required operation?

ParseBuilding ()

[sm_parser.cpp] and [sm_rdlparser.cpp]

Calls:

Match()
XUstrdup()

Variables (global & static local):

BuildingSym
AssignOp
ware
TokenBuffer
StringSym
SemiColon

Purpose:

COME BACK TO THIS!!!!

Warnings:

There are two listings for this function in the site manager directory. Which one will get picked by the compiler to perform the required operation?

ParseAcad ()

[sm_parser.cpp]

Calls:

Match()
ACPParseDXFFile()
fprintf()
XUstrdup()
atoi()
FindHouseLimits()

Variables (global & static local):

AcadSym
AssignOp
ware
TokenBuffer
stderr
StringSym
LParen
Literal
RParen
SemiColon

Purpose:

Parses AutoCAD files, so one can draw a site in that format and Site Manager will understand.

Warnings:

flags |= ACP_EXPLODE_BLOCKS is commented out.

ParseIv ()

[sm_parser.cpp]

Calls:

- Match ()
- XUstrdup()
- atoi()

Variables (global & static local):

- IvSym
- AssignOp
- ware
- TokenBuffer
- StringSym
- LParen
- Literal
- RParen
- SemiColon

Purpose:

Parses OpenInventor files and converts them into a format that Site Manager will understand.

Warnings:

Errors if necessary

ParseRobots ()

[sm_parser.cpp]

Calls:

- Match()
- LLNewAppend()
- XUstrdup()
- atoi()
- LLRetrieve()
- ReportParseError()

Variables (global & static local):

- RobotsSym
- LCurly
- CurrentToken
- RCurly
- StringSym
- AssignOp
- Literal
- LParen
- RParen
- SemiColon
- RCurly

Purpose:

COME BACK TO THIS!

Warnings:

ParseOrigin ()

[sm_parser.cpp]

Calls:

- Match()
- atof()

Variables (global & static local):

- OriginSym
- AssignOp
- ware
- TokenBuffer
- Literal
- SemiColon

Purpose:

COME BACK TO THIS!!

Warnings:

ParseTargets ()

[sm_parser.cpp]

Calls:

- Match()
- LLNewAppend()
- atoi()
- atof()

Variables (global & static local):

- TargetsSym
- LCurly
- CurrentToken
- RCurly
- ware
- Literal
- LParen
- RParen
- SemiColon

Purpose:

Warnings:

ParseMarkers()

[sm_parser.cpp]

Calls:

Match()
LLNewAppend()
atoi()
atof()

Variables (global & static local):

MarkersSym
LCurly
CurrentToken
RCurly
ware
TokenBuffer
Literal
LParen
RParen
SemiColon

Purpose:

Warnings:

ParseDocks()

[sm_parser.cpp]

Calls:

Match()
LLNewAppend()
atoi()
atof()
LLRetrieve()
ReportParseError()

Variables (global & static local):

DocksSym
LCurly
CurrentToken
RCurly
ware
TokenBuffer
Literal
LParen
RParen

Purpose:

What the function does.

Warnings:

Errors if necessary

ParsePaths()

[sm_parser.cpp]

Calls:

- Match()
- LLMalloc()
- LLCreateList()
- atoi()
- XUstrdup()
- FindMarker()
- ReportParseError()
- FindRobot()
- ParseSGV()
- LLAppend()

Variables (global & static local):

- PathsSym
- LCurly
- CurrentToken
- RCurly
- ware
- PathSym
- AisleSym
- AssignOp
- TokenBuffer
- Literal
- StringSym
- LParen
- RParen
- RCurly

Purpose:

Warnings:

ParseSGV ()

[sm_parser.cpp]

Calls:

- Match()
- ReportParseError()
- atoi()
- FindTarget()
- LLAppend()

Variables (global & static local):

- LCurly
- CurrentToken
- VisualSym
- AssignOp
- AutoSym
- HumanSym
- MoveSym
- InspectSym
- VisSym
- OtherSym
- TargetsSym
- TokenBuffer

SemiColon
RCurly

Purpose:

Warnings:

CleanUp()

[sm_parser.cpp] & [sm_rdlparser.cpp] & [alv_actions.cpp] & [alv_creature.cpp]

Calls:

In the sm_rdlparser.cpp file
CloseFile()
or in alv_* functions
ALVCloseFile()

Variables (global & static local):

Purpose:

Closes the file associated with the current parsing.

Warnings:

I am not sure if this function has been finished. There is still a commented section and the sm_parser.cpp function has no code in it whatsoever.

ReportParseError ()

[sm_scanner.cpp] & [mon_load.cpp]

Calls:

va_start()
vsprintf()
va_end()
fprintf()

Variables (global & static local):

stderr
CurrentFilename
LineNum
line

Purpose:

Reports an error in the parsing engine if one occurs.

Warnings:

CloseFile ()

[sm_scanner.cpp] & [mon_scanner.cpp]

Calls:

fclose()
free()

PopFileContext()

Variables (global & static local):

CurrentFile
line
TokenBuffer

Purpose:

Closes the file and frees the memory used by it.

Warnings:

The mon_scanner.cpp function returns an integer (0) whereas the sm_scanner returns void. There is also a difference in the order of the functions.

XUmalloc()

[xu_utility.cpp]

Calls:

malloc()
fprintf()
exit()

Variables (global & static local):

stderr

Purpose:

Allocates necessary memory resources. Reports error if necessary.

Warnings:

CreateFilePulldown()

[sm_db_xwin.cpp] & [sm_xwin.cpp]

Calls:

XmStringCreateSimple()
XmVaCreateSimplePulldownMenu()
XtNameToWidget()
XmStringFree()

Variables (global & static local):

Purpose:

Creates necessary buttons and events (keyboard accelerators, etc.) for the file pulldown menu.

Warnings:

Two occurrences of this function reside in the site manager directory. Overloads the new operator.

CreateEditPulldown ()

[sm_xwin.cpp]

Calls:

- XmStringCreateSimple ()
- XmVaCreateSimplePulldownMenu()
- XmStringFree()
- XtParent()

Variables (global & static local):

Purpose:

Creates all the necessary buttons and events (keyboard accelerators, etc.) for the edit pulldown menu.

Warnings:

Overloads the delete operator

CreateViewPulldown ()

[sm_xwin.cpp]

Calls:

- XmStringCreateSimple()
- XmVaCreateSimplePulldownMenu()
- XtVaSetValues()
- XtNameToWidget()
- XmStringFree()
- XtParent()

Variables (global & static local):

Purpose:

Creates all the necessary buttons and events (keyboard accelerators, etc.) for the view pulldown menu.

Warnings:

CreateRobotsPulldown()

[sm_xwin.cpp]

Calls:

- XmStringCreateSimple()
- XmVaCreateSimplePulldownMenu()
- XtNameToWidget()
- XmStringFree()
- XtParent()

Variables (global & static local):

- DisplayTog
- DiagTog

Purpose:

Creates all the necessary buttons and events (keyboard accelerators, etc.) for the robot pulldown menu.

Warnings:

Overloads the delete operator

CreateMapPulldown()

[sm_xwin.cpp]

Calls:

- XtMalloc()
- XmStringCreateSimple()
- XmVaCreateSimplePulldownMenu()
- XtNameToWidget()
- XtVaSetValues()
- XmStringFree()
- XtParent()

Variables (global & static local):

- Dxfw
- Ivw
- DXFUnits[]
- IVUnits[]

Purpose:

Creates all the necessary buttons and events (keyboard accelerators, etc.) for the map pulldown menu.

Warnings:

CreateBuildingPulldown()

[sm_xwin.cpp]

Calls:

- XmStringCreateSimple()
- XmVaCreateSimplePulldownMenu()
- XmStringFree()
- XtParent()

Variables (global & static local):

Purpose:

Creates all the necessary buttons and events (keyboard accelerators, etc.) for the map pulldown menu.

Warnings:

CreateHelpPulldown ()

[sm_xwin.cpp]

Calls:

- XmStringCreateSimple()
- XmVaCreateSimplePulldownMenu()
- XmStringFree()
- XtParent()

Variables (global & static local):

Purpose:

Creates button and necessary events for the help pulldown menu.

Warnings:

CreateMapIcons ()

[File.cpp]

Calls:

- XmCreateRowColumn()
- XtVaSetValues()
- XtMalloc()
- XCreatePixmapFromBitmapData()
- XtVaCreateManagedWidget()
- InstallHelpText()
- XtAddCallback()
- XtManageChild()

Variables (global & static local):

Purpose:

Creates the Map toolbar.

Warnings:

LoadGIF ()

[sm_xgifload.cpp]

Calls:

- SetupColorAndDisplayInfo()
- fopen()
- XUSystemError()
- fseek()
- ftell()
- malloc()
- fread()
- strncmp()
- XUProgError()
- fprintf()
- free()
- XCreateImage()
- ReadCode()
- AddToPixel()
- ColorDicking()

Variables (global & static local):

- fp
- RawGIF
- Raster
- id
- RWidth
- RHeight

HasColormap
BitsPerPixel
numcols
BitMask
Background
NEXTBYTE
Red[]
Green[]
Blue[]
used[]
numused
cols[]
LeftOfs
TopOfs
Width
Height
CodeSize
ClearCode
EOFCODE
FreeCode
InitCodeSize
MaxCode
ReadMask
Image
theImage
BytesPerScanline
CurCode
OldCode
FinChar
InCode
OutCode[]
OutCount
fname
FinChar
Prefix[]
Suffix[]

Purpose:

Loads a GIF formatted picture into memory and performs the necessary conversions in order to render the image on screen.

Warnings:

CreateProcessedLogos ()

[sm_xwin.cpp]

Calls:

XUmalloc()
XCreateImage()
XQueryColor()
XGetPixel()
XPutPixel()
XCreatePixmap()
XPutImage()

XPutImage()

Variables (global & static local):

display
theVisual
ZPixmap
Cmap
theImage
GrayPixmap
BlackPixmap

Purpose:

Changes the color values of the screen and places images on the screen.

Warnings:

CreateRobotOption ()

[sm_xwin.cpp]

Calls:

XmCreatePushButton()
XtVaSetValues()
XtAddCallback()
XmCreatePulldownMenu()
XmCreateOptionsMenu()
InstallHelpText()
XtManageChild()

Variables (global & static local):

ColorB
RobotMenu
RobotOption
RobotBs[]

Purpose:

Creates a button and menu for the robots.

Warnings:

CreateBottomArea ()

[sm_xpath.cpp]

Calls:

XmCreateRowColumn()
XCreatePixmapFromBitmapData()
XmCreatePushButton()
XtVaSetValues()
XtAddCallback()
XmCreateToggleButton()
XmCreateLabel()

XtManageChild()

Variables (global & static local):

ChooseTog

Purpose:

Creates all the components that are displayed at the bottom of the screen.

Warnings:

CreateInspectionArea()

[sm_xpath.cpp]

Calls:

XtMalloc()
XmCreateFrame()
XmCreateLabel()
XtVaSetValues()
XtManageChild()
XmCreateRowColumn()
XmCreateToggleButton()
XtAddCallback()
CreateSubForm()

Variables (global & static local):

FileLabels
frames[]
VisButton
OButton

Purpose:

Creates the Inspection, Visual, and Other buttons used for guiding the robot.

Warnings:

XUAddCloseProtocol ()

[xu_utility.cpp]

Calls:

XmInternAtom()
XtDisplay()
XmAddWMProtocols()
XmAddWMProtocolCallback()
XUProgError()

Variables (global & static local):

TopLevel

Purpose:

Given a shell widget, this function will communicate to the window manager what action should be taken when the window manager "close" option is selected.

Warnings:

XUCreateOptionsMenu ()

[xu_dialog.cpp]

Calls:

- XUmalloc()
- sizeof()
- GetButtonCount()
- XtMalloc()
- sprintf()
- XmCreatePulldownMenu()
- XmCreateOptionsMenu()
- XmStringCreateSimple()
- XtVaSetValues()
- XmCreatePushButton()
- XtAddCallback()
- XtManageChild()
- XmStringFree()

Variables (global & static local):

Purpose:

Creates the Menu and buttons for the option commands.

Warnings:

XUOptionsMenuButtons()

[xu_dialog.cpp]

Calls:

- XtVaGetValues()

Variables (global & static local):

Purpose:

Gets the list of the buttons in the option menu.

Warnings:

CreateBarGraph ()

[sm_rdl.cpp]

Calls:

- XmCreateLabel()
- XmCreateFrame()
- XmCreateDrawingArea()
- XtAddCallback()
- XtManageChild()

Variables (global & static local):

percent
graph

Purpose:

Creates a window with a graph in the drawing area.

Warnings:

CreateControls ()

[sm_tour.cpp]

Calls:

XmCreateForm()
XUWidgetInForm()
XmCreateRowColumn()
XUCreateThumbWheel()
XUGetThumbWheelWidgets()
XtVaSetValues()
ResetProximityWheel()
XmCreateForm()
XmCreateLabel()
XmCreateToggleButton()
XtAddCallback()
XtManageChild()
XUWidgetInForm()
XmCreateRadioBox()

Variables (global & static local):

ControlForm
detail
info
dw
proximity

Purpose:

This function creates a

Warnings:

This statement is commented out of one of the XtVaSetValues() calls:
/* SgNangleRange, MAX_DETAIL - MIN_DETAIL, */

XUWidgetInForm ()

[xu_utility.cpp]

Calls:

va_arg()
XtVaSetValues()
fprintf()
va_end()

Variables (global & static local):

Purpose:

Attaches values to the widget passed to the function.

Warnings:

ResetTraveller ()

[sm_tour.cpp]

Calls:

Variables (global & static local):

traveller
house

Purpose:

Sets the viewpoint's origin to the middle of the room.

Warnings:

ResetRealBase ()

[sm_tour.cpp]

Calls:

Variables (global & static local):

RealBase

Purpose:

Sets up initial parameters for the Robot's real base.

Warnings:

ResetRealTurret ()

[sm_tour.cpp]

Calls:

Variables (global & static local):

RealTurret

Purpose:

Sets up the initial parameters for the robot's turret.

Warnings:

ResetHistoryBase ()

[sm_tour.cpp]

Calls:

Variables (global & static local):
HistoryBase

Purpose:
Sets up the initial parameters for the history base.

Warnings:

ResetHistoryTurret ()

[sm_tour.cpp]

Calls:

Variables (global & static local):
HistoryTurret

Purpose:
Sets the history turret to its initial parameters.

Warnings:

XUCreateThumbWheel ()

[xu_wheel.cpp]

Calls:

- XUmalloc()
- sizeof()
- XmCreateForm()
- XtVaSetValues()
- XmCreateText()
- sprintf()
- XtAddCallback()
- XtVaCreateManagedWidget()
- XmCreateLabel()
- XtManageChild()

Variables (global & static local):
ThumbWheel

Purpose:
Creates a new widget with parameters specified.

Warnings:
The sprintf was originally listed as sprintf(buf, "%.2f", val), but then changed to sprintf(buf, "%g", val).

XUGetThumbWheelWidgets ()

[xu_wheel.cpp]

Calls:

- XtVaGetValues()

Variables (global & static local):
twheel

Purpose:
Sets the passed variables to the values of the thumbwheel.

Warnings:
*twheel is declared, but twheel is used in the function.

GUBuildHSItoRGBTable ()

[gu_twod.cpp]

Calls:
fprintf()
AllocateColorTable()
powf()
GUHSItoRGB()

Variables (global & static local):
stderr
GUTableResolution
GULookupShift
HSItoRGBTable[][]

Purpose:
Sets up the HSItoRGBTable to the colors that can be displayed on the system. Similar to a color cube.

Warnings:

***LLNewAppend()**

[ll_lists.cpp]

Calls:
LLMalloc()
LLAppend()

Variables (global & static local):

Purpose:
Appends a new item to the beginning of the list.

Warnings:

CreateDisplayLists ()

[sm_draw.cpp]

Calls:
gluNewQuadric()
glGenLists()
glNewList()

glLineWidth()
glBegin()
glVertex2f()
glEnd()
glEndList()
DrawModel()
DrawTarget()
DrawDock()
gluDisk()
DrawArrows()
Create2DdrumDLs()
sprintf()
ACPParseDXFFile()
fprintf()

Variables (global & static local):

quadobj
lorigin
LENGTH
lmodel
house
ltarget
ldock
lcircle
DLOutline
DLLidar
larrrows
flags
RobotModel
AriesRootDir

Purpose:

Creates the OpenGL display lists for all objects. This saves memory and speeds performance.

Warnings:

GUCopyPoint2D ()

[GU.h]

Calls:

memcpy()
sizeof()

Variables (global & static local):

Purpose:

Copies the memory location of “a” and places it in “b.”

Warnings:

Function is defined in GU.h.

SetAspectRatio ()

[sm_map.cpp]

Calls:

- LLRetrieve()
- GUCopyPoint2D()

Variables (global & static local):

- View
- ViewList
- info
- house
- MapHeight
- MapWidth

Purpose:

Corrects the perspective of the two-dimensional application.

Warnings:

ClearHelpText ()

[sm_utility.cpp]

Calls:

- sprintf()
- strcat()
- XULabelSetString()

Variables (global & static local):

- MyTime
- MyClear
- info

Purpose:

Resets the help dialog box to initial values.

Warnings:

LLQuickReadTraverse ()

[ll_lists.cpp]

Calls:

Variables (global & static local):

Purpose:

Moves through the list and sets the LastReadPtr to the found node. Also returns the same node.

Warnings:

UpdateCommunication ()

[sm_comm.cpp]

Calls:

- XtVaSetValues()
- XUProgError()
- ClearHelpText()
- LLDestroyList()
- LLCreateList()
- LLRetrieve()
- MakeConnection()
- XtRemoveWorkProc()
- EndComm()

Variables (global & static local):

- house
- info
- DisplayTog
- DiagTog
- TopLevel
- FitList
- FIDList
- FitDiagType
- FIDDiagType
- DisplayWorkProcId
- ConnectedToServer

Purpose:

Checks the communication between the robot and server. If no robots are present, displays an error message. Updates the FitList and the FIDList and processes work orders to the robots.

Warnings:

PushFileContext ()

[sm_scanner.cpp]

Calls:

- LLMalloc()
- strcpy()
- LLAppend()

Variables (global & static local):

- CurrentFile
- line
- TokenBuffer

ChPos
LineNum
ch
CurrentToken
CurrentFilename
FileList

Purpose:

Updates the file context with the current parameters being used.

Warnings:

PopFileContext ()

[sm_scanner.cpp]

Calls:

LLRetrieve()
strcpy()
LLDelete()

Variables (global & static local):

FileList
CurrentFilename
CurrentFile
line
TokenBuffer
ChPos
LineNum
ch
CurrentToken

Purpose:

Passes the current parameters the current values of the FileList. This function does the opposite of PushFileContext().

Warnings:

NextLine ()

[sm_scanner.cpp]

Calls:

fgetc()
fprintf()

Variables (global & static local):

TabOffset
CurrentFile
line[]
ChPos
ch

Purpose:

This function reads in the next line from the input file.

Warnings:

Eof ()

[sm_scanner.cpp]

Calls:

feof()

Variables (global & static local):

CurrentFile

Purpose:

Returns true or false based on the result of the feof function on the current file.

Warnings:

getch ()

[sm_scanner.cpp]

Calls:

NextLine()

Variables (global & static local):

ch
LineNum
line[]

Purpose:

Gets the current character and updates the line counter if the current character is an end of line command.

Warnings:

inspect ()

[sm_scanner.cpp]

Calls:

Variables (global & static local):

ch

Purpose:

Returns the value of the ch integer.

Warnings:

ProcessComment ()

[sm_scanner.cpp]

Calls:

advance()
getch()

Variables (global & static local):

Purpose:

Advances through and ignores the comment. Accepts both the /* and the // variations.

Warnings:

ProcessFilename ()

[sm_scanner.cpp]

Calls:

getch()

Variables (global & static local):

TokenBuffer[]
StringSym

Purpose:

Checks the validity of the TokenBuffer as a unix filename.

Warnings:

ProcessLiteral ()

[sm_scanner.cpp]

Calls:

getch()
isdigit()
advance()
inspect()

Variables (global & static local):

TokenBuffer[]
MinusEqualsOp
Literal

Purpose:

Checks the ALVTokenBuffer for its validity as a literal.

Warnings:

***XUstrdup ()**

[xu_utility.cpp]

Calls:

XUmalloc()
strlen()
sizeof()
strcpy()

Variables (global & static local):
source

Purpose:
Given a string, this function will malloc a new copy and return a pointer to the new memory. (exact functionality of strdup, but POSIX compliant).

Warnings:

* **ACPParseDXFFile ()**

[dxf_flow.cpp]

Calls:

- InitVars()
- setjmp()
- ACPOpenFile()
- ACPNextToken()
- CheckString()
- MatchString()
- ACPMatch()
- ParseHeaderSection()
- ParseTablesSection()
- ParseBlocksSection()
- ParseEntitiesSection()
- ACPError()
- ACPCloseFile()
- HandleOptions()
- BuildModelArray()

Variables (global & static local):
ExceptionBuf
ACPCurrentToken
SemanticRec

Purpose:
Converts a DXF format into an ACPModel.

Warnings:

FindHouseLimits()

[sm_utility.cpp]

Calls:

Variables (global & static local):

Purpose:
Determines the extents of the warehouse

Warnings:
There is still a commented section in this function.

ReportParseError ()

[sm_scanner.cpp]

Calls:

- va_start()
- vsprintf()
- va_end()
- fprintf()

Variables (global & static local):

- stderr
- CurrentFilename
- LineNum
- line

Purpose:

Prints out an error if one occurs in the parsing process.

Warnings:

LLMalloc ()

[ll_lists.cpp]

Calls:

- malloc()
- fprintf()

Variables (global & static local):

- stderr

Purpose:

Allocates the memory resources for the linked list. Reports error if one occurs.

Warnings:

* FindMarker ()

[sm_parser.cpp]

Calls:

- LLRetrieve()

Variables (global & static local):

- ware

Purpose:

Retrieves the marker from the linked list and checks to see if it is the one specified.

Warnings:

***FindRobot ()**

[sm_parser.cpp]

Calls:

LLRetrieve ()
strcmp()

Variables (global & static local):

ware

Purpose:

Gets the robot from the linked list and checks to make sure that was the one asked for.

Warnings:

LLAppend ()

[ll_lists.cpp]

Calls:

LLCreateNode()

Variables (global & static local):

Purpose:

Places a node at the end of the list with the data specified.

Warnings:

***FindTarget ()**

[sm_parser.cpp]

Calls:

LLRetrieve()

Variables (global & static local):

ware

Purpose:

Matches the target specified with the one in the liked list.

Warnings:

ALVCloseFile ()

[alv_scanner.cpp]

Calls:

fclose()
free()
PopFileContext()

Variables (global & static local):

CurrentFile
line
ALVTokenBuffer

Purpose:

Closes the file that is currently being used by the program. Also clears the memory contents.

Warnings:

SetupColorAndDisplayInfo ()

[sm_xgifload.cpp]

Calls:

DefaultScreen ()
RootWindow()
DefaultGC()
DefaultVisual()
DefaultColormap()

Variables (global & static local):

theScreen
display
rootW
theGC
theCmap

Purpose:

Gives the screen and all other color components the default values.

Warnings:

ReadCode ()

[sm_xgifload.cpp]

Calls:

Variables (global & static local):

Raster[]
CodeSize
ReadMask

Purpose:

Fetch the next code from the raster data stream. The codes can be any length from 3 to 12 bits, packed into 8-bit bytes, so we have to maintain our location in the Raster array as a BIT Offset. We compute the byte Offset into the raster array by dividing this by 8, pick up three bytes, compute the bit Offset into our 24-bit chunk, shift to bring the desired code to the bottom, then mask it off and return it.

Warnings:

AddToPixel ()

[sm_xgifload.cpp]

Calls:

Variables (global & static local):

- YC
- Height
- Image
- BytesPerScanline
- XC
- used[]
- numused
- Width
- Interlace
- Pass

Purpose:

If a non-interlaced picture, increments YC to the next scan line. If it's interlaced, deals with the interlace as described in the GIF spec. Puts the decoded scan line out to the screen if it hasn't gone past the bottom of it

Warnings:

ColorDicking ()

[sm_xgifload.cpp]

Calls:

- XAllocColor()
- fprintf()
- XQueryColors()
- abs()
- XUProgError()
- CopyMemory()
- XFreeColors()

Variables (global & static local):

- HasColormap
- nostrip
- numcols
- used[]
- defs[]
- Red[]
- Green[]
- Blue[]

display
theCmap
dispcells
stderr
numused
ptr

Purpose:

Converts to the colors in the image file. If no file, sets the color to default.

Warnings:

CreateSubForm ()

[sm_xpath.cpp]

Calls:

XmCreateForm ()
XmCreateLabel()
XtVaSetValues()
XmCreateRadioBox()
XtManageChild()
XmCreateToggleButton()
XmCreateRowColumn()
XmCreatePushButton()
XtAddCallback()

Variables (global & static local):

FileLabels[]
toggles[]

Purpose:

Creates all the necessary buttons, columns for the form area of the screen.

Warnings:

GetButtonCount ()

[xu_dialog.cpp]

Calls:

fprintf ()

Variables (global & static local):

Purpose:

Counts the number of buttons on the menu. If count exceeds 50, generates an error message.

Warnings:

ResetProximityWheel ()

[sm_tour.cpp]

Calls:

XUSetThumbWheel()

XtVaSetValues ()

Variables (global & static local):

house
info
pw

Purpose:

Sets the maximum and home positions of the proximity wheel to either 1 and 0 (respectively) if there is no model defined in the house structure. Else sets the maximum to the value of the house's extents and the home to half.

Warnings:

AllocateColorTable ()

[gu_twod.cpp]

Calls:

powf()
malloc()

Variables (global & static local):

GUTableResolution
HSItoRGBTable[][]

Purpose:

Allocates all the memory resources for the HSItoRGBTable.

Warnings:

GUHSItoRGB ()

[gu_twod.cpp]

Calls:

sinf()
cosf()
max()
min()

Variables (global & static local):

Purpose:

Converts H to R, S to G, and I to B.

Warnings:

DrawModel ()

[sm_draw.cpp]

Calls:

glColor3f()

```
glBegin()  
glVertex2fv()  
glEnd()
```

Variables (global & static local):
house

Purpose:
Draws the ACPModel that is passed to the function.

Warnings:

DrawTarget ()

[sm_draw.cpp]

Calls:
gluDisk()
glBegin()
glVertex2f()
glEnd()

Variables (global & static local):
D_LENGTH
LENGTH

Purpose:
Draws a target using OpenGL commands. Looks like it is a square.

Warnings:
glLineWidth(2.0) was commented out of the function.

DrawDock ()

[sm_draw.cpp]

Calls:
gluDisk()
glBegin()
glVertex2f()
glEnd()

Variables (global & static local):
quadobj
DOCK_LENGTH

Purpose:
Draws the AutoCharger using OpenGL commands.

Warnings:

DrawArrows ()

[sm_draw.cpp]

Calls:

```
glColor3f()
glBegin()
glVertex2f()
glEnd()
```

Variables (global & static local):

```
LENGTH
```

Purpose:

Draws an arrow using OpenGL commands.

Warnings:

Although the function changes the color used to yellow, it never changes the color back to what it was originally.

Create2DdrumDLs ()

[sm_drums.cpp]

Calls:

```
gluNewQuadric()
XtNumber()
glGenLists()
glNewList()
glColor3fv()
gluDisk()
glEndList()
```

Variables (global & static local):

```
sizes
DrumDLs[ ]
colors[ ]
```

Purpose:

Creates the display lists for the 2D drums.

Warnings:

LLDestroyList ()

[ll_lists.cpp]

Calls:

```
LLEraseList()
free()
```

Variables (global & static local):

Purpose:

Erases the linked list and frees the memory that was used.

Warnings:

MakeConnection ()

[sm_comm.cpp]

Calls:

- CheckComm()
- StartComm()
- XUProgError()
- XtVaSetValues()
- XtAppAddWorkProc()

Variables (global & static local):

- TopLevel
- info
- DisplayTog
- DiagTog
- house
- center
- DisplayWorkProcId

Purpose:

Checks the communication between the robot and the server. Starts if necessary. Also shows errors if there are ones present.

Warnings:

EndComm ()

[comm_ComUtil.cpp]

Calls:

- clock()
- CloseSocket()

Variables (global & static local):

- Client[]
- NORM
- HIGH
- ROBOTD
- CLOSE_CONNECTION
- REQ
- INACTIVE
- Time

Purpose:

This function terminates communication with the server. It sends a close connection signal to the server to let the server know to close the connection. This function closes both lines of communication and does not return until the server responds to the close connection request.

Warnings:

LLDelete ()

[ll_lists.cpp]

Calls:

fprintf()
LLQuickWriteTraverse()
destroy()
free()

Variables (global & static local):

stderr

Purpose:

Removes the selected node from a linked list.

Warnings:

NextLine ()

[sm_scanner.cpp]

Calls:

fgetc()
fprintf()

Variables (global & static local):

TabOffset
CurrentFile
Line[]
ChPos
ch

Purpose:

This function reads in the next line from the input file.

Warnings:

advance ()

[sm_scanner.cpp]

Calls:

Variables (global & static local):

ChPos
line[]
ch

Purpose:

Advances the character to the next line (prefixed).

Warnings:

InitVars ()

[dxf_flow.cpp]

Calls:

- ACmalloc()
- LLCreateList()

Variables (global & static local):

- model
- ACPModel
- LayerType
- LLlist
- EntityType

Purpose:

Sets the model parameters to default values.

Warnings:

ACPOpenFile ()

[dxf_scanner.cpp]

Calls:

- fopen ()
- ACPErrror()

Variables (global & static local):

- fp
- ACPLineNum

Purpose:

Opens the specified file. Reports error if one occurs.

Warnings:

ACPNextToken ()

[dxf_scanner.cpp]

Calls:

- NextLine()
- ParseInt()
- ACPErrror()
- GetBasicType()
- ParseHex()
- ParseFloat()

Variables (global & static local):

- line[]
- SemanticRec

Purpose:

Increments the current counter to the next token of the DXF file. Converts it into an appropriate ACP command.

Warnings:

MatchString ()

[dxf_scanner.cpp]

Calls:

CheckString()
ACPErrror()

Variables (global & static local):

SemanticRec

Purpose:

Compares two strings together and reports an error if they don't match.

Warnings:

ACPMatch ()

[dxf_scanner.cpp]

Calls:

ACPErrror()
ACPNextToken()

Variables (global & static local):

ACPCurrentToken

Purpose:

Matches the passed parameter to the current ACP token. If no match reports error and sets the ACP current token to the next token in the series.

Warnings:

ParseHeaderSection ()

[dxf_header.cpp]

Calls:

AllocateHeaderMemory()
ACPMatch()
ACPErrror()
AssignHeaderMemory()
MatchString()

Variables (global & static local):

ACPCurrentToken
HeaderVariables[]
SemanticRec

Purpose:

Parses the header section of code until a start code token is checked.

Warnings:

ParseTablesSection ()

[dxf_tables.cpp]

Calls:

ACPMatch()
CheckString()
ParseAppidTable()
ParseDimstyleTable()
ParseLtypeTable()
ParseLayerTable()
ParseStyleTable()
ParseUcsTable()
ParseViewTable()
ParseVportTable()
ACPError()

Variables (global & static local):

SemanticRec

Purpose:

Checks the stringy parameter of the SemanticRec class and then calls the function for the appropriate table.

Warnings:

ParseBlocksSection ()

[dxf_blocks.cpp]

Calls:

ACPMatch()
CheckString()
ParseSingleBlock()
ACPError()

Variables (global & static local):

SemanticRec

Purpose:

Calls the ParseSingleBlock function if there is a block in the current file.

Warnings:

ParseEntitiesSection ()

[dxf_entities.cpp]

Calls:

- ACPMatch()
- CheckString()
- StoreVertices()
- EntityIndex()
- ParseEntity()

Variables (global & static local):

- PolylineMode
- SemanticRec
- NextStart
- model
- polyline

Purpose:

While in the Entities section of the DXF file, calls all the necessary functions to complete the parsing of this section.

Warnings:

ACPErrror ()

[dxf_flow.cpp]

Calls:

Variables (global & static local):

- ACPErrrorMesg
- ACPErrrorLineNum
- ACPLineNum
- TerminateParse

Purpose:

Generates an error message for the AutoCAD Parser to call when errors arise.

Warnings:

ACPCloseFile ()

[dxf_scanner.cpp]

Calls:

- fclose()

Variables (global & static local):

- fp

Purpose:

Closes the file used by the AutoCAD parser.

Warnings:

HandleOptions ()

[dxf_tools.cpp]

Calls:

- ExplodeBlock()
- ChangeBylayer()
- ConvertSolid()
- SegmentArc()
- SegmentText()
- ConvertTrace()
- ConvertMesh()
- ExplodeDimension()
- ExplodePolyline()
- ConvertThickness()
- ConvertElevation()

Variables (global & static local):

Purpose:

Compares the option parameter to 11 different choices and calls the appropriate function.

Warnings:

BuildModelArray ()

[dxf_flow.cpp]

Calls:

- ACmalloc ()
- sizeof()
- LLRetrieve()

Variables (global & static local):

model

Purpose:

Moves the entity sub-list to a different section of the model linked list.

Warnings:

LLCreateNode ()

[ll_lists.cpp]

Calls:

malloc()
sizeof()
fprintf()

Variables (global & static local):

ListNode

Purpose:

Creates a new node on the linked list. Sets the prev and next sections to NULL.

Warnings:

LLeraseList ()

[ll_lists.cpp]

Calls:

LLDelete()

Variables (global & static local):

Variables

Purpose:

Deletes all the node entries in the linked lists.

Warnings:

CheckComm ()

[sm_comm.cpp]

Calls:

fork()
sprintf()
execl()
XULastFilename()
clock()
waitpid()
WIFEXITED()
WEXITSTATUS()

Variables (global & static local):

buffer
AriesRootDir

Purpose:

Checks the communication between robot and server.

Warnings:

StartComm ()

[comm_ComUtil.cpp]

Calls:

MAKEWORD()
WSAStartup()
printf()
LOBYTE()
HIBYTE()
WSACleanup()
fopen()
time()
InitializeClient()
InitializeK2AMem()
InitializeSpecialMem()
CreateSocket()
CloseSocket()
clock()
StartTimer()

Variables (global & static local):

ErrLog
stderr
TempId
ConnectedToServer

Purpose:

This function is called to create a socket connection and to connect to the server. It will wait until a connection is made before it exits the routine. This function calls two other routines. It will call a routine to initialize the client structures as well as call a routine to initialize a copy of the robots memory.

Warnings:

CloseSocket ()

[comm_Socket.cpp]

Calls:

ZeroMemory()
shutdown()

Variables (global & static local):

Client
ComStatus
ConnectedToServer

Purpose:

The purpose of this function is to close the socket connection off. It will return the state of the message structures to their initial values before closing the connection.

Warnings:

All the `bzero(Client[i].MessBuffer, MAX_MESS_SIZE)` statements have been converted to `ZeroMemory(Client[i].MessBuffer, MAX_MESS_SIZE)` statements.

LLQuickWriteTraverse ()

[ll_lists.cpp]

Calls:

Variables (global & static local):

Purpose:

Traverses the linked lists and returns the node that was searched for.

Warnings:

ACmalloc ()

[dxf_utility.cpp]

Calls:

malloc()
fprintf()
exit()

Variables (global & static local):

stderr

Purpose:

This function is used as a generic memory allocator coupled to the error handler of this application. When the function is called, the number of bytes to allocate is required, in addition to the name of the function calling ACmalloc() and the name of the variable being allocated. This helps with code development; if a malloc were to fail, having the additional information leaves no doubt which malloc failed, as dbx can be misleading.

Warnings:

ParseInt ()

[dxf_scanner.cpp]

Calls:

atoi()

Variables (global & static local):

line

Purpose:

Converts the contents of the current line to an integer.

Warnings:

GetBasicType ()

[dxf_header.cpp] & [dxf_scanner.cpp]

Calls:

ACPError ()

Variables (global & static local):

GroupCodes[]

Purpose:

Checks the GroupCodes structure and compares it to the current type. If a match occurs, returns the VarType. Otherwise reports an error.

Warnings:

There are two instances of this function. The only difference is the dxf_header function compares type to GroupCodes[i].GroupId. The dxf_scanner function compares type to GroupCodes[i].code.

ParseHex ()

[dxf_scanner.cpp]

Calls:

sscanf ()

Variables (global & static local):

line

Purpose:

Checks the line and determines if the value is present within the line..

Warnings:

ParseFloat ()

[dxf_scanner.cpp]

Calls:

atof ()

Variables (global & static local):

line

Purpose:

Converts the current line into a float value.

Warnings:

AllocateHeaderMemory ()

[dxf_header.cpp]

Calls:

GetBasicType()

ACmalloc()

sizeof()

Variables (global & static local):

IntIndex
FloatIndex
StringIndex
HeaderVariables[]
model

Purpose:

Allocates memory for the type of Header that is in the HeaderVariables structure.

Warnings:

AssignHeaderMemory ()

[dxf_header.cpp]

Calls:

GetBasicType()
ACstrdup()
ACPErrror()
ACPMatch()

Variables (global & static local):

ACPCurrentToken
model
SemanticRec
IntIndex
FloatIndex
StringIndex

Purpose:

Converts the passed variables to the value of the current token.

Warnings:

ParseAppidTable ()

[dxf_tables.cpp]

Calls:

ACPMatch()
CheckString()

Variables (global & static local):

SemanticRec
ACPCurrentToken

Purpose:

Converts all the tokens until a “ENDTAB” is reached.

Warnings:

ParseDimstyleTable ()

[dxf_tables.cpp]

Calls:

ACPMatch()
CheckString()

Variables (global & static local):

SemanticRec
ACPCurrentToken

Purpose:

Converts all the tokens until a “ENDTAB” is reached.

Warnings:

ParseLtypeTable ()

[dxf_tables.cpp]

Calls:

ACPMatch()
CheckString()

Variables (global & static local):

SemanticRec
ACPCurrentToken

Purpose:

Converts all the tokens until a “ENDTAB” is reached.

Warnings:

ParseLayerTable ()

[dxf_tables.cpp]

Calls:

ACPMatch()
CheckString()
ParseSingleTable()
ACPError()

Variables (global & static local):

SemanticRec
ACPCurrentToken

Purpose:

Converts all the tokens until a “ENDTAB” is reached. If there is no table under this layer generates an error or calls the ParseSingleTable function.

Warnings:

ParseStyleTable ()

[dxf_tables.cpp]

Calls:

ACPMatch()
CheckString()

Variables (global & static local):

SemanticRec
ACPCurrentToken

Purpose:

Converts all the tokens until a “ENDTAB” is reached.

Warnings:

ParseUcsTable ()

[dxf_tables.cpp]

Calls:

ACPMatch()
CheckString()

Variables (global & static local):

SemanticRec
ACPCurrentToken

Purpose:

Converts all the tokens until a “ENDTAB” is reached.

Warnings:

ParseViewTable ()

[dxf_tables.cpp]

Calls:

ACPMatch()
CheckString()

Variables (global & static local):

SemanticRec
ACPCurrentToken

Purpose:

Converts all the tokens until a “ENDTAB” is reached.

Warnings:

ParseVportTable ()

[dxf_tables.cpp]

Calls:

ACPMatch()

CheckString()

Variables (global & static local):

SemanticRec
ACPCurrentToken

Purpose:

Converts all the tokens until a “ENDTAB” is reached.

Warnings:

ParseSingleBlock ()

[dxf_blocks.cpp]

Calls:

LLCreateList()
sizeof()
CheckString()
StoreVertices()
EntityIndex()
ACPMatch()
ParseEntity()
LLAppend()

Variables (global & static local):

EntityType
PolylineMode
SemanticRec
NextStart
model
polyline

Purpose:

Parses until an “ENDBLK” is reached. While looping, converts the model’s VertexList values to the values of the polyline variable.

Warnings:

StoreVertices ()

[dxf_entities.cpp]

Calls:

ACmalloc()
sizeof()
LLRetrieve()
CopyACPoint3D()
LLEraseList()

Variables (global & static local):

Purpose:

Converts the Entity’s data into the data passed by the list. Then copies the points into a local entity, before deleting the list.

Warnings:

EntityIndex ()

[dxf_entities.cpp]

Calls:

CheckString ()

Variables (global & static local):

EntityFormats[]

SemanticRec

Purpose:

Checks the name of the Entity against the SemanticRec.stringv value. If a match occurs, function returns the integer value of the location.

Warnings:

ParseEntity ()

[dxf_entities.cpp]

Calls:

ACPMatch ()

AllocateEntity()

ParseEntityHeader()

IsXCoord()

ParseCoordinates()

ParseExtrusion()

IsExtendedEntityData()

GetTypeIndex()

StoreParameter()

IsNormal()

ACPErrror()

LLAppend()

Variables (global & static local):

format

EntityFormats[]

ACPCurrentToken

PolylineMode

model

polyline

Purpose:

Converts an entity and places it on the model list.

Warnings:

ExplodeBlock ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()
- CheckString()
- CopyBlockToEntities()
- LLDelete()

Variables (global & static local):
model

Purpose:

Parses the block and copies the contents to a linked list before deleting the model->EntityList.

Warnings:

ChangeBylayer ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()
- CheckString()
- LLDelete()

Variables (global & static local):
model

Purpose:

Changes the color so that each layer will be a different one.

Warnings:

ConvertSolid ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()
- CopyACPoint3D()

Variables (global & static local):
model

Purpose:

Creates a new list and places the points for this instance in their correct position and sets the id to something recognizable by the program

Warnings:

SegmentArc ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()
- CalcArbitraryAxes()
- CopyACPoint3D()
- AllocateEntity()
- CopyHeader()
- cos()
- sin()
- OrientLineECS()
- LLAppend()
- LLDelete()

Variables (global & static local):

- model

Purpose:

Creates an arc from the AutoCAD file.

Warnings:

SegmentText ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()
- strlen()
- AllocateEntity()
- CopyHeader()
- CopyPoint2D()
- OrientLineECS()
- LLAppend()
- LLDelete()

Variables (global & static local):

- model
- Fonts[]

Purpose:

Converts the text from DXF files into a font that the Site Manager program can understand and use. Appends the text to the EntityList of model. Then deletes a node from it.

Warnings:

ConvertTrace ()

[dxf_tools.cpp]

Calls:

Variables (global & static local):

Purpose:

There is nothing in this function!

Warnings:

I don't think this function has been completed. There is nothing in it.

ConvertMesh ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()
- LLDelete()
- AllocateEntity()
- CopyHeader()
- CopyACPoint3D()
- LLAppend()
- LLDelete()

Variables (global & static local):

model

Purpose:

Converts the DXF mesh to a usable Site Manager class.

Warnings:

ConvertPolyfaceMesh(poly) was commented out of the function.

ExplodeDimension ()

[dxf_tools.cpp]

Calls:

Variables (global & static local):

Purpose:

There is nothing in this function!!!

Warnings:

I do not think this function was completed. There is nothing in this function!!!

ConvertThickness ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()
- LineThickness()
- LLDelete()

Variables (global & static local):

model

Purpose:

Updates the thickness of the lines to the current value of the entity.

Warnings:

ExplodePolyline ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()
- AllocateEntity()
- CopyHeader()
- CopyACPoint3D()
- OrientLineECS()
- LLAppend()
- LLDelete()

Variables (global & static local):
model

Purpose:

Creates a polyline class that is used for splines (I think).

Warnings:

ConvertElevation ()

[dxf_tools.cpp]

Calls:

- LLRetrieve()

Variables (global & static local):
model

Purpose:

Goes through the entity list and converts the AutoCAD parameters into an elevation parameter.

Warnings:

InitializeSpecialMem ()

[comm_initialize.cpp]

Calls:

Variables (global & static local):
SpecialRoot

Purpose:

This function initializes the header node for the linked list of special variables. The header is initialized to values that ensure that it will never be used or deleted in the linked list.

Warnings:

Although the comments in this function say not to set any of the variables to negative values, that is all this function does.

InitializeK2AMem ()

[comm_Initialize.cpp]

Calls:

Variables (global & static local):
K2AMem[]

Purpose:

This function is called from the StartComm function. This function initializes the copy of the robots memory to zero.

Warnings:

InitializeClient ()

[comm_Initialize.cpp]

Calls:

ZeroMemory()

Variables (global & static local):
Client[]

Purpose:

This function is called to reset some important values in the message structure. The Id value for the structure, pointers, and the socket buffers are set to some initial values.

Warnings:

CreateSocket ()

[comm_Socket.cpp]

Calls:

ZeroMemory()

htons()

gethostbyname()

fprintf()

CopyMemory()

socket()

perror()

SetSocketOptions()

connect()

sizeof()

Variables (global & static local):
Client[]

Purpose:

The purpose of this function is to create a socket and make a connection. Another function is called to set up the socket options.

Warnings:

There is a LOT of commented out sections in this function.

StartTimer ()

[comm_initialize.cpp]

Calls:

Variables (global & static local):

Purpose:

Warnings:

Everything in this function has been commented out!

ParseSingleTable ()

[dxf_tables.cpp]

Calls:

- LLMalloc()
- ACPMatch()
- ACstrdup()
- ACPErr()
- LLAppend()

Variables (global & static local):

- model
- ACPCurrentToken
- SemanticRec

Purpose:

Parses a single table of AutoCAD information and converts it into usable site manger data.

Warnings:

***AllocateEntity ()**

[dxf_entities.cpp]

Calls:

- LLMalloc()
- ACmalloc()

Variables (global & static local):

- EntityFormats[]
- model

Purpose:

Allocates the memory resources for the entity that will be created.

Warnings:

ParseEntityHeader ()

[dxfl_entities.cpp]

Calls:

- SetEntityDefault()
- ACstrdup()
- ACPErr()
- ACPMatch()

Variables (global & static local):

- ACPCurrentToken
- SemanticRec

Purpose:

Goes through the header and extracts all relevant information.

Warnings:

ParseCoordinates ()

[dxfl_entities.cpp]

Calls:

- ACPMatch()

Variables (global & static local):

- ACPCurrentToken
- SemanticRec

Purpose:

Converts the AutoCAD coordinates into a system that site manager can use.

Warnings:

- Errors

ParseExtrusion ()

[dxfl_entities.cpp]

Calls:

- ACPMatch()

Variables (global & static local):

- SemanticRec

Purpose:

Converts the SemanticRec values into the entity's values.

Warnings:

GetTypeIndex ()

[dxf_entities.cpp]

Calls:

ACPErrror()

Variables (global & static local):

ACPCurrentToken

GroupCodes[]

format

Purpose:

Passes the type and index parameter the values of the ACPCurrentToken's data.

Warnings:

StoreParameter ()

[dxf_entities.cpp]

Calls:

ACstrdup()

ACPErrror()

Variables (global & static local):

SemanticRec

Purpose:

Stores the parameters of the SemanticRec class into the entity list.

Warnings:

IsNormal ()

[dxf_entities.cpp]

Calls:

Variables (global & static local):

format

Purpose:

Checks to see if any of the format list entries match the passed variable.

Warnings:

CopyBlockToEntities ()

[dxf_blocks.cpp]

Calls:

MatLoadId()
InsertBlock()

Variables (global & static local):

BlockMat

Purpose:

Calls two other functions.

Warnings:

CalcArbitraryAxes ()

[dxf_tools.cpp]

Calls:

CopyACPoint3D()
sqrt()
fabs()
ACPCrossProduct()

Variables (global & static local):

Wy
Wz

Purpose:

Performs the entire math that is needed to calculate the axes relative to Site Manager.

Warnings:

CopyHeader ()

[dxf_tools.cpp]

Calls:

Variables (global & static local):

Purpose:

Copies all the parameters of entity “b” into entity “a.”

Warnings:

OrientLineECS ()

[dxf_tools.cpp]

Calls:

CalcArbitraryAxes()
CopyACPoint3D()

Variables (global & static local):

Purpose:

Orients the line in the Site Manager's world.

Warnings:

Errors

LineThickness ()

[dxf_tools.cpp]

Calls:

AllocateEntity()
CopyHeader()
LLAppend()

Variables (global & static local):

Purpose:

Creates a new entity and converts its points to reflect the thickness of the imitated line.

Warnings:

SetSocketOptions ()

[comm_Socket.cpp]

Calls:

setsockopt()
sizeof()
perror()
MakeSocketEfficient()

Variables (global & static local):

Client[]

Purpose:

This function sets up the various options needed on the socket in order to ensure the socket will remain active and operate asynchronously.

Warnings:

There are two if statements that have been commented out by Mr. Paul McCarty.

ACstrdup ()

[dxf_utility.cpp]

Calls:

ACmalloc()
sizeof()
strcpy()

Variables (global & static local):

Purpose:

Given a string, this function will malloc a new copy and return a pointer to the new memory. (exact functionality of strdup, but POSIX compliant).

Warnings:

MatLoadId ()

[dxf_blocks.cpp]

Calls:

Variables (global & static local):

Purpose:

Converts the matrix that is passed into the function into an identity matrix.

Warnings:

Errors

InsertBlock ()

[dxf_blocks.cpp]

Calls:

LLRetrieve()
AdjustBlockMatrix()
MatTranslate()
AllocateEntity()
CopyEntity()
OrientCopy()
CheckString()
MatPush()
InsertBlock()
MatPop()
LLAppend()

Variables (global & static local):

BlockMat
model

Purpose:

Recursively calls itself (until the NodeCount of the BlockList reaches 0) so the function can perform all the necessary translates for the entity being placed on screen.

Warnings:

Errors

ACPCrossProduct ()

[dxf_math.cpp]

Calls:

Variables (global & static local):

Purpose:

Performs the entire math needed for a matrix cross product.

Warnings:

MakeSocketEfficient ()

[comm_Socket.cpp]

Calls:

setsockopt()
sizeof()
perror()

Variables (global & static local):

Client[]

Purpose:

This function sets up the socket so that it will be more efficient.

Warnings:

AdjustBlockMatrix ()

[dxf_blocks.cpp]

Calls:

MatTranslate()
MatRot()
MatScale()

Variables (global & static local):

BlockMat
attr[]

Purpose:

Performs a translate, rotation, and scaling of the matrix.

Warnings:

MatTranslate ()

[dxf_blocks.cpp]

Calls:

MatLoadId()
MatCopy()
MatMultiply()

Variables (global & static local):

Purpose:

Translates (moves) a matrix to the position specified in the passed parameters.

Warnings:

CopyEntity ()

[dxf_blocks.cpp]

Calls:

ACmalloc()
sizeof()
CopyACPoint3D()

Variables (global & static local):

Purpose:

Copies the entity “b” into entity “a.”

Warnings:

OrientCopy ()

[dxf_blocks.cpp]

Calls:

MatPntMultiply()

Variables (global & static local):

BlockMat

Purpose:

Calls the function MatPntMultiply() on all the vertexes of the entity to be copied

Warnings:

Errors

MatPush ()

[dxf_blocks.cpp]

Calls:

LLCreateList()
LLMalloc()
CopyMatrix()
LLAppend()

Variables (global & static local):

MatrixList
ACPMatrix

Purpose:

Creates a new matrix with the same properties as the original matrix.

Warnings:

MatPop()

[dxf_blocks.cpp]

Calls:

- LLRetrieve()
- CopyMatrix()
- LLDelete()

Variables (global & static local):

- MatrixList
- BlockMat

Purpose:

Removes the matrix that was created by the MatPush() function from the MatrixList.

Warnings:

MatRot ()

[dxf_blocks.cpp]

Calls:

- cos()
- sin()
- MatLoadId()
- MatCopy()
- MatMultiply()

Variables (global & static local):

- Variables

Purpose:

Rotates the matrix by the angle listed in the second parameter.

Warnings:

MatCopy ()

[dxf_blocks.cpp]

Calls:

Variables (global & static local):

Purpose:

Places all the entries of the source matrix into the destination matrix.

Warnings:

MatScale ()

[dxf_blocks.cpp]

Calls:

MatLoadId()
MatCopy()

Variables (global & static local):

Purpose:

Multiplies the matrix by the scaling factor.

Warnings:

MatMultiply ()

[dxf_tools.cpp]

Calls:

Variables (global & static local):

Purpose:

Adds the multiplication of the first two matrices into the result matrix. Function also clears the entry with 0.0 before placing the result.

Warnings:

Errors

MatPntMultiply ()

[dxf_blocks.cpp]

Calls:

CopyACPoint3D()

Variables (global & static local):

Purpose:

Multiplies a three dimensional ACPoint by the matrix passed.

Warnings:

Errors

Table of Contents

| | |
|-------------------------------------|----|
| Program Flow: Outline..... | 5 |
| Program Flow: Ring One Detail..... | 6 |
| Program Flow: Ring Two Detail | 9 |
| Header File Structure | 18 |
| Main.c {Include files}..... | 20 |
| main()..... | 21 |
| StartTasks() | 22 |
| StartServer() | 23 |
| InitializeRobot()..... | 24 |
| WaitToStart()..... | 26 |
| StartRecording() | 26 |
| ReferenceRobot() | 27 |
| GetToNextSubgoal() | 28 |
| CompleteNextSubgoal() | 29 |
| ReturnHome()..... | 31 |
| DockRobot() | 33 |
| ShutDownRobot() | 34 |
| EvlInit()..... | 35 |
| StartCom()..... | 36 |
| StartMemory() | 37 |
| InitializePower() | 38 |
| InitializeWarnLight()..... | 38 |
| CheckArray() {Thread} | 39 |
| HandleMessages() {thread}..... | 39 |
| Init_Winsock()..... | 41 |
| InitializeServer()..... | 41 |
| CreateSocket() | 42 |
| ServerCommand() {Thread} | 42 |
| InitializeOffsets()..... | 43 |
| InitSuperCom()..... | 43 |
| SyncClock() | 44 |
| InitializeNodeTable()..... | 44 |
| InitializePathNode()..... | 45 |
| InitializeInspectionTable() | 45 |
| InitializeFileTable() | 46 |
| ParseMission() | 47 |
| InitializePathDatabase()..... | 49 |
| InitializeMissionList()..... | 49 |
| CopyDrumDatabase()..... | 51 |
| OpenDrumDatabase() | 51 |
| CreateDrumDatabase()..... | 52 |
| GetNodeNumber()..... | 53 |
| InitializeHostVariables()..... | 53 |
| InitializeDataSet()..... | 54 |
| InitializeLift() | 54 |
| InitializeBarcode() | 55 |
| WriteOutImageDirectory() | 55 |
| // InitializeVision() | 56 |
| RecordMission() {thread} | 56 |
| GetReferenceActionFilename() | 58 |
| GetFileNameNumber()..... | 58 |

| | |
|--------------------------------|----|
| InsertMissionListData() | 59 |
| ExecuteProgram() | 60 |
| ResetMissionList() | 61 |
| ReadNextSubgoal() | 61 |
| FindPath()..... | 62 |
| RemoveProgramFromDB() | 62 |
| GoBackToLastNode() | 63 |
| MarkPathBlocked()..... | 63 |
| ResetVariables()..... | 65 |
| ReportProgramError()..... | 65 |
| GetDockActionFilename() | 66 |
| StopRecording()..... | 66 |
| AbortMission() | 67 |
| TransferAllData() | 67 |
| DiscontinueTasks() | 68 |
| ResSysOff()..... | 68 |
| InitializeSupervisor() | 69 |
| InitializeControl() | 69 |
| Supervisor() {thread}..... | 70 |
| Control() {thread} | 72 |
| InitializeK2AMem() | 73 |
| InitializeActiveBlocks() | 73 |
| Memory() {Thread} | 74 |
| MonitorPower() {Thread} | 74 |
| WarnProc() {Thread} | 75 |
| FillInArrayAgain()..... | 75 |
| SolveNxNArray()..... | 77 |
| GetResponse()..... | 77 |
| SetSocketOptions() | 78 |
| ReadFromClient() | 78 |
| ProcessClientCommands() | 79 |
| GetNewClient()..... | 80 |
| WriteToClient() | 80 |
| GetTimeFromServer()..... | 81 |
| LostTime()..... | 81 |
| BuildMissionFileName()..... | 83 |
| InitIO() | 83 |
| NextToken() | 84 |
| GetSiteName() | 84 |
| GetBuildingName()..... | 86 |
| GetStartNodeName()..... | 86 |
| GetHomeNodeName()..... | 87 |
| GetReferenceAction()..... | 87 |
| StartMission() | 88 |
| GetStartTime()..... | 88 |
| GetOffset()..... | 89 |
| CloseIO()..... | 89 |
| GetHomeNode()..... | 90 |
| GetActionFile()..... | 90 |
| BuildDatabaseFile() | 91 |
| RemoveFiles()..... | 91 |
| CreatePathDatabase()..... | 93 |
| BuildNodeTable() | 93 |
| BuildFileTable()..... | 94 |

| | |
|--------------------------------|-----|
| BuildPathList() | 94 |
| DeletePathDatabase() | 96 |
| BuildArray() | 96 |
| FillInArray() | 97 |
| MHCopyFile() | 97 |
| DBOpen() | 98 |
| DBCreateOnly() | 98 |
| DBWriteUserData() | 99 |
| MoveBBAbsolute() | 99 |
| InitPort() | 101 |
| SetScannerOperation() | 101 |
| GetValue() | 102 |
| Convert2ByteSigned() | 102 |
| GetFileName() | 103 |
| FillInPathName() | 103 |
| ReadInstructions() | 104 |
| AdjustRelativeOffsets() | 104 |
| AddHaltToProgram() | 105 |
| WritePathProgram() | 105 |
| MonitorMovement() | 107 |
| UpdateSubgoalStatus() | 108 |
| GetNodeName() | 108 |
| AdjustCurrentPosition() | 109 |
| RemoveFromList() | 109 |
| BlockFromList() | 110 |
| WriteOutMissionReport() | 110 |
| CloseDrumDatabases() | 111 |
| TransferDrumDatabase() | 111 |
| // *** TransferImages() | 112 |
| TransferRecordFile() | 112 |
| InformOffboardDatabase() | 113 |
| TransferLogFile() | 113 |
| ClearSupervisorPort() | 114 |
| InitializeSemaphores() | 114 |
| ClearControlPort() | 115 |
| HandleSupervisorWriteAndRead() | 115 |
| ReadFromControlPort() | 116 |
| WriteToControlPort() | 116 |
| PollNeededData() | 117 |
| ReadDataDirectly() | 117 |
| //AssertWarn() | 119 |
| //DeAssertWarn() | 119 |
| FillInVariable() | 120 |
| MakeSocketEfficient() | 120 |
| UpdateReadBuffer() | 121 |
| ExtractMess() | 121 |
| ReadSocketLine() | 122 |
| CloseSocket() | 122 |
| Send_Id() | 123 |
| Read_Block() | 123 |
| Write_Block() | 124 |
| Read_Var() | 124 |
| Read_Special() | 125 |
| Write_Special() | 125 |

| | |
|-------------------------|-----|
| Close_Connection() | 126 |
| Down_Load() | 126 |
| Dis_Asm() | 127 |
| Load_Status() | 127 |
| Com_Status() | 128 |
| Mission_Status() | 128 |
| Unknown_Command() | 129 |
| GetConnection() | 129 |
| CreateMess() | 130 |
| WriteSocketLine() | 130 |
| Connect() | 132 |
| SendMessageMH() | 132 |
| NextLine() | 133 |
| Eof() | 133 |
| ProcessFilename() | 134 |
| ProcessId() | 134 |
| ProcessLiteral() | 135 |
| Match() | 135 |
| GetAisleBehavior() | 136 |
| AddToNodeTable() | 136 |
| AddToFileTable() | 137 |
| AddToList() | 137 |
| SetLiftValues() | 138 |
| MoveLift() | 138 |
| SendCommand() | 139 |
| IsRelativeInstruction() | 139 |
| Communicate() | 140 |
| DownloadPath() | 140 |
| LoadDriveAndSteer() | 142 |

Program Flow: Outline

I. Mission() [Main.c]

A. Start() [Main.c]

1. StartTasks()[Main.c]
2. InitializeRobot() [Initialize.c]
3. StartServer() [StartServer.c]
4. WaitToStart() [Main.c]
5. ReferenceRobot() [Reference.c]
6. StartRecording() [Main.c]
7. GetToNextSubgoal() [Subgoal.c] {Main loop!}
8. CompleteNextSubgoal() [Subgoal.c]
9. CompleteNextSubgoal() [Subgoal.c] {End main loop}
10. ReturnHome() [Subgoal.c]
11. DockRobot() [Reference.c]
12. ShutDownRobot() [Main.c]

Program Flow: Ring One Detail

I. Mission() [Main.c]

A. Start() [Main.c]

1. StartTasks()[Main.c]

- 1.1 EvlInit() [Resource.c]
- 1.2 StartCom() [Resource.c]
- 1.3 StartMemory() [Memory.c]
- 1.4 InitializePower() [Power.c]
- 1.5 InitializeWarnLight() [WarnLight.c]
- 1.6 CheckArray() [Main.c]
- 1.7 HandleMessages() [Mesg.c]

2. InitializeRobot() [Initialize.c]

- 2.1. InitializeOffsets() [Initialize.c]
- 2.2. InitSuperCom() [ComUtil.c]
- 2.3. SyncClock() [Comm.c]
- 2.4. InitializeNodeTable() [NodeTable.c]
- 2.5. InitializePathNode() [List.c]
- 2.6. InitializeInspectionTable() [Initialize.c]
- 2.7. InitializeFileTable() [FileTable.c]
- 2.8. ParseMission() [Parser.c]
- 2.9. InitializePathDatabase() [Initialize.c]
- 2.10. InitializeMissionList() [MissionList.c]
- 2.11. CopyDrumDatabase() [Database.c]
- 2.12. OpenDrumDatabase() [Database.c]
- 2.13. CreateDrumDatabase() [Database.c]
- 2.14. GetNodeNumber() [NodeTable.c] {Start}
- 2.15. GetNodeNumber() [NodeTable.c] {Home}
- 2.16. InitializeHostVariables() [Initialize.c]
- 2.17. InitializeDataSet() [DrumData.c]
- 2.18. InitializeLift() [Lift.c]
- 2.19. InitializeBarcode() [Barcode.c]
- 2.20. WriteOutImageDirectory() [Copy.c]
- 2.21. InitializeVision[Vision.c]

3. StartServer() [Resource.e StartServer.c]

- 3.1. InitializeServer() [Initialize.c]
- 3.2. InitializeCom() [Initialize.c]
- 3.3. CreateSocket() [Socket.c]
- 3.4. ServerCommand() [StartServer.c]

4. WaitToStart() [Main.c]

- 4.1. time()
- 4.2. localtime()

5. ReferenceRobot() [Reference.c]

- 5.1. GetNodeNumber() [NodeTable.c] {to}
- 5.2. GetNodeNumber() [NodeTable.c] {from}
- 5.3. GetReferenceActionFilename() [Reference.c]
- 5.4. GetFileNameNumber() [FileTable.c]
- 5.5. InsertMissionListData() [MissionList.c]
- 5.6. ExecuteProgram() [ReadFile.c]

6. StartRecording() [Main.c]

- 6.1. CreateDirectory()
- 6.2. fopen()
- 6.3. RecordMission() [Main.c]

7. GetToNextSubgoal() [Subgoal.c] {Main loop!}

- 7.1. ResetMissionList() [MissionList.c]
- 7.2. ReadNextSubgoal() [Subgoal.c]
- 7.3. FindPath() [Array.c]
- 7.4. ExecuteProgram() [ReadFile.c]
- 7.5. ResetVariables() [Status.c]

8. CompleteNextSubgoal() [Subgoal.c]

- 8.1. ResetMissionList() [MissionList.c]
- 8.2. FindPath() [Array.c]
- 8.3. FindInspectionPath() [Inspection.c]
- 8.4. ExecuteProgram() [ReadFile.c]
- 8.5. UpdateSubgoalStatus() [Subgoal.c]
- 8.6. ResetVariables() [Status.c]

9. CompleteNextSubgoal() [Subgoal.c] {End main loop}

- 9.1. ResetMissionList() [MissionList.c]
- 9.2. FindPath() [Array.c]
- 9.3. FindInspectionPath() [Inspection.c]
- 9.4. ExecuteProgram() [ReadFile.c]
- 9.5. UpdateSubgoalStatus() [Subgoal.c]
- 9.6. ResetVariables() [Status.c]

10. ReturnHome() [Subgoal.c]

- 10.1. ResetMissionList() [MissionList.c]
- 10.2. FindPath() [Array.c]
- 10.3. ExecuteProgram() [ReadFile.c]
- 10.4. ResetVariables() [Status.c]

11. DockRobot() [Reference.c]

- 11.1. GetNodeNumber() [NodeTable.c] {to}
- 11.2. GetNodeNumber() [NodeTable.c] {from}
- 11.3. GetDockActionFilename() [Reference.c]
- 11.4. GetFileNameNumber() [FileTable.c]
- 11.5. InsertMissionListData() [MissionList.c]
- 11.6. ExecuteProgram() [ReadFile.c]

12. ShutDownRobot() [Main.c]

- 12.1. StopRecording() [Main.c]
- 12.2. taskNameToId()
- 12.3. taskDelete() {Server}
- 12.4. AbortMission() [Subgoal.c]
- 12.5. TransferAllData() [Copy.c]
- 12.6. DiscontinueTasks() [Main.c]
- 12.7. ResSysOff() [Resource.c] {That's all folks!}

Program Flow: Ring Two Detail

I. Mission() [Main.c]

A. Start() [Main.c]

1. StartTasks()[Main.c]

1.1 **EvlInit()** [Logger.c] - Starts Event Logger.

1.1.1. EvlTask() [Logger.c] { *Thread }

1.1 **StartCom()** [Communication.c] - Low-level serial interface

1.2.1. msgQCreate()

1.2.2. InitializeIOBoard() [Digital.c]

1.2.3. DeAssert() [Digital.c]

1.2.4. InitializeSupervisor() [Communication.c]

1.2.5. InitializeControl() [Communication.c]

1.2.6. Supervisor() [Communication.c] { *Thread }

1.2.7. Control() [Communication.c] { *Thread }

1.3. **StartMemory()** [Memory.c]

1.3.1. InitSuperCom() [ComUtil.c]

1.3.2. InitializeK2Amem() [Memory.c]

1.3.3. InitializeActiveBlocks() [Memory.c]

1.3.4. Memory() [Memory.c] { *Thread }

1.4. **InitializePower()** [Power.c]

1.4.1. MonitorPower() [Power.c] { *Thread }

1.5. **InitializeWarnLight()** [WarnLight.c]

1.5.1. WarnProc() [WarnLight.c] { *Thread }

1.6. **CheckArray()** [Main.c] { *Thread }

1.7. **HandleMessages()** [Mesg.c] { *Thread }

2. InitializeRobot() [Initialize.c]

2.1. **InitializeOffsets()** [Initialize.c] - Calculate 'DrumDistance' and 'DrumAngle'

2.2. **InitSuperCom()** [ComUtil.c] - Get a Message Queue ID

2.3. **SyncClock()** [Comm.c] - Really screwed up!

2.4. **InitializeNodeTable()** [NodeTable.c] - Init.

2.5. **InitializePathNode()** [List.c] - Init.

2.6. **InitializeInspectionTable()** [Initialize.c] - Init.

2.7. **InitializeFileTable()** [FileTable.c] - Init.

2.8. **ParseMission()** [Parser.c]

2.8.1. BuildMissionFileName() [Parser.c]

2.8.2. InitIO() [Scanner.c]

2.8.3. NextToken() [Scanner.c]

2.8.4. CloseIO() [Scanner.c]

2.8.5. GetActionFile() [Initialize.c]

2.8.6. BuildDatabaseFile [Parser.c]

2.8.7. RemoveFiles() [Parser.c]

2.9. **InitializePathDatabase()** [Initialize.c]

2.9.1. CreatePathDatabase() [List.c]

2.9.2. BuildNodeTable() [NodeTable.c]

2.9.3. BuildFileTable() [FileTable.c]

2.9.4. BuildPathList() [List.c]

2.9.5. DeletePathDatabase() [List.c]

2.9.6. BuildArray() [Array.c]

2.9.7. FillInArray() [Array.c]

2.9.8. SolveNxNArray() [Array.c]

- 2.10. **InitializeMissionList()** [MissionList.c] - Init.
- 2.11. **CopyDrumDatabase()** [Database.c] - File copy
- 2.12. **OpenDrumDatabase()** [Database.c]
 - 2.12.1. DBOpen() [commands.c]
- 2.13. **CreateDrumDatabase()** [Database.c]
 - 2.13.1. DBCreateOnly() [commands.c]
 - 2.13.2. DBWriteUserData() [??]
- 2.14. **GetNodeNumber()** [NodeTable.c] {Start} - Init.
- 2.15. **GetNodeNumber()** [NodeTable.c] {Home}
- 2.16. **InitializeHostVariables()** [Initialize.c] - Init.
- 2.17. **InitializeDataSet()** [DrumData.c] - Init.
- 2.18. **InitializeLift()** [Lift.c]
 - 2.18.1. MoveBBAbsolute() [Lift.c]
- 2.19. **InitializeBarcode()** [Barcode.c]
 - 2.19.1. InitPort() [Barcode.c]
 - 2.19.2. InitScanner() [Barcode.c]
- 2.20. **WriteOutImageDirectory()** [Copy.c] - mkdir
- 2.21. **InitializeVision()** [Vision.c] - This would be Breck.

3. StartServer() [StartServer.c]

3.1. **InitializeServer()** [\socket\Initialize.c] - Init.

3.2. **InitializeCom()** [\socket\Initialize.c]

3.2.1. **InitSuperCom()** [ComUtil.c]

3.3. **CreateSocket()** [Socket.c] - Set up sockets

3.4. **ServerCommand()** [StartServer.c] { *Thread }

4. WaitToStart() [Main.c] - Just waiting

4.1. **time()**

4.2. **localtime()**

5. ReferenceRobot() [Reference.c]

5.1. **GetNodeNumber()** [NodeTable.c] {to} - Resolve node # from name

5.2. **GetNodeNumber()** [NodeTable.c] {from} - Resolve node # from name

5.3. **GetReferenceActionFilename()** [Reference.c] - Assumes only one!

5.4. **GetFileNameNumber()** [FileTable.c] - Resolve file # from filename

5.5. **InsertMissionListData()** [MissionList.c] - Insert a (?) path

5.6. **ExecuteProgram()** [ReadFile.c]

5.6.1. **FillInPathName()** [ReadFile.c]

5.6.2. **GetFileName()** [FileTable.c]

5.6.3. **ReadInstructions()** [ReadFile.c]

5.6.4. **AdjustRelativeOffsets()** [ReadFile.c]

5.6.5. **AddHaltToProgram()** [ReadFile.c]

5.6.6. **WritePathProgram()** [Download.c]

5.6.7. **MonitorMovement()** [Monitor.c]

6. StartRecording() [Main.c]

6.1. **CreateDirectory()**

6.2. **fopen()**

6.3. **RecordMission()** [Main.c] { *Thread } - Playback info

7. GetToNextSubgoal() [Subgoal.c] {Main loop!}

7.1. **ResetMissionList()** [MissionList.c] - Free list memory

7.2. **ReadNextSubgoal()** [Subgoal.c]

7.2.1. GetNodeNumber() [NodeTable.c] {start & end}

7.2.2. UpdateSubgoalStatus() [Subgoal.c] {start & end}

7.3. **FindPath()** [Array.c]

7.3.1. GetNodeName() [NodeTable.c]

7.3.2. InsertMissionListData() [MissionList.c] - Insert a (?) path

7.4. **ExecuteProgram()** [ReadFile.c]

7.4.1. FillInPathName() [ReadFile.c]

7.4.2. GetFileName() [FileTable.c]

7.4.3. ReadInstructions() [ReadFile.c]

7.4.4. AdjustRelativeOffsets() [ReadFile.c]

7.4.5. AddHaltToProgram() [ReadFile.c]

7.4.6. WritePathProgram() [Download.c]

7.4.7. MonitorMovement() [Monitor.c]

7.5. **ResetVariables()** [Status.c] - VERY minor

8. CompleteNextSubgoal() [Subgoal.c]

8.1. **ResetMissionList() [MissionList.c]** - Free list memory

8.2. FindPath() [Array.c]

8.2.1. GetNodeName() [NodeTable.c]

8.2.2. InsertMissionListData() [MissionList.c] - Insert a (?) path

8.3. FindInspectionPath() [Inspection.c]

8.3.1. GetInspectionActionFilename() [Inspection.c]

8.3.2. GetFileNameNumber() [FileTable.c]

8.3.3. InsertMissionListData() [MissionList.c] - Insert a (?) path

8.4. ExecuteProgram() [ReadFile.c]

8.4.1. FillInPathName() [ReadFile.c]

8.4.2. GetFileName() [FileTable.c]

8.4.3. ReadInstructions() [ReadFile.c]

8.4.4. AdjustRelativeOffsets() [ReadFile.c]

8.4.5. AddHaltToProgram() [ReadFile.c]

8.4.6. WritePathProgram() [Download.c]

8.4.7. MonitorMovement() [Monitor.c]

8.4.8. UpdateSubgoalStatus() [Subgoal.c]

8.5.1. ReadDataDirectly() [Memory.c]

8.6. **ResetVariables() [Status.c]** - VERY minor

9. CompleteNextSubgoal() [Subgoal.c] {End main loop}

9.1. ResetMissionList() [MissionList.c]

9.2. FindPath() [Array.c]

9.2.1. GetNodeName() [NodeTable.c]

9.2.2. InsertMissionListData() [MissionList.c] - Insert a (?) path

9.3. FindInspectionPath() [Inspection.c]

9.3.1. GetInspectionActionFilename() [Inspection.c]

9.3.2. GetFileNameNumber() [FileTable.c]

9.3.3. InsertMissionListData() [MissionList.c] - Insert a (?) path

9.4. ExecuteProgram() [ReadFile.c]

9.4.1. FillInPathName() [ReadFile.c]

9.4.2. GetFileName() [FileTable.c]

9.4.3. ReadInstructions() [ReadFile.c]

9.4.4. AdjustRelativeOffsets() [ReadFile.c]

9.4.5. AddHaltToProgram() [ReadFile.c]

9.4.6. WritePathProgram() [DownLoad.c]

9.4.7. MonitorMovement() [Monitor.c]

9.5. UpdateSubgoalStatus() [Subgoal.c]

9.5.1. ReadDataDirectly() [Memory.c]

9.6. **ResetVariables() [Status.c]** - VERY minor

10. ReturnHome() [Subgoal.c]

10.1. **ResetMissionList() [MissionList.c]** - Free list memory

10.2. **FindPath() [Array.c]**

10.2.1. GetNodeName() [NodeTable.c]

10.2.2. InsertMissionListData() [MissionList.c] - Insert a (?) path

10.3. **ExecuteProgram() [ReadFile.c]**

10.3.1. FillInPathName() [ReadFile.c]

10.3.2. GetFileName() [FileTable.c]

10.3.3. ReadInstructions() [ReadFile.c]

10.3.4. AdjustRelativeOffsets() [ReadFile.c]

10.3.5. AddHaltToProgram() [ReadFile.c]

10.3.6. WritePathProgram() [DownLoad.c]

10.3.7. MonitorMovement() [Monitor.c]

10.4. **ResetVariables() [Status.c]** - VERY minor

11. DockRobot() [Reference.c]

11.1. **GetNodeNumber() [NodeTable.c] {to & from}** - Resolve # from name

11.2. **GetDockActionFilename() [Reference.c]** - Assumes only ONE dock action!

11.3. **GetFileNameNumber() [FileTable.c]** - Resolve # from filename

11.4. **InsertMissionListData() [MissionList.c]** - Insert a (?) path

11.5. **ExecuteProgram() [ReadFile.c]**

11.5.1. FillInPathName() [ReadFile.c]

11.5.2. GetFileName() [FileTable.c]

11.5.3. ReadInstructions() [ReadFile.c]

11.5.4. AdjustRelativeOffsets() [ReadFile.c]

11.5.5. AddHaltToProgram() [ReadFile.c]

11.5.6. WritePathProgram() [DownLoad.c]

11.5.7. MonitorMovement() [Monitor.c]

12. ShutDownRobot() [Main.c]

12.1. **StopRecording() [Main.c]** - Kill recorder

12.2. **taskNameToId()**

12.3. **taskDelete() {Server}**

12.4. **AbortMission() [Subgoal.c]** - Subgoal = ABORTED

12.4.1. UpdateSubgoalStatus() [Subgoal.c]

12.5. **TransferAllData() [Copy.c]**

12.5.1. WriteOutMissionReport() [Report.c]

12.5.2. CloseDrumDatabases() [Database.c]

12.5.3. TransferDrumDatabase() [Copy.c]

12.5.4. TransferImages() [Copy.c]

12.5.5. TransferRecordFile() [Copy.c]

12.5.6. InformOffboardDatabase() [Comm.c]

12.5.7. TransferLogFile() [Copy.c]

12.6. **DiscontinueTasks() [Main.c]** - Kill 'MonitorMessages', 'CheckArray', 'MonitorPower', 'WarnLight', 'K2Amemory', 'Supervisor', 'Control', and 'EvlTask'

12.6.1. taskNameToId()

12.6.2. taskDelete()

12.7. **ResSysOff() [Resource.c] {That's all folks!}**

Header File Structure

Includes.h {VxWorks specific except for ComUtil.h, laser.h, vision.h. }

```
“ComUtil.h” [\langland\com\tty] { *Local }
<vxWorks.h> [\h] { Wind River }
<taskLib.h> [\h] { Wind River }
<stdio.h> [\h] { Wind River }
<ioLib.h> [\h] { Wind River }
<string.h> [\h] { Wind River }
<msgQLib.h> [\h] { Wind River }
<selectLib.h> [\h] { Wind River }
<sys/times.h> [\h] { Wind River }
<vxWorks.h> [\h] { Wind River }
<taskLib.h> [\h] { Wind River }
<ioLib.h> [\h] { Wind River }
<stdio.h> [\h] { Wind River }
<stdlib.h> [\h] { Wind River }
<string.h> [\h] { Wind River }
<time.h> [\h] { Wind River }
<sys/types.h> [\h] { Wind River }
<sys/stat.h> [\h] { Wind River }
<dirent.h> [\h] { Wind River }
<unistd.h> [\h] { Wind River }
<msgQLib.h> [\h] { Wind River }
<selectLib.h> [\h] { Wind River }
<sys/times.h> [\h] { Wind River }
<hostLib.h> [\h] { Wind River }
<sockLib.h> [\h] { Wind River }
<fcntl.h> [\h] { Wind River }
<netinet/in.h> [\h] { Wind River }
<sys/socket.h> [\h] { Wind River }
<errnoLib.h> [\h] { Wind River }
<sysLib.h> [\h] { Wind River }
<math.h> [\h] { Wind River }
<itxcore.h> [??] **
```


<dpx.h> [??] **
<amc1.h> [??] **
<ima.h> [??] **
<clu.h> [??] **
<hf.h> [??] **
<laser.h> [\\langland\\mission\\vision] { *Local }
<vision.h> [\\langland\\mission\\vision] { *Local }

Main.c {Include files}

```
"Mission.h"  
    "Includes.h" - Trouble!  
    "Structures.h" {Which one?}  
        "Defines.h" {Which one?}  
    "HostVariables.h" - Not much  
    "Logger.h"  
    "States.h" {**Changed "OVERFLOW" to "OVRFLOW"}  
    "Lift.h"  
    "DC_01.h"  
    "Digital.h"  
"Resource.h" {Which one?}  
"Variables.h" {Which one?} - Enum for computer variables
```

main()

[main.cpp]

Calls:

- StartTasks() [main.cpp]
- StartServer() [StartServer.cpp]
- InitializeRobot() [Initialize.cpp]
- WaitToStart() [main.cpp]
- StartRecording() [main.cpp]
- ReferenceRobot() [Reference.cpp]
- GetToNextSubgoal() [Subgoal.cpp]
- CompleteNextSubgoal() [Subgoal.cpp]
- ReturnHome() [Subgoal.cpp]
- DockRobot() [Reference.cpp]
- ShutDownRobot() [main.cpp]

Vars (global):

- MissionStatus [int, Declarations.cpp]
- InspectMode [int, Subgoal.cpp]

Purpose:

Calls initialization routines StartTasks(), StartServer(), InitializeRobot(), WaitToStart(), StartRecording(), and ReferenceRobot(); then enters the main program loop where GetToNextSubgoal() and CompleteNextSubgoal() are called; and then calls shutdown routines ReturnHome(), DockRobot(), and ShutDownRobot().

Warnings:

"while (Flag != QUIT)" QUIT = -2, but Flag values are BOOL;
GetToNextSubgoal()
might return QUIT!

StartTasks()

[Main.cpp]

Calls:

```
EvIInit() [Logger.cpp]
StartCom() [Communication.cpp]
StartMemory() [Memory.cpp]
InitializePower() [Power.cpp]
// InitializeWarnLight() [WarnLight.cpp]
CheckArray() {thread} [Main.cpp]
HandleMessages() {thread} [Mesg.cpp]
```

Vars (global):

```
hCheckArray [HANDLE, Main.cpp]
CheckArrayID [DWORD, Main.cpp]
hHandleMessages [HANDLE, Main.cpp]
HandleMessagesID [DWORD, Main.cpp]
```

Purpose:

Calls initialization routines `EvIInit()`, `StartCom()`, `StartMemory()`, `InitializePower()`, and `InitializeWarnLight()`; starts threads `CheckArray()` and `HandleMessages()`.

Warning:

`InitializeWarnLight()` is commented out right now.

StartServer()

[StartServer.cpp]

Calls:

- Init_Winsock() [StartServer.cpp]
- InitializeServer() [CommInit.cpp]
- CreateSocket() [Socket.cpp]
- ServerCommand() (thread) [StartServer.cpp]

Vars (global):

- hServerCommand [HANDLE, Main.cpp]
- ServerCommandID [DWORD, Main.cpp]

Purpose:

Calls Init_Winsock(), InitializeServer(), and CreateSocket(); and starts the ServerCommand() thread.

Warnings:

InitializeCom() commented out (function in CommInit.cpp is commented out and the important call to InitSuperCom() can be found in InitializeRobot()).

InitializeRobot()

[Initialize.cpp]

Calls:

```
InitializeOffsets() [Initialize.cpp]
InitSuperCom() [ComUtil.cpp]
SyncClock() [Comm.cpp]
InitializeNodeTable() [NodeTable.cpp]
InitializePathNode() [List.cpp]
InitializeInspectionTable() [Initialize.cpp]
InitializeFileTable() [FileTable.cpp]
ParseMission() [Parser.cpp]
InitializePathDatabase() [Initialize.cpp]
InitializeMissionList() [MissionList.cpp]
CopyDrumDatabase() [Database.cpp]
OpenDrumDatabase() [Database.cpp]
CreateDrumDatabase() [Database.cpp]
GetNodeNumber() [NodeTable.cpp]
InitializeHostVariables() [Initialize.cpp]
InitializeDataSet() [DrumData.cpp]
// InitializeLift() [Lift.cpp]
// InitializeBarcode() [Barcode.cpp]
WriteOutImageDirectory() [Copy.cpp]
// InitializeVision() [Vision.cpp]
```

Vars (global):

```
CurrentLocation [int, Global.cpp]
HomePosition [int, Global.cpp]
```

Purpose:

Calls InitializeOffsets(), InitSuperCom(), SyncClock(), InitializeNodeTable(), InitializePathNode(), InitializeInspectionTable(), InitializeFileTable(), ParseMission(), InitializePathDatabase(), InitializeMissionList(), CopyDrumDatabase(), OpenDrumDatabase(), and CreateDrumDatabase(); then CurrentLocation and HomePosition are set to the value returned by GetNodeNumber(); then calls are made to InitializeHostVariables(), InitializeDataSet(), InitializeLift(), InitializeBarcode(), WriteOutImageDirectory(), InitializeVision().

Warnings:

InitializeLift(), InitializeBarcode(), and InitializeVision() are commented out!

WaitToStart()

[Main.cpp]

Calls:

Vars (global):

- StartYear [int, global.cpp]
- StartMonth [int, global.cpp]
- StartDay [int, global.cpp]
- StartHour [int, global.cpp]
- StartMinute [int, global.cpp]

Purpose:

Synchronizes clock onboard robot with Site Manager.

Warnings:

Seems to be logically flawed. Modifications have been made to handle this that have not been tested!

StartRecording()

[Main.cpp]

Calls:

RecordMission() (thread) [Main.cpp]

Vars (global):

- RecordFile [static FILE*, Main.cpp]
- HistoryName [char[], Global.cpp]
- hRecordMission [HANDLE, Main.cpp]
- RecordMissionID [DWORD, Main.cpp]

Purpose:

Creates (or recreates) the History file and kicks off the RecordMission() thread.

Warnings:

No longer creates a "History" directory of the same name (this is not legal under NT).

ReferenceRobot()

[Reference.cpp]

Calls:

- GetNodeNumber() [NodeTable.cpp]
- GetReferenceActionFilename() [Reference.cpp]
- GetFileNameNumber() [FileTable.cpp]
- InsertMissionListData() [MissionList.cpp]
- ExecuteProgram() [ReadFile.cpp]

Vars (global):

- ReferenceActionTo [char[], Global.cpp]
- ReferenceActionFrom [char[], Global.cpp]
- MissionAnswer [int*, Global.cpp]

Purpose:

Calls GetNodeNumber() on ReferenceActionTo to determine "to" (local var) and calls GetNodeNumber() on ReferenceActionFrom to determine "from" so that the proper reference action may be called. The function GetReferenceActionFilename() returns a file pointer (local var) to the proper reference file. MissionAnswer[0] is given the returning value of GetFileNameNumber() and is given to InsertMissionListData() along with local vars "to" and "from". Finally, ExecuteProgram() is called.

Warnings:

Note similarities to DockRobot(). Also note the opportunity for a reference action to have different starting and ending points.

GetToNextSubgoal()

[Subgoal.cpp]

Calls:

- ResetMissionList() [MissionList.cpp]
- ReadNextSubgoal() [Subgoal.cpp]
- FindPath() [Array.cpp]
- ExecuteProgram() [ReadFile.cpp]
- RemoveProgramFromDB() [Subgoal.cpp]
- ResetVariables() [Status.cpp]
- RetryCurrentSubgoal() [Subgoal.cpp]
- ShutDownRobot() [Main.cpp]
- MarkPathBlocked() [Subgoal.cpp]
- AbortMission() [Subgoal.cpp]
- ReportProgramError() [Main.cpp]
- UpdateSubgoalStatus() [Subgoal.cpp]

Vars (global):

- CurrentMode [static int, Subgoal.cpp]
- CurrentLocation [int, Global.cpp]
- StartNodeNumber [int, Global.cpp]
- HaltCondition [int, Global.cpp]

Purpose:

Calls are made to ResetMissionList(), ReadNextSubgoal(), and FindPath().

The Status

that FindPath() returns switches several cases: END_OF_FILES, calls ExecuteProgram(),

if there is an error, several cases are switched on global variable

HaltCondition:

ERROR_IN_PROGRAM, calls RemoveProgramFromDB(), ResetVariables(), RetryCurrentSubgoal(), and ShutDownRobot() if it fails;

PATH_BLOCKED, calls

MarkPathBlocked(), ResetVariables(), RetryCurrentSubgoal(), and

ShutDownRobot() if

it fails; SYSTEM_NOT_READY calls ShutDownRobot(); GO_HOME calls AbortMission(); CONTINUE, ERROR_WHILE_EXECUTING, STUCK, and DC_IGNORE call the default case which calls ResetVariables(),

RetryCurrentSubgoal(),

and ShutDownRobot() if it fails. If ExecuteProgram() succeeds several cases

are switched

on global variable HaltCondition: ERROR_IN_PROGRAM, calls RemoveProgramFromDB(); PATH_BLOCKED, calls MarkPathBlocked();

SYSTEM_NOT_READY calls ShutDownRobot(); GO_HOME sets local

variable Status

to QUIT; CONTINUE, ERROR_WHILE_EXECUTING, STUCK, and DC_IGNORE call the default case. Finally, still in case END_OF_FILES, ResetVariables() is called.

If FindPath() returns NO_PATH ReportProgramError() and UpdateSubgoalStatus() are called. If FindPath() returns ALREADY_THERE the function ends. If FindPath() returns PATH_ERROR ShutDownRobot() is called. The default case calls ReportProgramError() and UpdateSubgoalStatus().

Warnings:

Bear in mind similarities between this function and CompleteNextSubgoal() and ReturnHome()!

CompleteNextSubgoal()

[Subgoal.cpp]

Calls:

ResetMissionList() [MissionList.cpp]
FindPath() [Array.cpp]
FindInspectionPath() [Inspection.cpp]
ExecuteProgram() [ReadFile.cpp]
UpdateSubgoalStatus() [Subgoal.cpp]
RemoveProgramFromDB() [Subgoal.cpp]
GoBackToLastNode() [Subgoal.cpp]
ShutDownRobot() [Main.cpp]
MarkPathBlocked() [Subgoal.cpp]
ResetVariables() [Status.cpp]
ReportProgramError() [Main.cpp]

Vars (global):

InspectMode [int, Subgoal.cpp]
CurrentLocation [int, Global.cpp]
EndNodeNumber [int, Global.cpp]
CurrentMode [static int, Subgoal.cpp]
HaltCondition [int, Global.cpp]
StartNodeNumber [int, Global.cpp]

Purpose:

First ResetMissionList() is called, and depending on whether or not the path is an inspection path, FindPath() or FindInspectionPath() is called. The Status that FindPath()

or FindInspectionPath() returns switches several cases: END_OF_FILES, calls ExecuteProgram(), if there is an error, several cases are switched on global variable HaltCondition: ERROR_IN_PROGRAM, calls UpdateSubgoalStatus(), RemoveProgramFromDB(), GoBackToLastNode() and ShutDownRobot() if it fails; PATH_BLOCKED, calls UpdateSubgoalStatus(), MarkPathBlocked(), GoBackToLastNode(), and ShutDownRobot() if it fails; SYSTEM_NOT_READY calls UpdateSubgoalStatus() and ShutDownRobot(); GO_HOME calls UpdateSubgoalStatus(), GoBackToLastNode(), and AbortMission(); CONTINUE, ERROR_WHILE_EXECUTING, STUCK, and DC_IGNORE call the default case which calls UpdateSubgoalStatus(), GoBackToLastNode(), and ShutDownRobot() if it fails.

If ExecuteProgram() succeeds several cases are switched on global variable HaltCondition: ERROR_IN_PROGRAM, calls RemoveProgramFromDB(); PATH_BLOCKED, calls MarkPathBlocked(); SYSTEM_NOT_READY calls ShutDownRobot(); GO_HOME sets local variable Status to QUIT; CONTINUE, ERROR_WHILE_EXECUTING, STUCK, and DC_IGNORE call the default case. Finally, still in case END_OF_FILES, ResetVariables() is called. (The same as GetToNextSubgoal())

If FindPath() returns NO_PATH ReportProgramError() and UpdateSubgoalStatus() are called. If FindPath() returns ALREADY_THERE UpdateSubgoalStatus() is called. If FindPath() returns PATH_ERROR ShutDownRobot() is called. The default case calls ReportProgramError() and UpdateSubgoalStatus().

Warnings:

Bear in mind similarities between this function and UpdateSubgoalStatus() and ReturnHome()!

ReturnHome()

[Subgoal.cpp]

Calls:

- ResetMissionList() [MissionList.cpp]
- FindPath() [Array.cpp]
- ExecuteProgram() [ReadFile.cpp]
- RemoveProgramFromDB() [Subgoal.cpp]
- ResetVariables() [Status.cpp]
- ShutDownRobot() [Main.cpp]
- ReportProgramError() [Main.cpp]

Vars (global):

- EndNodeNumber [int, Global.cpp]
- HomePosition [int, Global.cpp]
- CurrentMode [static int, Subgoal.cpp]
- CurrentLocation [int, Global.cpp]
- HaltCondition [int, Global.cpp]

Purpose:

Calls are made to ResetMissionList() and FindPath(). The Status that FindPath() returns switches several cases: END_OF_FILES, calls ExecuteProgram(), if there is an error, several cases are switched on global variable HaltCondition: ERROR_IN_PROGRAM, calls RemoveProgramFromDB() and ResetVariables(); PATH_BLOCKED, calls MarkPathBlocked() and ResetVariables(); SYSTEM_NOT_READY calls ShutDownRobot(); GO_HOME sets local variable Status to FALSE; CONTINUE, ERROR_WHILE_EXECUTING, STUCK, and DC_IGNORE call the default case which calls ResetVariables().

If ExecuteProgram() succeeds several cases are switched on global variable HaltCondition: ERROR_IN_PROGRAM, calls RemoveProgramFromDB(); SYSTEM_NOT_READY calls ShutDownRobot(); GO_HOME sets local variable Status to FALSE; PATH_BLOCKED, CONTINUE, ERROR_WHILE_EXECUTING, STUCK, and DC_IGNORE call the default case. Finally, still in case END_OF_FILES, ResetVariables() is called.

If FindPath() returns NO_PATH ReportProgramError() is called. If FindPath() returns

ALREADY_THERE the function ends. If FindPath() returns PATH_ERROR ShutDownRobot() is called. The default case calls ReportProgramError().

Warnings:

Bear in mind similarities between this function and UpdateSubgoalStatus() and CompleteNextSubgoal()! Take note of PATH_BLOCKED moved to default case.

DockRobot()

[Reference.cpp]

Calls:

- GetNodeNumber() [NodeTable.cpp]
- GetDockActionFilename() [Reference.cpp]
- GetFileNameNumber() [FileTable.cpp]
- InsertMissionListData() [MissionList.cpp]
- ExecuteProgram() [ReadFile.cpp]

Vars (global):

- HomeNode [char[], Global.cpp]
- MissionAnswer [int*, Global.cpp]

Purpose:

Calls GetNodeNumber() on HomeNode to determine "to" (local var) and calls GetNodeNumber() on HomeNode to determine "from" so that the proper docking action may be called. The function GetDockActionFilename() returns a file pointer (local var) to the proper reference file. MissionAnswer[0] is given the returning value of GetFileNameNumber() and is given to InsertMissionListData() along with local vars "to" and "from". MissionAnswer[1] is set to END_OF_FILES. Finally, ExecuteProgram() is called.

Warnings:

Note similarities to ReferenceRobot().

ShutDownRobot()

[Main.cpp]

Calls:

- StopRecording() [Main.cpp]
- AbortMission() [Subgoal.cpp]
- TransferAllData() [Copy.cpp]
- DiscontinueTasks() [Main.cpp]
- ResSysOff() [Resource.cpp]

Vars (global):

- MissionStatus [Declarations.cpp]
- hServerCommand [Main.cpp]

Purpose:

Shuts down the robot. Stops recording (StopRecording()), then terminates the

ServerCommand() thread, calls AbortMission(), transferAllData(), DiscontinueTasks(), and ResSysOff(), which exits the program.

Warnings:

Terminates everything. Might want to shut down in a civil manner, or keep running.

Evllnit()

[Logger.cpp]

Calls:

Vars (global):

RobotsName [char[], Global.cpp]

hLogFile [HANDLE, Logger.cpp]

Purpose:

Creates a log file from Computer's name.

Warnings:

May want to change to "append" and put a time/date stamp in!

StartCom()

[Communication.cpp]

Calls:

- InitializeSupervisor() [Communication.cpp]
- InitializeControl() [Communications.cpp]
- Supervisor() {thread} [Communications.cpp]
- Control() {thread} [Communications.cpp]

Vars (global):

- Comm_timeout [int, Main.cpp]
- SuperMsgQID [HANDLE, Communication.cpp]
- ControlMsgQID [HANDLE, Communication.cpp]
- MissionMsgQID [HANDLE, Communication.cpp]
- hSupervisor [HANDLE, Main.cpp]
- SupervisorID [DWORD, Main.cpp]
- hControl [HANDLE, Main.cpp]
- ControlID [DWORD, Main.cpp]

Purpose:

Creates the Super, Control, and Mission pipes; calls InitializeSupervisor()
and InitializeControl(); and starts the Supervisor() and Control() threads.

Warnings:

Might want to consider "message" pipes as opposed to "byte" pipes.
Need to insure that pipe message architecture is NOT a bottleneck!

StartMemory()

[Memory.cpp]

Calls:

- InitSuperCom() [ComUtil.cpp]
- InitializeK2AMem() [Memory.cpp]
- InitializeActiveBlocks() [Memory.cpp]
- Memory() {Thread} [Memory.cpp]

Vars (global):

- ReadDirectSem [HANDLE, Memory.cpp]
- WriteDirectSem [HANDLE, Memory.cpp]
- MemoryComLink [HANDLE, Memory.cpp]
- ReadDirectlyComLink [HANDLE, Memory.cpp]
- WriteDirectlyComLink [HANDLE, Memory.cpp]
- hMemory [HANDLE, Main.cpp]
- MemoryID [DWORD, Main.cpp]

Purpose:

Creates the Read and Write Direct semaphores (ReadDirectSem and WriteDirectSem).

Calls InitSuperCom() on MemoryComLink, ReadDirectlyComLink, and WriteDirectlyComLink. Then calls InitializeK2AMem(), InitializeActiveBlocks(), and starts the Memory() thread.

Warnings:

InitializePower()

[Power.cpp]

Calls:

MonitorPower() {Thread} [Power.cpp]

Vars (global):

hPower [HANDLE, Main.cpp]

PowerID [DWORD, Main.cpp]

Purpose:

Kicks off the MonitorPower() thread.

Warnings:

InitializeWarnLight()

[WarnLight.cpp]

Calls:

WarnProc() {Thread} [WarnLight.cpp]

Vars(global):

hWarn_Light [HANDLE, Main.cpp]

Warn_LightID [DWORD, Main.cpp]

Purpose:

Kicks off Warning Light thread.

Warnings:

CheckArray() {Thread}

[Main.cpp]

Calls:

FillInArrayAgain() [Array.cpp]
SolveNxNArray() [Array.cpp]

Vars(global):

MissionSizeOfArray [int, Global.cpp]
MissionPathList [PathListPntr, Global.cpp]

Purpose:

Who knows?

Warnings:

Local variable "flag" is an int and might should be BOOL.
Sleeps for 30 minutes!

HandleMessages() {thread}

[Mesg.cpp]

Calls:

GetResponse() [Mesg.cpp]

Vars(global):

MissionMsgQID [Communication.cpp]

Purpose:

This thread loops forever waiting on a connection from the MissionMsgQID pipe. The pipe is read and the name of the sender (pipe) is determined. The remaining message is given to GetResponse(). A pipe is opened to the sender, and data is returned.. The replying pipe is closed and the waiting pipe disconnects.

Warnings:

Error handling is crude at best (all the exit(1)'s). Consider "message" pipes as opposed

to "byte" pipes. Need to check for bottlenecks.

Init_Winsock()

[StartServer.cpp]

Calls:

Vars(global):

Purpose:

Start up Winsock. Currently we are using version 1.1.

Warnings:

Need to call WASCleanUp() in Shutdown(), or something like that.

InitializeServer()

[CommInit.cpp]

Calls:

Vars(global):

Purpose:

Functionality duplicated (and called earlier) in InitializeRobot().

Warnings:

May want to remove.

CreateSocket()

[Socket.cpp]

Calls:

SetSocketOptions() [Socket.cpp]

Vars(global):

Server [SocketDesc, Declarations.cpp]

Purpose:

This function gets the host (robot) name to get the IP. A socket is created and bound to host IP, ect. The socket is then set to listen and SetSocketOptions() is called.

Warnings:

ServerCommand() {Thread}

[StartServer.cpp]

Calls:

ReadFromClient() [ServerUtil.cpp]

ProcessClientCommands() [ProcessCommand.cpp]

GetNewClient() [ServerUtil.cpp]

WriteToClient() [ServerUtil.cpp]

Vars(global):

Server [SocketDesc, Declarations.cpp]

Client [SocketDesc[], Declarations.cpp]

DataReady [int, StartServer.cpp]

Purpose:

Does some FD_??? that I don't understand and calls select(). Then, while DataReady is TRUE, ReadFromClient(), ProcessClientCommands(), GetNewClient(), and WriteToClient().

Warnings:

"fd" may not be used, what ill effects does this have?

InitializeOffsets()

[Initialize.cpp]

Calls:

Vars (global):

DrumAngle [int, Global.cpp]

Purpose:

Calculates global var "DrumAngle" from defines in Mission.h.

Warnings:

May need to check math and "representation" of vars (float, int, ect).

InitSuperCom()

[ComUtil.cpp]

Calls:

Vars (global):

Purpose:

Accepts a pointer to a HANDLE and a name (string) and creates a pipe of that name and returns the HANDLE.

Warnings:

SyncClock()

[Comm.cpp]

Calls:

 GetTimeFromServer() [Comm.cpp]
 LostTime() [Comm.cpp]

Vars (global):

 Time [static struct tm, Comm.cpp]

Purpose:

 Calls GetTimeFromServer() and then LostTime(). If there is a discrepancy, the system time is updated.

Warnings:

 Year 2000 problems? Not sure who/where the "time server" runs.

InitializeNodeTable()

[NodeTable.cpp]

Calls:

Vars (global):

 MissionRootNodeTable [struct NodeData, NodeTable.cpp]

Purpose:

 Initializes NodeData structure.

Warnings:

InitializePathNode()

[List.cpp]

Calls:

Vars (global):

MissionRootPathNode [struct PathData, Global.cpp]

Purpose:

Initializes PathData structure.

Warnings:

InitializeInspectionTable()

[Initialize.cpp]

Calls:

Vars (global):

MissionRootInspectionTable [struct InspectionTable, Global.cpp]

Purpose:

Initializes InspectionTable structure.

Warnings:

Status of "-1", no enum?

InitializeFileTable()

[FileTable.cpp]

Calls:

Vars (global):

MissionRootFileTable [struct FileData, FileTable.cpp]

Purpose:

Initializes FileData structure.

Warnings:

ParseMission()

[Parser.cpp]

Calls:

- BuildMissionFileName() [Parser.cpp]
- InitIO() [Scanner.cpp]
- NextToken() [Scanner.cpp]
- GetSiteName() [Parser.cpp]
- GetBuildingName() [Parser.cpp]
- GetStartNodeName() [Parser.cpp]
- GetHomeNodeName() [Parser.cpp]
- GetReferenceAction() [Parser.cpp]
- StartMission() [Parser.cpp]
- GetStartTime() [Parser.cpp]
- GetOffset() [Parser.cpp]
- CloseIO() [Scanner.cpp]
- GetHomeNode() [Initialize.cpp]
- GetActionFile() [Initialize.cpp]
- BuildDatabaseFile() [Parser.cpp]
- RemoveFiles() [Parser.cpp]

Vars (global):

- ErrorString [static char[], Parser.cpp]
- CurrentToken [int, Scanner.cpp]
- State [static int, Parser.cpp]

Purpose:

Calls BuildMissionFileName(), InitIO(), and NextToken(). Switches on token returned.

SITE calls GetSiteName(), BUILDING calls GetBuildingName(), STARTING_NODE calls GetStartNodeName(), HOME_NODE calls GetHomeNodeName(), REFERENCE_ACTION calls GetReferenceAction(), BEGIN_MISSION calls StartMission(), START_TIME calls GetStartTime(), OFFSET calls GetOffset(), and the default case returns ERROR. Then CloseIO() is called. Global variable "State" is checked against GOT_SITE_NAME, GOT_BUILDING_NAME, GOT_MISSION, GOT_START, GOT_REFERENCE_ACTION, and GOT_OFFSET. If State does not indicate GOT_HOME, GetHomeNode() is called. Then GetActionFile(), BuildDatabaseFile(), and RemoveFiles() are called.

Warnings:

No GOT_DOCK, or some such thing? Need to test for errors on
GetHomeNode(),
GetActionFile(), and BuildDatabaseFile().

InitializePathDatabase()

[Initialize.cpp]

Calls:

- CreatePathDatabase() [List.cpp]
- BuildNodeTable() [NodeTable.cpp]
- BuildFileTable() [FileTable.cpp]
- BuildPathList() [List.cpp]
- DeletePathDatabase() [List.cpp]
- BuildArray() [Array.cpp]
- FillInArray() [Array.cpp]
- SolveNxNArray() [Array.cpp]

Vars (global):

- MissionAnswer [int*, Global.cpp]
- MissionSizeOfArray [int, Global.cpp]

Purpose:

Calls CreatePathDatabase(), BuildNodeTable(), BuildFileTable(), BuildPathList(), DeletePathDatabase(), BuildArray(), FillInArray(), and SolveNxNArray().

Then

allocates memory (MissionAnswer) for MissionSizeOfArray integers.

Warnings:

Dynamic memory allocation. . . is it set free?

InitializeMissionList()

[MissionList.cpp]

Calls:

Vars (global):

- MissionListRoot [struct MissionList, Global.cpp]

Purpose:

Initializes the MissionList structure, MissionListRoot.

Warnings:

Might want to enum "-1".

CopyDrumDatabase()

[Database.cpp]

Calls:

MHCopyFile() [Copy.cpp]

Vars (global):

SiteName [char[], Global.cpp]

BuildingName [char[], Global.cpp]

DatabaseFile [char[], Global.cpp]

Purpose:

Calls MHCopyFile() after assembling the proper Read/Write directory names. Copies drum database to temp directory.

Warnings:

Might want to #define hard-coded ".pddb" extension.

OpenDrumDatabase()

[Database.cpp]

Calls:

DBOpen() [Commands.cpp]

Vars (global):

DatabaseFile [char[], Global.cpp]

ReadDesc [static DBDesc*, Database.cpp]

ReadOnBoardDatabase [int, Global.cpp]

Purpose:

Calls DBOpen() on temp drum database.

Warnigns:

ReadOnBoardDatabase is an int, wants to be a BOOL. Might want to #define hard-coded ".pddb" extension.

CreateDrumDatabase()

[Database.cpp]

Calls:

DBCCreateOnly() [Commands.cpp]
DBWriteUserData() [Commands.cpp]

Vars (global):

RobotsName [char[], Global.cpp]
DatabaseTemplate [char[], Global.cpp]
WriteDesc [static DBDesc*, Database.cpp]
WriteOnBoardDatabase [int, Global.cpp]
SiteName [char[], Global.cpp]
BuildingName [char[], Global.cpp]

Purpose:

Calls DBCreateOnly() on local var DatabaseReport, and global var DatabaseTemplate.

Calls DBWriteUserData() on local var HeaderInfo.

Warnings:

Might want to change HeaderInfo[512] to HeaderInfo[some #define].

Might want to #define hard-coded ".sddb" extension.

No longer create a same-name subdirectory (NT/NFS incompatibility)

GetNodeNumber()

[NodeTable.cpp]

Calls:

Vars (global):

MissionRootNodeTable [struct NodeData, NodeTable.cpp]

Purpose:

Receives local var NodeName and searches for same name within MissionRootNodeTable and returns NodeNumber if successful.

Warnings:

Slow search method?

InitializeHostVariables()

[Initialize.cpp]

Calls:

Vars (global):

HostVariable [int[], Global.cpp]

Purpose:

initializes HostVariable[] array to 0.

Warnings:

InitializeDataSet()

[DrumData.cpp]

Calls:

Vars (global):

DrumDataSet [WriteDataSet[], Global.cpp]

InventoryId [char[][]], Global.cpp]

ImageNames [char[][][]], Global.cpp]

Purpose:

Initialize InventoryId array, DrumDataSet array, and ImageNames array.

Warnings:

Check space allocated vs space REQUIRED!

InitializeLift()

[Lift.cpp]

Calls:

MoveBBAbsolute() [Lift.cpp]

Vars (global):

DrumStatus [int, Declarations.cpp]

LevelStatus [int, Declarations.cpp]

Purpose:

Calls MoveBBAbsolute() to LEVEL0 and SIZE_55GALLON, and initializes global variables DrumStatus and LevelStatus.

Warnings:

Commented out in InitializeRobot(). Who knows if it works!

InitializeBarcode()

[Barcode.cpp]

Calls:

- InitPort() [Barcode.cpp]
- SetScannerOperation() [Barcode.cpp]

Vars (global):

Purpose:

Initializes barcode hardware by calling InitPort() and SetScannerOperation().

Warnings:

Is commented out in InitializeRobot(). Has not been tested yet!

WriteOutImageDirectory()

[Copy.cpp]

Calls:

Vars (global):

- SiteName [char[], Global.cpp]
- BuildingName [char[], Global.cpp]

Purpose:

Composes two strings, a filename based on SiteName and BuildingName, and a filename for images within the "tmp" subdirectory. The first filename (string) is written into the second when it is opened.

Warnings:

// InitializeVision()

[Vision.cpp]

Calls:

- init_system() []
- init_doe_iluts() []
- init_find_tb() []
- init_laser() []

Vars (global):

Not sure.

Purpose:

Initializes vision.

Warnings:

Ignored for now. And commented out!

RecordMission() {thread}

[Main.cpp]

Calls:

- GetValue() [Lift.cpp]
- Convert2ByteSigned() [Lift.cpp]

Vars (global):

- ContinueToRecord [static int, Main.cpp]
- RecordFile [static FILE, Main.cpp]
- DoneRecording [static int, Main.cpp]

Purpose:

Records aspects of the robot's motion for playback. While ContinueToRecord is TRUE the following values are recorded: CPS_CPS1POS, CPS_CPS2POS, CPS_CPS4POS, DC01_XPOS, DC01_YPOS, and DC01_AZIMUTH. They are written to RecordFile in that order, every half a second. When ContinueToRecord is TRUE and the loop ends and the global variable DoneRecording is set to TRUE.

Warnings:

ContinueToRecord should be a BOOL. DoneRecording should be a BOOL.
Dependent on SLEEPRECORDER (Includes.h).

GetReferenceActionFilename()

[Reference.cpp]

Calls:

 GetFileName() [FileTable.cpp]

Vars (global):

 MissionPathList [PathListPntr *, Global.cpp]

Purpose:

 Searches MissionPathList for a path program which satisfies "to" and "from" and is also a REFERENCE action. Returns the appropriate action file name (NULL otherwise).

Warnings:

 Is "NULL" tested for when called?

GetFileNameNumber()

[FileTable.cpp]

Calls:

Vars (global):

 MissionRootFileTable [struct FileData, FileTable.cpp]

Purpose:

 Searches for a filename, Name, in the MissionRootFileTable and returns a number.

Warnings:

 Returns a "-1" if file not found.

InsertMissionListData()

[MissionList.cpp]

Calls:

Vars (global):

MissionListRoot [struct MissionList, Global.cpp]

Purpose:

Allocates a MissionList structure and initializes it, passing it to the global variable

MissionListRoot.

Warnings:

Deallocation?

ExecuteProgram()

[ReadFile.cpp]

Calls:

- FillInPathName() [ReadFile.cpp]
- GetFileName() [FileTable.cpp]
- ReadInstructions() [ReadFile.cpp]
- AdjustRelativeOffsets() [ReadFile.cpp]
- AddHaltToProgram() [ReadFile.cpp]
- WritePathProgram() [Download.cpp]
- MonitorMovement() [Monitor.cpp]

Vars (global):

- MissionListRoot [struct MissionList, Global.cpp]
- BatteryFlag [int, Global.cpp]
- HaltCondition [int, Global.cpp]
- IndexToBuffer [int, ReadFile.cpp]
- MissionAnswer [int*, Global.cpp]
- NumberOfInstructions [int, ReadFile.cpp]
- ProgramBuffer [unsigned char[], ReadFile.cpp]
- Acceleration [unsigned char[], ReadFile.cpp]

Purpose:

First, BatteryFlag is checked and then a loop begins which lasts until global variable

MissionAnswer (incremented) indicates END_OF_FILES. Within this loop, FillInPathName() is called on what GetFileName() returns. The program length is

checked and then ReadInstructions() and AdjustRelativeOffsets() are called. If the

program is longer than 255 instructions, AddHaltToProgram() is called along with

WritePathProgram() and MonitorMovement(). Outside of the while loop, AddHaltToProgram() is called along with WritePathProgram() and MonitorMovement().

Warnings:

Local var InspectMode is also a global name, is there confusion? Code replication

{AddHaltToProgram(), WritePathProgram() and MonitorMovement()} might cause

trouble. Also need to check the 255 interactions!

ResetMissionList()

[MissionList.cpp]

Calls:

Vars (global):

MissionListRoot [struct MissionList, Global.cpp]

Purpose:

This function free()'s the Mission List.

Warnings:

What if MissionListRoot is NULL?

ReadNextSubgoal()

[Subgoal.cpp]

Calls:

GetNodeNumber() [NodeTable.cpp]

UpdateSubgoalStatus() [Subgoal.cpp]

Vars (global):

CurrentSubgoal [InspectionPntr, Subgoal.cpp]

StartNodeNumber [int, Global.cpp]

EndNodeNumber [int, Global.cpp]

Purpose:

The CurrentSubgoal pointer is incremented and if not equal to NULL, StartNodeNumber and EndNodeNumber are set and tested for validity.

The function

returns with its variable parameter set to TRUE (or Quit on an error).

Warnings:

StartNodeNumber and EndNodeNumber are members of MissionList too!

Var

parameter TRUE vs QUIT values (need BOOL). How is return(ERROR) handled?

FindPath()

[Array.cpp]

Calls:

 GetNodeName() [NodeTable.cpp]
 InsertMissionListData() [MissionList.cpp]

Vars (global):

 MissionSizeOfArray [int, Global.cpp]
 MissionAnswer [int*, Global.cpp]
 MissionArray [ArrayPntr*, Array.cpp]

Purpose:

 Calls GetNodeName() for debug purposes. Then allocates memory for local variable "Files". Tests to be sure that parameters "i" and "j" are not equal. Then creates an array of filename "numbers" (?) in local var "Files". Calls InsertMissionListData() on parameter "j" then increments "j" along "MissionArray[][]From". The "Files" memory is then released.

Warnings:

 Produces debug data that you may want to curb.

RemoveProgramFromDB()

[Subgoal.cpp]

Calls:

 AdjustCurrentPosition() [MissionList.cpp]
 RemoveFromList() [List.cpp]
 FillInArrayAgain() [Array.cpp]
 SolveNxNArray() [Array.cpp]

Vars (global):

 ProgramCounter [int, Global.cpp]

Purpose:

First AdjustCurrentPosition() is called on global var ProgramCounter, the return value is given to RemoveFromList() and the functions FillInArrayAgain() and SolveNxNArray() are called.

Warnings:

Global Var ProgramCounter has same name as "MissionList.ProgramCounter" .

GoBackToLastNode()

[Subgoal.cpp]

Calls:

AdjustCurrentPosition() [MissionList.cpp]
ResetMissionList() [MissionList.cpp]
FindPath() [Array.cpp]
ExecuteProgram() [ReadFile.cpp]

Vars (global):

ProgramCounter [int, Global.cpp]
CurrentMode [static int, Subgoal.cpp]

Purpose:

Calls AdjustCurrentPosition(), ResetMissionList(), and FindPath(). If FindPath() returns "END_OF_FILES" ExecuteProgram() is called.

Warnings:

MarkPathBlocked()

[Subgoal.cpp]

Calls:

AdjustCurrentPosition() [MissionList.cpp]
BlockFromList() [List.cpp]
FillInArrayAgain() [Array.cpp]
SolveNxNArray() [Array.cpp]

Vars (global):

ProgramCounter [int, Global.cpp]

Purpose:

First `AdjustCurrentPosition()` is called on global var `ProgramCounter`, the return value is given to `BlockFromList()` and the functions `FillInArrayAgain()` and `SolveNxNArray()` are called.

Warnings:

Very similar to `RemoveProgramFromDB()`.

ResetVariables()

[Status.cpp]

Calls:

Vars (global):

Oldstate [static int, Status.cpp]

Counter [static int, Status.cpp]

HaltCondition []

Purpose:

Reinitialize status (?) variables.

Warnings:

Enum (-1) for "oldstate" ?

ReportProgramError()

[Main.cpp]

Calls:

GetNodeName() [NodeTable.cpp]

Vars (global):

CurrentLocation [int, Global.cpp]

EndNodeNumber [int, Global.cpp]

Purpose:

Calls GetNodeName() to set the local vars "Start" and "End" . Gives error report.

Warnings:

GetDockActionFilename()

[Reference.cpp]

Calls:

 GetFileName() [FileTable.cpp]

Vars (global):

 MissionPathList [PathListPntr *, Global.cpp]

Purpose:

 Searches MissionPathList for a path program which satisfies "to" and "from" and is also a DOCKING action. Returns the appropriate action file name (NULL otherwise).

Warnings:

 Is return value "NULL" tested for when called?

StopRecording()

[Main.cpp]

Calls:

Vars (global):

 ContinueToRecord [static int, Main.cpp]

 DoneRecording [static int, Main.cpp]

Purpose:

 Sets global var ContinueToRecord to FALSE and waits until global var DoneRecording is set to TRUE. Sleeping for a SLEEPRECORDER quantumn.

Warnings:

 Need to make these BOOL' s!

AbortMission()

[Subgoal.cpp]

Calls:

UpdateSubgoalStatus() [Subgoal.cpp]

Vars (global):

CurrentSubgoal [InspectionPntr, Subgoal.cpp]

Purpose:

Traverses CurrentSubgoal calling UpdateSubgoalStatus(ABORTED).

Warnings:

TransferAllData()

[Copy.cpp]

Calls:

WriteOutMissionReport() [Report.cpp]
CloseDrumDatabases() [Database.cpp]
TransferDrumDatabase() [Copy.cpp]
// TransferImages() [Copy.cpp]
TransferRecordFile() [Copy.cpp]
InformOffboardDatabase() [Comm.cpp]
TransferLogFile() [Copy.cpp]

Vars (global):

Purpose:

Calls WriteOutMissionReport(), CloseDrumDatabases(),
TransferDrumDatabase(),
TransferImages(), TransferRecordFile(), InformOffboardDatabase(), and
TransferLogFile() [Copy.cpp]

Warnings:

The function call to TransferImages() is commented out!

DiscontinueTasks()

[Main.cpp]

Calls:

Vars (global):

- hHandleMessages [HANDLE, Main.cpp]
- hCheckArray [HANDLE, Main.cpp]
- hPower [HANDLE, Main.cpp]
- hMemory [HANDLE, Main.cpp]
- hSupervisor [HANDLE, Main.cpp]
- hControl [HANDLE, Main.cpp]

Purpose:

Calls TerminateThread() on the global vars hHandleMessages, hCheckArray, hPower, hMemory, hSupervisor, and hControl; shutting down those threads.

Warnings:

Might want to kill threads in a more "civilized" manner.

ResSysOff()

[Resource.cpp]

Calls:

Vars (global):

Purpose:

Calls exit(1) and shuts down Mission handler.

Warnings:

Might want a more "civilized" ending.

InitializeSupervisor()

[Communication.cpp]

Calls:

ClearSupervisorPort() [TtyUtil.cpp]
InitializeSemaphores() [ComUtil.cpp]

Vars (global):

hSuper [HANDLE, Communication.cpp]

Purpose:

Creates global var handle, hSuper, to Supervisor COM port (SUPERCOM, Includes.h).

Gets COM state and modifies the DCB, then sets COM state. Gets COM timeouts and

modifies the timeouts structure, then sets COM timeouts. Calls ClearSupervisorPort() and InitializeSemaphores().

Warnings:

Timeouts are the same as those for pipes (READTIMEOUT and WRITETIMEOUT, Includes.h).

InitializeControl()

[Communications.cpp]

Calls:

ClearControlPort() [TtyUtil.cpp]

Vars (global):

hCtrl [HANDLE, Communication.cpp]

Purpose:

Creates global var handle, hCtrl, to Supervisor COM port (CONTROLCOMM,

Includes.h). Gets COM state and modifies the DCB, then sets COM state. Gets COM

timeouts and modifies the timeouts structure, then sets COM timeouts.
Calls
ClearControlPort().

Warnings:

Timeouts are the same as those for pipes (READTIMEOUT and WRITETIMEOUT, Includes.h). Has much in common with InitializeSupervisor(), should consolodate code!

Supervisor() {thread}

[Communications.cpp]

Calls:

HandleSupervisorWriteAndRead() [TtyUtil.cpp]

Vars (global):

SuperMsgQID [HANDLE, Communication.cpp]

Purpose:

Loops forever. Waits for a connection on SuperMsgQID pipe. Reads COM_MAX_LEN into a buffer. Searches the buffer for a "," and divides it into sender's name (pipe_name) and message (buffer2). The message is cat'd with "\r\n" and given to HandleSupervisorWriteAndRead(). Another while(TRUE) loop is created. Inside, an attempt is made to create a handle to pipe "pipe_name" and wait for it. Break from the loop if a timeout. If a connection is successfully made with the response pipe, the data returned from HandleSupervisorWriteAndRead() (buffer2) is written to the response pipe unless that operation failed (Size < 0), in which case the ORIGINAL message (buffer) is returned. The handle is closed. Finally, the SuperMsgQID pipe is disconnected.

Warnings:

Lots of "exit(1)"'s. Need error handling! Could replace "while(TRUE)" with "while(BOOL)" and exit more gracefully than killing the threads. Second "while(TRUE)" probably NEVER loops. ***Move SuperMsgQID pipe disconnect to right after read?***

Control() {thread}

[Communications.cpp]

Calls:

ReadFromControlPort() [TtyUtil.cpp]
WriteToControlPort() [TtyUtil.cpp]

Vars (global):

ControlMsgQID [HANDLE, Communication.cpp]

Purpose:

Loops forever. If ReadFromControlPort() returns TRUE, the following happens. A pipe is created (hMissionQ), and a buffer (buffer2) containing the string CTRLMSGQID, is written and the handle is closed. Then the function waits for a connection to the pipe ControlMsgQID. The buffer (buffer) is filled by reading the pipe. An integer (Status) and a message (buffer2) are taken from the buffer (buffer). "\r\n" are appended to the message (buffer2). The message is then passed to the function WriteToControlPort(). The pipe (ControlMsgQID) is then disconnected.

Warnings:

Lots of "exit(1)"'s. Need error handling! Could replace "while(TRUE)" with "while(BOOL)" and exit more gracefully than killing the threads. ***Move ControlMsgQID pipe disconnect to right after read?*** Might want to remove EVL_INFO messages that are represent statuses and not errors! Handle "scanf()" differently in Control() and Supervisor()!

InitializeK2AMem()

[Memory.cpp]

Calls:

Vars (global):

K2AMem [static int, Memory.cpp]

Purpose:

Sets all members of K2AMem (NUM_VARIABLES) to zero.

Warnings:

InitializeActiveBlocks()

[Memory.cpp]

Calls:

Vars (global):

ActiveBlock [int, Memory.cpp]

Purpose:

Sets all members of ActiveBlock (NUM_BLOCKS) to zero.

Warnings:

Memory() {Thread}

[Memory.cpp]

Calls:

 PollNeededData() [Poll.cpp]

Vars (global):

Purpose:

 Calls PollNeededData() and sleeps for MEM_POLL ms.

Warnings:

 Put var instead of TRUE in while() to gracefully shut down (do-while?).

MonitorPower() {Thread}

[Power.cpp]

Calls:

 ReadDataDirectly() [Memory.cpp]

Vars (global):

 BatteryFlag [Global.cpp]

Purpose:

 Loops until local var (Continue) becomes FALSE. Calls ReadDataDirectly()
on

 DC01_BATT2. If the value at DC01_BATT2 is below
 MINIMUM_BATTERY_VALUE for MAX_MINUTES_BELOW_THRESHOLD
 (checks every minute), global var BatteryFlag is set to BATTERY_BAD and
the thread
 ends.

Warnings:

 Why battery 2? Is ReadDataDirectly() necessary? Probably need to test!

WarnProc() {Thread}

[WarnLight.cpp]

Calls:

AssertWarn() [Digital.cpp]
DeAssertWarn() [Digital.cpp]

Vars (global):

MissionStatus [int, Declarations.cpp]
LiftMode [int, Declarations.cpp]

Purpose:

Checks to see if global var MissionStatus indicates that the robot is referencing, moving, returning, or if the lift is moving. If so, calls AssertWarn(); if not, calls DeAssertWarn().

Warnings:

Calls DeAssertWarn() a lot, is that okay? AssertWarn() and DeAssertWarn() don't do anything right now!

FillInArrayAgain()

[Array.cpp]

Calls:

Vars (global):

MissionSizeOfArray [int, Global.cpp]
MissionArray [ArrayPntr*, Array.cpp]
MissionNumberOfFiles [int, Global.cpp]
MissionBigInt [int, Global.cpp]
MissionPathList [PathListPntr*, Global.cpp]

Purpose:

Initializes the MissionArray matrix. Fills the MissionArray matrix with MissionPathList array. Sets the identity of the MissionArray matrix.

Warnings:

SolveNxNArray()

[Array.cpp]

Calls:

Vars (global):

MissionSizeOfArray [int, Global.cpp]

MissionArray [ArrayPntr*, Array.cpp]

Purpose:

Calculates cost from x to y (and gives filename).

Warnings:

MissionSizeOfArray x MissionSizeOfArray x MissionSizeOfArray solution!

GetResponse()

[Mesg.cpp]

Calls:

FillInVariable() [Mesg.cpp]

Vars (global):

Purpose:

Tests for ":" message. Eats white space at the end of the message. Calls FillInVariable()

on local var (str). Calculates checksum and adds it to "str".

Warnings:

Who knows if it works!

SetSocketOptions()

[Socket.cpp]

Calls:

MakeSocketEfficient() [Socket.cpp]

Vars (global):

Server [SocketDesc, Declarations.cpp]

Client [SocketDesc, Declarations.cpp]

Purpose:

Calls setsockopt() on either Server or Client global vars (sockets), then calls MakeSocketEfficient() on the said socket.

Warnings:

ReadFromClient()

[ServerUtil.cpp]

Calls:

UpdateReadBuffer() [Socket.cpp]

ExtractMess() [PropMesg.cpp]

ReadSocketLine() [Socket.cpp]

CloseSocket() [Socket.cpp]

Vars (global):

DataReady [int, StartServer.cpp]

Client [SocketDesc[], Declarations.cpp]

Purpose:

Initialize global var DataReady to FALSE. Loop local var "i" for MAX_CLIENTS. Test client (i) for "ACTIVE" and a message present. Call UpdateReadBuffer() and ExtractMess() and continue. Call select() on Client, and then if FD_ISSET() call ReadSocketLine() on it and ExtractMess(). If there is an error call CloseSocket() and set DataReady to TRUE if any clients are active or messages are present.

Warnings:

Global var DataReady should be a BOOL.

ProcessClientCommands()

[ProcessCommand.cpp]

Calls:

Send_Id() [ProcessCommand.cpp]
Read_Block() [ProcessCommand.cpp]
Write_Block() [ProcessCommand.cpp]
Read_Var() [ProcessCommand.cpp]
Read_Special() [ProcessCommand.cpp]
Write_Special() [ProcessCommand.cpp]
Close_Connection() [ProcessCommand.cpp]
Down_Load() [ProcessCommand.cpp]
Dis_Asm() [ProcessCommand.cpp]
Load_Status() [ProcessCommand.cpp]
Com_Status() [ProcessCommand.cpp]
Mission_Status() [ProcessCommand.cpp]
Unknown_Command() [ProcessCommand.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

Loop through MAX_CLIENTS, testing for ACTIVE and for "bad" messages.

Call

proper function (see above) for respective message command.

Warnings:

GetNewClient()

[ServerUtil.cpp]

Calls:

 GetConnection() [Socket.cpp]
 CloseSocket() [Socket.cpp]

Vars (global):

 Server [SocketDesc, Declarations.cpp]
 Client [SocketDesc[], Declarations.cpp]

Purpose:

 The function select() is called on the global var Server. Loops waiting for a client to connect (polling). Calls GetConnection() on global var Client[index]. If there are too many clients, the temp client (TmpClient) is connected then disconnected. "ERROR" is returned if no connection is made

Warnings:

 Timeout is set at 0? Local var "TmpClient" is never initialized before going to

 CloseSocket().

 **Might want to put polling in thread or put in 500msec delay!

WriteToClient()

[ServerUtil.cpp]

Calls:

 CreateMess() [PrepMesg.cpp]
 WriteSocketLine() [Socket.cpp]

Vars (global):

 Client [SocketDesc[], Declarations.cpp]

Purpose:

 Checks all (Client) connections for ACTIVE and writes the message.

Warnings:

Error checking?

GetTimeFromServer()

[Comm.cpp]

Calls:

Connect() [Comm.cpp]
SendMessageMH() [Comm.cpp]

Vars (global):

DatabaseFD [static int, Comm.cpp]
Time [static struct tm, Comm.cpp]

Purpose:

Waits for a connection (poll with a sleep spec by SLEEPRECORDER) for MAX_ATTEMPTS. Asks (the database server?) for the time. Gets time and sets global variable Time.

Warnings:

Only asks once for time (comp with loop for connect).

LostTime()

[Comm.cpp]

Calls:

Vars (global):

Time [static struct tm, Comm.cpp]

Purpose:

Gets system time and compares it with global variable Time. Returns "TRUE" if time is off.

Warnings:

BuildMissionFileName()

[Parser.cpp]

Calls:

Vars (global):

HistoryName [char[], Global.cpp]

OutputHistoryName [char[], Global.cpp]

Purpose:

Build strings for "MissionName" (parameter) and global vars
"HistoryName" and
"OutputHistoryName".

Warnings:

See "defines.h" for #defines! Extensions ".mission" and ".history" are
hard coded
(should be #defines)!

InitIO()

[Scanner.cpp]

Calls:

NextLine() [Scanner.cpp]

Vars (global):

LineNum [int, Scanner.cpp]

Purpose:

Begins parsing MDL file (xxx.mission). Sets LineNum to 1 and calls
NextLine();

Warnings:

Exported var "LineNum" present in DB package. Be wary of complications!

NextToken()

[Scanner.cpp]

Calls:

- Eof() [Scanner.cpp]
- ProcessFilename() [Scanner.cpp]
- ProcessId() [Scanner.cpp]
- ProcessLiteral() [Scanner.cpp]

Vars (global):

- TokenBuffer [char[], Scanner.cpp]
- LineNum [int, Scanner.cpp]

Purpose:

Returns the next token type. Ignores comments, calls ProcessFilename(), ProcessId(), and ProcessLiteral() if required. Returns appropriate token for symbols. Test for end of file.

Warnings:

Could re-implement as table.

GetSiteName()

[Parser.cpp]

Calls:

- Match() [Scanner.cpp]

Vars (global):

- SiteName [char[], Global.cpp]
- TokenBuffer [char[], Scanner.cpp]
- State [static int, Parser.cpp]

Purpose:

Gets site name from MDL file and sets appropriate variables (global SiteName and local State).

Warnings:

GetBuildingName()

[Parser.cpp]

Calls:

Match() [Scanner.cpp]

Vars (global):

BuildingName [char[], Global.cpp]

TokenBuffer [char[], Scanner.cpp]

State [static int, Parser.cpp]

Purpose:

Gets building name from MDL file and sets appropriate variables (global BuildingName and local State).

Warnings:

GetStartNodeName()

[Parser.cpp]

Calls:

Match() [Scanner.cpp]

Vars (global):

StartNode [char[], Global.cpp]

TokenBuffer [char[], Scanner.cpp]

State [static int, Parser.cpp]

Purpose:

Gets StartNode from MDL file and sets appropriate variables (global StartNode and local State).

Warnings:

GetHomeNodeName()

[Parser.cpp]

Calls:

Match() [Scanner.cpp]

Vars (global):

HomeNode [char[], Global.cpp]

TokenBuffer [char[], Scanner.cpp]

State [static int, Parser.cpp]

Purpose:

Gets HomeNode from MDL file and sets appropriate variables (global HomeNode and local State).

Warnings:

GetReferenceAction()

[Parser.cpp]

Calls:

Match() [Scanner.cpp]

Vars (global):

ReferenceActionFrom [char[], Global.cpp]

TokenBuffer [char[], Scanner.cpp]

State [static int, Parser.cpp]

Purpose:

Gets ReferenceActionFrom from MDL file and sets appropriate variables (global ReferenceActionFrom and local State).

Warnings:

StartMission()

[Parser.cpp]

Calls:

Match() [Scanner.cpp]
GetAisleBehavior() [Parser.cpp]

Vars (global):

MissionRootInspectionTable [struct InspectionTable, Global.cpp]
CurrentToken [int, Scanner.cpp]
TokenBuffer [char[], Scanner.cpp]
State [static int, Parser.cpp]

Purpose:

Adds entries to the MissionRootInspectionTable. Calls GetAisleBehavior()
for aisle
actions(?). Sets "GOT_MISSION" flag in State global var.

Warnings:

Status hardwired at "-1" for init?

GetStartTime()

[Parser.cpp]

Calls:

Match() [Scanner.cpp]

Vars (global):

CurrentToken [int, Scanner.cpp]
State [static int, Parser.cpp]
StartYear [int, Global.cpp]
StartMonth [int, Global.cpp]
StartDay [int, Global.cpp]
StartHour [int, Global.cpp]
StartMinute [int, Global.cpp]

Purpose:

Gets start time from MDL file and sets global var "State" indicating
presence of start date

in the "Start*" global variables.

Warnings:

GetOffset()

[Parser.cpp]

Calls:

Match() [Scanner.cpp]

Vars (global):

XcoordinateOffset [float, Global.cpp]

YcoordinateOffset [float, Global.cpp]

TokenBuffer [char[], Scanner.cpp]

State [static int, Parser.cpp]

Purpose:

Gets offset (global vars "XcoordinateOffset" and "YcoordinateOffset") and sets global var "State".

Warnings:

ClosesIO()

[Scanner.cpp]

Calls:

Vars (global):

fp [FILE*, Scanner.cpp]

Purpose:

Closes MDL file.

Warnings:

Beware of DB package fp!

GetHomeNode()

[Initialize.cpp]

Calls:

Vars (global):

HomeNode [char[], Global.cpp]

StartNode [char[], Global.cpp]

Purpose:

Copies global var HomeNode to global var StartNode.

Warnings:

Could check for NULL?

GetActionFile()

[Initialize.cpp]

Calls:

Vars (global):

ActionFilePath [char[], Global.cpp]

SiteName [char[], Global.cpp]

BuildingName [char[], Global.cpp]

Purpose:

Initializes global var "ActionFilePath" with #defines from "defines.h" and global vars

"SiteName" and "BuildingName".

Warnings:

BuildDatabaseFile()

[Parser.cpp]

Calls:

Vars (global):

DatabaseTemplate [char[], Global.cpp]

SiteName [char[], Global.cpp]

BuildingName [char[], Global.cpp]

DatabaseFile [char[], Global.cpp]

Purpose:

Initializes global var "DatabaseTemplate" with #defines from "defines.h" and global vars

"SiteName" and "BuildingName". Takes "/" out of global var "DatabaseFile".

Warnings:

Looks for '/' where now maybe it should look for '\'?

RemoveFiles()

[Parser.cpp]

Calls:

Vars (global):

RobotsName [char[], Global.cpp]

Purpose:

Initializes local var "MissionReport" and local var "MissionHistory" with #defines from

"defines.h" and global var "RobotsName". Then removes the files indicated by local var

"MissionReport" and local var "MissionHistory".

Warnings:

At one time may have also removed the MDL file. File extensions ".report" and

".history" are hard coded.

CreatePathDatabase()

[List.cpp]

Calls:

Vars (global):

MissionRootPathNode [struct PathData, Global.cpp]

SiteName [char[], Global.cpp]

BuildingName [char[], Global.cpp]

DatabaseFile [char[], Global.cpp]

Purpose:

Initializes local var "PathDatabase" with #defines from "defines.h" and global var s "SiteName" and "BuildingName" and "DatabaseFile". Opens file and reads in path database and puts it in a linked list (global var MissionRootPathNode).

Warnings:

File extension, "*.pdb" is hard-wired. No error checking.

BuildNodeTable()

[NodeTable.cpp]

Calls:

AddToNodeTable() [NodeTable.cpp]

Vars (global):

MissionRootPathNode [struct PathData, Global.cpp]

MissionBigInt [int, Global.cpp]

MissionSizeOfArray [int, Global.cpp]

Purpose:

Traverses node list (global var "MissionRootPathNode") calling AddToNodeTable() on each "To" and "From". Global var MissionBigInt stores the cost of the path with the

highest cost, then multiplies by global var "MissionSizeOfArray".

Warnings:

No error checking.

BuildFileTable()

[FileTable.cpp]

Calls:

AddToFileTable() [FileTable.cpp]

MHCopyFile() [Copy.cpp]

Vars (global):

MissionRootPathNode [struct PathData, Global.cpp]

ActionFilePath [char[] , Global.cpp]

Purpose:

Initializes local var "Destination" with #defines from "defines.h". Builds file table (?) by

calling AddToFileTable() on action files in list. Copies files from list (global var

"MissionRootPathNode") from global var "ActionFilePath" directory to "Destination"

directory by calling MHCopyFile().

Warnings:

Error checking?

BuildPathList()

[List.cpp]

Calls:

GetNodeNumber() [NodeTable.cpp]

AddToList() [List.cpp]

Vars (global):

MissionPathList [PathListPntr*, Global.cpp]

MissionSizeOfArray [int, Global.cpp]

MissionRootPathNode [struct PathData, Global.cpp]

Purpose:

Allocates memory and assigns it to global var "MissionPathList". Initializes the array to NULL, and then calls GetNodeNumber() and AddToList(). Builds path matrix.

Warnings:

Where is memory freed?

DeletePathDatabase()

[List.cpp]

Calls:

Vars (global):

MissionRootPathNode [struct PathData, Global.cpp]

Purpose:

Frees memory associated with global var "MissionRootPathNode".

Warnings:

BuildArray()

[Array.cpp]

Calls:

Vars (global):

MissionArray [ArrayPntr*, Array.cpp]

MissionSizeOfArray [int, Global.cpp]

MissionNumberOfFiles [int, Global.cpp]

MissionBigInt [int, Global.cpp]

MissionSizeOfArray [int, Global.cpp]

Purpose:

Allocates memory for an array "MissionSizeOfArray" x "MissionSizeOfArray" (MissionArray). Initializes elements in the array to values NODE_UNKNOWN, MissionNumberOfFiles, MissionBigInt, MissionSizeOfArray.

Warnings:

Allocates a LOT of memory! Where is it set free?

FillInArray()

[Array.cpp]

Calls:

Vars (global):

MissionSizeOfArray [int, Global.cpp]

MissionPathList [PathListPntr*, Global.cpp]

MissionArray [ArrayPntr*, Array.cpp]

Purpose:

Builds global array "MissionArray" from global array "MissionPathList".

Warnings:

MHCopyFile()

[Copy.cpp]

Calls:

Vars (global):

Purpose:

Takes input parameters "WriteDir", "ReadDir", and "File" and copies the "File" from the "ReadDir" to the "WriteDir".

Warnings:

A section is commented out which may not make a difference.

DBOpen()

[Commands.cpp]

Calls:

- AssertNull() [DBLib.h]
- CreateDesc() []
- DBReportError() [utility.cpp]
- FreeDesc() [commands.cpp]
- VerifyAriesDB() [create.cpp]
- ExtractDescription() [manage.cpp]
- CacheRecordTable() [tables.cpp]

Vars (global):

Purpose:

Attempts to open ARIES v2.0 (drum) database file, verifies ("VerifyAriesDB()") that it is said database, calls ExtractDescription() and CacheRecordTable() on it.

Warnings:

DBCreateOnly()

[Commands.cpp]

Calls:

- AssertNull() [DBLib.h]
- CreateDesc() [commands.cpp]
- DBReportError() [utility.cpp]
- FreeDesc() [commands.cpp]
- ParseDescFile() [DBParser.cpp]
- WriteNewHeaders() [create.cpp]
- AssertError() [DBLib.h]
- CacheRecordTable() [tables.cpp]

Vars (global):

Purpose:

Calls CreateDesc() to create a "DBDesc", opens the file "dbfile" (parameter) calls

ParseDescFile() and WriteNewHeaders() and then AssertError() on CacheRecordTable()

Warnings:

DBWriteUserData()

[Commands.cpp]

Calls:

- AssertNull() [DBLib.h]
- DBReportError() [utility.cpp]
- CopyInc() [DBLib.h]
- GotoBlock() [DBLib.h]
- WriteBlock() [DBLib.h]

Vars (global):

Purpose:

Adds a field to the database.

Warnings:

Lots of "sizeof()" calls. Make sure none were missed!

MoveBBAbsolute()

[Lift.cpp]

Calls:

- SetLiftValues() [Lift.cpp]
- MoveLift() [Lift.cpp]

Vars (global):

- LiftMode [int, Declarations.cpp]
- LevelStatus [int, Declarations.cpp]
- DrumStatus [int, Declarations.cpp]

Purpose:

Calls SetLiftValues() then MoveLift() while setting global vars "LiftMode", "LevelStatus", and "DrumStatus".

Warnings:

InitPort()

[Barcode.cpp]

Calls:

Vars (global):

hBarcode [Barcode.cpp]

Purpose:

Initializes comm port (BARCODE_PORT) used for barcode reader.

Warnings:

Test?

SetScannerOperation()

[Barcode.cpp]

Calls:

SendCommand() [Barcode.cpp]

Vars (global):

hBarcode [Barcode.cpp]

Purpose:

Warnings:

Return "ERROR" or "FALSE"? See InitPort() above! Calls to WriteEsc() commented out. Who knows if it works?

GetValue()

[Lift.cpp]

Calls:

 ReadDataDirectly() [Memory.cpp]

Vars (global):

Purpose:

 Reads the value of a variable directly (calls ReadDataDirectly()). If attempt fails after MAX_ATTEMPTS, returns error.

Warnings:

 MAX_ATTEMPTS is set pretty low (2).

Convert2ByteSigned()

[Lift.cpp]

Calls:

Vars (global):

Purpose:

 Converts unsigned int (16-bit value) to signed value.

Warnings:

 Test!

GetFileName()

[FileTable.cpp]

Calls:

Vars (global):

MissionRootFileTable [FileTable.cpp]

Purpose:

Returns filename from filename "number" .

Warnings:

FillInPathName()

[ReadFile.cpp]

Calls:

Vars (global):

CompleteFileName [char[], ReadFile.cpp]

NumberOfInstructions [int, ReadFile.cpp]

Purpose:

Gets "stat" of copy of path program (in temp dir) and sets global var NumberOfInstructions to it (# of instructions) size.

Warnings:

ReadInstructions()

[ReadFile.cpp]

Calls:

Vars (global):

CompleteFileName [char[], ReadFile.cpp]
NumberOfInstructions [int, ReadFile.cpp]
ProgramBuffer [unsigned char[], ReadFile.cpp]
Acceleration [unsigned char[], ReadFile.cpp]

Purpose:

Reads action file and adds its instructions to "ProgramBuffer" (global var).

Warnings:

Global var "ProgramBuffer" is hard-coded "256*6". Where does "Index"
(par) get updated? Which "Acceleration" value is used (should be lowest).

AdjustRelativeOffsets()

[ReadFile.cpp]

Calls:

IsRelativeInstruction() [ReadFile.cpp]

Vars (global):

NumberOfInstructions [int, ReadFile.cpp]
ProgramBuffer [unsigned char[], ReadFile.cpp]

Purpose:

Checks if instruction is "relative" and recalculates offset (links).

Warnings:

"Index <=" (guess not)?

AddHaltToProgram()

[ReadFile.cpp]

Calls:

Vars (global):

ProgramBuffer [unsigned char[], ReadFile.cpp]

Purpose:

Puts a halt (between linked programs?).

Warnings:

Maybe use "BYTES_PER_INSTRUCTION" instead of "6" in ReadFile.cpp!
Opcode for "Halt" is hard-coded at "20". Is this why robot always halts?

WritePathProgram()

[Download.cpp]

Calls:

Communicate() [Download.cpp]
DownloadPath() [Download.cpp]
LoadDriveAndSteer() [Download.cpp]
// InitializeLift() [Lift.cpp]
ClearSignal() [Download.cpp]

Vars (global):

LinkFailureCounter [int, Global.cpp]

Purpose:

Sets K2A variable "DC01_PLOADED" to "0", calls DownloadPath() and LoadDriveAndSteer(). Sets K2A variable "DC01_LASTINS" to "NumOfInstructions - 1" (par), then sets K2A variable "DC01_PLOADED" to "1", and calls InitializeLift(). Sets K2A variable "DC01_MODE" to "AMODE" and initializes global var "LinkFailureCounter" to 0. Calls ClearSignal().

Warnings:

Call to InitializeLift() is commented out! Does InitializeLift() need to be called each time?

MonitorMovement()

[Monitor.cpp]

Calls:

- GetMode() [Monitor.cpp]
- CheckStatus() [Status.cpp]
- GetProgramCounter() [Monitor.cpp]
- AdjustCurrentPosition() [MissionList.cpp]
- GetStatus() [Monitor.cpp]

Vars (global):

- ModeOfRobot [int, Global.cpp]
- ProgramCounter [int, Global.cpp]
- HaltCondition [int, Global.cpp]
- StatusExceptions [StatusException, Global.cpp]
- StatusOfRobot [int, Global.cpp]

Purpose:

Calls GetMode() until global var "ModeOfRobot" is "HMODE". While the global var "ModeOfRobot" is not "HMODE" functions CheckStatus() and GetMode() are called (in between sleeping). Calls GetProgramCounter() and AdjustCurrentPosition(). If global var "HaltCondition" is "DC_IGNORE" (natural halt?) GetStatus() is called and global var "HaltCondition" is set. If AdjustCurrentPosition() does not return NULL the status is reported (as an error)

Warnings:

Made an assumption about the size required for local array "Temp[]".

What about

"GetMode2()"?

UpdateSubgoalStatus()

[Subgoal.cpp]

Calls:

 ReadDataDirectly() [Memory.cpp]

Vars (global):

 CurrentSubgoal [InspectionPntr, Subgoal.cpp]

Purpose:

 Gets X and Y position (DC01_?POS) and updates global var
 "CurrentSubgoal".

Warnings:

 Global var "CurrentSubgoal" is really global var
 "MissionRootInspectionTable".

GetNodeName()

[NodeTable.cpp]

Calls:

Vars (global):

 MissionRootNodeTable [struct NodeData]

Purpose:

 Searches global struct "MissionRootNodeTable" for (input param)
 "NodeNumber" and
 returns a character pointer to the name ("NodeName").

Warnings:

AdjustCurrentPosition()

[MissionList.cpp]

Calls:

Vars (global):

MissionListRoot [struct MissionList, Global.cpp]

CurrentLocation [int, Global.cpp]

Purpose:

Traverses global var "MissionListRoot" until it reaches the end, or the member var

"ProgramCounter" exceeds parameter "PC" . Global var

"CurrentLocation" is updated

and NULL or the current position on "MissionListRoot" is returned

depending on

whether or not the end of the list was reached.

Warnings:

Returns "NULL" on else.

RemoveFromList()

[List.cpp]

Calls:

Vars (global):

MissionPathList [PathListPntr*, Global.cpp]

Purpose:

Removes an element(s) from global var "MissionPathList" .

Warnings:

BlockFromList()

[List.cpp]

Calls:

Vars (global):

MissionPathList [PathListPntr*, Global.cpp]

Purpose:

Traverses global var MissionPathList and sets element(s) described by input parameters to "NO_ACCESS" .

Warnings:

WriteOutMissionReport()

[Report.cpp]

Calls:

WriteOutHeader() [Report.cpp]
WriteOutBehavior() [Report.cpp]
WriteOutStatus() [Report.cpp]
WriteOutPosition() [Report.cpp]

Vars (global):

MissionRootInspectionTable [struct InspectionTable, Global.cpp]
RobotsName [char, Global.cpp]

Purpose:

Composes mission report filename, tries alternate location (temp) on failure. Calls WriteOutHeader() to set up file, then WriteOutBehavior(), WriteOutStatus(), and WriteOutPosition(). Closes the file.

Warnings:

Might want to #define ".report". Why try to open a temp file if main fails?

CloseDrumDatabases()

[Database.cpp]

Calls:

DBCclose() [commands.cpp]

Vars (global):

ReadOnBoardDatabase [int (bool?), Global.cpp]

ReadDesc [DBDesc*, Database.cpp]

WriteOnBoardDatabase [int, Global.cpp]

WriteDesc [DBDesc*]

Purpose:

Tests global vars "ReadOnBoardDatabase" and "WriteOnBoardDatabase" and calls

DBCclose() on global vars "ReadDesc" and "WriteDesc respectively" .

Warnings:

Might want to redirect to EvIMsg.

TransferDrumDatabase()

[Copy.cpp]

Calls:

Vars (global):

RobotsName [char[], Global.cpp]

SiteName [char[], Global.cpp]

BuildingName [char[], Global.cpp]

Purpose:

Builds directory names (drum database)

Warnings:

Might want to #def the extension ".sddb" .

// * TransferImages()**

[Copy.cpp]

Calls:

 IsImageFile() [Copy.cpp]

Vars (global):

 SiteName [char[], Global.cpp]

 BuildingName [char[], Global.cpp]

Purpose:

 Copies image files offboard.

Warnings:

 Not ported!

TransferRecordFile()

[Copy.cpp]

Calls:

 MHCopyFile() [Copy.cpp]

Vars (global):

 RobotsName [char[], Global.cpp]

 HistoryName [char[], Global.cpp]

Purpose:

 Calls MHCopyFile() on "RobotsName.history" file in
 "*_MISSION_REPORT" (see
 #def) directories. Then removes the file referred to by global var
 "HistoryName".

Warnings:

 Might want to #def the extension ".history".

InformOffboardDatabase()

[Comm.cpp]

Calls:

Connect() [Comm.cpp]
SendMessageMH() [Comm.cpp]

Vars (global):

SiteName [char[], Global.cpp]
BuildingName [char[], Global.cpp]
RobotsName [char[], Global.cpp]
DatabaseFD [static int, Comm.cpp]

Purpose:

Waits for successful call to Connect() then builds message passed to SendMessageMH().

Times out after "MAX_ATTEMPTS" tries.

Warnings:

Might want to #def the extension ".sddb".

TransferLogFile()

[Copy.cpp]

Calls:

MoveLogFiles() [Copy.cpp]

Vars (global):

RobotsName [char[], Global.cpp]

Purpose:

Builds dirs and filenames, creates a dir with the robot's name (global var "RobotsName").

Calls MoveLogFiles(), checks for an existing file, then copies the log file.

Warnings:

ClearSupervisorPort()

[TtyUtil.cpp]

Calls:

Vars (global):

 HSuper [HANDLE, Communication.cpp]

Purpose:

 Purges Supervisor (port-through) COM port.

Warnings:

InitializeSemaphores()

[ComUtil.cpp]

Calls:

Vars (global):

 WriteSem [static HANDLE, ComUtil.cpp]

 ReadSem [static HANDLE, ComUtil.cpp]

Purpose:

 Creates and initializes read/write semaphores.

Warnings:

ClearControlPort()

[TtyUtil.cpp]

Calls:

Vars (global):

hCtrl [HANDLE, Communication.cpp]

Purpose:

Purges Control COM port.

Warnings:

HandleSupervisorWriteAndRead()

[TtyUtil.cpp]

Calls:

GetSizeOfResponse() [TtyUtil.cpp]

GetChecksum() [TtyUtil.cpp]

ClearSupervisorPort() [TtyUtil.cpp]

WriteToSupervisorPort() [TtyUtil.cpp]

ReadFromSupervisorPort() [TtyUtil.cpp]

Vars (global):

Purpose:

Receives a message (func param), validates it by calling GetSizeOfResponse() and GetChecksum(). Purges the COM port ("ClearSupervisorPort()"), writes the message ("WriteToSupervisorPort()"), and listens for a response ("ReadFromSupervisorPort()").

Warnings:

"Supervisor" really means "port-through". Timeout on listen ("ReadFromSupervisorPort()") is the only way out if there is no answer.

ReadFromControlPort()

[TtyUtil.cpp]

Calls:

IsMessForMission() [TtyUtil.cpp]
ClearControlPort() [TtyUtil.cpp]

Vars (global):

hCtrl [HANDLE, Communication.cpp]

Purpose:

Reads a byte from Control port to determine what type of message is incoming and reads the data.

Warnings:

Is similar to ReadFromSupervisorPort(). Sure we only read 8 on a ";" message?

WriteToControlPort()

[TtyUtil.cpp]

Calls:

Vars (global):

hCtrl [HANDLE, Communication.cpp]

Purpose:

Writes message to Control COM port.

Warnings:

Very similar to "WriteToSupervisorPort()". No error checking (complete write &etc)!

PollNeededData()

[Poll.cpp]

Calls:

RequestBlock() []
PollExtraData() []

Vars (global):

BLOCK0 [enum, Blocks.h]

Purpose:

Calls RequestBlock() on global var "BLOCK0" (default), then calls PollExtraData() to get necessary data that remains.

Warnings:

ReadDataDirectly()

[Memory.cpp]

Calls:

IntegerToHex() [Memory.cpp]
WriteSuper() [ComUtil.cpp]
ReadSuper() [ComUtil.cpp]
ConvertData() [Memory.cpp]

Vars (global):

ReadDirectSem [Memory.cpp]
Variables [Variables.cpp]

Purpose:

Waits for "ReadDirectSem" Semaphore (global), builds a message using IntegerToHex() and the global var "Variables" (generated by the "Listings" application).

Calls

WriteSuper() with the message and ReadSuper() for the response. Calls ConvertData() on the response.

Warnings:

Local var "Variable" (param) vs global var "Variables"! Several
"IntegerToHex()" functions (3).

//AssertWarn()

[Digital.cpp]

Calls:

Vars (global):

Purpose:

Commented out; does nothing.

Warnings:

//DeAssertWarn()

[Digital.cpp]

Calls:

Vars (global):

Purpose:

Commented out; does nothing.

Warnings:

FillInVariable()

[Mesg.cpp]

Calls:

Vars (global):

HostVariable [int[]], Global.cpp]

Purpose:

Gets a value for global var "HostVariable" from input param message ("mess").

Warnings:

MakeSocketEfficient()

[Socket.cpp]

Calls:

Vars (global):

Server [SocketDesc, Declarations.cpp]

Client [SocketDesc[], Declarations.cpp]

Purpose:

Increases the size of the send/receive buffers on the (indexed) socket.

Warnings:

UpdateReadBuffer()

[Socket.cpp]

Calls:

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

Compares message buffer with read buffer and copies any remaining data.

Checks for a

complete message and flags if not done when function ends.

Warnings:

Lots of pointer action.

ExtractMess()

[PropMesg.cpp]

Calls:

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

Fills the "To", "From", "Command", "Size", and "Status" fields in the global var "Client".

Checks the "Command" field for "DOWN_LOAD" or "DIS_ASM".

Warnings:

"i" vs "j".

ReadSocketLine()

[Socket.cpp]

Calls:

UpdateReadBuffer() [Socket.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

Checks the socket receive buffer and puts more data in if there is room.

Calls

UpdateReadBuffer() when done to sync message/read buffers.

Warnings:

CloseSocket()

[Socket.cpp]

Calls:

Vars (global):

Purpose:

Reinitializes the SocketDesc structure (*Sock) passed to it (param).

Warnings:

"UNKNOWN2" vs "UNKNOWN". Modified calls to "shutdown()" and "close()" (now "closesocket()"), NEED TO TEST!!!

Send_Id()

[ProcessCommand.cpp]

Calls:

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function will process the Id of the sender if the status flag is set appropriately. In order to communicate with the server a process must send its Id so the server knows who it is Talking to.

Warnings:

Read_Block()

[ProcessCommand.cpp]

Calls:

GetComStatus() [Memory.cpp]

GetBlockData() [Memory.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function retrieves a block of data for the caller. The block number being requested is sent as the first integer in the data array. Comm is tested with GetComStatus(), global var "Client" is reset on failure. The data is requested with a call to GetBlockData().

Warnings:

Write_Block()

[ProcessCommand.cpp]

Calls:

WriteVariableData() [Modem.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function allows a client to write a block of data to the robot. The client will receive the status of the write in the return message.

Warnings:

Read_Var()

[ProcessCommand.cpp]

Calls:

GetComStatus() [Memory.cpp]

GetVariableData() [Memory.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function reads a particular variable for the calling process. The variable that is being requested is identified by the first integer in the data array.

Warnings:

Read_Special()

[ProcessCommand.cpp]

Calls:

ReadSpecialMessage() [Modem.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function allows the caller to specify an area of memory to be examined in one of the computers on board the robot. This requires that the caller send the size of the variable, the address and the computer number of the variable. This information is sent in the first three integers of the data field.

Warnings:

Write_Special()

[ProcessCommand.cpp]

Calls:

WriteSpecialMessage() [Modem.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function allows a caller to write to an area of memory that is not specified by a variable name. The caller must give the size of the variable in bytes, the address and the computer number of the variable. This information is sent in the first three integers of the data array.

Warnings:

Close_Connection()

[ProcessCommand.cpp]

Calls:

CreateMess() [PrepMesg.cpp]
WriteSocketLine() [Socket.cpp]
CloseSocket() [Socket.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function allows a client to close its socket connection gracefully.

Although this is

not necessary it is recommended so that the system will can shutdown the socket and recover more quickly.

Warnings:

Down_Load()

[ProcessCommand.cpp]

Calls:

Com_WritePathProgram() [Modem.cpp]

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function is called by the control program. This function allows the client to

download a path program to the DC-01 for execution. The success of this operation is

returned in the reply message.

Warnings:

Dis_Asm()

[ProcessCommand.cpp]

Calls:

 GetInstructionBlock() [Modem.cpp]

Vars (global):

 Client [SocketDesc[], Declarations.cpp]

Purpose:

 This function is called by the control program. It allows the client to extract the last path program that was downloaded. The client sends the number of instructions in the first integer in the data field.

Warnings:

Load_Status()

[ProcessCommand.cpp]

Calls:

 GetLoadStatus() [Memory.cpp]

Vars (global):

 Client [SocketDesc[], Declarations.cpp]

Purpose:

 This function is generally called by the control program. It gives the caller an idea of how heavily loaded the communication system is (ie how many blocks of data are actively being polled from the robot).

Warnings:

Com_Status()

[ProcessCommand.cpp]

Calls:

 GetComStatus() [Memory.cpp]

Vars (global):

 Client [SocketDesc[], Declarations.cpp]

Purpose:

 This function is generally called by the control program to determine if communications is intact.

Warnings:

Mission_Status()

[ProcessCommand.cpp]

Calls:

Vars (global):

 Client [SocketDesc[], Declarations.cpp]

 MissionStatus [int, Declarations.cpp]

 BarcodeStatus [int, Declarations.cpp]

 VisionStatus [int, Declarations.cpp]

 PanStatus [int, Declarations.cpp]

 LevelStatus [int, Declarations.cpp]

 DrumStatus [int, Declarations.cpp]

 LiftMode [int, Declarations.cpp]

 DC01Status [int, Declarations.cpp]

 DC01Mode [int, Declarations.cpp]

Purpose:

 Puts mission status variables (global) into Client (global) for transmission.

Warnings:

Unknown_Command()

[ProcessCommand.cpp]

Calls:

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function sets the command in the socket to UNKNOWN. This is used to determine when a valid message is present. If the command is set to UNKNOWN then there is not a message present.

Warnings:

GetConnection()

[Socket.cpp]

Calls:

SetSocketOptions() [Socket.cpp]

Vars (global):

Server [SocketDesc, Declarations.cpp]

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function checks to see if any client has connected to the server.

Warnings:

Returns both "MH_ERROR" and "TRUE/FALSE". Lots of "FD_*)" calls.

No

timeouts. "i" vs "j". Call to "select()" uses "PhoneNo + 1"??

CreateMess()

[PrepMesg.cpp]

Calls:

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

This function takes information passed to it in the socket structure and puts it into a string format that can be sent over the ethernet. The information used to build the appropriate message is as follows TTFCCSSSSXXDDDDDDDD....\r\n. TT is a two digit hex number determining who the message is to. FF is a two digit hex number determining who the message is from. CC is a two digit hex number determining what the command is. SSSS is a four digit hex number determining the number of integers that follow. XX is a two digit hex number determining the status of the message being sent. DDDDDDDD is an 8 digit hex number representing an integer. This is the data part of the message that might be needed.

Warnings:

"i", "j", and "k". '0' should be '\0'?? "Size & 0XFFF" should be "Size & 0XFFFF"??

WriteSocketLine()

[Socket.cpp]

Calls:

Vars (global):

Client [SocketDesc[], Declarations.cpp]

Purpose:

The purpose of this function is to write out a message that is in the write message buffer to the socket descriptor.

Warnings:

Connect()

[Comm.cpp]

Calls:

Vars (global):

DatabaseFD [static int, Comm.cpp]

Purpose:

Sets up socket (DATABASE_MACHINE), calls setsockopt() and connect().

Warnings:

See similarity in CreateSocket() (Socket.cpp). Linger?

SendMessageMH()

[Comm.cpp]

Calls:

GotMessage() [Comm.cpp]

Vars (global):

DatabaseFD [static int, Comm.cpp]

Purpose:

Sends message to database computer (TimeSever) and listens for a reply for 30 seconds (?).

Warnings:

Timeout (Timeout.tv_sec) is set to 30 seconds!? Could loop forever if no data is ever sent (need another timeout?) same for read loop?

NextLine()

[Scanner.cpp]

Calls:

Vars (global):

- fp [FILE*, Scanner.cpp]
- ChPos [int, Scanner.cpp]
- ch [static int, Scanner.cpp]
- TabOffset [int, Scanner.cpp]
- line [char[], Scanner.cpp]

Purpose:

Puts the next line (from MDL file-global var "fp") in a buffer (global var "line").

Warnings:

There are "fp", "ChPos", and "ch" member variables in the database package! DB uses same general scanner.

Eof()

[Scanner.cpp]

Calls:

Vars (global):

- fp [FILE*, Scanner.cpp]

Purpose:

Does EOF (feof()) test on global var "fp".

Warnings:

ProcessFilename()

[Scanner.cpp]

Calls:

 getch() [Scanner.cpp]

Vars (global):

 TokenBuffer [char[], Scanner.cpp]

Purpose:

 Copies chars in between quotes (from call to "getch()") to global var "TokenBuffer".

Warnings:

 Why pass in "ch2"?

ProcessId()

[Scanner.cpp]

Calls:

 inspect() [Scanner.cpp]

 advance() [Scanner.cpp]

 CheckReserved() [Scanner.cpp]

Vars (global):

 TokenBuffer [char[], Scanner.cpp]

Purpose:

 Checks the validity of an identifier.

Warnings:

Dear sir:

ProcessLiteral()

[Scanner.cpp]

Calls:

- getch() [Scanner.cpp]
- inspect() [Scanner.cpp]
- advance() [Scanner.cpp]

Vars (global):

- TokenBuffer [char[], Scanner.cpp]

Purpose:

Checks the TokenBuffer for its validity as a literal. Looks like it will handle unary minus, decimals, and exponential notation.

Warnings:

- Unary "+"? Test?

Match()

[Scanner.cpp]

Calls:

- NextToken() [Scanner.cpp]

Vars (global):

- CurrentToken [int, Scanner.cpp]
- LineNum [int, Scanner.cpp]
- TokenBuffer [char[], Scanner.cpp]
- ErrorTString [static char[], Scanner.cpp]

Purpose:

Matches global var "CurrentToken" with input param "token" and handles errors.

Warnings:

GetAisleBehavior()

[Parser.cpp]

Calls:

Match() [Scanner.cpp]

Vars (global):

CurrentToken [int, Scanner.cpp]

Purpose:

Used when parsing the MDL file to determine if an aisle is "VISUAL" or "OTHER" .

Warnings:

AddToNodeTable()

[NodeTable.cpp]

Calls:

Vars (global):

MissionRootNodeTable [struct NodeData, NodeTable.cpp]

MissionSizeOfArray [int, Global.cpp]

Purpose:

Checks global struct "MissionRootNodeTable" for node and adds (increments global var "MissionSizeOfArray") it if not found.

Warnings:

Memory allocated; set free? Similar to AddToFileTable().

AddToFileTable()

[FileTable.cpp]

Calls:

Vars (global):

MissionRootFileTable [struct FileData, FileTable.cpp]

MissionNumberOfFiles [int, Global.cpp]

Purpose:

Checks global struct "MissionRootFileTable" for node and adds (increments global var "MissionNumberOfFiles") it if not found.

Warnings:

Memory allocated; set free? Similar to AddToNodeTable().

AddToList()

[List.cpp]

Calls:

GetNodeNumber() [NodeTable.cpp]

GetFileNameNumber() [FileTable.cpp]

Vars (global):

MissionPathList [PathListPntr*, Global.cpp]

Purpose:

Allocates memory for an element (list) on global var "MissionPathList" if needed. Gets node numbers (to, from; GetNodeNumber()) and filename (number; GetFileNameNumber()).

Warnings:

Allocated memory set free (global array "MissionPathList")?

SetLiftValues()

[Lift.cpp]

Calls:

Communicate() [Download.cpp]

Vars (global):

Purpose:

Bounds tests input params ("Level" and "Size"). Queries "CPS_DRUMSIZ" and "CPS_DRUMLEV" with calls to Communicate().

Warnings:

Still handle 110's?

MoveLift()

[Lift.cpp]

Calls:

CheckIfLiftWillMove() [Lift.cpp]

SetLiftMode() [Lift.cpp]

GetLiftStatus() [Lift.cpp]

GetLiftMode() [Lift.cpp]

RecordLiftStatus() [Lift.cpp]

Vars (global):

Purpose:

Calls CheckIfLiftWillMove(), then SetLiftMode() ("CPS_PAUTO"). Sleeps for 5 seconds

(?), then loops looking for a status of "CPS_PBUSY" or "CPS_OPERLIM" while in

"CPS_PAUTO" mode. Calls GetLiftStatus() and GetLiftMode() and sleeps for 0.5

seconds. Gets the Status and calls RecordLiftStatus().

Warnings:

"MoveLift2()" ? What does "CheckIfLiftWillMove()" return? Five (and 0.5) second sleep?!

SendCommand()

[Barcode.cpp]

Calls:

 CheckResponse() [Barcode.cpp]

Vars (global):

 hBarcode [HANDLE, Barcode.cpp]

Purpose:

 Builds message to barcode (<ESC><STX><XXXX><CR><ESC>; where <XXXX> is the command). Calls CheckResponse() for results.

Warnings:

 "strlen()" vs 8?

IsRelativeInstruction()

[ReadFile.cpp]

Calls:

Vars (global):

Purpose:

 Tests input param "Instruction" to see if it is a relative instruction ("JUMP", "JUMP_GREATER", "JUMP_LESS", "JUMP_EQUAL", "CALL", "CALL_GREATER", "CALL_LESS", "CALL_EQUAL", "JUMP_NOT_EQUAL", "CALL_NOT_EQUAL").

Warnings:

 Is this all of them?

Communicate()

[Download.cpp]

Calls:

WriteDataDirectly() [Memory.cpp]

Vars (global):

Purpose:

Calls WriteDataDirectly() on input params "Variable" and "Value". Tries "MAX_ATTEMPTS" (2) times.

Warnings:

"MAX_ATTEMPTS" only 2?

DownloadPath()

[Download.cpp]

Calls:

BuildPathMessage() [MesgUtil.cpp]

WriteSuper() [ComUtil.cpp]

ReadSuper() [ComUtil.cpp]

Vars (global):

SuperComLink [HANDLE, Declarations.cpp]

Purpose:

This function takes the instructions from the message buffer and downloads them to the

K2A. Breaks the program in to X 30 char blocks. Calls BuildPathMessage() to build

message from buffer (input param). Xfers program with calls to WriteSuper() and

ReadSuper() (for response).

Warnings:

Looks like "0x2409" is hard coded! May want to replace "SuperComLink" with a #def.

See near duplicate (DownloadPath()) in Modem.cpp.

LoadDriveAndSteer()

[Download.cpp]

Calls:

- IntegerToHex() [MesgUtil.cpp]
- Checksum() []
- WriteSuper() [ComUtil.cpp]
- ReadSuper() [ComUtil.cpp]

Vars (global):

- SuperComLink [HANDLE, Declarations.cpp]

Purpose:

Builds a message to set Drive and Steer. Calls IntegerToHex() and Checksum()

Warnings:

May want to #def ":042A0301". May want to replace "SuperComLink" with a #def.

TABLE OF CONTENTS

| | |
|--|----|
| INTRODUCTION | 2 |
| PASM: THE VIRTUAL PATH LANGUAGE ASSEMBLER..... | 4 |
| ENHANCEMENTS | 11 |
| PSU: THE PASM SUPPORT UTILITY | 20 |
| RESULTS | 22 |
| CONCLUSION | 24 |
| APPENDIX A: THE REFERENCE PROGRAM..... | 26 |
| BIBLIOGRAPHY..... | 27 |

INTRODUCTION

Since the initial offering of Cybermotion* mobile robots to the research community in 1984, it has been necessary to provide a means of introducing a path program to the robot for execution. These path programs tell the robot where to go, what to do, and what to expect along the way. The path programs are compositions of the Virtual Path Language (VPL). The Virtual Path Language is a collection of simple instructions that are native to the robot and very similar to machine code instructions. Writing path programs in Virtual Path Language is very similar to writing programs in assembly language.



Figure 1. ARIES

For the robot to navigate confidently, even in relatively small areas, quite a few path programs are required. Small programs can be manually entered directly into the robot. However, this approach becomes impractical as the number and size of path programs increases.

Now that Cybermotion mobile robots patrol the hallways of large office buildings providing security and environmental monitoring (Figure 2), the need for management of the literally thousands of path programs is critical. Until the availability of tools such as PathCAD, which would automatically generate the path programs, simplification of the path programming task comes through the use of the path language assembler, or PASM.

The initial PASM source code came from the University of South Carolina's implementation of PASM for the ARIES (Autonomous Robotic Inspection Experimental System, Figure 3) hazardous waste inspection system (Byrd, 1996). A UNIX version of PASM was needed to be integrated with "Site Manager" (Figure 4), the operator control software, which ran on a Silicon Graphics workstation.

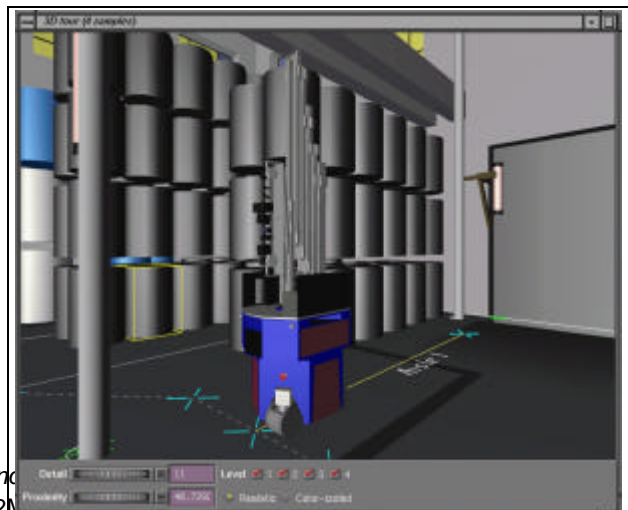
* Cybermotion, Inc., Salem, VA. World-Wide Web: www.cybermotion.com

Site Manager is functionally analogous to Cybermotion's "Dispatcher" program (Figure 5). Dispatcher is the high-level user interface to the robot. Real time feedback and interactions with the robot are possible, as well as diagnostics and path programming. Because PASM was specifically tailored for the ARIES project, it lacked certain features that Cybermotion's PASM provided for Dispatcher. On the other hand, PASM provided a rich set of debug tools, advanced arithmetic expression evaluation, a number of listing file options, and a structure conducive to the modification and addition of new instructions via the PASM Support Utility (PSU).

The productization phase of the ARIES contract specifies that portions of the software run on Windows NT platforms, although it would be commercially desirable for the entire code base to be ported. In anticipation of this, coupled with the desire to provide Cybermotion with a next-generation replacement for their current implementation of PASM, the following milestones had to be met.

The code had to be ported from the UNIX platform to the MS-DOS, Windows, and Windows NT platforms (both sixteen and thirty-two bit). Integral to this process would be the migration of the source code from C to C++ in order to leverage the Microsoft Foundation Classes (MFC) as a future expansion path. In terms of providing the functionality of its commercial compatriot, PASM required the addition of concurrent process compilation and disassembly capabilities. This also necessitated the upgrading of the PASM Support Utility to provide support for concurrent processes. Wildcard operation was identified as a user requirement, but this could only be accomplished through the restructuring of PASM to support operation over multiple files. Finally, an extensive automated test suite was needed to ensure compilation integrity and to speed future modifications and additions.

This document is organized to provide suitable background information on PASM and its mission in order to emphasize the relevance of the enhancements that have been made. Succinct descriptions of the major enhancements and additions to PASM are given with examples (where



appropriate). A brief, but necessary digression is made concerning the PASM Support Utility (or PSU) and its impact upon the growth and expandability of PASM. Results of extensive testing are given which showcase PASM against its commercial counterpart, and concluding remarks are made with regard to the implications of the enhancements and functionalities of PASM. Finally, information that is pertinent, but not critical to the body of this document, such as the actual data from the test suite, source code listings, and excerpts from the Virtual Path Language Reference (Cybermotion, 1995) has been relegated to the Appendices.

PASM: THE VIRTUAL PATH LANGUAGE ASSEMBLER

The native control language (or Virtual Path Language) of the Cybermotion mobile robotic platform is composed of a number of opcodes, or instructions, which can have up to three parameters. The opcodes range from the very simple TURN and RUN instructions, to the more complex DODRUM instruction. The TURN and RUN instructions do just as their name implies, while the DODRUM instruction causes the robot to locate and inspect a drum. There are branch and loop opcodes as well as opcodes that may only be legal within certain constraints.

PASM is basically an assembler. PASM is used to resolve variable names, labels, and mnemonics with their numeric values while also verifying adherence to any parameter limits or rules for opcode usage. To best exemplify PASM's utility, a path programming example is appropriate.

A simple, but common scenario is depicted in Figure 7. The task is to command the robot to travel from the place designated as "DOCKPOS," through the aisle of drums, to the place designated as "F." These "places," or nodes, are coordinates on the robot's world map. The robot is oriented and positioned on this world map by "referencing."

Referencing the robot can occur in several different ways. The method that is used in this situation is via docking beacon. The docking beacon is a structured light source that the robot finds and uses to bring itself to a known position and orientation. The "DOCK" instruction is used to inform the robot that it is to search for, and use, a particular docking beacon.

Once in the known position and orientation, the robot must be told where it is and in which direction it is facing with regard to the world map. This is done with the "SETXY" instruction and the "MEANAZ" instruction, respectively. At this point, the robot may be sent to any X, Y coordinates on the world map.

Generally, it is desirable to have a greater number of simpler path programs than to have fewer, more complicated, path programs. This gives the operator greater flexibility in dispatching the robot to particular nodes. However, in this case, only two path programs and one reference program are needed.

Similar to header files in computer programs, positions and constants of interest are kept in separate "definition" files. Such a definition file for the area in Figure 9 is given in Table 1.

The default unit of length for Cybermotion mobile robots is one one-hundredths of a foot. The keywords "DEFC" and "DEFP" stand for "define constant" and "define position," respectively. The position and constant definitions in the definition file are used throughout the following path programs. This substitution of intuitive labels for numbers and coordinates is one of the key advantages to using an assembler.

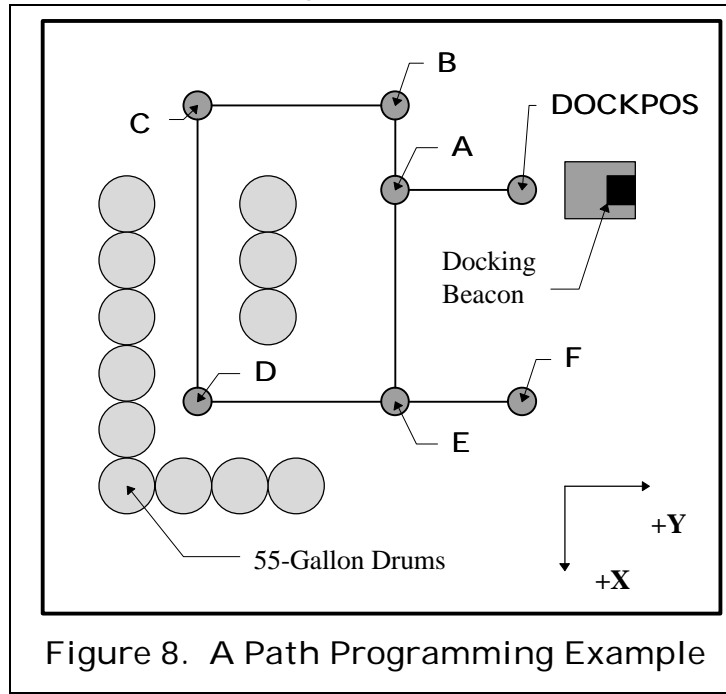


Figure 8. A Path Programming Example

| | | |
|------|---------|-------------------|
| DEFC | MED | 150 |
| DEFC | DOCK_Y | 0 |
| DEFC | DOCK_X | 0 |
| DEFC | COL1_Y | -400 |
| DEFC | ROW1_X | -300 |
| DEFC | COL2_Y | -1000 |
| DEFC | ROW2_X | 800 |
| DEFP | DOCKPOS | DOCK_X, DOCK_Y |
| DEFP | A | DOCK_X, COL1_Y |
| DEFP | B | ROW1_X, COL1_Y |
| DEFP | C | ROW1_X, COL2_Y |
| DEFP | D | ROW2_X, COL2_Y |
| DEFP | E | ROW2_X, COL1_Y |

| | | |
|------|---|-------------------|
| DEFP | F | ROW2_X, DOCK_Y |
|------|---|-------------------|

Table 2. A Definition File, "TEST.DEF"

The reference program is the means by which the robot is positioned upon the world map. The "DOCK" instruction is used such that the robot will be positioned at the node, "DOCKPOS," facing the docking beacon. In this orientation, the positive X and Y directions are as shown in Figure 10. From this frame of reference, the coordinates in the definition file (Table 3) were obtained by careful measurement. Information from this definition file, as well as several other definition files provided by Cybermotion are used in the reference program "DOCKREF.SGV" (Table 4).

This reference program is compiled with PASM. This may be done from within Dispatcher by typing "Makeact". The resulting action file (or compiled path program, "*.ACT") is fairly lengthy and complex due to the initialization code found in the other definition files. For the sake of brevity, the compiled reference program is assumed to be functional and correctly compiled (to see the disassembled action file, please refer to Table 5 in Appendix C). The individual path programs, however, merit closer inspection.

| | |
|---------|------------------|
| INCLUDE | Defaults. def |
| INCLUDE | Local.def |
| INCLUDE | Global.de f |
| INCLUDE | Presets.d ef |
| INCLUDE | Test.def |
| DOCK | 1, 300, 100 |
| MEANAZ | 0 |
| SETXY | DOCKPO S |
| UNDOCK | 1, 0 |

Table 6. A Reference Program, "DOCKREF.SGV"

When writing path programs, it is mandatory that each path from one node to another have a corresponding return path between the same two nodes. The two programs need only have common beginning and ending nodes and do not have to follow the same routes. For this example problem, a path will be created that runs from node "DOCKPOS" to node "F" so that it traverses nodes "A," "B," "C," "D," and "E." The returning path from "F" to "DOCKPOS" will traverse "E" and "A." This will give us the ability to navigate the drum aisle and then return to the docking beacon.

The basic path program from "DOCKPOS" to "F" might resemble that shown in Table 7. When PASM is used to compile this path program an action file ("*.ACT") is created. This action file is ready to be downloaded to the robot and executed. The entire action file, as it would appear on disk, or within the robot, is given in Table 8. The data are in hexadecimal.

| | |
|---------|------------|
| INCLUDE | Test.def |
| RUNON | MED, A |
| RUNON | MED, B |
| RUNON | MED, C |
| AVOID | 1, 50, 50 |
| RUNON | MED, D |
| AVOID | 1, 100, 50 |
| RUNON | MED, E |
| RUNON | MED, F |

Table 9. A Path Program, "DOCKPOS_F.SGV"

The data shown in Table 10 are in low-high byte format (remember, this code is downloaded to a Z80). The two sixteen bit words appended to the end of the file are the drive and steer accelerations. A disassembly ("pasm -a") of the action file is given in Table 11 for comparison with the path program in Table 12.

| | |
|-----|---|
| 10: | 29 00 96 00 00 00 70 FE 29 00 96 00 D4 FE 70 FE |
| 20: | 29 00 96 00 D4 FE 18 FC 20 00 01 00 32 00 32 00 |
| 30: | 29 00 96 00 20 03 18 FC 20 00 01 00 64 00 32 00 |
| 40: | 29 00 96 00 20 03 70 FC 29 00 96 00 20 03 00 00 |
| 50: | 04 00 0A 00 |

Table 13. An Action File, "DOCKPOS_F.ACT"

To complete this example, the path program from node "F" to node "DOCKPOS" must be written (Table 14). The only requirement for this path program is that it begins at "F" and ends at "DOCKPOS." There is no requirement that states that a path must retrace its previous route, so this path program takes a shortcut back to "DOCKPOS" avoiding the drum aisle.

| | | | |
|---|-----------|-----------|-----------|
| Virtual Path Language Assembler v1.50 | | | |
| Disassembler mode ... disassembling [dockpo~1.act] | | | |
| File contains [8] instructions, [4] additional bytes | | | |
| Command: | s: | x: | y: |
| runon (41) | 150 | 0 | -400 |
| runon (41) | 150 | -300 | -400 |
| runon (41) | 150 | -300 | -1000 |
| avoid (32) | 1 | 50 | 50 |
| runon (41) | 150 | 800 | -1000 |
| avoid (32) | 1 | 100 | 50 |
| runon (41) | 150 | 800 | -400 |
| runon (41) | 150 | 800 | 0 |
| Drive Acceleration: | [4] | | |
| Steer Acceleration: | [10] | | |

Table 15. Disassembly of "DOCKPOS_F.ACT"

Now the robot can be sent through the drum aisle to the node "F," and then the robot can return to the node "DOCKPOS" by going around the drum aisle. Typing the commands "Send F" or "Send DOCKPOS" in the Dispatcher command area will dispatch the robot to the proper node (assuming that these same names have been entered into the action database).

This very simple example was given to illustrate the properties and utility of PASM in the generic sense. Appendix A contains data from the compilation of many path programs which can be used as a watermark of functionality. However, that data is in comparison form, and not rendered in the detail found in this example. A further example that deals with

concurrent processes will be used to both give insight into the inner workings of PASM and emphasize the portions of PASM that have been modified and enhanced.

| | |
|---------|-----------------|
| INCLUDE | TEST.DEF |
| RUNON | MED, E |
| RUNON | MED, A |
| RUNON | MED, DOCKPOS |

Table 16. A Path Program, "F_DOCKPOS.SGV"

ENHANCEMENTS

- **VISUAL C++**

The original source code for PASM was written in C. In order to take advantage of the utility provided by Microsoft development tools and the expansion path afforded by C++, the code was tweaked to compile under Microsoft Visual C++ version 1.52c. This version of Visual C++ was the last sixteen-bit compiler for Windows and MS-DOS that Microsoft produced. A sixteen-bit version of PASM was maintained to ensure backward compatibility with Cybermotion's legacy version of PASM. However, once fit for compilation on Visual C++ 1.52c, the code is readily moved to Microsoft's current thirty-two bit compiler, Visual C++ 4.2.

At this point the code can be expanded to leverage Win32 API calls and the Microsoft Foundation Classes (MFC). MFC greatly simplifies the development of user-friendly, intuitive user interfaces while also providing classes that strengthen and simplify the code base.

For example, many development suites incorporate what is termed an "integrated development environment" (IDE). In its simplest implementation, the IDE is a text editor tailored to a particular programming environment with hooks into the compiler that automate and simplify its use. The creation of such an IDE for PASM will be greatly simplified by the use of MFC.

- **WILDCARD OPERATION**

One of the most fundamental user features that was lacking in the initial version of PASM was the ability to undertake batch compilations by invoking wildcards in the program names. In other words, each path program had to be individually compiled, and each filename had to be individually entered on the command line. Wildcard support gives the user the option to compile every path program in a given subdirectory by typing "*.SGV" on PASM's command line.

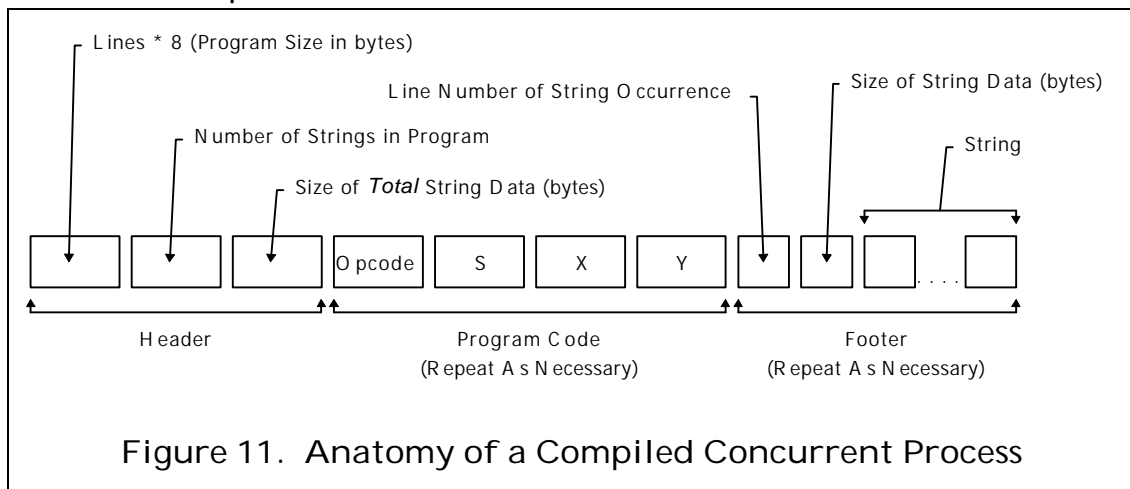
Adding this functionality resulted in the necessary reconstruction of the fundamental structure of the code. Whereas before, one execution of

PASM resulted in one compiled path program, now one execution of PASM had to compile many path programs. Indeed, some commercial installations of Cybermotion mobile security robots employ thousands of path programs. The original PASM code operated under the assumption that it would only execute once. No resource de-allocation or state resets were necessary.

Although greater detail has been relegated to Appendix B, the addition of the wildcard capability itself was trivial. Microsoft provides a library called "setargv.obj" that, when linked with a program, will expand all wildcards on the command line. This is a Microsoft specific extension and is not portable. The difficult part was modifying the structure of PASM so that it would free the appropriate structures and resources then reinitialize the appropriate variables.

• **CONCURRENT PROCESSES**

The most critical addition to PASM in terms of providing the functionality of Cybermotion's commercial version, was that of concurrent processes. The main difference between a compiled concurrent process and an action file, besides the non-Virtual Path Language extensions, is the ability to handle command strings. This necessitates a slightly different form for the compiled code.



As can be seen in Figure 12, the compiled concurrent process ("*.CCP") is comprised of three main parts. The header contains the size (in bytes) of the executable (or the number of instructions times eight), the total number of strings that occur in the program, and the total size (in bytes) that is occupied by the string data. This header is six bytes, or three sixteen-bit words. The body, or program code, is merely composed of the opcodes with their respective S, X, and Y parameters. Concurrent process opcodes begin at 1000 as opposed to Virtual Path Language opcodes which begin at 0, so each program instruction, along with its three parameters,

takes eight bytes, or four sixteen-bit words. The footer, or string data, is variable in size depending upon the usage of strings within the concurrent process. Each string is organized as follows: A byte value for the line number of the string's occurrence, a byte value for the size of the string itself, and then the bytes that compose the string.

A concurrent process has a vocabulary which is partially a subset of the Virtual Path Language with some Dispatcher specific commands and some Cybermotion General Peripheral Interface (GPI) specific commands. The syntax is distinctive with all commands beginning with a period (Table 17).

| | | | |
|---------|---------|----------|---------|
| .READW | .READB | .WRITEW | .WRITEB |
| .JUMP | .JUMP= | .JUMP> | .JUMP< |
| .JUMP!= | .DOCKIN | .DOCKOUT | .WAIT |
| .SOUND | .LOCK | .UNLOCK | .SEND |
| .LOG | .PATROL | .MAP | .CMND |
| .KILLCP | .GPIN | .GPOUT | .GPSET |
| .GPCLR | | | |

Table 18. Concurrent Process Vocabulary

Concurrent processes are interpreted by Dispatcher in the same manner as user commands. Indeed, the utility of concurrent processes is to automate routine Dispatcher tasks in much the same way as batch files automate often used MS-DOS commands. As such, concurrent processes possess several qualities that path programs do not, the most significant being the ability to contain strings. In addition, concurrent processes are the means by which the GPI is controlled. Through the use of the GPI, the robot can control external devices, such as elevators.

To continue with the example centered around Figure 13, a brief concurrent process will be composed that causes the robot to negotiate the drum aisle, return to the docking beacon, and then repeat the process indefinitely. This would normally require the operator to continually type in the commands "send F" and "send DOCKPOS."

Through the use of a concurrent process, the operator interaction can be automated where the operator needs only to invoke the concurrent process (by entering its name) and terminate it (using the Dispatcher commands "endcp" or "killcp"). The concurrent process itself is fairly simple, but it will demonstrate the unique features of concurrent processes. The source for the concurrent process, "LOOP.CP," is given in Table 19.

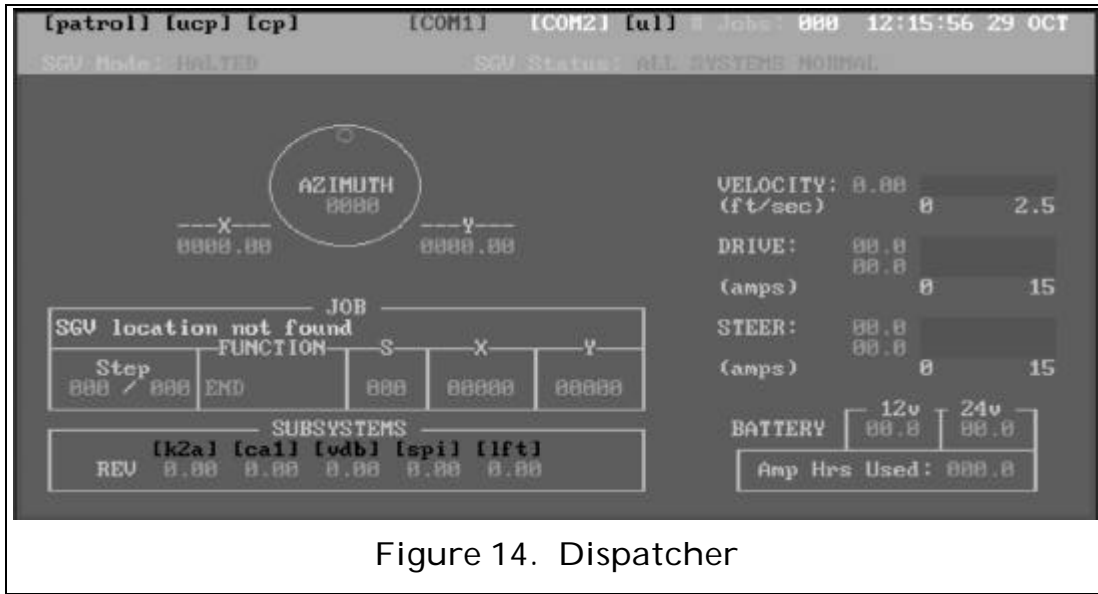


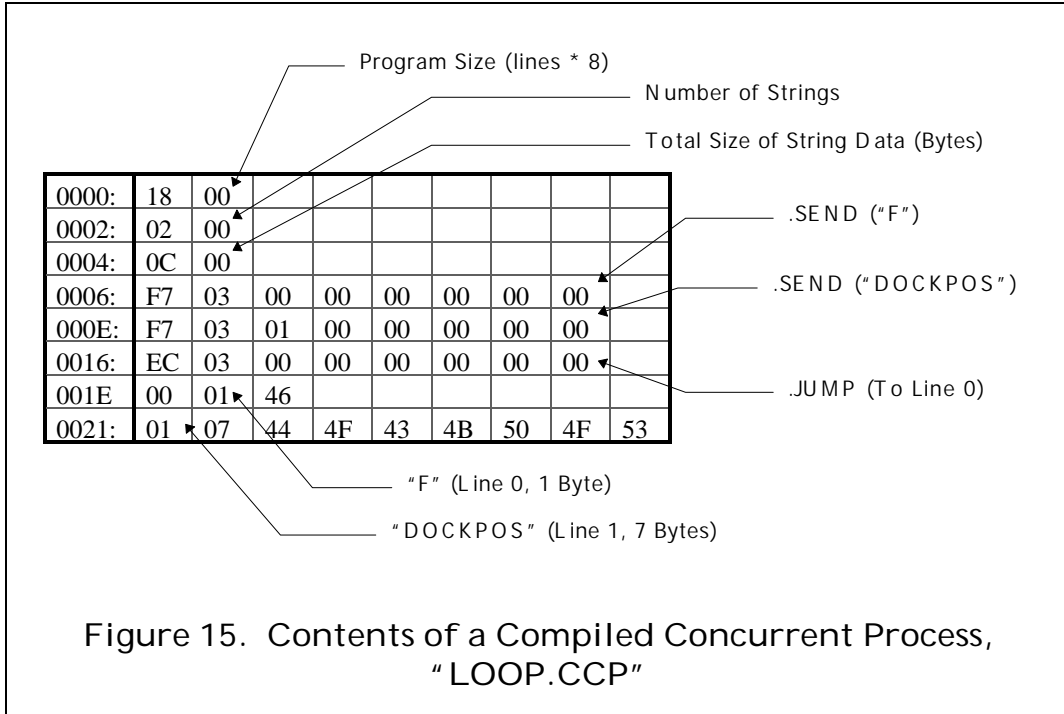
Figure 14. Dispatcher

The concurrent process executes just as if the operator sat down and typed in the commands. The definition file "TEST.DEF" is not necessary because the data in the concurrent process is purely string data, and the node names "F" and "DOCKPOS" are already known to Dispatcher through the action database.

| | |
|--------|---------|
| START: | |
| .SEND | F |
| .SEND | DOCKPOS |
| .JUMP | START |

Table 20. Concurrent Process File, "LOOP.CP"

Now that the concurrent process has been composed, it must be compiled. To do so, the "/c" or "-c" command line option is chosen. This is compatible with the current implementation of Cybermotion's PASM. From within Dispatcher, "Makeccp" (compiled concurrent process) would be typed to invoke the compiler. Upon successful compilation of the concurrent process, a file called "LOOP.CCP" will be created. This is analogous to the action file created from the compilation of a path program.



Keeping Figure 16 in mind, Figure 17 shows the contents of the actual compiled concurrent process file ("LOOP.CCP") in hexadecimal. The program has three instruction lines, so the size of the program is twenty-four. There are two strings, one for each of the ".SEND" commands. Including the line number, size, and the actual string, there are twelve bytes comprising the total string data for both strings. The three instructions along with their S, X, and Y parameters follow. Then the string data concludes the file. The first string is one byte long ("F") and occurs on line zero. The second string is seven bytes long ("DOCKPOS") and occurs on line one. There are no NULL terminators on the strings.

| Virtual Path Language Assembler v1.50 | | | | |
|--|----|----|----|---------|
| Disassembler mode ... disassembling [loop.ccp] | | | | |
| Command: | s: | x: | y: | string: |
| .send (1015) | 0 | 0 | 0 | F |
| .send (1015) | 1 | 0 | 0 | DOCKPOS |
| .jump (1004) | 0 | 0 | 0 | |

Table 21. Disassembly of "LOOP.CCP"

A disassembly of the "LOOP.CCP" file is given in Table 22. When comparing Figure 18 and Table 23, bear in mind that the sixteen bit words are in low-high byte order. So, for example, the ".SEND" instruction (1015) appears as "F703" in hexadecimal. To ensure the maximum versatility when debugging, the disassembly of concurrent processes does

purposely show X and Y parameters for those concurrent process commands whose only parameter is a string. Incidentally, the S parameter for these commands is the index to the string.

- **THE STRUCTURE OF PASM**

Upon invocation, PASM begins by parsing the command line (**Error! Not a valid link.**). The wildcard expansion routine ("setargv.obj") simply places every file specified by the wildcard on the command line delimited by a space. Command line parameters can be preceded by a "/" or a "-", the latter is not supported by Cybermotion's PASM. Command line parameters may also occur before or after the input file specification. To view all of the command line options, the user can enter the "-h" or "/h" command line option to view a brief help screen (Table 24).

Once the command line has been parsed and the various states of the compiler have been set, a file or an array of files to be compiled is given to the main loop of the compiler (Code Listing 1). The name of the particular file to be compiled is retrieved, and initialization of the error package, symbol table, and stack takes place.

PASM is a two-pass compiler that is loosely implemented around recursive descent (Pettus, 1994, p. 2). The fact that the Virtual Path Language lacks formal specification and is continually growing and changing lend to the difficulty in crafting a suitable compiler. Recursive descent is the simplest form of parsing by replacement (Fischer & LeBlanc Jr., p. 33). This relatively forgiving framework is ideal for the unpredictable nature of the Virtual Path Language. However, the majority of information concerning the Virtual Path Language is stored in tables that are easily updated as the language changes, thus simplifying the addition of new instructions and features to PASM.

In the first pass, the input file is read line by line and each word is converted to a symbolic representation, or "token," which is examined to determine whether or not it represents an identity (ID) or value. If found to be an ID, the symbol table is checked for prior references. If the symbol table does not contain the ID, it is inserted into the symbol table and the next token is fetched. If the token is found to represent a value or an arithmetic expression, it is evaluated as a floating point or integer value, whichever is appropriate.

| | |
|---|--|
| Virtual Path Language Assembler v1.50 | |
| The Path Assembler is invoked by the command | |
| <code>pasm { <options> <source filename> }</code> | |
| The following options are available: | |
| -a | Disassembler mode |
| -c | Compiles concurrent processes |
| -d | Debug mode |
| -g | Inhibits creation of .act file |
| -h | Help |
| -i | Print include files in the listing |
| -L | Print listing in landscape format |
| -l | Print listing file |
| -n | Set symbol table size |
| -o <filename> | Allows specifying output file name |
| -p | Parameter checking |
| -q | Disable checking of sequence errors |
| -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 |

Table 25. PASM Command Line Options

If the ID is identified as a command, it is given to the command parser (Code Listing 2). The command in question is checked against various sequence rules for the appropriateness of its occurrence within the program. Loop records are created to keep track of the branches and loops within the program.

The S, X, and Y parameters are then parsed. If an arithmetic expression is present, it is evaluated. The value is stored for semantic checking during the second pass. This process is repeated for each of the three parameters, as necessary.

When compiling a concurrent process, the commands are tested for those which represent intrinsic Dispatcher commands (Table 26). Some of these commands may be followed by a string. These strings go into a string buffer of predefined size (2048 bytes, as defined by Cybermotion). The size of the string and the line number of its occurrence are recorded. These commands are not parsed for their S, X, and Y parameters.

| | | |
|---------|---------|---------|
| .LOCK | .PATROL | .SOUND |
| .UNLOCK | .SEND | .LOG |
| .MAP | .CMND | .KILLCP |

Table 27. Dispatcher Commands Found in Concurrent Processes

If the token in question is a pseudo operand, it is handled by a separate function. A pseudo operand can be one of the following: A defined constant (DEFC), a defined position (DEFP), a set drive acceleration (DEFD), a set steer acceleration (DEFS), an external reference (EXTERN), or an include file (INCLUDE).

The pseudo operands which specify a constant or constants, check for an arithmetic expression to evaluate, then assign appropriate values, and test for parameters that are in violation of set limits. The external declarations and references are handled separately. The includes are handled with a stack.

In preparation for the second pass, the error package is reinitialized. When the second pass begins, if a concurrent process is being compiled, the header (Code Listing 3) is written to disk. The program is parsed in very much the same way as it was during pass 1. The symbol table is queried to resolve labels with values. The commands are parsed again, and this time the values are checked against a table to see whether or not they are within the range specified for the instruction.

The program data is now written to the output file. This is the action file in the case of path programs, or this is the compiled concurrent process file. In the latter case, the output must be modified slightly (Code Listing 4) to account for the sixteen bit word opcode.

Finally, a footer is added to the output file. In the case of the action file, the footer is the drive and steer accelerations. If the file is a compiled concurrent process, the footer is the string data from the Dispatcher specific commands, if there are any (Code Listing 5). The resources are then deallocated (Code Listing 6), all of the files are closed, and if files remain to be compiled the proper variables are reinitialized (Code Listing 7). If compilation is complete, the program terminates.

This completes a general overview of the structure of PASM. Particular attention was paid to the nature of the concurrent process compilation in order to highlight the major enhancements. The source code listings, along with additional detail pertaining to the code itself may be found in Appendix B.

- **TEST SUITE**

The greatest risk any piece of software faces is that of undergoing modification or enhancement. In the process of fixing a bug or adding new capabilities, something may be overlooked or accidentally changed that has adverse effects upon the program. Even more disturbing is the prospect that such an erroneous introduction might go unnoticed for such a length of time that its origin becomes uncertain. This can greatly increase the difficulty in finding and fixing the problem.

During the course of this work, many changes were made to PASM, and it was critical to be assured that none of the modifications had a

negative influence. In response to this threat, an extensive automated test suite was developed that quickly and easily demonstrated the integrity of the compiler. Implemented as a batch file and largely centered around the Microsoft MS-DOS file comparison utility, "FC.EXE," this test suite is both portable and easily expanded.

The main files, the PASM executable and the test suite batch file, reside in a root subdirectory. In this root subdirectory are any number of subdirectories that contain either path programs or concurrent process files along with their necessary include and definition files. From the root subdirectory, the PASM executable is copied to each child subdirectory and the respective path or concurrent process files are compiled. The compiled files are compared (using the file comparison utility) with files compiled before any changes were made to PASM.

The file comparison utility simply performs a binary comparison between two files, and its results are written to disk in a "summary file." These summary files are linked together into one large summary file which shows the file by file comparison for the entire test suite. This file is then compared with the same file produced from the unmodified version of PASM. This way very subtle changes (even at the *bit* level) in PASM's output can be detected easily and quickly.

All that is needed to add additional files to the test suite is a new subdirectory that contains the files and a small addition to the batch file. Of course, as new instructions and methods are added to PASM that do not exist in the unmodified version, special care must be taken to verify that the output is correct. This can be done via the disassembly options or by examining the actual contents of the output files with a binary editor. Once the correctness of the output is assured, the test suite may be used as before.

PSU: THE PASM SUPPORT UTILITY

The PASM Support Utility (PSU) provides a convenient means of updating and maintaining PASM as changes occur in the Virtual Path Language. The majority of command information is stored in tables within PASM. Changes or additions to the language are, to the greatest extent, reflected in the tables with the exception of any specialized code that is necessarily added directly. PSU is used to automatically generate updated tables for insertion into the PASM source code.

Both PASM and PSU share the same header file ("pasm.h"). From this file they have access to the terminals enumeration (Code Listing 8). From this terminals enumeration, the majority of data for the rest of the tables is obtained. PSU has several internal tables that succinctly represent the instructions (Code Listing 9). To add a new instruction, these tables are modified along with the terminals enumeration.

Once this is done, PSU generates the tables that are used throughout PASM. These tables can be directed to a file from which they may be copied and pasted directly into the PASM source code. The "CommandNames[]" and "ECommandNames[]" disassembly tables are put into the main ("main.cpp") file, and a reference to "CommandNames[]" in the header file ("pasm.h") must also be updated. The "commands[]" and "ecommands[]" tables, which contain the primary enumeration, instruction type, and parameter limits are put into the parser ("parser.cpp"). The index of parameter limits "ParLimits[]" is also placed in the parser. The scanner ("scanner.cpp") must have the new "keywords[]" table added to it as well as the "TokenValue[]" and "IdValue[]" tables. Figure 19 provides a summary of these relations.

The "DigitValue[]" and "FileChar[]" tables found in the scanner are generated with PSU as well as the parameter sign tables, "ParameterSign[]" and "EParameterSign[]," that are used by the disassembly functions ("main.cpp"). However, they are less likely to change. PSU also produces several other tables whose functions are no longer needed.

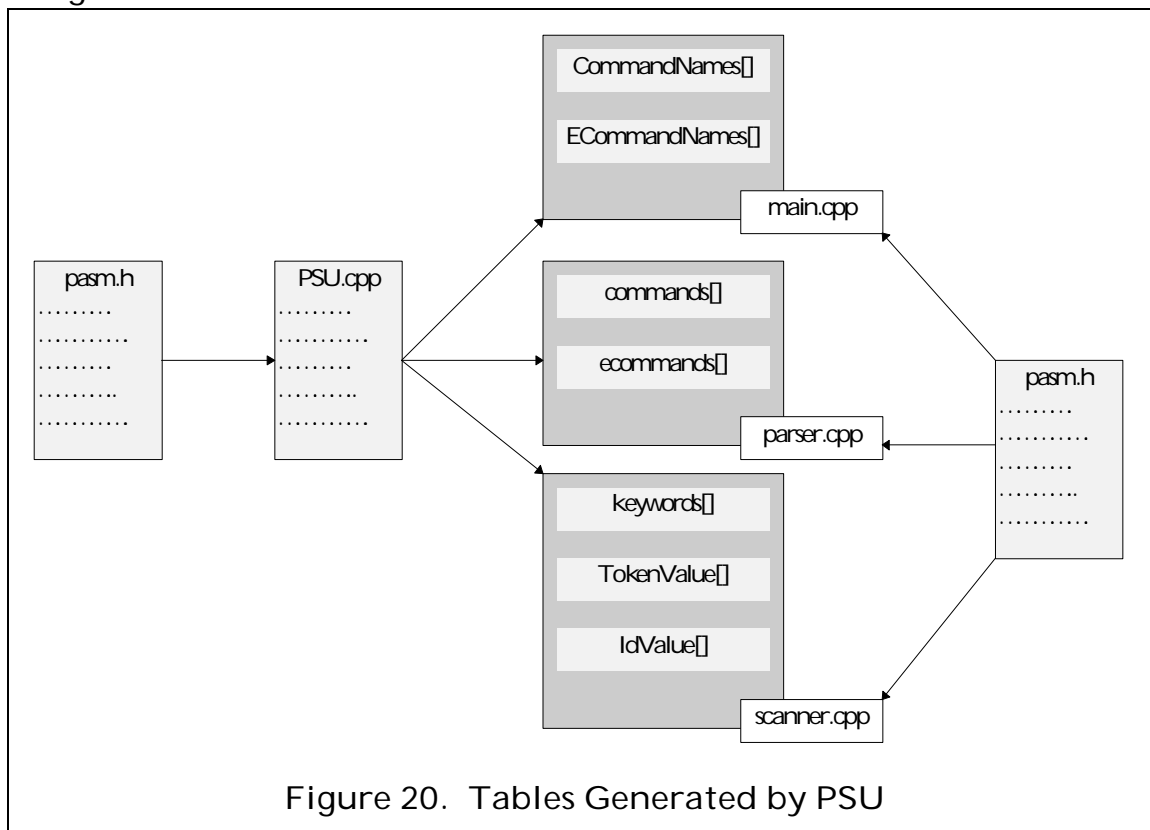


Figure 20. Tables Generated by PSU

In order to accommodate the changes imposed by the compilation of concurrent processes, and to offer the same ease of addition of concurrent process commands, PSU had to be restructured and modified. Three new

tables now have to be generated. These are the "ECommandNames[]" disassembly table (Code Listing 10), which is created by the "BuildEDisassemblerTable()" function (Code Listing 11); the "ecommands[]" commands table (Code Listing 12), which is created by the "BuildECommandTable()" function (Code Listing 13); and the "EParameterSign[]" table ([Error! Not a valid link.](#)) also used in disassembly, which is created by the "BuildEParameterSignTable()" function (Code Listing 14).

Because some concurrent process commands are identical to path language commands (with the exception of "." syntax and 1000+ opcode base) a method was devised to share compilation of these commands with the path language compiler. A cross-reference table "E_to_Norm[]" (Code Listing 15) was placed in the "pasm.h" header file along with the terminals enumeration. This cross-reference table indexes the zero-based enumeration value of the concurrent process terminals with the appropriate path language command terminal. If the concurrent process command has no corollary within the path language command set it is given an out of range value that the command parser looks for.

This approach allows both concurrent process commands and path language commands to share the same mechanisms where applicable. Concurrent process commands that have string parameters or concurrent process commands that do not have path language functions are handled by the parser as extensions to the path language.

RESULTS

With regard to hard results, PASM was tested over five separate test suites for a total of one hundred and sixty-three path programs and thirty concurrent processes. The output of PASM was evaluated against that of Cybermotion's PASM. Of the one hundred and sixty-three path programs, PASM compiled all one hundred and sixty-three (100%). Differences between the outputs of PASM and Cybermotion's PASM occurred in none. Cybermotion's PASM compiled only one hundred and thirty-five of the one hundred and sixty-three path programs (83%). The data are given in Appendix A.

Of the thirty concurrent processes files, PASM compiled all thirty (100%). Differences between the outputs of PASM and Cybermotion's PASM occurred in two (7%). The data are given in Appendix A.

| | |
|--------|---------------------------------------|
| .gpset | xxx, 1 2 3 4 5 6 7 8 9 10 11 12 13 14 |
| .gpset | xxx, 1 |

| | |
|--------|----------|
| .gpset | xxx, 2 |
| . | . |
| .gpset | xxx, 16 |
| .gpset | xxx, all |

Table 28. Examples of ".gpset" Usage

Within these two concurrent process files that did not compile correctly, the culprit was the ".gpset" command, which is a pseudo operand of the ".gpout" command. This command sets relays in the GPI according to a bit mask. For example, to set relays number one, four, and seven the correct value for the ".gpset" parameter written to disk in the compiled concurrent process file is seventy-three ($2^0 + 2^3 + 2^6$). To Cybermotion's implementation of PASM, the correct syntax for the ".gpset" command in the concurrent process file would be ".gpset xxx, 1 4 7" (where "xxx" denotes the proper GPI). However, this syntax is in violation of the trend for the Virtual Path Language to *always* separate parameters with commas.

In Table 29 several uses of the ".gpset" GPI command are shown. The first example of ".gpset" usage in Table 30 is the usage that is present in the two concurrent process files that did not compile properly. This is the syntax that PASM rejects. The other two examples, using multiple calls to ".gpset" or using the "all" keyword, are legal ways within PASM to accomplish what the first example does.

| Compiler | Compile Time |
|---------------------|-----------------|
| Cybermotion's PASM | 1.00.46 minutes |
| Sixteen bit PASM | 55.40 seconds |
| Thirty-two bit PASM | 44.45 seconds |

Table 31. Compilation Times

The largest single test suite contained seventy path programs. For the purpose of benchmarking compilation time, this test suite was compiled ten times (for a total of seven hundred path programs) by Cybermotion's PASM, the sixteen bit version of PASM, and thirty-two bit version of PASM. This was all driven from a batch file to minimize activity not directly associated with the compilers. The results are given in Table 32.

CONCLUSION

To emphasize the aspects of PASM that have been enhanced or added throughout the course of this work, two programming examples have been given. A general path programming example served as an introduction to PASM, its mission, and its core functionalities. An example in concurrent processes illustrated the underlying architecture of PASM with particular detail given to the extensions and additions that embody the majority of effort put into the code. Portions of the source code itself may be found in Appendix B. These modules represent the focus of modifications and development.

All code was ported to the Microsoft Visual C++ environment. This includes both Visual C++ 1.52 and Visual C++ 4.2. From within each of these respective environments, sixteen bit and thirty-two bit versions of both PASM and PSU were created and tested. Wildcard operation was added, and all of the restructuring of the code that continuous operation required was completed and tested. Concurrent process compilation and disassembly capabilities were added to PASM, and the functions and outputs necessary for PSU to support concurrent processes were added. To ensure the integrity of compilation, an extensive automated test suite was created.

PASM was evaluated against its commercial counterpart. The test suite of path programs and concurrent processes was used to identify compilation inconsistencies between the two. Benchmarking was performed to quantify relative execution times. In each case, PASM was proven to be competitive, if not superior.

The motivation for this expansion of PASM's capabilities was in response to interest that Cybermotion has expressed in PASM. The ease

with which PASM may be updated to support new instructions or methods, along with additional capabilities such as expression evaluation, universal units of measure, and expanded debug resources should ensure and solidify its acceptance in the commercial arena.

APPENDIX A: THE REFERENCE PROGRAM

```

Virtual Path Language Assembler v1.50
Disassembler mode ... disassembling [dockref.act]
File contains [22] instructions, [4] additional bytes

```

| Command: | s: | x: | y: |
|---------------------|-------|-------|-------|
| avoid (32) | 0 | 300 | 100 |
| warn (33) | 1 | 75 | 0 |
| radius (40) | 0 | 120 | 0 |
| cdeflect (59) | 0 | 32 | 1 |
| wdeflect (60) | 0 | 15 | 3 |
| state (81) | 0 | 0 | 33280 |
| writew (6) | 1 | 11890 | 10 |
| writew (6) | 1 | 12700 | 100 |
| writew (6) | 1 | 12702 | 36 |
| writew (6) | 1 | 11898 | 150 |
| writew (6) | 1 | 11475 | 241 |
| readb (7) | 1 | 11622 | 0 |
| jump= (12) | 15 | 0 | 0 |
| wait (3) | 0 | 1 | 0 |
| jump (9) | 16 | 0 | 0 |
| writeb (5) | 1 | 16012 | 1 |
| writew (6) | 1 | 13347 | 50 |
| writew (6) | 1 | 11633 | 15 |
| dock (21) | 1 | 300 | 100 |
| meanaz (31) | 0 | 0 | 0 |
| setxy (23) | 0 | 0 | 0 |
| undock (22) | 1 | 0 | 0 |
| Drive Acceleration: | [6] | | |
| Steer Acceleration: | [15] | | |

Table 33. Disassembly of "DOCKREF.ACT"

BIBLIOGRAPHY

Byrd, J. S. (1996). Low-Level Stored Waste Inspection Using Mobile Robots. Paper presented at ISRAM '96 Sixth International Symposium on Robotics and Manufacturing, Montpellier, France.

Cybermotion, Inc. (1991). Cybermotion Dispatcher User's Manual. (rev 1.2). Salem, VA: Cybermotion, Inc.

Cybermotion, Inc. (1995). Virtual Path Language Reference. (rev. 4.1). Salem, VA: Cybermotion, Inc.

Fischer, C. N. & LeBlanc Jr., R. J. (1991). Crafting a Compiler with C. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc.

Pettus, R. O. (1995). PASM Support Utility Manual. (Unpublished programmer's reference). University of South Carolina.

Pettus, R. O. (1994). Path Assembler Technical Manual (Unpublished programmer's reference). University of South Carolina.

THE PORTABLE PATH ASSEMBLER (PASM)

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.

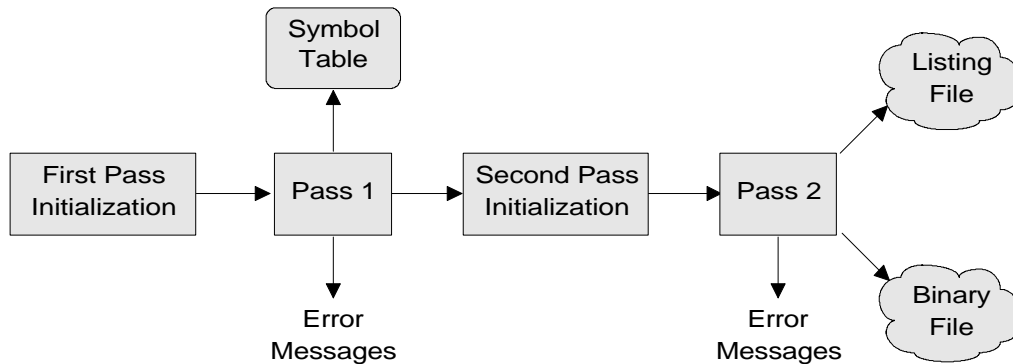


Figure 1. PASM operation

PASM operates as a typical two-pass assembler in order to resolve the problem of forward references (see figure 1). 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.

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 loosely¹ implemented using a top-down approach known as recursive descent. Due to the lack of a formal grammar, it cannot be guaranteed that 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 The term “loosely” is used here because the PASM grammar is not been formally defined due to the manner in which it has evolved.

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

Restriction 2 — For the case 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.

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 2 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 2. PASM help screen

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 include 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 3. 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 3. 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. 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 5. Listing file for example program 2.

The output obtained from the disassembler option (pa -a t2) is given in figure 6. 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 6. Disassembler output for program 2.

PASM STRUCTURE

PASM is implemented in seven files, `main.c`, `scanner.c`, `parser.c`, `symbol.c`, `utility.c`, `error.c`, and `pasm.h`, as shown in figure 7. 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

```

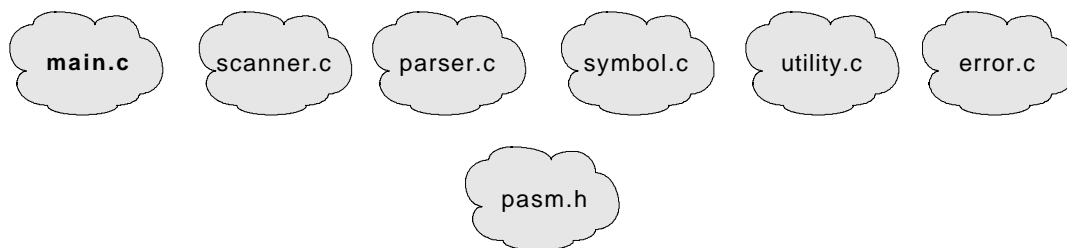


Figure 7. PASM file structure.

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 linkage¹. 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 <code>ParseCommand</code> and <code>ParsePseudop</code> |
| symbol.c | Symbol table and access routines |
| utility.c | Stack routines, units conversion table |
| error.c | Error handling routines (<code>RecordError</code> , <code>OutputErrors</code>) plus various record and control variables for the error system. |

Table 1. Contents of PASM files

¹ 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.

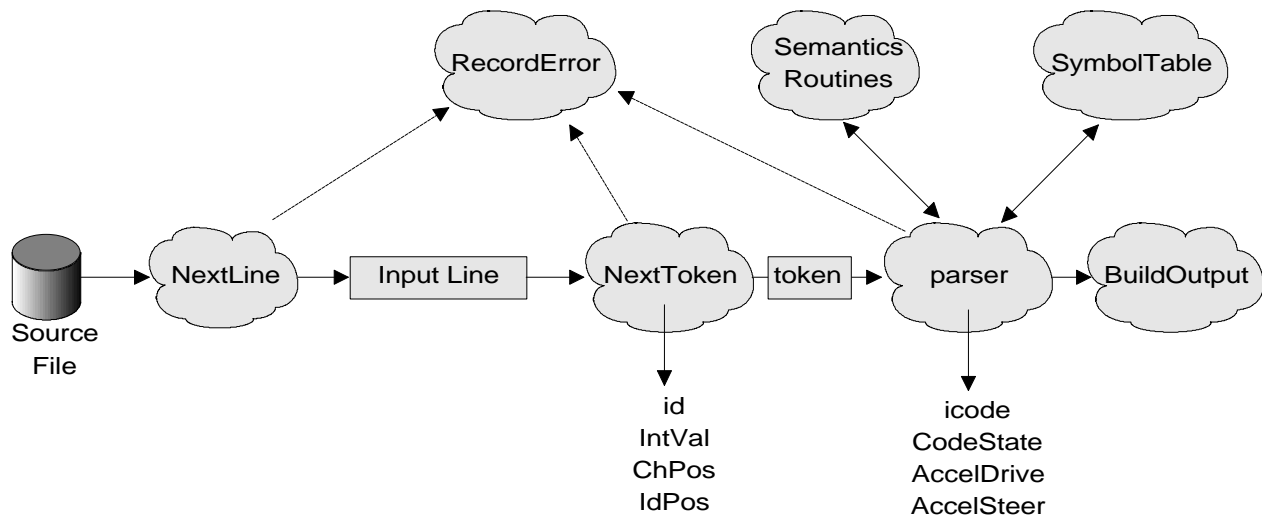
PASM OPERATION

Figure 8. PASM data flow model

PASM Dataflow

A data flow model of PASM is given in figure 8. 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 variables `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 a parameter 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.

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.

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.

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`

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.

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 identifier 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;

```

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.

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;
```

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;

```

The Pass 1 and Pass 2 Functions

```

{
SynType      *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 undefine 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 9. Annotated pass1 code.

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 9. 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.

The NextLine Function

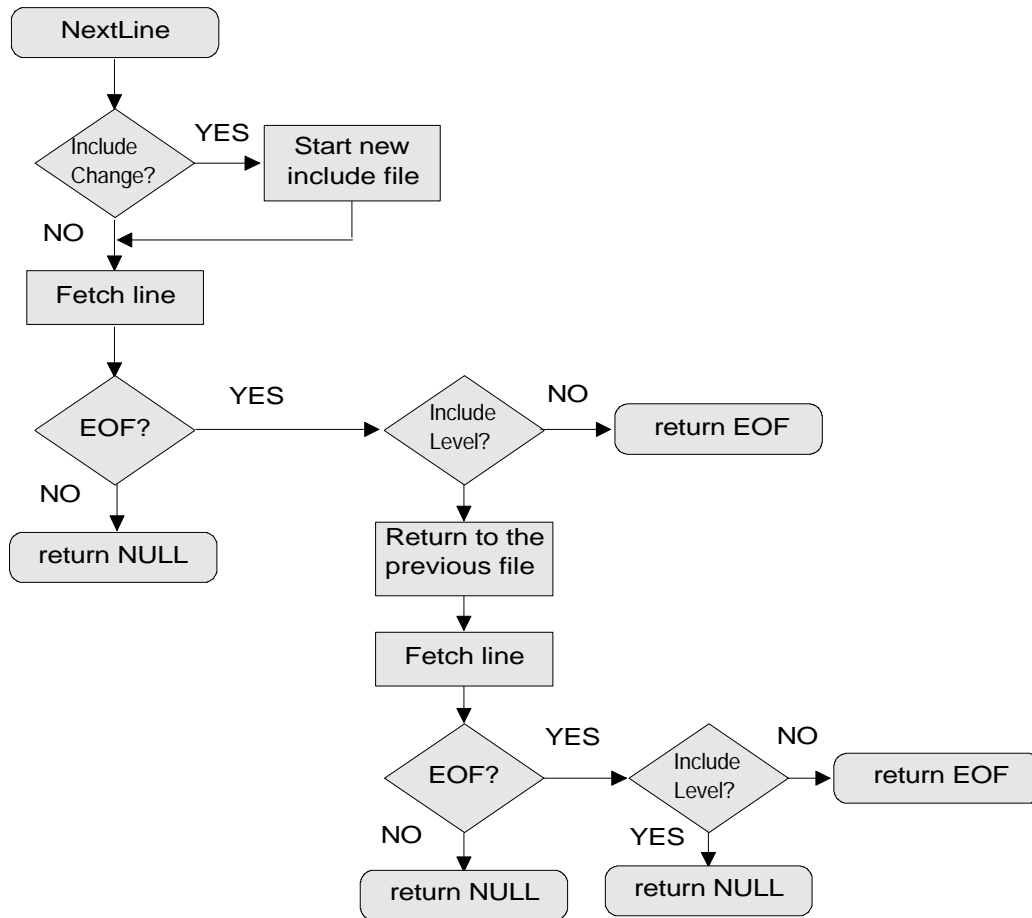


Figure 10. NextLine simplified flowchart

A simplified flowchart of the `NextLine` function is given in figure 10. 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.

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.

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).

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.

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.

The Scanner



Figure 11. 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 12. 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 12. PASM vocabulary

A flowchart of the operation of `NextToken` is given in figure 13. `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.

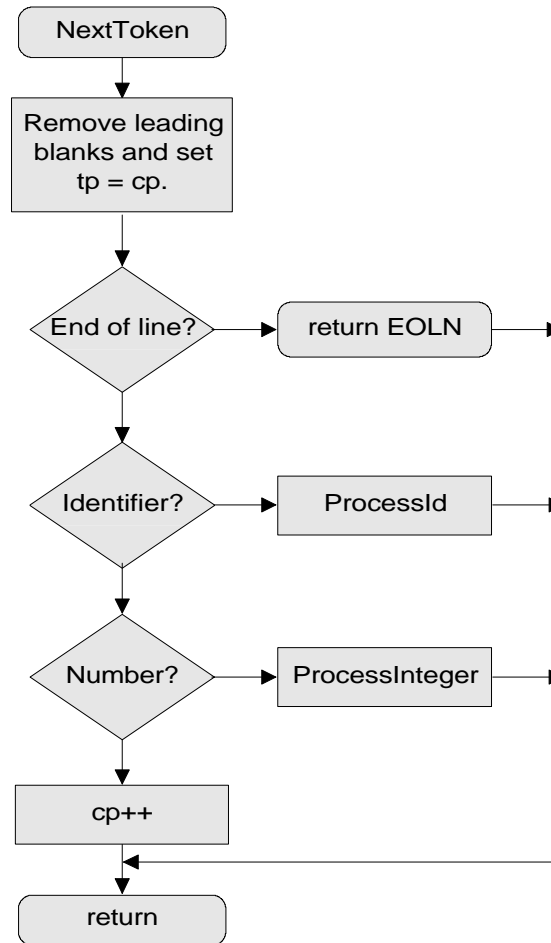


Figure 13. Flowchart of operation of `NextToken`

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 14.

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 14. NextToken code

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.

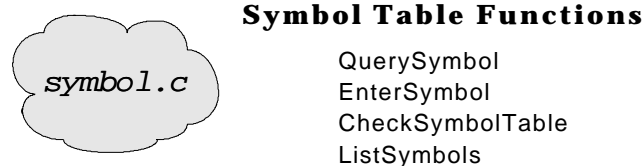


Figure 15. Symbol table access functions

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.

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 16). 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.

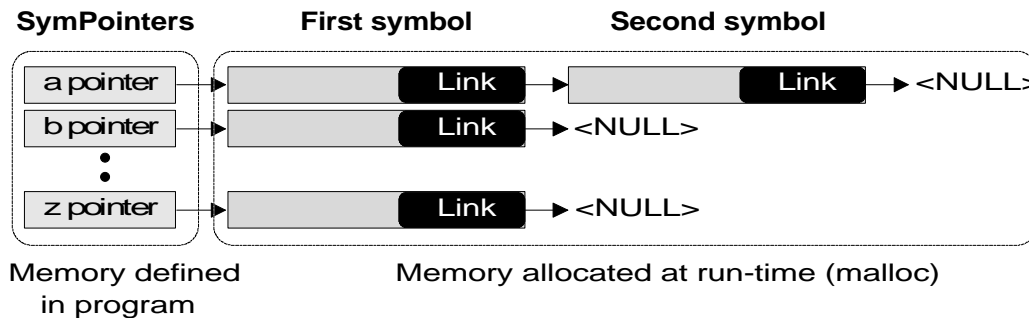


Figure 16. Symbol table structure

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(e1);
  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 be increased. 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.

The QuerySymbol and EnterSymbol Function

| FUNCTION | WHERE CALLED | PURPOSE |
|--------------------------|---------------------------------------|--|
| <code>QuerySymbol</code> | <code>pass2 (main.c)</code> | Checking labels to be sure they are in table. |
| | <code>primary (parser.c)</code> | Checking identifiers to determine if they are in the symbol table while parsing parameters. |
| <code>EnterSymbol</code> | <code>pass1 (main.c)</code> | Entering labels into the symbol table |
| | <code>GetDefineName (parser.c)</code> | Processing identifiers associated with <code>DEFB</code> and <code>DEFP</code> . <code>EnterSymbol</code> is used since the identifiers do not have to have been previously defined. |
| | <code>ProcessExtern (parser.c)</code> | Enter identifier associated with an <code>EXTERN</code> statement into the symbol table. |
| | <code>primary (parser.c)</code> | 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 = SymPtrs[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.

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 the various formatting commands but it is relatively straight forward.

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.

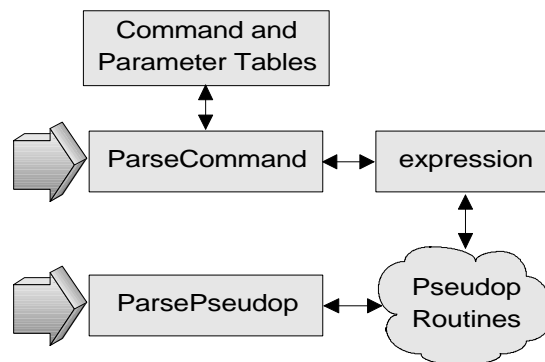


Figure 17. Parser structure.

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;
} SRTyp;

```

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:

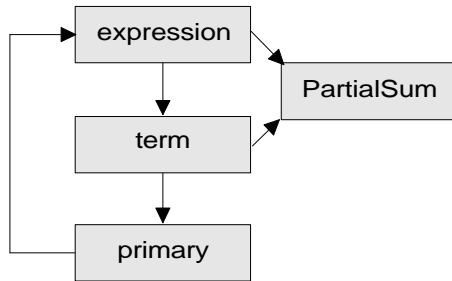
```
static SRTyp e ErrSR = { ERR_TYPE, 0, 0, 0 }, NullSR = { 0, 0, 0, 0 };
```

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

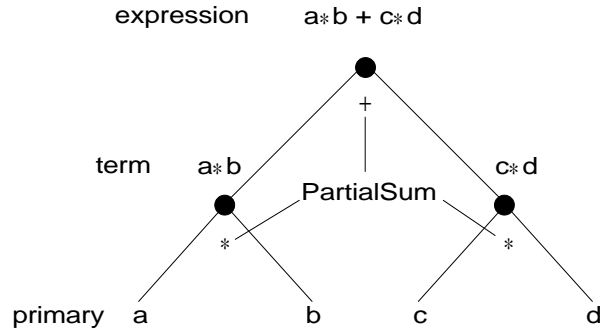
The Expression Evaluator

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 18. Expression evaluator structure and operation

The general structure and operation of the expression evaluator is shown in figure 18. 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 19. Simplified expression code.

The general form of `expression` is given in figure 19. 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 20. 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 20. 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 1, 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 21. Simplified `primary` code.

A simplified version of the code for `primary` is shown in figure 21. 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 D TYPE | ACTUAL TYPE | | | |
|-----------------------|-------------|----------|----------|-----------|
| | LABEL | CONSTANT | POSITION | UNDEFINED |
| <code>S_TYPE</code> | A1 | A2 | E1 | A3 |
| <code>C_TYPE</code> | E1 | A4 | E1 | E2 |
| <code>CP_TYPE</code> | 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: | <code>sr.s = p->s</code> |
| A2: | <code>sr.s = p->x</code> |
| A3: | <code>if(pass == 2)</code> <code>E2</code> |
| A4: | <code>sr.x = p->x</code> |
| A5: | <code>sr.x = p->x, sr.type = C</code> |

| | |
|-----|---------------------------------------|
| 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 22.

```

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 22. 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 23, 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 23. Parameter checking code

Parsing of Machine Commands

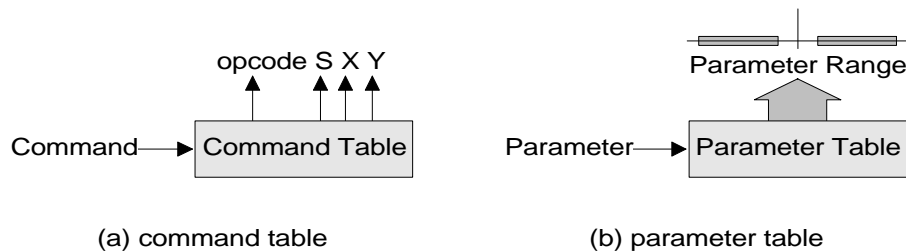


Figure 24. Primary parser tables

ParseCommand uses two tables (see figure 24), 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*¹ parameter values are:

¹ 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.

```

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.

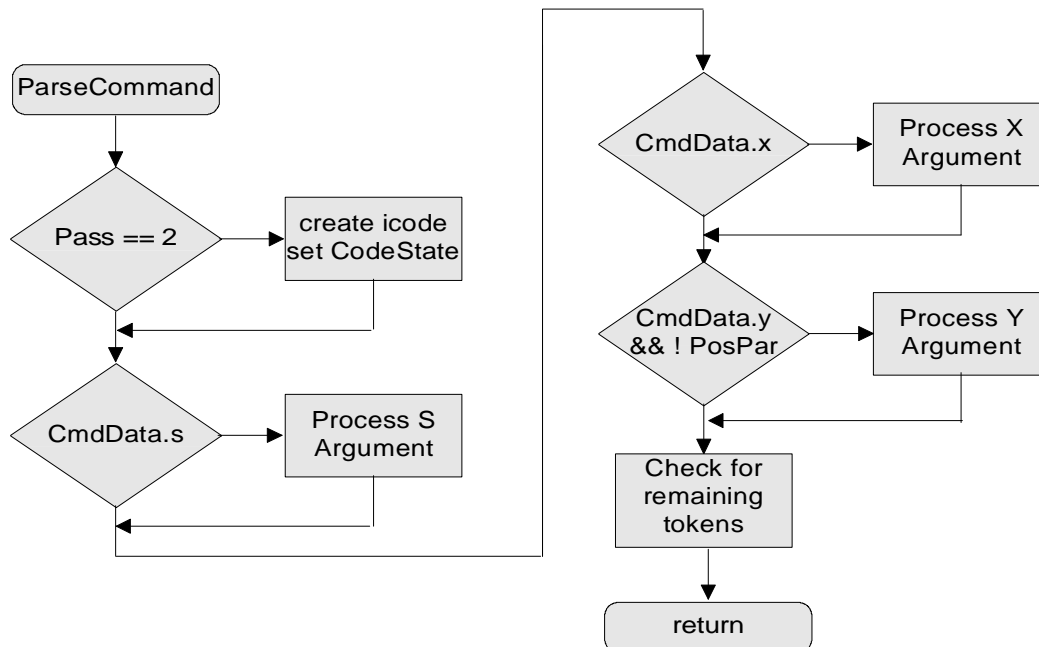


Figure 25. Flowchart of ParseCommand

A flowchart given the operation of `ParseCommand` is given in figure 25. 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.

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 26.

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 26. ParsePseudop

The DEFC and DEFP Commands

Both `DefineConstant` and `DefinePosition` first call the function `GetDefineName` (see figure 27) 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 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 28. 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 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 variable 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 27. `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 28. Code for `DefineConstant`

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 an operand is required. These commands are implemented by the SetDriveAccel and SetSteerAccel functions. The code for SetDriveAccel is given in figure 29. 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 29. SetDriveAccel code

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 30. ProcessInclude code.

The code to process the include directive, `ProcessInclude`, is given in figure 30. 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.

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.

Error Handler

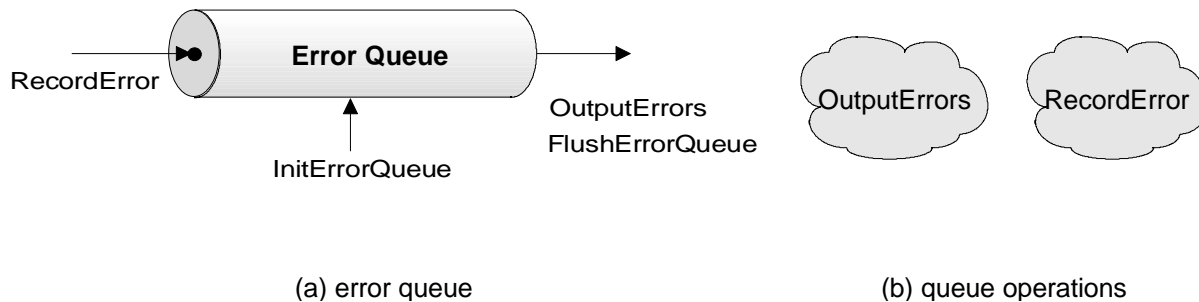


Figure 31. Error queue

The primary components of the error handler (see figure 31) 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.

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 32. 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 32. Error entry and error queue code.

Error System Variables

The variables used by the error system are given in figure 33. 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 33. Error system variables

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 33.

```

/*
!           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 33. Error message strings

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`.

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;
```

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.

Utility Package

The file `utility.c` is used to hold various utility routines of general use. It currently contains the table used for converting the units and the stack routines used for the include stack.

THE PASM SUPPORT UTILITY (PSU)

INTRODUCTION

PSU is a support program which is used to automate the creation of the various PASM tables. It is designed to run in any (ANSI) environment and has a simple menu interface for ease of use. PSU is not designed to be an end user program and is not written to the same standards as PASM. It is implemented as a single, rather large (>50K bytes for v1.40), file (`psu.c`). Because PSU uses the same header files as PASM, most PASM changes are automatically reflected in PSU. However, PSU is the mechanism by which most PASM parameter values are defined, so these changes are first made in the appropriate PSU tables, then a new copy of the appropriate PASM table is created and inserted in the program. These tables define most of the base values for the Path Assembler so their accuracy is essential.

```

PASM Support Utility      v1.40
Vocabulary entries:      [126] entries
Command table:           [89] entries
Alias table:              [11] entries
Parameter table:         [38] entries
Pseudop table:           [6] entries
Last command table entry: [dowhile=]
Last alias table entry:   [wends]
Last parameter table entry: [Special processing
required]
Last pseudop table entry: [include]

Press [RETURN] for next screen ..

```

Figure 1. PSU opening screen

The PSU opening screen is given in figure 1. The opening screen gives the sizes of various tables along with the last entry. This information has no particular use and is included primarily for a “sanity check” on the program. The value given for “Vocabulary entries” is the value of `LAST_TERMINAL` in `pasm.h`. The vocabulary enumeration is the primary item from the header file which used by PSU. The command, alias, parameter, and pseudop tables are the primary sources of information used by PSU. Their sizes and last entry are displayed to convey the current state of PSU.

The PSU main menu is given in figure 2. This menu contains the commands to construct the primary tables used by PASM. The token value and keyword tables are used by the scanner (in `scanner.c`). The token value table is used for initial assignments of the token values representing the appropriate vocabulary component. The keyword table is used by

the scanner to look up identifiers to see if they match a PASM keyword. If so, the token value is changed from ID to the appropriate keyword value.

The command and parameter tables are used by the parser (in parser.c) for the information required to parse machine commands. These two tables are the primary “intelligence” of PSU and are fundamental to the operation of PASM. The structure of these tables will be discussed in a section to follow.

```

-----
PASM Support Utility    v1.40

[1]:  Menu 2 Items
[2]:  Build Token Value Table
[3]:  Build Keywords Table
[4]:  Build Command Table
[5]:  Build Parameter Table
[6]:  Build Paramater Sign Table
[7]:  Build Parameter List
[8]:  Build Disassembler Table
[F]:  Specify Files
[H]:  Help
[M]:  Display Menu
[Q]:  Exit Program
-----
Output File:  [Not Specified]
-----

(PASM Support Utility) Select Option [ ]

```

Figure 2. PSU Main menu

```

-----
PASM Support Utility (Menu2)    v1.40

[1]:  Generate Command Documentation
[2]:  Generate Parameter Documentation
[3]:  Generate PASM Test Suite
[4]:  Generate PASM Parameter Limits Test Suite
[5]:  BuildToLowerTable
[6]:  BuildIsDigitTable
[7]:  BuildIsHexTable
[F]:  Specify Files
[H]:  Help
[M]:  Display Menu2
[R]:  Return to Main Menu
[Q]:  Exit Program
-----
Output File:  [Not Specified]
-----

(Menu 2 ) Select Option [ ]

```

Figure 3. PSU menu 2 items

Menu 2, shown in figure 3, contains the commands for several secondary operations. These include general of command and parameter documentation, creation of some simple test suites, and the construction of several utility tables.

PSU DATA TABLES

PSU uses two sources of information about PASM, the PASM header file and several internal tables, the command table, the parameter table, and the pseudop table. The internal tables, shown in figure 4, are the only source of information on the actual commands and parameters, so it is vital that they be kept up to date.

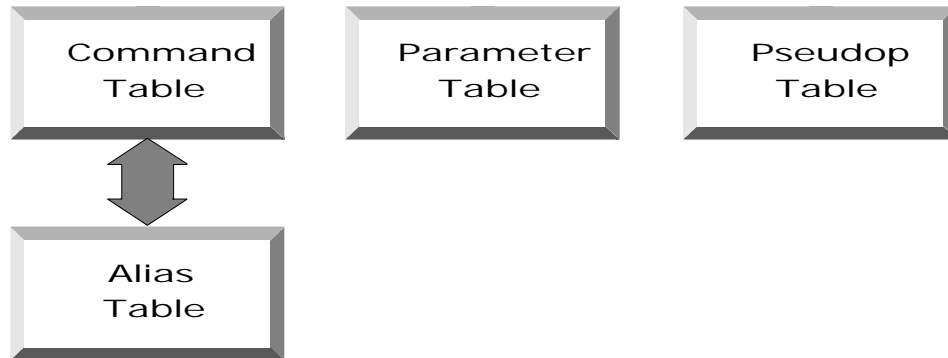


Figure 4. Primary PSU tables

The Command Table

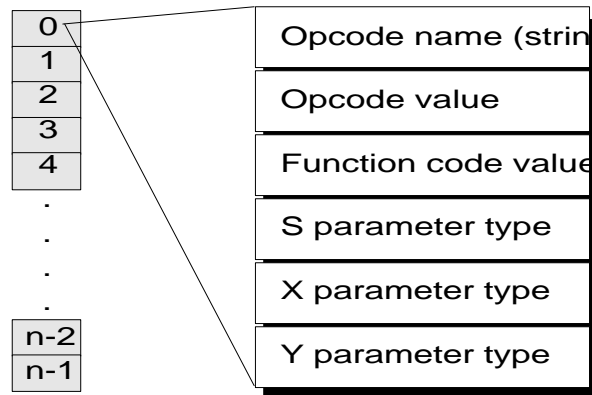


Figure 5. Command table entry

The command table (illustrated in figure 5) contains an entry for each command (and for each unused opcode within the contiguous opcode map) in the path language. Each entry consists of (i) the opcode name, stored as a string, (ii) the opcode value, (iii) the actual function code assigned to the opcode, and (iv) information about each of the three possible command parameters. The opcode value is the same as the index for the location and is used to reference the table. The function code is the same value if the opcode is used, 0 otherwise. This structure allows the use of the opcode as an index to the table while han-

dling “holes” in the opcode map. The parameter types are indices to the parameter table. The C code used to define each command table entry is given in figure 6.

```
typedef struct
{
  char cmd[16]; /* command identifier */
  int opcode; /* opcode value, not always assigned */
  int Fcode; /* Function code for assigned opcodes */
  int s; /* index for S parameter in parameters table */
  int x; /* index for X parameter in parameters table */
  int y; /* index for Y parameter in parameters table */
} CmdType ;
```

Figure 6. C struct used to define command table entry.

The command table itself is shown in figure 7. The table is statistically initialized.

```
CmdType commands[LAST_COMMAND+1] =
{
  {"nop", 0, NOP, NONE, NONE, NONE },
  {"run", 1, RUN, MAX_SPEED, S_WORD, S_WORD },
  {"turn", 2, TURN, NONE, MAX_AZIMUTH, NONE },
  .
  .
  {"dostop@", 92, DOSTOP_AT, NUM_DO_INSTR, S_WORD, S_WORD },
  {"apprwall", 93, APPROACH, NONE, DIST_WALL, NONE },
  {"apprjunk", 94, APPROACH, NONE, NONE, DIST_JUNK },
  {"", 0, 0, 0, 0 },
};
```

Figure 7. Command table (partial code).

Notice that the last two entries are the alternative forms (apprwall, apprjunk) of the approach command. They have the same value of function code but take different arguments. All unused opcode values are assigned nop's.

```
typedef struct
{
  char name[32];
  char token[32];
} NameType;

NameType AliasNames[16] =
{
  {"call!=", "CALL_NE" },
  {"callneq", "CALL_NE" },
  .
  .
  {"wend", "WENDS_AT" },
  {"wends", "WENDS_AT" },
  {"", "" }
};
```

Figure 8. Alias table.

The alias table, shown in figure 8, allows assigning multiple names to a given command. It allows compatibility with previous versions of PASM. This technique is preferred to allowing abbreviations since it is more controllable and less error prone.

The Parameter Table

| PARAMETER NAME | TOKEN | VALUE |
|-------------------------------------|-----------------|---------------------|
| Unsigned byte (generic value) | U_BYTE | 0 ⇒ 255 |
| Unsigned word (generic value) | U_WORD | 0 ⇒ 65,535 |
| Signed word (generic value) | S_WORD | -32,737 ⇒ 32,767 |
| Wall distance (X, Y) | DIST_WALL | 300 ⇒ 700 |
| Junk distance (Y) | DIST_JUNK | 100 ⇒ 1,000 |
| Binary variable (generic value) | BINARY | 0 ⇒ 1 |
| Collision avoidance distance (X, Y) | DIST_AVOID | 0 ⇒ 600 |
| Maximum speed (S) | MAX_SPEED | 0 ⇒ 250 |
| Maximum deflection (X) | MAX_DEFLECTION | 0 ⇒ 85 |
| Maximum deflection rate (Y) | MAX_DEFL_RATE | 0 ⇒ 5 |
| Maximum excursion distance (X, Y) | MAX_EXC_DIST | 0 ⇒ 800 |
| Maximum steer velocity (Y) | MAX_STEER_VEL | 0 ⇒ 350 |
| Drive motor current limit | DRIVE_CUR_LIMIT | 0 ⇒ 350 |
| Steer motor current limit (Y) | STEER_CUR_LIMIT | 0 ⇒ 250 |
| Number of DO instructions (S) | NUM_DO_INSTR | 1 ⇒ 25 |
| Maximum dock number (S) | MAX_DOCK_NUM | 0 ⇒ 31 |
| Distance from dock to stop (X) | MAX_DOCK_STOP | 50 ⇒ 600 |
| Distance from dock to obstacle (Y) | MAX_DOCK_OBS | 0 ⇒ 600 |
| Door type (S) | DOOR_TYPE | 0 ⇒ 6 |
| Distance to door center (X) | DOOR_DISTANCE | 200 ⇒ 600 |
| Width of door opening (Y) | DOOR_WIDTH | 300 ⇒ 800 |
| Follow distance | MAX_FOLLOW_DIST | 0 ⇒ 600 |
| Gate distance (X) | MAX_GATE_DIST | ±200 ⇒ ±700 |
| Gate width (Y) | MAX_GATE_WIDTH | 0 ⇒ 255 |
| Distance to hall wall (X) | MAX_HALL_DIST | -600 ⇒ 600 |
| Maximum azimuth value (X) | MAX_AZIMUTH | 0 ⇒ 1023 |
| Maximum address value (X, Y) | MAX_ADDRESS | 0 ⇒ 65,535 |
| Maximum absolute direction (Y) | MAX_REL_DIRECT | 0 ⇒ ±512 |
| SPI mode (S) | SPI_MODE | 0 ⇒ 2 |
| Maximum RUNON radius (X) | MAX_RADIUS | 50 ⇒ 1,000 |
| Maximum steer acceleration (X) | MAX_STEER_ACCEL | 0 ⇒ 10 |
| Maximum drive acceleration (Y) | MAX_DRIVE_ACCE | 0 ⇒ 35 |

| | | |
|--------------------------------|-----------------|------------------|
| | L | |
| Maximum tilt angle (Y) | MAX_TILT_ANGLE | -999 ⇒ 999 |
| Maximum undocking distance (X) | MAX_UNDOCK_DIST | -32,767 ⇒ 0 |
| Positive word | POS_WORD | 0 ⇒ 32,767 |
| Unlimited range | UNLIMITED | -32,767 ⇒ 65,535 |
| Special processing required | SPECIAL | 0 |

Figure 9. Parameter table information

The parameter table information is used to (i) determine if the parameter is required and (ii) define the legal values for the parameter. The table given in figure 9 gives the information contained in the parameter table. The process by which this information was gathered was to start at the beginning of the opcode sequence and identify each unique parameter. Many of the parameters, the maximum value of an address for instance, are overloaded (used in several different situations). The same numerical value will occur more than once where the types of parameters differ substantially and there is a good chance that one might be changed without affecting the other. (**Note:** It may be necessary to separate some of the existing over-loaded parameters in the future.) The parameter values are assigned by an enumeration in `pasm.h`. The C code for the parameter table entry and for the table itself is shown in figures 10 and 11.

The enumeration also includes `NONE` in the first position (i.e., `NONE = 0`) to be used where no parameter is required.

```
typedef struct
{
    char    par[32];      /* parameter identifier */
    char    Ptoken[16]; /* string for ParameterType enum entry */
    int     Pcode; /* Numeric value of function code (opcode) */
    int     Dcode; /* Symbolic value of function code */
    long    NegMin;      /* */
    long    NegMax;      /* */
    long    PosMin;      /* */
    long    PosMax;      /* */
} ParType;
```

Figure 10. C struct for parameter table entry

```
ParType parameters[LAST_PARAMETER+1] =
{
    { "None", "NONE", 0, NONE, 0, 0, 0, 0 },
    { "Unsigned byte (generic value)", "U_BYTE", 1, U_BYTE, 0, 0, 0, 255 },
    { "Unsigned word (generic value)", "U_WORD", 2, U_WORD, 0, 0, 0, 65535 },
    { "Signed word (generic value)", "S_WORD", 3, S_WORD, -32737, 0, 0, 32767 },
    { "Wall distance", "DIST_WALL", 4, DIST_WALL, 0, 0, 300, 700 },
    { "Junk distance", "DIST_JUNK", 5, DIST_JUNK, 10, 0, 00, 1000 },
    .
    .
    { "Maximum drive acceleration", "MAX_DRIVE_ACCEL", 32, MAX_DRIVE_ACCEL, 0, 0, 0, 35 },
    { "Maximum tilt angle", "MAX_TILT_ANGLE", 33, MAX_TILT_ANGLE, -999, 0, 0, 999 },
    { "Maximum undocking distance", "MAX_UNDOCK_DIST", 34, MAX_UNDOCK_DIST, -32767, 0, 0, 0 },
    { "Positive word", "POS_WORD", 35, POS_WORD, 0, 0, 0, 32767 },
    { "Unlimited range", "UNLIMITED", 36, UNLIMITED, -32767, 0, 0, 65535 },
}
```

```
{ "Special processing required", "SPECIAL", 37, SPECIAL, 0, 0, 0, 0 },
{ "", "", 0, 0, 0, 0 },
};
```

Figure 11. Partial code for parameter table

The Pseudop Table

```
typedef      struct
{
    char      pseudop[32];
    char      PseudopToken[16];
    int       PseudopCode;
} PseudopType;
```

Figure 12. C struct for pseudops

The final table is the pseudop table, which holds the definitions of the PASM pseudops (or assembler directives).

```
PseudopType      pseudops[LAST_PSEUDOP - LAST_COMMAND + 1] =
{
    { "defc", "DEFC", DEFC },
    { "defd", "DEFD", DEFD },
    { "defp", "DEFP", DEFP },
    { "defs", "DEFS", DEFS },
    { "extern", "EXTERN", EXTERN },
    { "include", "INCLUDE", INCLUDE },
    { "", "", 0 }
};
```

Figure 13. pseudop table

PARAMETER DOCUMENTATION

PSU can create simple documentation on either commands (figure 14) or parameters (figure 15). The primary use of this feature is for checking the contents of the tables themselves.

```
run (1) (position command)
    S-Parameter: Maximum speed
    X-Parameter: Signed word (generic value)
    Y-Parameter: Signed word (generic value)

turn (2)
    S-Parameter: None
    X-Parameter: Maximum azimuth value
    Y-Parameter: None
    .
    .
downhen!= (87)
    S-Parameter: Number of DO instructions
    X-Parameter: Maximum address value
    Y-Parameter: Signed word (generic value)

dowhile= (88)
    S-Parameter: Number of DO instructions
```

X-Parameter: Maximum address value
 Y-Parameter: Signed word (generic value)

Figure 14. Example command documentation

| | | |
|--------------------------------|-----------------|--------------------------|
| None | NONE | [0..0], [0..0] |
| Unsigned byte (generic value) | U_BYTE | [0..0], [0..255] |
| Unsigned word (generic value) | U_WORD | [0..0], [0..65535] |
| Signed word (generic value) | S_WORD | [-32737..0], [0..32767] |
| Wall distance | DIST_WALL | [0..0], [300..700] |
| Junk distance | DIST_JUNK | [10..0], [0..1000] |
| Binary variable | BINARY | [0..0], [0..1] |
| Collision avoidance distance | DIST_AVOID | [0..0], [0..600] |
| Maximum speed | MAX_SPEED | [0..0], [0..250] |
| Maximum deflection | MAX_DEFLECTION | [0..0], [0..85] |
| Maximum deflection rate | MAX_DEFL_RATE | [0..0], [0..5] |
| Maximum excursion distance | MAX_EXC_DIST | [0..0], [0..800] |
| Maximum steer velocity | MAX_STEER_VEL | [0..0], [0..350] |
| Drive motor current limit | DRIVE_CUR_LIMIT | [0..0], [0..350] |
| Steer motor current limit | STEER_CUR_LIMIT | [0..0], [0..250] |
| Number of DO instructions | NUM_DO_INSTR | [0..0], [1..25] |
| Maximum dock number | MAX_DOCK_NUM | [0..0], [0..31] |
| Distance from dock to stop | MAX_DOCK_STOP | [0..0], [50..600] |
| Distance from dock to obstacle | MAX_DOCK_OBS | [0..0], [0..600] |
| Door type | DOOR_TYPE | [0..0], [0..6] |
| Distance to door center | DOOR_DISTANCE | [0..0], [200..600] |
| Width of door opening | DOOR_WIDTH | [0..0], [300..800] |
| Follow distance | MAX_FOLLOW_DIST | [0..0], [0..600] |
| Gate distance | MAX_GATE_DIST | [-700..-200], [200..700] |
| Gate width | MAX_GATE_WIDTH | [0..0], [0..255] |
| Distance to hall wall | MAX_HALL_DIST | [-600..0], [0..600] |
| Maximum azimuth value | MAX_AZIMUTH | [0..0], [0..1023] |
| Maximum address value | MAX_ADDRESS | [0..0], [0..65535] |
| Maximum absolute direction | MAX_REL_DIRECT | [-512..0], [0..512] |
| SPI mode | SPI_MODE | [0..0], [0..2] |
| Maximum RUNON radius | MAX_RADIUS | [0..0], [50..1000] |
| Maximum steer acceleration | MAX_STEER_ACCEL | [0..0], [0..10] |
| Maximum drive acceleration | MAX_DRIVE_ACCEL | [0..0], [0..35] |
| Maximum tilt angle | MAX_TILT_ANGLE | [-999..0], [0..999] |
| Maximum undocking distance | MAX_UNDOCK_DIST | [-32767..0], [0..0] |
| Positive word | POS_WORD | [0..0], [0..32767] |
| Unlimited range | UNLIMITED | [-32767..0], [0..65535] |
| Special processing required | SPECIAL | [0..0], [0..0] |

Figure 15. Parameter documentation.

DEPENDENCIES

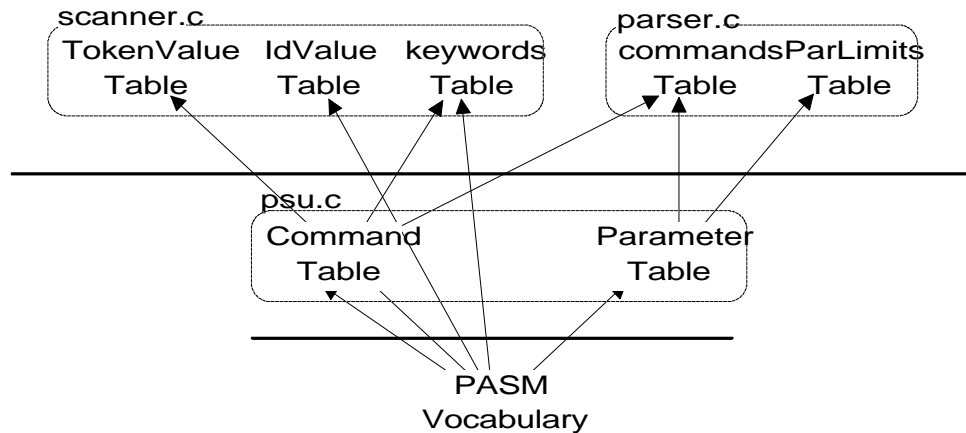


Figure 16. Dependencies

There are a number of dependencies among the PASM vocabulary and the PASM and PSU tables, as illustrated in figure 16.

1. Changes in the PASM vocabulary require that PSU be re-compiled and all dependent PASM tables be replaced.
2. Changes in either the command table or parameter table require that PSU be re-compiled and the dependent PASM tables be replaced.
3. Changes in the alias table require that PSU be re-compiled and that the scanner keywords table be replaced.

PSU STRUCTURE

The functions which make up PSU can be divided into four groups, (i) the main program, (ii) initialization, (iii) the application, and (iv) utility. The program is largely structured about the menu tasks and there is relatively little coupling between the modules other than the shared data structures. The four groups are given below.

Main Program Functions — These functions provide the user interface and handle the processing of commands.

```

int      GetCommand(void) ;
int      ProcessCommand(int) ;
int      ProcessCommand2(void) ;
void     HandleException(char *) ;
void     help(void) ;
void     MainMenu(void) ;
void     ProcessHelp(int) ;
void     SpecifyFiles(void) ;
void     StatusBar(void) ;

```

Initialization Code — These functions handle the program initialization.

```
void    init(int, char *[ ]);
void    InitTables(void);
void    ListTableSizes(void);
```

Application Routines — These are the functions which perform the actual program tasks. Generally there is a one to one correlation with the menu items, although several tasks require more than one function. This is indicated by the indented lines of code.

```
void    BuildCommandTable(void);
void    BuildDisassemblerTable(void);
void    BuildIsDigitTable(void);
void    BuildIsHexTable(void);
void    BuildKeywordsTable(void);
void    void    MkKeywordsArray(void);
void    void    MkToken(char [ ], char [ ]);
void    void    SortKeywords(void);
void    BuildParameterList(void);
void    BuildParameterSignTable(void);
void    BuildParameterTable(void);
void    BuildTokenValueTable(void);
void    BuildToLowerTable(void);
void    CommandDocumentation(void);
void    void    SingleCommand(char [ ]);
void    GenerateLimitsSuite(void);
void    GenerateTestSuite(void);
void    int    IsJumpOrCall(char *);
void    ParameterDocumentation(void);
```

Utility Functions — The utility routines provide several I/O functions for the program.

```
int     ScreenFull(int);
void    ClearScreen(void);
void    ScreenActivity(void);
void    WaitForNewLine(void);
```

Appendix B

ARIES: An Intelligent Inspection and Survey Robot

COMPUTER VISION SYSTEM

Department of Electrical & Computer Engineering
Clemson University

B. VISION SYSTEM

B.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.

B.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.

B.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).

B.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.

B.3.2 Time and Power

B.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.

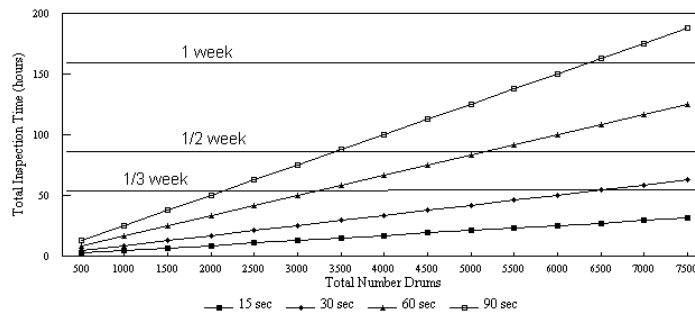


Figure 5-1: Inspection Capacity as Function of Single Drum Inspection Time.

B.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 sec per drum. Further speedup may be possible, if faster strobe recharge times are enabled, and with further algorithm recoding.

B.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.

B.5 SEGMENTATION ALGORITHMS

B.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 learning algorithm is provided which, given inputs from a human "tutor", adjusts the algorithm parameters.

B.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.

B.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)

B.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.

B.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.

B.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.

B.5.7 Rust Segmentation

B.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.

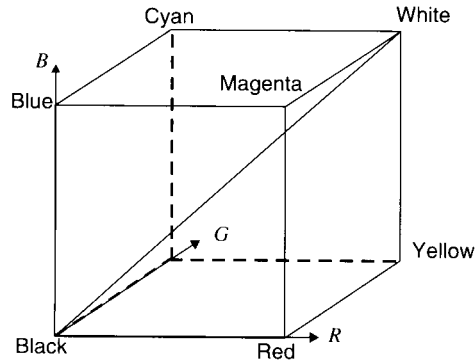


Figure 5-2: RGB Coordinate System

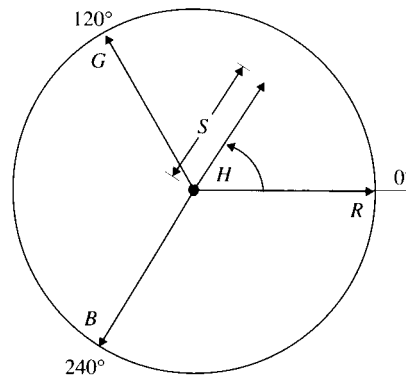


Figure 5-3: Hue Space Color Map (Red = 0 deg)

- 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.

B.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.

B.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:

$$(x(i,j) - x(i - \text{DELTA } i, j) > 30 \text{ quad AND}$$

$$x(i,j) - x(i + \text{DELTA } i, j) > 30 \text{ quad AND}$$

$$x(i,j) - x(i, j - \text{DELTA } j) > 30 \text{ quad AND}$$

$$x(i,j) - x(i, j + \text{DELTA } j) > 30 \text{ quad AND}$$

$$x(i,j) > 50)$$

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.

B.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 operator's 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.

B.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.

B.8 ARIES #1 VISION HARDWARE AND ALGORITHM IMPLEMENTATION

B.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.

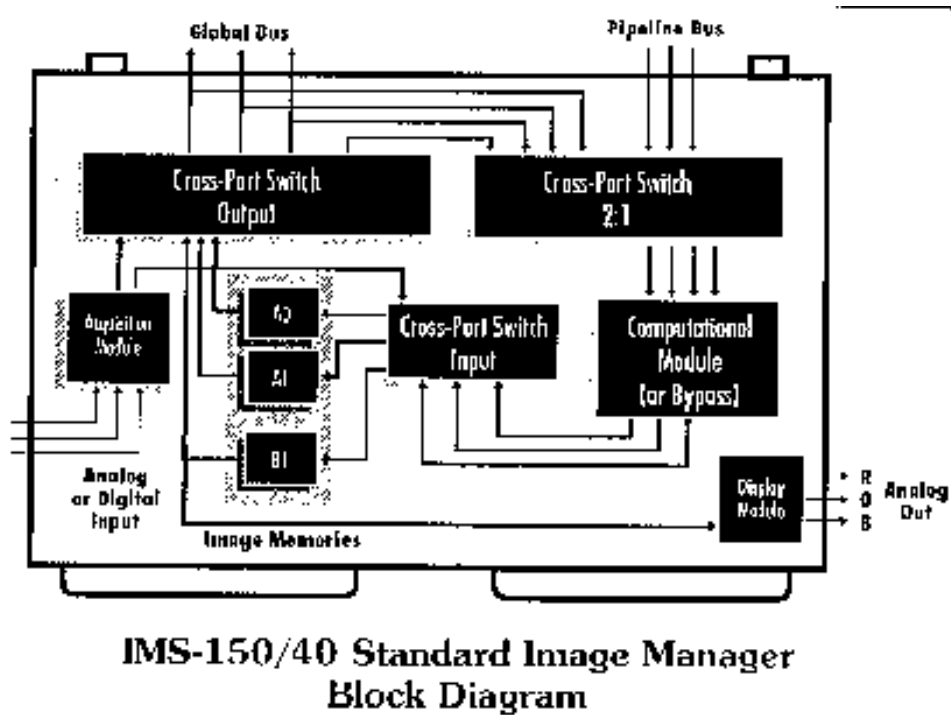


Figure 5-4: ITI IMA

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 $(768 \times 572 \times 3 \times 25 = 32.94 \text{ Mbytes/second})$. 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.

B.8.2 Image Acquisition System and Hardware

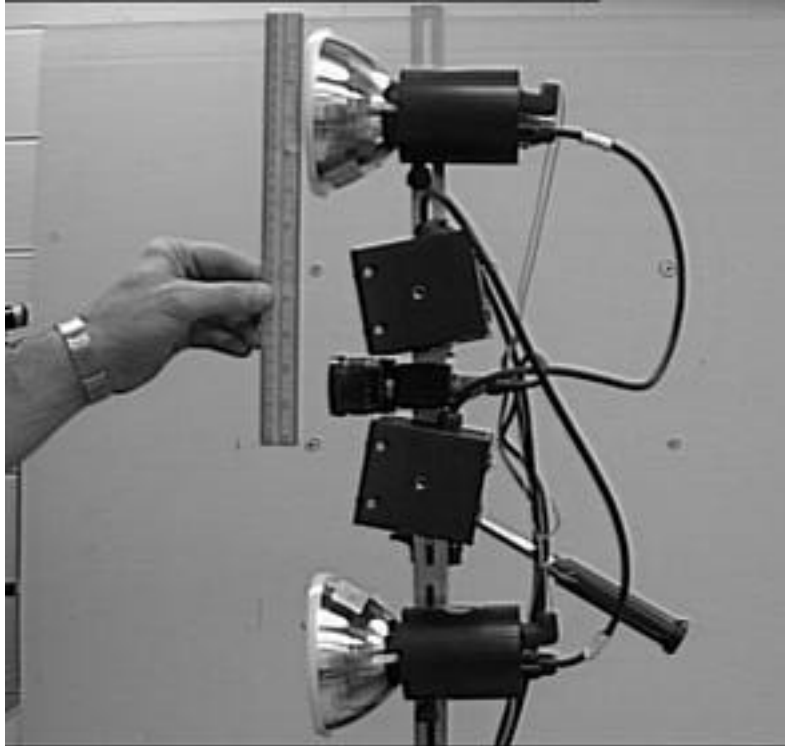


Figure 5-5: Basic Imaging Head (Dot Projector Not Shown)

B.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.

B.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.

B.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:

B.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.

B.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.

B.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.

B.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.

ARIES #2 Vision System

Brecht Carver, RJ Schalkoff

revised by RJS 9/8/97

for internal use only;

NOT FOR DISTRIBUTION

Contents

| | | |
|----------|-----------------|----------|
| 1 | Hardware | 2 |
| 2 | Software | 6 |

List of Figures

| | | |
|---|---|---|
| 1 | Placement of vision components on ARIES panning unit. . . . | 4 |
| 2 | ARIES Flash controller schematic diagram. | 5 |

List of Tables

| | | |
|---|---|---|
| 1 | The components of the ARIES imaging subsystem and the locations on the robot. | 3 |
| 2 | The cables for the ARIES imaging subsystem. | 3 |
| 3 | Descriptions of the routines in vision.cpp. | 7 |

Overview

The ARIES vision system was designed to provide the ARIES robot with the capabilities of inspecting waste-storage drums for rust, dents, and tilting. This document describes the construction, installation, and testing of the ARIES vision system. It also provides a summary of the software components.

1 Hardware

The hardware components of the ARIES vision system and their mounting locations are listed in Table 1. The precise locations of the components mounted on the panning unit are shown in Figure 1. The cables necessary for interconnecting the components are listed in Table 2. Note that, other than cables, the only component which must be manufactured is the flash controller. This TTL circuit is controlled by the ITI TTL outputs and in turn controls the lasers and flash heads. A circuit schematic is provided in Figure 2.

Many of the components have hardware settings which should be set prior to operation. With the exception of those for the color camera, the settings are listed in the third column of Table 1. The color camera has parameters which are set via the front panel switches and others which can only be accessed via a displayed menu system. The front panel switches should be set as follows:

- The cluster of four toggle switches should be set, starting with the GAIN and proceeding in a clockwise direction, to OFF, HOLD, MANU, and CAM.
- The toggle switch labelled ELC should be set OFF.
- The two rotary color gain dials should be initially centered. Note that these will be altered later.

Unfortunately, the camera will only display the menu system to either the S-Video or the Video Out ports on the rear of the camera body, neither of which are connected to the acquisition module of the ITI system. Thus, when these need to be altered (or checked), the BNC normally connected to the B/W camera should be extended and connected to either Video Out

Table 1: The components of the ARIES imaging subsystem and the locations on the robot.

| Component | Location on Robot | Setting |
|--|-------------------|-------------|
| Panasonic GP-US502E Color Camera Head | Panning Unit | |
| Panasonic GP-US502E Color Camera Body | Tower | (see text) |
| Cosmicar C60607 6mm Lens | Color Camera | f16, 1m |
| Pulnix TM6-CN B/W Camera | Panning Unit | 1.0,MGC,FLD |
| Cosmicar C60607 6mm Lens | B/W Camera | f2, 1m |
| Edmund Scientific E43141, 690nm filter (25mm) | TM6-CN Camera | |
| 2 Lumedyne #008 Modular Flash Heads | Panning Unit | |
| 2 Lumedyne #065Z 200WS Power Packs | Tower | FST, 50WS |
| 2 Lasiris SNF-XXX-690-30, 690 \pm 2nm Lasers | Panning Unit | |
| 1 Lasiris SLH-513D 13-dot head | Upper laser | |
| 2 Lasiris M-75 Mounting Brackets | Panning Unit | |
| Flash Controller | Robot Base | |
| ITI IMPCI Image Manager | PCI Bus Computer | |
| ITI CMC-PCI-CLU Module Controller | PCI Bus Computer | |
| ITI CMHF Histogram Module | CMC-PCI slot #1 | |
| ITI CMMEM-16 Memory Module | IMPCI CM slot | |
| ITI AMCLR Color Acquisition Module | IMPCI AM slot | |
| FBV15040-VME-2 Frontplane Bus | CMC-PCI to IMPCI | |

Table 2: The cables for the ARIES imaging subsystem.

| Name | From | To | Cable / Signal |
|------------------------|-------------------|-------------------|-------------------------|
| Power Pack Power #1 | +12VDC Source | Power Pack #1 | +12VDC, 28A |
| Power Pack Power #2 | +12VDC Source | Power Pack #2 | +12VDC, 28A |
| Flash Power #1 | Power Pack #1 | Flash Head #1 | Lumedyne 034P |
| Flash Power #2 | Power Pack #2 | Flash Head #2 | Lumedyne 034P |
| Flash Control #1 | Flash Controller | Flash Head #1 | TTL |
| Flash Control #2 | Flash Controller | Flash Head #2 | TTL |
| Laser Power | Flash Controller | 2 Lasers | +5VDC, 250mA |
| Flash Controller Power | +5VDC Source | Flash Controller | +5VDC, 500mA |
| IMPCI Strobe #1 | IMPCI | Flash Controller | TTL |
| IMPCI Strobe #2 | IMPCI | Flash Controller | TTL |
| Color Camera Power | +12VDC Source | Camera Controller | +12VDC, 1A |
| Color Camera Signal | Camera Controller | Camera Head | Panasonic GPCA-63 |
| Color Camera Video | IMPCI | Camera Controller | 4 - 75 Ω Coaxial |
| B/W Camera Power | +12VDC Source | B/W Camera | +12VDC, 250mA |
| B/W Camera Video | IMPCI | B/W Camera | 75 Ω Coaxial |

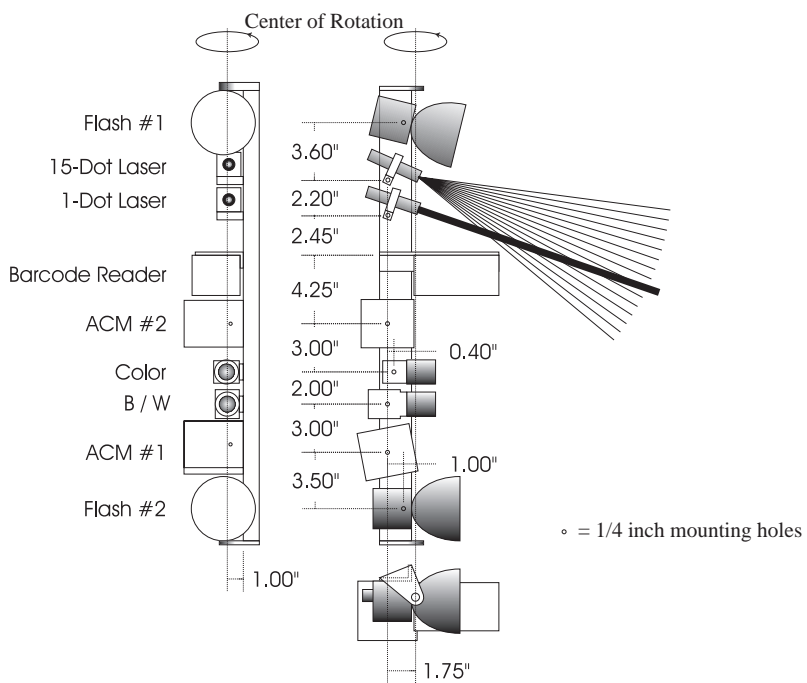


Figure 1: Placement of vision components on ARIES panning unit.

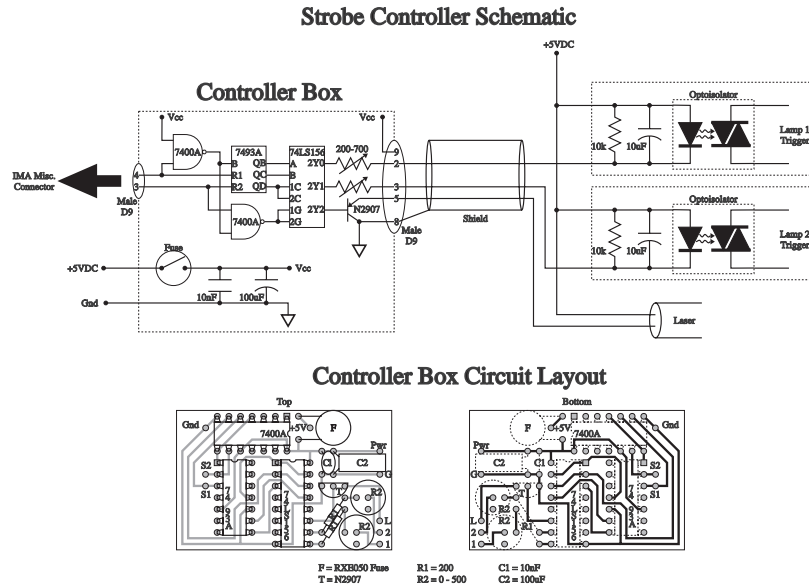


Figure 2: ARIES Flash controller schematic diagram.

port. To view the menu, run program Test, enter the command 'gl', then hold the recessed button labelled PAGE (front panel of color camera) down for two to three seconds. The camera's menu should display on the ITI's output monitor.

Using the four recessed buttons (PAGE, ITEM, <, >), set the menu parameters, from top to bottom, to OFF, OFF, OFF, MANU, INT, MANU, ABC, and USER. When the cursor is on USER, push the PAGE button to enter a second menu. Again using the four buttons, set the parameters, from top to bottom, to OFF, 4.0V, FLD, center, center, center. Finally, move the cursor to RET, press > to move it to END, then press PAGE to exit the menu system. These parameters need to be set only once after a camera is purchased (or returned from servicing).

The final settings for the color camera are the black balance and the white balance. Both of these set after the camera has been on at least 15 minutes and should be redone periodically (multiple months or during routine servicing) and anytime the camera has been off for a long period of time (multiple days). The black balance is accomplished by blocking the lens then momentarily setting the toggle switch labelled AWC/HOLD/ABC to

ABC. Once the red LED beside the switch stops blinking, the black balance is set.

Setting the white balance is accomplished by adjusting the two knobs, labelled R (red) and B (blue), on the front panel of the camera body until the image of a neutral gray object (the Delta #22010 18% Gray Card, readily available at most photography stores, is particularly suited) illuminated by a flash has a low saturation value (< 30 over the entire object). Both acquiring and reading the saturation values are accomplished using the program Test. Command 'ah' acquires a HSI flash image whereas the command 'v' allows one to peruse (the cursor can be moved about the image using the keys h [left 1 pixel], j [down], k [up], l [right], H [left 20 pixels], J, K, and L) the values of the image in the ITI IMPCI memory.

After the hardware components are installed and all hardware settings correctly set, the program Test should be used to verify the functionality of the individual components. Test provides the ability to test each subsystem separately and in unison. Once all components are operational, the lasers pointing directions should be set using program LaserCal.

2 Software

The primary software for the ARIES vision system consists of four routine libraries: vision, doe, laser, and util. They are provided as C++ source code, each consisting of a .cpp and a .h file. Note that, only those routines in vision.cpp are directly called by the mission handler. The other libraries are called by vision or by ancillary programs (Test, Lasercal, and Train).

vision.cpp

The routines contained in vision.cpp provide the interface between the mission handler and the vision system. Table 3 lists these routines and provides a brief description for each.

Table 3: Descriptions of the routines in vision.cpp.

| Routine | Description |
|------------------|--|
| InitializeVision | Initializes ITI cards Loads rust-finding parameters and laser calibration data Allocates memory |
| CloseVision | Deallocates memory |
| vision | Scans lasers to locate and determine number and size of drums Images each drum and inspects for rust Detects dents using laser scan Detects tilt based on two (lower and upper) laser scans |
| vision_save_a | Saves image of the furthest drum from ITI to disk |
| vision_save_b1 | Saves image of the nearest drum from ITI to memory |
| vision_save_b2 | Saves image saved by vision_save_b1 to disk |
| do_color_cal | Uses gray-patch on robot to calibrate color camera |

ARIES VISION SYSTEM

CALIBRATION AND TRAINING GUIDE

Color/B&W balance: Wait for the cameras to warm up.

Camera Focus: Between 0.5 and infinity.

B&W aperture: 4.0.

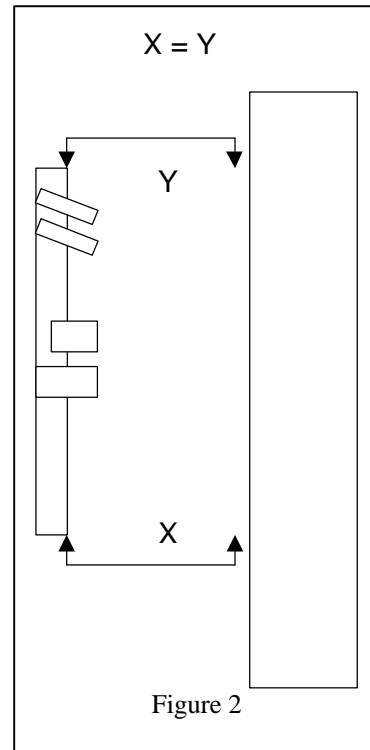
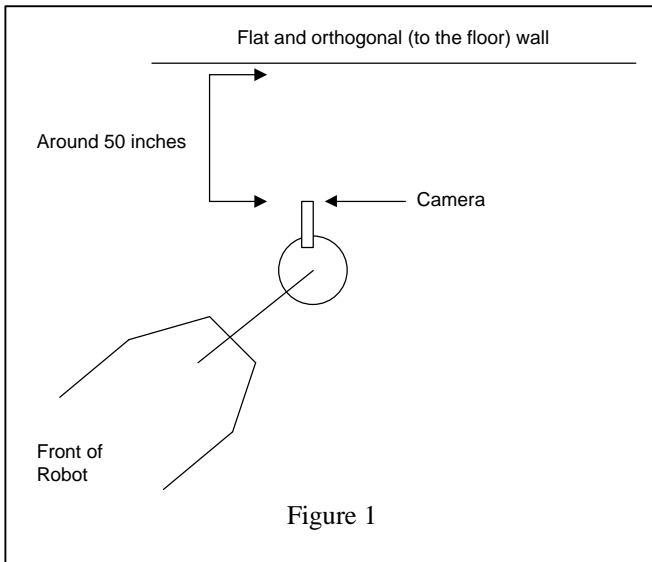
Color aperture: 16.0.

Be sure to focus lasers.

Calibrating the Lasers - Lasercal.exe

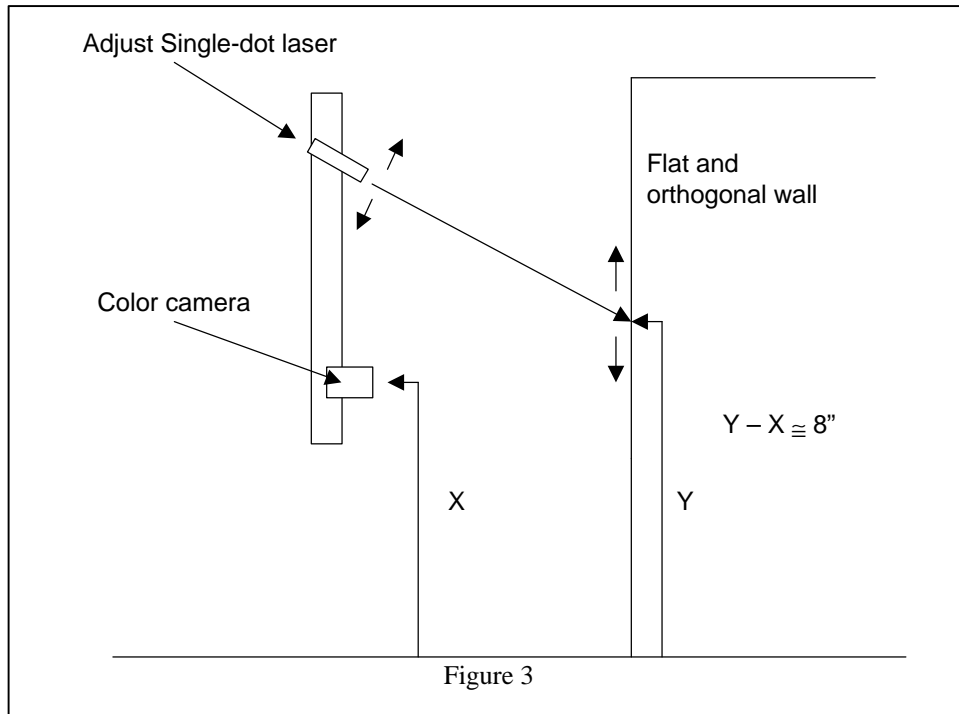
Step 1 - Position ARIES

- Deploy the lift to Level Three using the "Test" application ("T3" command).
- Pan the camera unit to ~120 degrees with respect to the robot using the "Test" application.
- Using the joystick, manually drive the robot until the front of the color camera is about 50 inches from a flat, smooth, relatively wide, and orthogonal to the floor wall or surface (Figure 1).
- Verify that the surface is orthogonal (Figure 2).



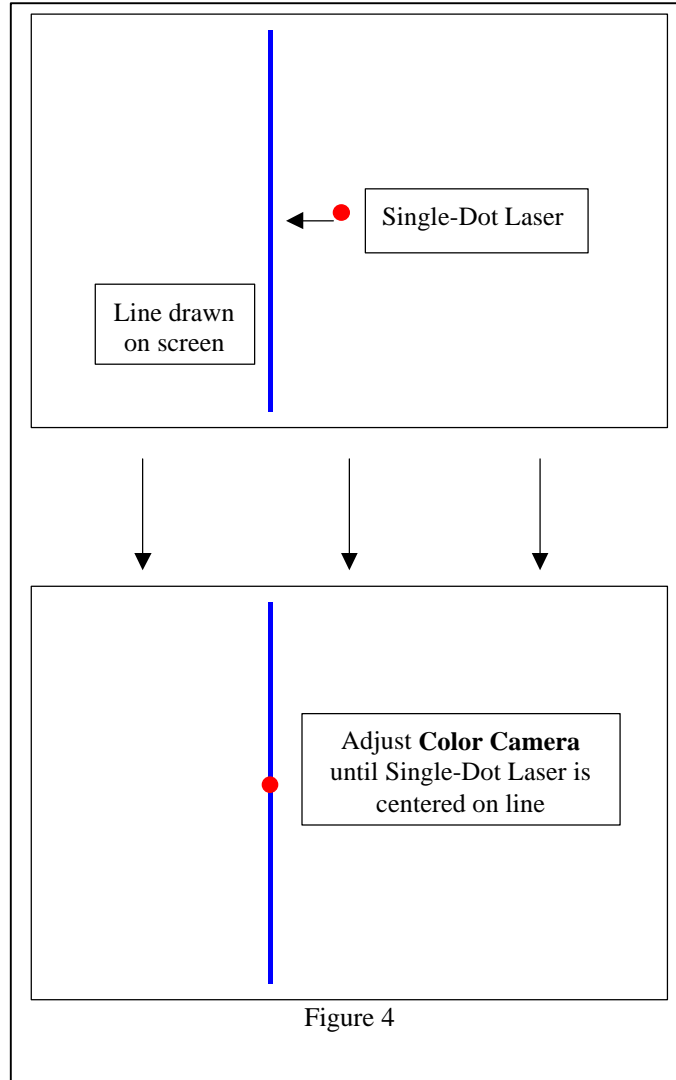
Step 2 - Position Single Dot Laser

- Measure the distance from the color camera to the floor (X).
- Measure the distance from the Single-Dot Laser (as opposed to the multi-dot diffraction grating laser) on the wall to the floor (Y).
- The difference between the two should be around eight inches ($Y - X \approx 8''$).
- See Figure 3.
- At this point, it's also a good idea to make sure both the lasers are in focus.



Step 3 - Position Color Camera

- Observe position of Single-Dot Laser relative to line drawn on the screen.
- Adjust the position of the **color camera** until the laser dot is centered on the line (Figure 4).



Step 4 - Position Black and White Camera

- Observe position of Single-Dot Laser relative to line drawn on the screen.
- Adjust the position of the **black and white camera** until the laser dot is centered on the line (Figure 5).
- Note: Consistency of Steps 3 and 4 is very important!

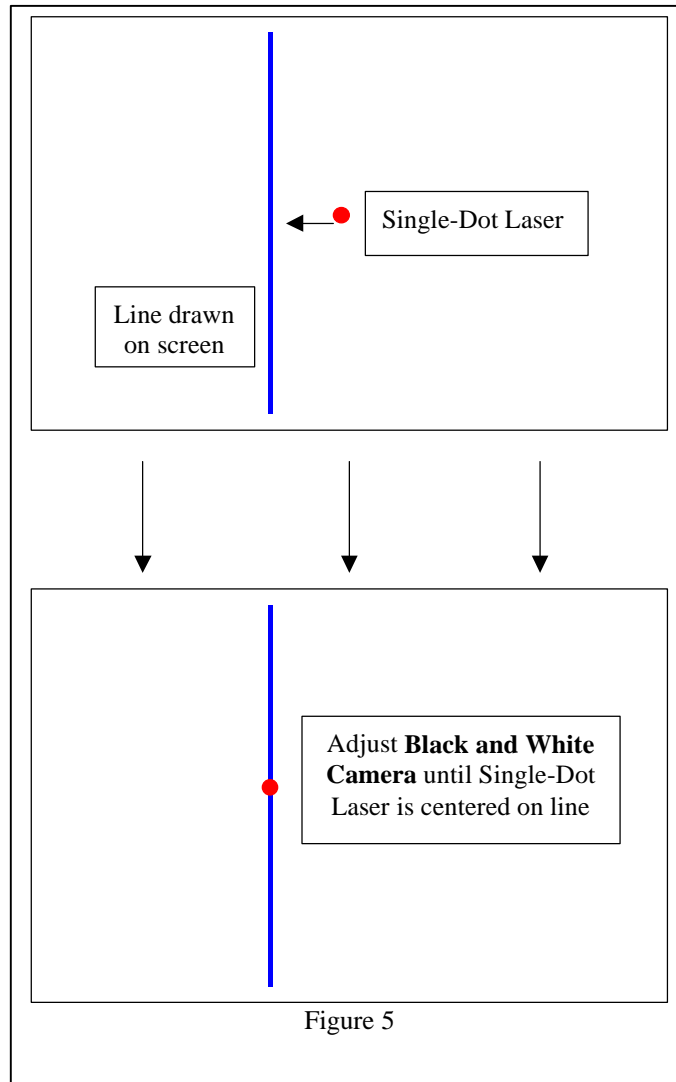
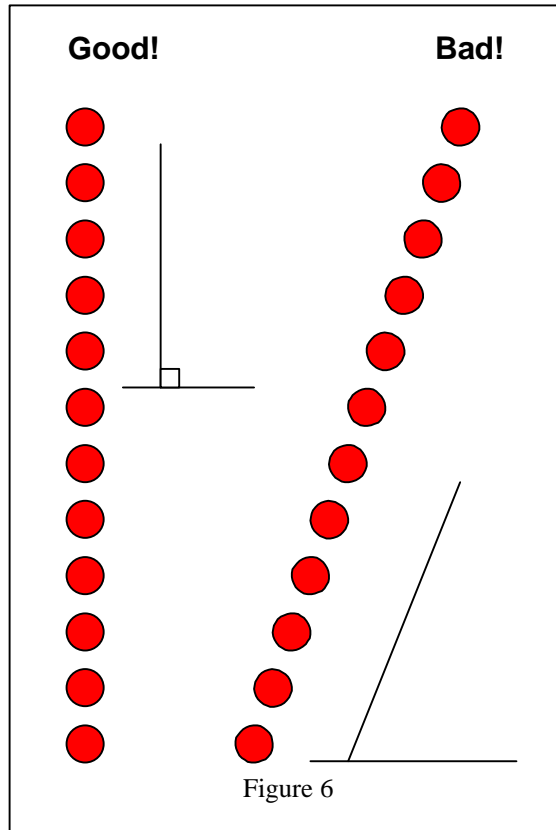


Figure 5

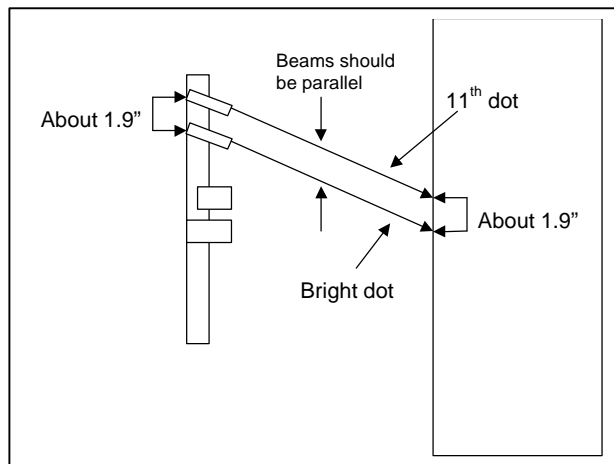
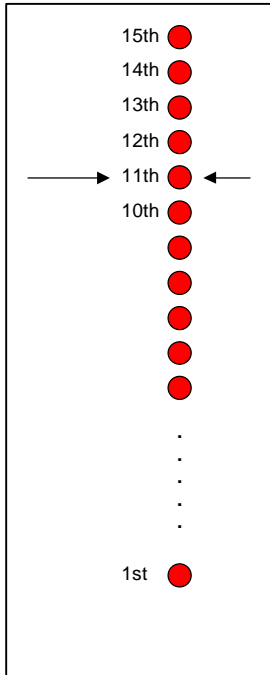
Step 5 - Align the multi-dot laser (rotational)

- Observe the alignment of the Multi-Dot Laser. Rotate the Laser until the dots are perpendicular to the plane of the floor (Figure 6).



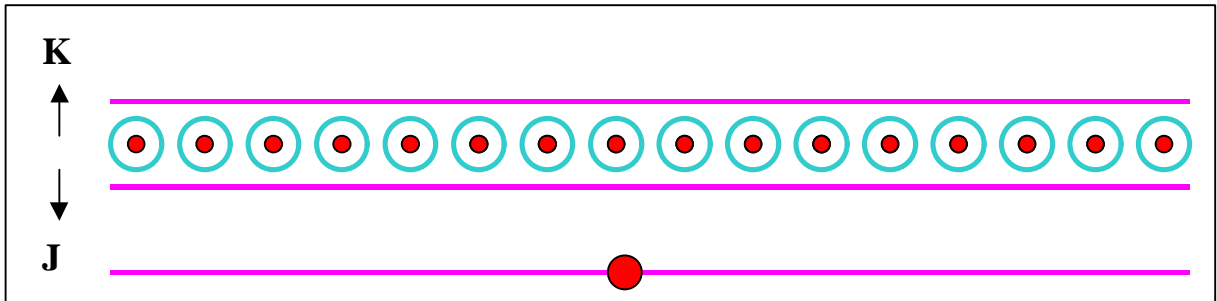
Step 6 - Align the multi-dot laser (lateral)

-



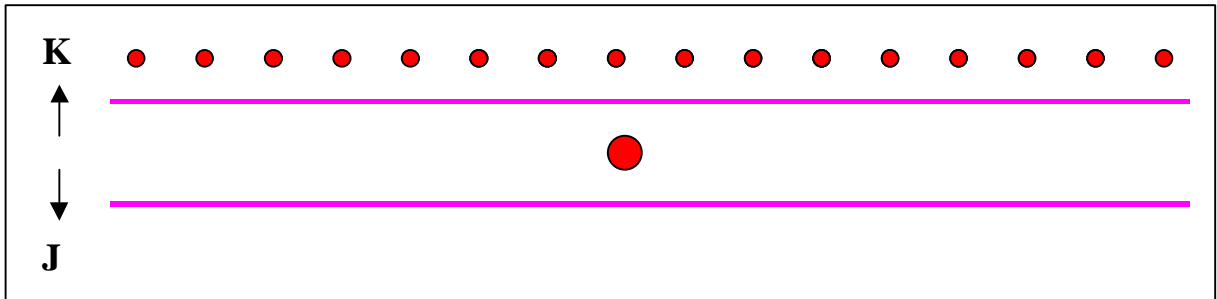
Step 7 - Rotate the Multi-Dot Laser

-



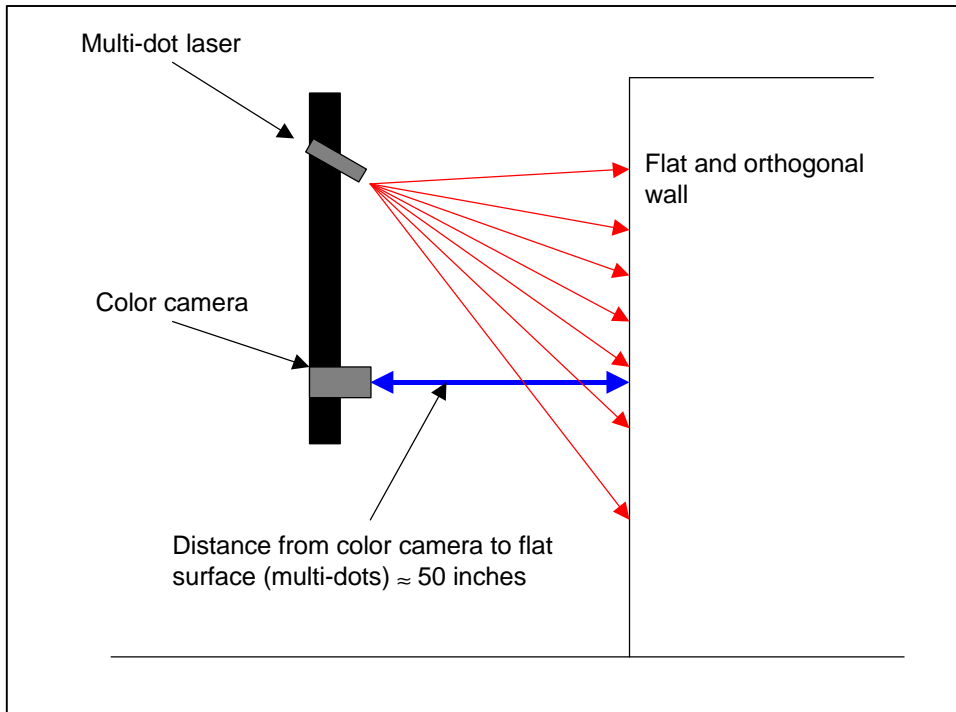
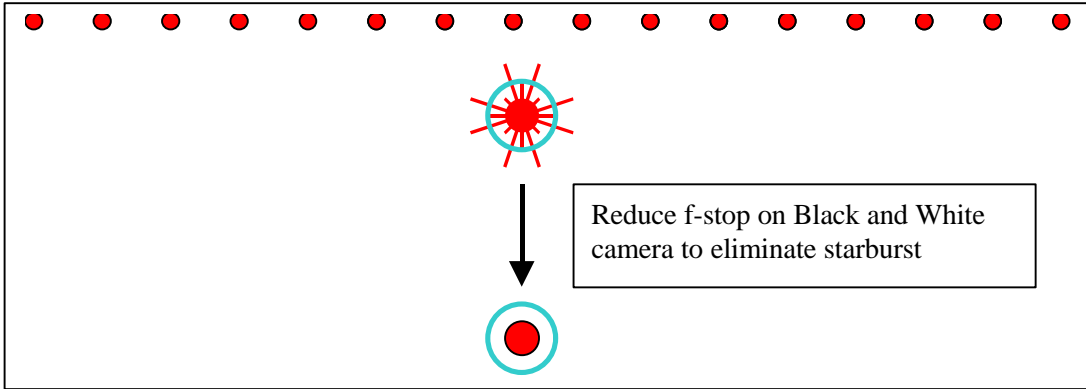
Step 8 - Pan Single-Dot Laser

-



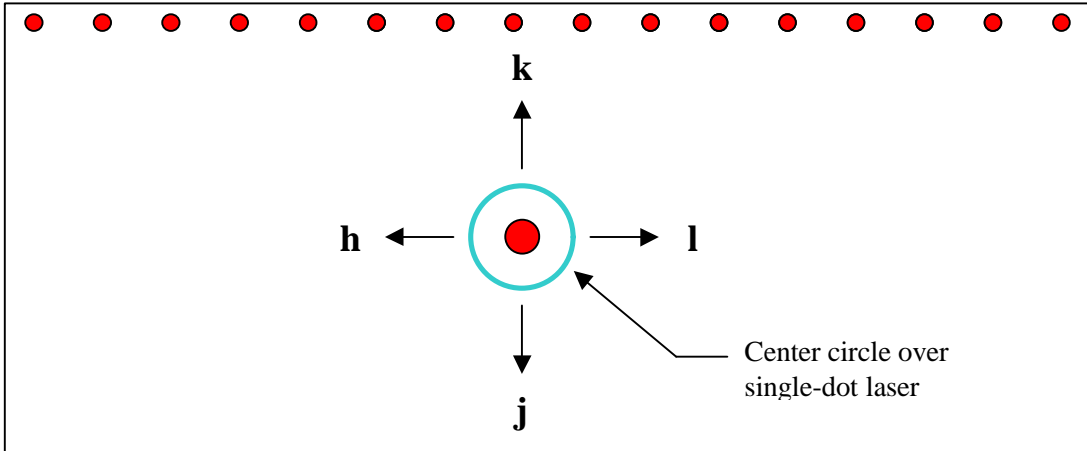
Step 9 - Reduce Black and White camera f-stop

-



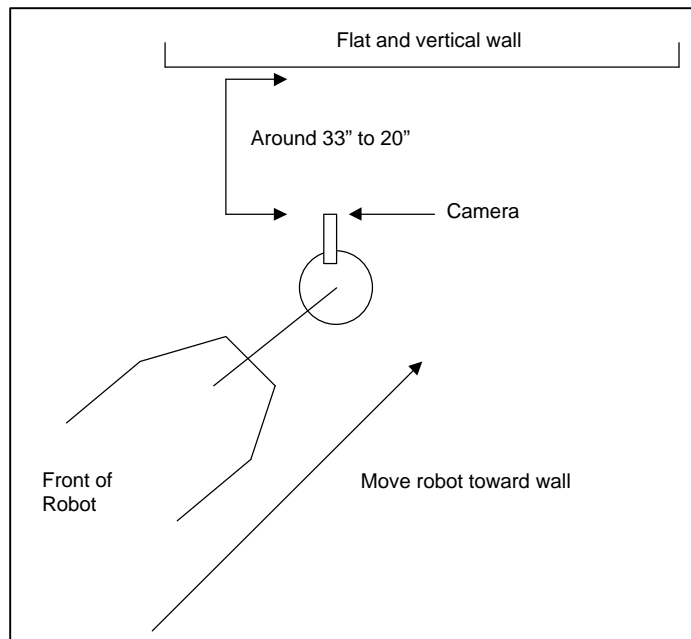
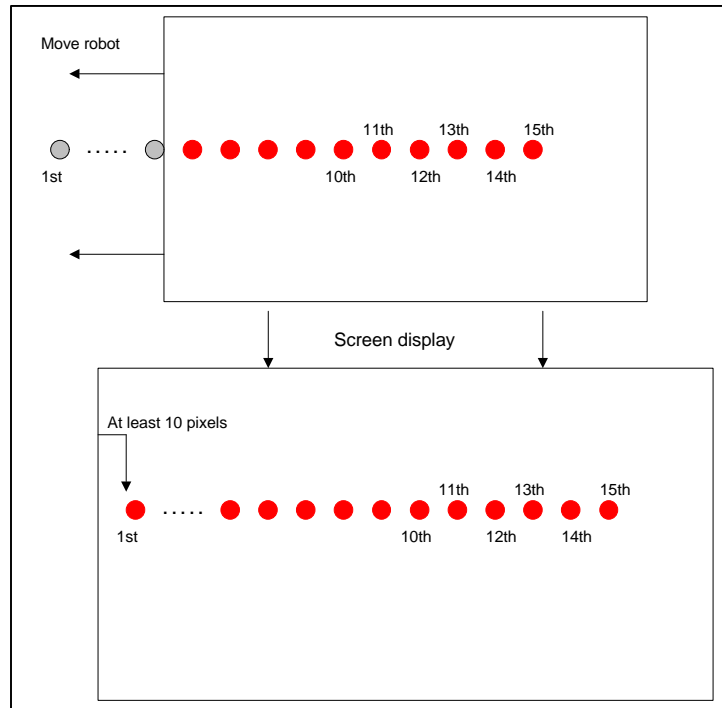
Step 10 - Locate Single-Dot

-



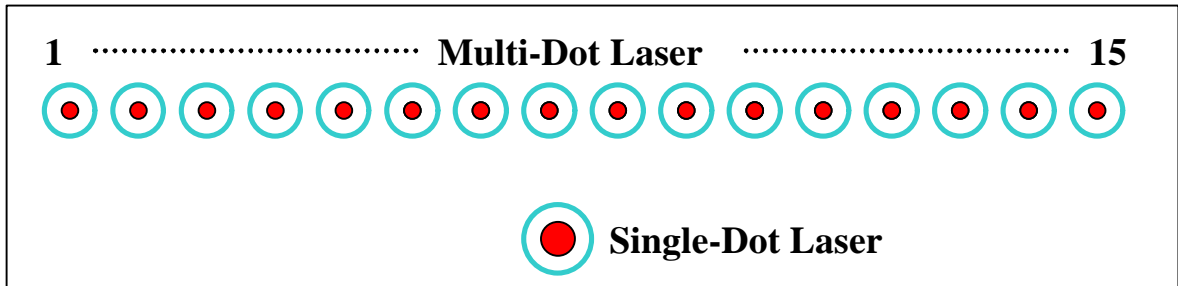
Step 11 - Drive ARIES towards surface

- Must have at least 10 dots in view (in the screen).
- Measure distance from color camera to multi-dot laser. This distance should always be between thirty-three and twenty inches.



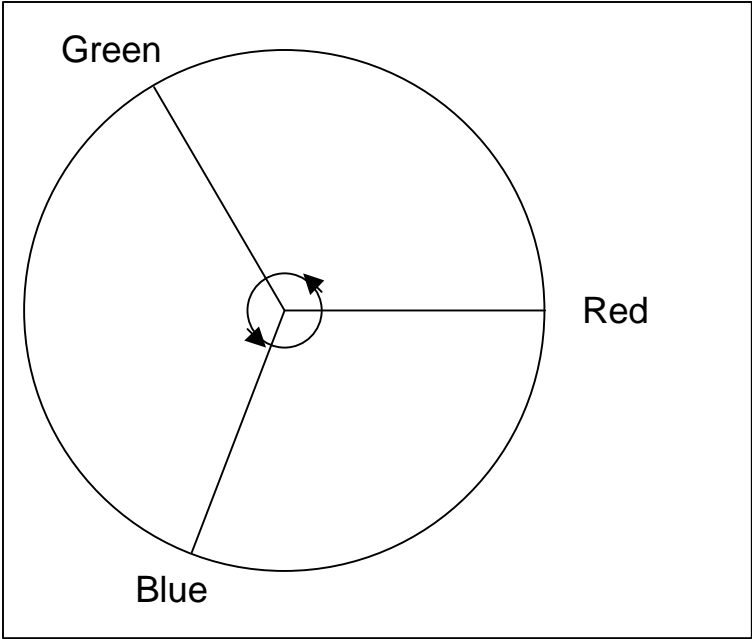
Step 12 - Check alignment

- Must have at least 15 dots in view (in the screen).
- Verify that the distance from the color camera to the multi-dots on the surface is between twenty and thirty-three inches.



Step 13 - Drive ARIES towards surface

- Must have at least 10 dots in view (in the screen).



Training

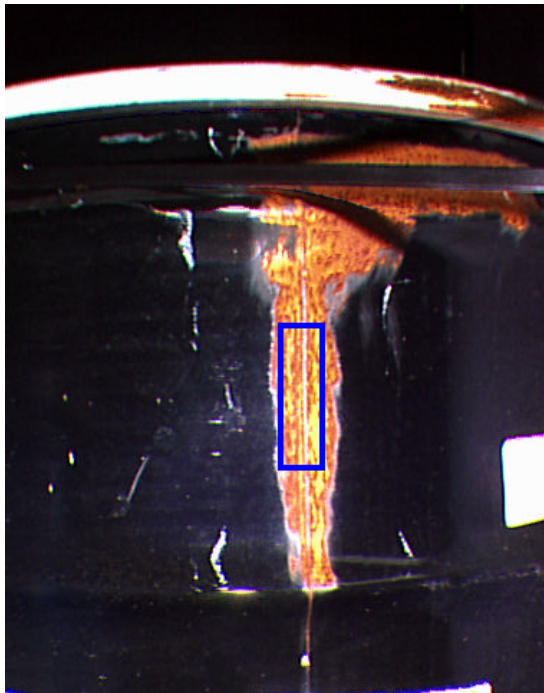
Need to run "filelist.bat" to get a list of image files in a file.

Train.exe needs to run in a 129x97 (raster 8x8) Dos box.

Need to name (or rename) to "rust.lut".

Command "l" loads image.

"Box-in drum"



Use mouse to draw box in "rusty" areas (or in anything else that you want to identify, for example, if you wanted to find all drums with green stickers, you could train for them).

Command "p" (positive) identifies rust.

Command "n" (negative) would be used to eliminate stickers (or whatever).

NEVER mark black negative! White/Yellow/Steel would be okay.

Command "mr" (mark rust) identifies all perceived rust in image.

Command "u" undoes last "n" or "p" command.

Parameter "d" has to do with the number of neighboring pixels that are bad before a given pixel is also considered bad.

Commands "N" and "P" display the negative and positive sets, respectively.

Appendix C

ARIES: An Intelligent Inspection and Survey Robot

ELECTRICAL SYSTEMS

Cybermotion Incorporated

C. ELECTRICAL SYSTEMS

Synopsis of energy storage units for mobile Robotics

Dr. Jerry Hudgins, C. Richard Shoop and Erica Benjamin

Abstract--The period over which mission critical tasks can be performed in a mobile platform will depend on the amount of energy available. In this paper, the authors will give a synopsis of present and future technologies in the area of energy storage, which are available for mobile Robotics. Present, *Prevailer* batteries are of sufficient capacity for the original design, however with the addition of power intensive devices and computers a different strategy will be required to meet future needs.

C.1 INTRODUCTION

Intelligent autonomous mobile robots are being used in surveillance, inspection and monitoring environments. The use of these robots can reduce human exposure in hazardous environments and minimize manpower requirements. Presently, research is being conducted to implement a system for inspecting low level waste containers.

The *Cybermotion* platform uses two 12V, 85A*h (34W*h/kg) sealed lead-acid batteries, the *Prevailer*, produced by *InterAction Battery Corporation*. These batteries are connected in series to produce a 24V supply. The *Prevailer* battery is orientation independent, which is desirable due to space constraints. This energy storage device possess a form factor of 12.75" x 6.75" x 9.875". Presently, the robot is capable of 20h of surveillance with standard *Cybermotion* equipment.

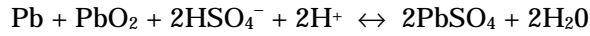
Our mission requires extra equipment to inspect waste containers and process received data. A larger amount of energy will be required to realize these goals. To provide this energy, a search of current and future battery technology has been completed and the following report summarizes this search.

C.2 THE LEAD-ACID BATTERY

The most common electrochemical storage device marketed to date is the lead-acid battery. This technology is theoretically capable of energy densities of 161 W*h/kg (see figure 1) [1], with attainable values close to 41 W*h/kg. This theoretical energy density value was derived from complete reaction equations, considering that the real components are pure elements not chemical compounds or alloys. Since this technology cannot operate in pure sulfuric acid, due to damaging corrosion and low conductance, water must be added.

The water and excess acid result in an increased weight, reducing the energy density (see figure 2) [2].

reaction equation:



weight of the reaction participants per formula-turnover

$$[(207.2 + 239.2 + 2) * 97.00] + (2 * 1.008) = 642.4 \text{ g}$$

transformed amount of energy

$$2 * 96,500 \text{ A*s} = 53.61 \text{ A*h}$$

presumptive cell equilibrium voltage of

$$E_0 = 1.928 \text{ V} \quad (\text{activity of the acid} = 1 \text{ mol/l})$$

the (hypothetical) energy per weight amounts to

$$(53.61 \text{ A*h})(1.928 \text{ V}) / 0.6424 \text{ kg} = 160.9 \text{ W*h/kg}$$

Figure 1: Theoretical storage ability (lead-acid)

Also, additional weight results from the products of the reaction not being 100% convertible. Structural supports of the active material and the separators increase the nonproductive weight and the resulting energy density decreases [2].

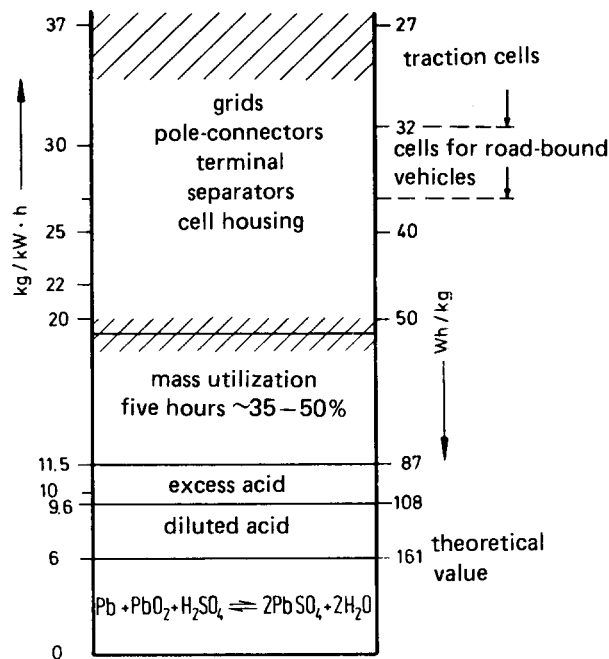


Figure 2: Relative weight vs. energy density

Not much room is left for a value-added design of the lead-acid battery. However, slight performance improvements are being realized by many manufactures as the demand increases for new higher energy density batteries in the automotive market (electric cars).

C.3 PARAMETERS FOR OUR SEARCH

For our search the following parameters will be used to narrow the field of interest.

- a) Low internal resistance (quick recharge and deep discharge)
- b) High energy density ($>34 \text{ W}\cdot\text{h}/\text{kg}$)
- c) Off the shelf availability (presently manufactured and sold)
- d) Electrical specifications
 - Terminal Voltage = 12V or 24V
 - Minimum A*h Capacity = 85A*h
- e) Mechanical specifications
 - Form factor(Prevailer) $\approx 12.75'' \times 6.75'' \times 9.875''$
 - Sealed case (orientation independent)

C.4 THE LEAD-ACID BATTERIES

As previously noted, the *Prevailer* battery is available in a 12V, 85A*h model 8G27. This battery utilizes a totally sealed recombinant system and can be mounted in almost any orientation. There are no corrosive acid fumes or potential for electrolyte spillage under normal operating conditions. The *Prevailer* has a low self discharge rate and can be stored for 24 months with very little effect on capacity. Low internal resistance allows a 90% recharge rate in 3.5 hours. Also, this battery will provide 400, 50% depth of discharge cycles. The form factor is 12.75" x 6.75" x 9.875" with a weight of 64 lbs.

The *Union* battery, marketed by *Synergistic Battery, Inc.*, offers a 12V, 100A*h (37.5 W*h/kg) model PW121000. This battery has a low internal resistance, 3 mΩ at 26°C, allowing fast recharge and deep discharge. Cycle lives of 250 charge/discharge cycles at 100% DOD (Depth Of Discharge) and greater than 600 charge/recharge cycles at 50% DOD are possible. The form factor is 13.28" x 6.96" x 8.56", which should be a workable substitute for the *Prevailer* [3].

The *Guardzman*, by *Douglas Battery*, offers a 12V, 95A*h (38.8 W*h/kg) model DG12-95. This battery will self-discharge at 3% per month at a temperature of 77°F. Cycle lives of 120 charge/discharge cycles at 100% DOD and greater than 600 charge/recharge cycles at 50% DOD are possible. The form factor is 12.60" x 6.6" x 8.13" with a weight of 63 lb., this also is a possible substitute for the *Prevailer* battery.

It should be noted that *Douglas* offers a 12V, 160A*h (DG12-160) and 200A*h (DG12-200) battery. The form factors are 20.7" x 8.7" x 8.86" (weight 132 lbs.) and 20.7" x 10.95" x 8.66" (weight 159 lbs.), respectively [4].

Power-Sonic Corporation offers a 12V, 80A*h (38.4 W*h/kg) model PS-12800. Cycle lives of 250 charge/discharge cycles at 100% DOD and 500 charge/discharge cycles at 50% DOD are possible. The battery's low internal resistance allows discharge currents of ten times the rated capacity. The form factor is 12" x 6.6" x 8.2", with a weight of 55 lbs.

Also, *Power-Sonic* offers a 12V, 100 A*h battery model PS121000. This battery has a form factor of 13.07" x 6.85" x 8.43 and a weight of 70.5 lbs. This battery could be substituted with modifications to the Cybermotion platform [5].

The final lead-acid battery is manufactured by *Electrosorce, Inc.* and offers valve-regulation, with high specific energy (50 W*h/kg). This newly developed battery is for the electric car industry and has a projected life span listed as 80,000 miles. This battery has higher power capability 300 W/kg vs. 90 W/kg (conventional lead-acid) @ 20% charge [6].

The fundamental importance of this technology is the fiber-glass reinforced lead wire used as a grid material in the woven-mesh, bi-polar grids. Sheathed with a high-density lead-tin alloy, this coaxial wire reduces the weight of the positive plate grid and extends its useful life. The fiber-glass core of this composite wire prevents growth of the positive grid and provides a dimensionally stable foundation for the positive plate.

These improved positive-grid attributes allow the design of thin positive plates capable of long-life under deep-cycle operation. In turn, such plates allow the design of sealed, recombinant cells and modules capable of significantly higher power and energy per unit of battery weight compared to conventional lead-acid batteries [7].

This battery has some good attributes such as the ability to recharge to 99% capacity in 30 minutes (low internal resistance), sealed valve-regulated technology, and high specific energy. However, the dimensions of 29" x 5" x 5.22" and weight of 60 lbs. are outside our target form factor.

The depth of discharge (DOD) will effect the life time of the battery. Typically manufactures will provide curves to estimate the number of cycles that will be realized over a given temperature range of operation and for a typical charge/discharge ratio (see figure 3) [8].

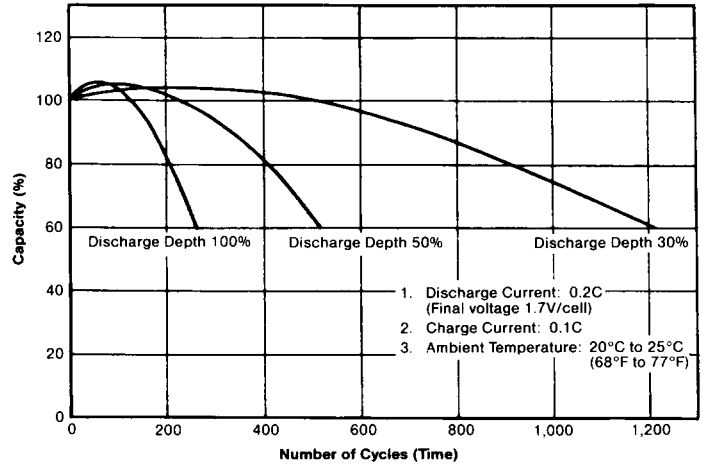


Figure 3: Number of cycles vs. DOD (Power Sonic)

The operating temperature range, for the lead-acid battery, is approximately 0 to 120° F. It should be realized that the ambient temperature will effect the capacity rating on the battery (see figure 4) [8]. When operating outside the range of room temperature the manufacture specifications should be consulted.

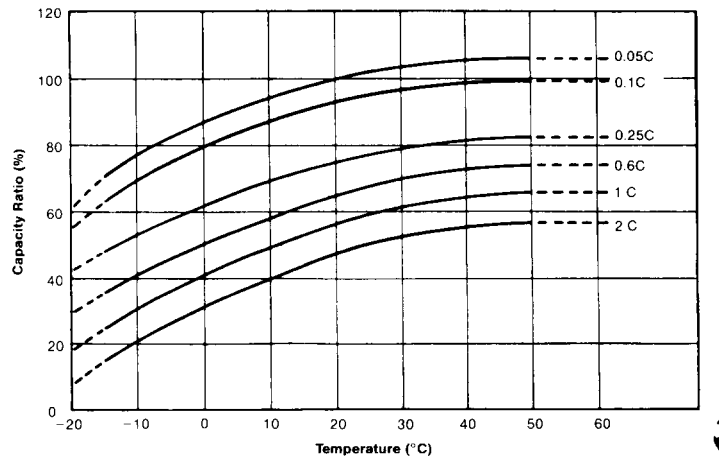


Figure 4: Ambient temperature effect on rated capacity (Power Sonic)

All of the lead-acid batteries described are totally sealed, maintenance free, and may be mounted in any orientation. There are no corrosive acid fumes or potential for electrolyte spillage under normal operating conditions. All batteries are equipped with a pressure safety valve to relieve the excessive gas pressure (nominally 2 to 6 psi.) caused by abnormal charging. The one way valve ensures that no air gets into the battery where the addition of oxygen would react with the plates causing internal discharge.

C.5 OTHER BATTERY TECHNOLOGIES

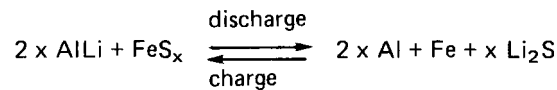
Nickel-Iron Batteries

Eagle-Picher, Inc presently offers a 6V, 215 A*h (53.1 W*h/kg) battery. This battery has a cycle life of approximately 900 charge/discharge cycles at 100% DOD. The form factor is 10.25" x 7.07" x 8.86" and it weighs 54 lbs. Water levels must be maintained at each charge making this technology orientation dependent and maintenance intensive. The lack of a sealed system can lead to environmental hazards during operation and charging. Charging at rates above manufacture specifications will cause excess hydrogen gas to form, leading to a possible explosion. Therefore, special charging/monitoring equipment must be used, increasing the cost above the purchase price of \$1400. However, unlike lead-acid, disposal of a spent nickel-iron battery is environmentally benign [9].

Lithium-Iron Disulfide

Westinghouse Electric Corporation produces a 24V, 30A*h (75 W*h/kg) battery. The low internal resistance of this battery allows for continuous high power discharge and rapid charge cycles. This is a maintenance free battery, totally sealed, and is orientation independent. The form factor is 8" x 8" x 9" and weighs 30 lbs.

In order to sustain functionality, the electrolyte has to be kept in a molten state, which presupposes a temperature of 380 to 500°C. The design of the cell looks similar to a cell operating at room temperature (see figure 5). This symmetrical cell has an iron disulfide electrode placed in the middle of a molybdenum current conductor and an yttrium oxide texture. On both sides is a boron nitride felt, which is soaked with the molten salt electrolyte, lithium-aluminum electrodes and a high-grade steel housing. The housing is the negative terminal. The overall reaction can be described by the following equation [10]:



The volatile reaction of lithium with water requires the operating environment be void of water, to preclude an accident. Under normal operating condition this should not be of concern. Research within this technology is focused on finding a less expensive method of manufacturing the boron felt, reduce positive electrode swelling during discharge, and lowering the cell voltage during charge cycles [11].

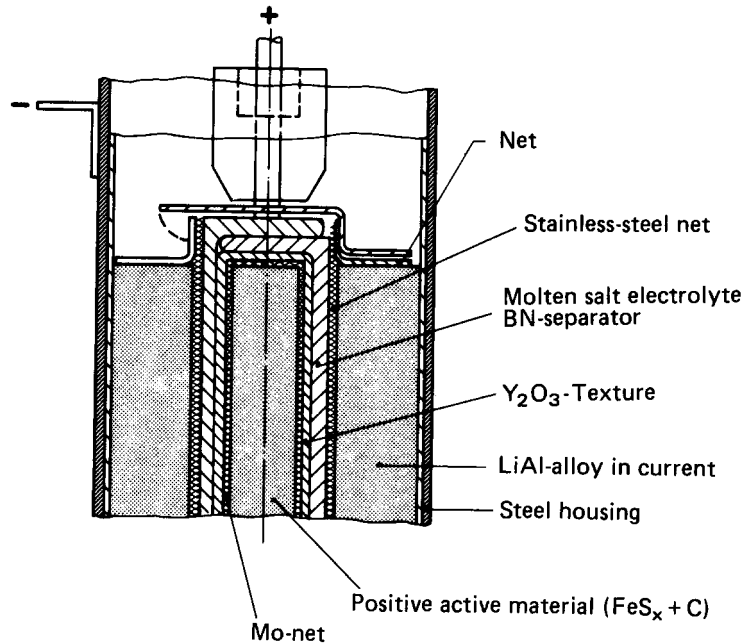


Figure 5: Lithium-Iron Disulfide battery [10]

Nickel-Metal Hydride

Ovonic Battery Inc produces batteries with an increased current capacity of 30-50% over nickel-cadmium or lead-acid batteries. Most of their products are used in small consumer electronics and laptop computers (low A*h ratings). The service life of these batteries has increased by approximately 50% over their counterparts. Charging systems for this technology require stringent electronic control to avoid a possible explosion due to overcharging and limitation of cycle life due to prolonged trickle charge.

Nickel-metal hydride batteries exhibit no memory effect, have longer lifetimes, and are void of highly poisonous cadmium. This will allow for more environmentally friendly disposal of fewer less toxic batteries [12].

Zinc-Bromine

Johnson Controls is producing a 48V, 126 A*h (60 W*h/kg) battery that operates near room temperature. In this system, an active stack of graphite plates and a reservoir of electrolyte (see figure 6) [13], minimizes self-discharge and provides optimal output in high power applications. A network of pumps, pipes and valves are used to transport electrolyte to the plates and provide an ideal coolant. The non-sealed nature of this battery should be corrected when the battery is commercially available (five years).

During recharge of a battery zinc is deposited on the cathode, and bromine is generated at the anode. In a drop configuration bromine reacts with the complex activating substance guided along the electrode and the drops deposit on the right side of the container. During discharge the bromine complex deposits bromine into the electrolyte. Therefore, dissolved bromine on the anode and zinc on the cathode are available for the electrochemical reaction to generate electrical energy. During the reaction $ZnBr_2$ is generated. The salt solutes in the electrolyte and the $ZnBr_2$ concentration increases during charging while there is a decrease in the $ZnBr_2$ concentration during recharge [13].

This technology is environmentally hazardous because of the highly reactive and noxious bromine solution. In the event of an accident chemical leaks would prove to be devastating to equipment and any humans who may come in contact with the spill. The size and large internal resistance of this battery will be interesting problems to overcome in future designs.

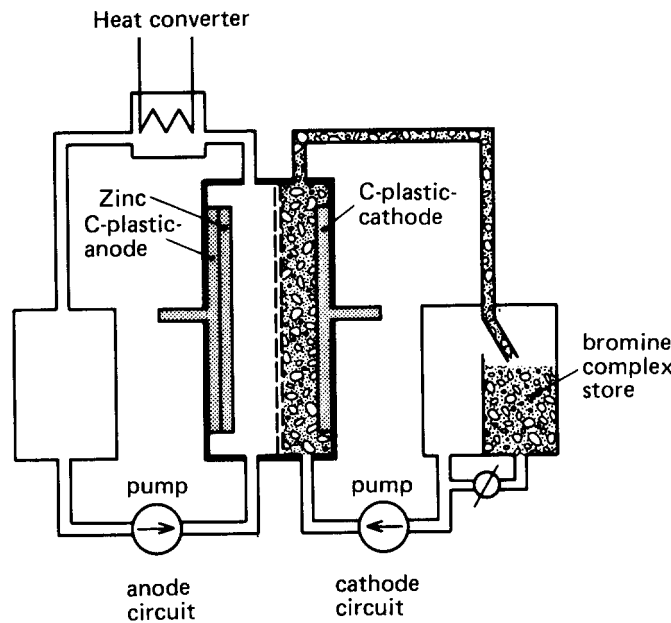


Figure 6: Zinc-Bromine battery principles

Sodium-Sulfur

The main components of a sodium-sulfur battery (see figure 7) [14] are molten sodium and sulfur. Enclosed in a metal housing, this battery operates near $350^{\circ}C$ to maintain these molten reactional substances. This metal housing functions as the positive electrode. A ceramic cylinder functions as a separator and an electrolyte, allowing the passage of sodium ions but no electrons. This also enables the battery to hold a charge for long periods of time. This housing and the ion-conducting beta-aluminum oxide (ceramic)

cylinder are connected in a manner to impede the entrance of air and the exit of sodium or sulfur. The sulfur +C felt increases the conductivity of the sulfur electrode, hence reducing the internal resistance.

During the discharge of the cell, sodium ions penetrate the electrolyte cylinder. In the negative electrode compartment sodium is consumed and in the positive electrode compartment the sodium ions react with the sulfur to produce Na_2S_x ($5 \geq x \geq 3$). This process can proceed until the total amount of sodium is consumed or until the whole sulfur is transformed to Na_2S_3 . During recharge all reactions proceed in the other direction [14].

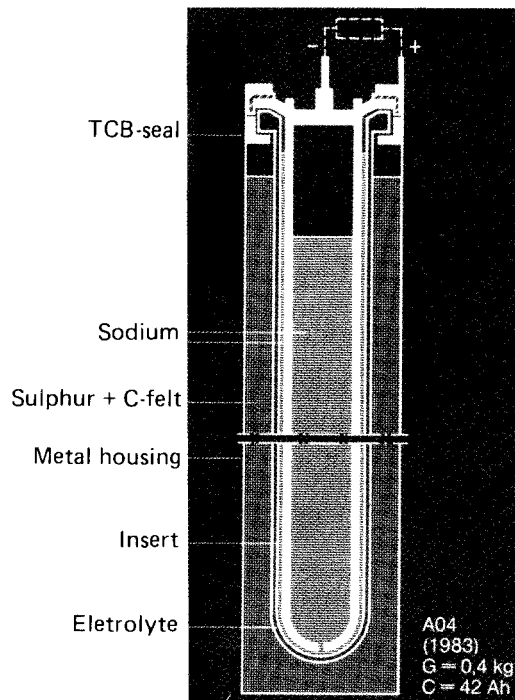


Figure 7 : Sodium-Sulfur battery cell.

A sodium-sulfur battery, 8V 300A*h, manufactured by *Chloride Silent Power Ltd.* with *Sandia National Laboratories*, was tested on the FORD ETX-II vehicle and retained 85% of its initial 292 A*h capacity (3h rate) after 700 cycles. The materials used in the manufacture of this technology are lightweight and plentiful. Long-term goals for this technology will provide high energy density batteries (200 W*h/kg) for use in commercial electric vehicles [15].

C.6 CONCLUSIONS

At present the lead-acid battery is still the optimal technology for our mission. Improvements have been realized in this technology and higher capacity units are available. *Union, Guardsman* and *Power Sonic* offer suitable replacements for the *Prevailer* battery.

However, the substitution of primary batteries will not allow our goals to be realized. A proposed solution is to upgrade the primary batteries to a higher capacity unit and add batteries in the turret area for dedicated service to the additional electronics above the *Cybermotion* base.

References:

- [1] H. A. Kiehne, "Battery Technology Handbook," New York, NY: Marcel Dekker, Inc. 1989 (pg 5)
- [2] H. A. Kiehne, "Battery Technology Handbook," New York, NY: Marcel Dekker, Inc. 1989 (pg 12-15)
- [3] Mr. Casper (CAS), Synergistic Batteries Inc., Maritta, GA: (1-800-634-6000)
- [4] Douglas Battery Co., Winston-Salem, NC: (1-910-650-7000)
"GURB 1-94," "GMB 2-94"
- [5] Power Sonic, "Sealed Lead-Acid Batteries," Redwood City, CA: (1-415-364-5001)
- [6] Chemical Engineering magazine, November 1993, pg 21
- [7] B. Jays, A. Datta, C. Mathews, R. Blayner, "Performance of the HORIZON® Advanced Lead-Acid Battery," Electrosource Inc., Austin, TX: (1-512-445-6606)
- [8] Power Sonic, "Sealed Lead-Acid Batteries," Redwood City, CA: pg 6-7 (1-415-364-5001)
- [9] Technical Data Sheet "Battery number NIF-200-5, Nickel-Iron Module," J. Whitford, Eagle Picher, Joplin, MO. (1-417-623-800)
- [10] H. A. Kiehne, "Battery Technology Handbook," New York, NY: Marcel Dekker, Inc. 1989 (pg 214-216)
- [11] D. Spak, Westinghouse Electric Co., Electrical Power Systems, Cleveland OH: (1-216-486-8300)
- [12] P. Gifford, Ovonic Battery Co., Troy, MI: (1-313-362-1750)
- [13] H. A. Kiehne, "Battery Technology Handbook," New York, NY: Marcel Dekker, Inc. 1989 (pg 207 - 210)
- [14] H. A. Kiehne, "Battery Technology Handbook," New York, NY: Marcel Dekker, Inc. 1989 (pg 210-213)
- [15] Battery an electrical vehicle update, "Automotive Engineering," September 1992

Systems Powered by the Turret Batteries

| ITEM | SPECS | QTY | 12V | 5V | -12V | 24V |
|------------------------|--|--------------|------------|------------|------|-----|
| IMA 150/40 Board | 4 A @ 5V | 2 | - | 40 | - | |
| Camera Box | 8.4 W @ 12V | 4 | 33.6 | | | |
| AM-CLR Board | 0.6 A @ 5V 0.22 A @ 12V 0.06 A @ -12V | 2 | 5.28 | 6 | 1.44 | |
| CM-HF Board | 1 A @ 5V | 1 | - | 5 | - | |
| CM-CLU Board | 1.8 A @ 5V | 1 | - | 9 | - | |
| *Strobe | Heads | 14.4 A @ 12V | 8 | *see below | - | - |
| *Strobe Powerpack | 5 A spike @ 12V | 4 | *see below | | | |
| *Laser diodes | 60 mA @ 5V | 4 | - | *see below | - | |
| TTL logic circuit | under 50 mA @12V | 1 | 1 | - | - | |
| *Raster Barcode Reader | 500 mA @ 12V | 4 | *see below | - | - | |
| HKMips Board | 5.5 A @ 5 V norm. 0.1 A @ 12 V max. 0.1 A @ -12 V max. | 1 | 1.2 | 40 | 1.2 | |
| I/O Board | 1.5 A @ 5 V max. | 1 | - | 7.5 | - | |
| Serial Board | 4.6 A @ 5 V 190 mA @ 12 V 190 mA @ -12 V | 1 | 2.28 | 23 | 2.28 | |
| 1 GB Hard Disk Drive | (startup values) 1.6 A @ 12 V max. 700 mA @ 5 V max. | 1 | 19.2 | 3.5 | - | |
| Lidar System | 0.4 A @ 24 V | | | | | 9.6 |
| Panaflow Brushless Fan | 0.16 A @ 12 V | 3 | 5.76 | | | |
| TOTAL | | | 68.32 | 134 | 4.92 | 9.6 |

Total Continuous Power Consumption : 216.84 W

Appendix D

ARIES: An Intelligent Inspection and Survey Robot

MECHANICAL SYSTEMS

**Department of Mechanical Engineering
University of South Carolina**

D. MECHANICAL SYSTEMS

D.1 ABSTRACT

Presented here is a review of the mechanical design and analysis of a camera positioning system (CPS) developed for the Department of Energy's ARIES project. The ARIES (Autonomous Robotic Inspection Experimental System) project was executed through a joint effort of three parties: University of South Carolina (USC), Clemson University, and Cybermotion, Inc., of Salem, Virginia. The goal of the project was to develop an autonomous mobile robot that positions a data acquisition package (DAP) which surveys drums containing hazardous materials in Department of Energy (DOE) warehouses. The three positioning systems designed, constructed, analyzed, and tested throughout three phases are discussed. Though all three systems (CPS-I, CPS-IE, and CPS-II) are presented, most emphasis is placed on CPS-IE. The unique mechanical design of each CPS is comprised of distinct components including a lift mechanism, a fourbar mechanism, and a camera panning mechanism. The components are integrated in a manner that allows the DAP to be positioned from 0 to 16 feet off the ground while attached atop a Cybermotion K3A autonomous mobile platform.

D.2 INTRODUCTION

Located throughout the United States at various Department of Energy (DOE) sites are warehouses that store large quantities of low-level radioactive wastes and other hazardous materials in steel drums (See Figure 1). The drums come in 55-, 85-, and 110-gallon sizes and are arranged on pallets and then homogeneously stacked on top of each other, forming columns of drums ranging in heights from 1 to 4 drums high (sixteen feet maximum). The columns of drums are aligned and arranged in aisles three feet wide. Up to 10,000 or more of these drums are located in a single warehouse. Currently, DOE site personnel visually inspect these drums at least once a week for leaks, corrosion, or containment failure. The DOE recognized this monotonous, yet critical inspection process as an excellent application for an autonomous robot—thus the advent of ARIES.



FIGURE 1: ARIES Team in a DOE Low-Level Nuclear Waste Storage Facility

DOE established several initial constraints that this robot must adhere to:

- must operate untethered and in an autonomous manner;
- must perform inspections in a timely fashion;
- must position itself vertically to survey drums from 0 to 16 feet;
- must be capable of lowering its' center-of-gravity for negotiating spillway berms with minimum 9% grades;
- must safely traverse and turn in a 3-foot aisle.

The project was broken into three distinct phases. Phase I was defined to survey current technology and prove the conceptual worth of the system. Phase II focused on the design and construction of Camera Positioning System I (CPS-I), a prototype. Phase III was then intended to analyze CPS-I, enhance its design, and using this information construct CPS-IE (enhanced). Phase I began in the fourth quarter of 1992 and the contract with DOE ceased at the end of Phase III, or the first quarter of 1997. Following Phase III the project was turned over to Cybermotion, Inc. to produce CPS-III, a productized version of the system.

The ARIES system consists of a mobile robotic platform (Cybermotion's K3A) (See Figure 2), the CPS which sits atop the K3A, and a remote command center where all operations are processed. For increased stability, the system is navigated through a warehouse in a compact (stowed) position. Upon entering an aisle of drums the system extends a Data Acquisition Package (DAP) down to acquire data from the lowest drum using a fourbar-mechanism. A panning-mechanism then rotates the DAP through a 93° sweep while collecting data on the drum. Finally, a lifting-mechanism provides a vertical translation of the DAP to the three remaining drums. The process is repeated for each drum column.

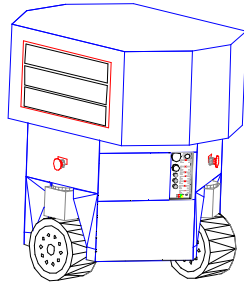


FIGURE 2: Cybermotion's K3A Mobile Robotic Platform

The DAP was developed by Clemson University. A single package consists of a color camera, a black and white camera, two flash lamps, two light sensors, two laser light projectors, and a barcode reader (See Figure 3). To capture the surface of a drum from top to bottom, the color camera must be positioned at two different locations on each drum. The positions are: ± 8 in. from the center of a 55-gallon drum; ± 10 in. from the center of an 85-gallon drum; and ± 12 in. from the center of a 110-gallon drum.

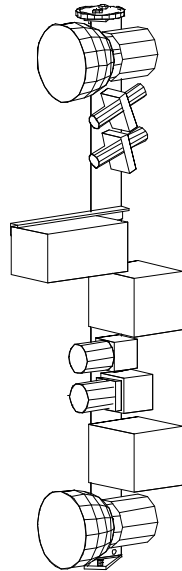


FIGURE 3: Data Acquisition Package (DAP)

D.3 PHASES I & II

D.3.1 Phase I: Proof of Concept

The first phase of the ARIES project allowed team members to ascertain the design concept initially envisioned. The ARIES team was to deliver reports and demonstrations as proof of this concept. There were few mechanical aspects taken into account during Phase I.

The ability of an autonomous vehicle to inspect low-level radiation sites was proven through the technology developed in Phase I. Issues such as navigation, vision, remote supervision, on and off board controls, and radiation dose-rate hardness for on board systems were all developed [2].

D.3.2 Phase II: Introduction of CPS-I

The main consideration during Phase II was to design and construct a prototype (CPS-I) that satisfied the project constraints, while demonstrating its ability to position a camera package properly and efficiently. DOE would test this prototype at the University of South Carolina's test facility.

D.3.2.1 CPS Conceptualization

Several systems were studied during conceptualization to avoid redesigning a pre-existing mechanism. These systems 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" [5]. The industrial work platform seemed the most suitable system for achieving the camera height translation. This type of platform is created from a series of interlocking rail elements, each translating with respect to one another as a result of cable/pulley associations, of which one element is fixed to 'ground'. The motion is a direct result of some type of actuator: hydraulic, electric, or pneumatic.

D.3.2.2 Existing Systems

The ability to manufacture this system was also a very important issue to study. Several production concepts were studied and considered before the design process ensued. The first concept was to simply retrofit a commercially available system to the K3A, however no readily available system was found to satisfy the conditions required. The next concept was to construct CPS-I using 'off the shelf' components. Finding these types of components deemed unattainable at the time. The third and final concept was to design and construct a positioning mechanism, unique to the market, able to be easily productized, and capable of fulfilling all requirements. This concept, at the time, seemed the most reasonable route due to the constraints applied to the system.

The concept envisioned for CPS-I contained an interlocking rail element lift mechanism and a parallelogram fourbar mechanism (See Figure 4). The lift would be an assembly of five rail elements with three individual DAPs attached on individual elements

such that, for every drum stack, each of the three top drums would have its own camera package retrieving data. The fourth DAP would be attached to a parallelogram fourbar mechanism that would extend down and away from the CPS in order to clear the K3A platform and to retrieve data from the lowest of the four drums. This design provides a dedicated DAP for each drum in the stack.

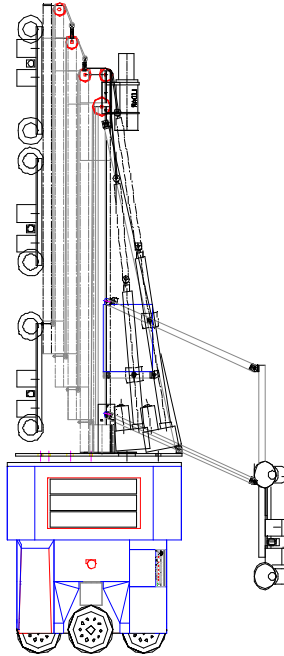


FIGURE 4: CPS-I

D.3.2.3 CPS Components

Because CPS-I was designed and manufactured as a prototype, the need for a versatile construction process arose. The components had to be easily procured, easily machined, and easily fastened to one another. Item Products', Inc. modular fabrication system was chosen to meet these needs. This system is made of high strength, precise pre-engineered components. The components range from custom 6005-T5 or 6061-T6 aluminum extrusions to eccentric bearing units and all types of connections in between. The versatility and availability of the Item components made this the system of choice.

D.3.3 CPS-I Lift Mechanism

The design of the lift mechanism required that the three camera packages attached to three of the elements be positioned at specific locations with respect to the center of each drum in the stack. These centers are dependent on the respective drum's size. The homogeneous stacks of drums placed in non-homogeneous rows meant that the positioning of the cameras could be different for each stack. To accommodate different drum sizes it was necessary to not only position the elements at a certain height, but to also maintain a distance between each package. This motion is defined by two degrees of freedom (DOF) which requires two inputs. One of the inputs defines the height the lift can attain with respect to ground, and the other defines the separation difference between the three camera

packages. The camera packages must be at different heights and separations to retrieve data from different size drum stacks.

D.3.3.1 Element Design

The lift's interlocking element design was constructed of one stationary aluminum extrusion or grounded element and four smaller extrusions moving with respect to one another by means of Item's roller bearing units. Item's 40 x 80 mm. precut extruded beams are used as the four moving elements (Elements 2, 3, 4, and 5) on the lift mechanism. The one stationary element (Element 1) is an 80 x 80 mm.-extruded beam. Attached to the beams are pairs of 14 mm. double bearing units produced by Item. These roller-type bearings ride along 14 mm.-diameter rails that are also attached to the subsequent elements. The eccentricity of the bearings allows the user to adjust bearing preload for a proper fit, which aids in the support of a lateral load applied to the elements. Placed on each of the bearing units is a lubricated end cap. This end cap functions as a lubrication reservoir and through a felt pad, and keeps a thin layer of oil on the rails to maintain a smooth fluid motion.

D.3.3.2 Lift Actuation

The motion of the lift is provided by a system of five pulleys, five cables, and two linear actuators. Each linear actuator controls one of the two DOF associated with the lift mechanism. Both actuators are the Electrak 100 models manufactured by Warner Electric. Eighteen and twenty-four inch linear actuators were used. Both have a rated continuous-load capacity of 500 lb. and life expectancies of 33,000 and 27,000 cycles, respectively, under normal loading conditions.

D.3.3.3 Load Analysis

Static and dynamic force analyses (method discussed in Chapter 4) were performed on the actuators to determine their appropriate load-capacity. The cable attached to actuator 2 has a maximum calculated static load of about 153 lb. The maximum speed attained by the actuators is 2.5 in./sec. To obtain a reasonable acceleration value for dynamic-loading, tests were performed using a load cell placed inline with the actuator's cable. The average static-load determined by force-analysis was 151.7 lb., while the average load obtained with the load cell was 146.9 lb. These static-load values fall within 3% of one another. Dynamically, a maximum of 175.9 lb. was measured when accelerating and decelerating the lift at approximately 5 ft./sec². Theoretical calculation of the dynamic load using this acceleration yields 176.75 lb., which is very close to the measured value.

The actuators transmit all motion through 0.125 in. diameter steel cables. These cables are associated with a group of strategically sized and placed pulleys to form cable/pulley associations throughout all five elements. These associations lead to amplifications of an element's movement when the linear actuator moves. The relative amplification of each element is determined by geometrically plotting its movement as a function of the actuators' motion. The positions are determined based on the constant length of the cables. Each element moves with its own distance ratio (DR) due to the cable/pulley association it has with the ground. Based on the assumption that all cables remain parallel to the elements, theoretical DR are determined for CPS-I as follows: Element 2 — 1:1, Element 3 — 3:1, Element 4 — 5:1, and Element 5 — 7:1. This states that theoretically a 1 in. actuation will result in a 7 in. translation of element 5. The actual

amplification factor of the mechanism's fifth element, however, is calculated as 6.5:1 as a result of angles introduced in the cables. The velocities and accelerations follow the same pattern.

From the conservation of energy principle, the gain of the DR creates a loss in mechanical advantage (MV). This exchange is illustrated by the definition of MV. According to Sahag [7], the applied load multiplied by the distance it moves is equal to the output load multiplied by the distance it moves (a conservation of energy principle). The MV is then the ratio of similar terms.

$$F_{out} \bullet d_{out} = F_{in} \bullet d_{in}$$

$$MV = F_{out} / F_{in} = d_{in} / d_{out} \quad (1)$$

Where F_{OUT} = output load
 d_{OUT} = distance the output load moved
 F_{IN} = applied load
 d_{IN} = distance the applied load moved
 MV = Mechanical Advantage
 d_{OUT}/d_{IN} = Distance Ratio

Rearranging the terms, it is clear that MV is the inverse of DR. A MV of 7, according to this definition, states that on output of 700 lb. requires an input of 100 lb. These systems, however, are the opposite. That is, a 100 lb. output theoretically requires a 700 lb. input. This results in a MV less than one. A mechanical disadvantage? Recall the advantage is not in the load, but in the DR, which in fact is the inverse of the MV. To take advantage of this knowledge, a method of calculating the applied load at the actuator as a function of the MV of each element in these systems was developed. This method will be detailed in Chapter 4.

D.3.3.4 Lift Mechanism Equations of Motion

The equations of motion for the lift mechanism were developed by Rocheleau [5] as a method for calculating the 3 DAP camera heights. They are approximated as linear functions based on an analytical geometry model using planes and lines. The heights of the cameras attached to Elements 3, 4, and 5 from the floor are denoted as z_3 , z_4 , and z_5 (measured in inches). The camera heights are functions of the retraction of actuators 1 and 2, denoted as x_1 and x_2 . The equations take into account the changes in cable angles as the mechanism extends upward. Three triplets of equations are developed for each of the 55-, 85-, and 110-gallon drum columns that may be encountered. The following sets of equations are used with an accuracy of ± 0.5 in.:

55-gallon drums:

$$z_3 = 1.829x_1 + 0.987x_2 + 46.366 \quad (2a)$$

$$z_4 = 2.646x_1 + 1.958x_2 + 72.716 \quad (2b)$$

$$z_5 = 3.450x_1 + 2.913x_2 + 101.066 \quad (2c)$$

85-gallon drums:

$$z_3 = 1.864x_1 + 0.952x_2 + 46.366 \quad (3a)$$

$$z_4 = 2.720x_1 + 1.894x_2 + 72.716 \quad (3b)$$

$$z_5 = 3.565x_1 + 2.825x_2 + 101.066 \quad (3c)$$

110-gallon drums:

$$z_3 = 1.881x_1 + 0.947x_2 + 46.366 \quad (4a)$$

$$z_4 = 2.755x_1 + 1.886x_2 + 72.716 \quad (4b)$$

$$z_5 = 3.619x_1 + 2.815x_2 + 101.066 \quad (4c)$$

The maximum drum height was constrained to four 110-gallon drums and their respective pallets. A 110-gallon drum is approximately 43 in. tall, and along with a 5 in. pallet the maximum height of a four-drum column is 192 in. Because the camera must be positioned at +/-12 in. from the center of the 110-gallon drum, the maximum height the camera's focal point needs to be 182.5 in., or 15 ft. 2.5 in. This information is imperative during the design process of a lift mechanism. Since this height fell within the initial constraint imposed by DOE of 16 ft., CPS-I was designed to reach this 16 ft.-height.

D.3.3.5 Element Geometry

To reach this 16 ft.-constraint, the mechanism's motion must be unhindered throughout its range of movement. The length of travel the bearing units must undergo along their respective rails is a contributing factor to define the element lengths. The magnification ratios are once again called upon to determine how much one element moves with respect to another. Based on these ratios and contributing assembly constraints, the calculated element lengths are as follows: Element 1 - 67.5 in.; Element 2- 67.5 in.; Element 3 - 67.75 in.; Element 4 - 67.75 in.; Element 5 - 62 in. Once assembled onto a 3/4 in. aluminum base plate, the minimum height of the lift stands approximately 81 in. Once assembled atop the 32 in.-tall K3A, ARIES stands approximately 113 in., or just under 9.5 ft. Added navigational equipment pushes this height over 10 ft.

D.3.4 CPS-I Fourbar Mechanism

To retrieve information from the lowest drum in each stack, a fourbar mechanism lowers a DAP out away from the K3A and down to the lower position. Because ARIES must turn around in a 3 ft.-aisle, the CPS that sits atop the K3A must be compact enough such that none of its components overhang the outer edges of the K3A; that is, from an overhead view the outer most edge is the K3A itself. To maintain this compactness and still be able to retrieve data from the lowest drums, a mechanism that moves the DAP away from the platform and down to its proper height was necessary. The mechanism chosen is a special-case-Grashof parallelogram fourbar [4].

The constraints on the fourbar were established as:

- it must provide full retraction of the DAP over the K3A
- the DAP is deployed out enough so the camera's field of view is not obstructed by the sides of the K3A
- the lateral displacement of the lower fourbar position should be equal to, or very close to the lateral displacement of the upper fourbar position

The final constraint is intended to maintain a constant vertical line of action along which the cameras may take data. This will result in similar data taken at the upper and lower positions of the lowest drum.

D.3.4.1 Conceptualization

In the conceptual design process of this mechanism several mechanisms were studied during the type-synthesis phase. A Watt's sixbar linkage was considered, but was dismissed due to its complexity. The fourbar linkage was then considered. During the

qualitative-synthesis phase the parallelogram geometry of the chosen special case Grashof mechanism (See Figure 5) was the best package for the application at hand.

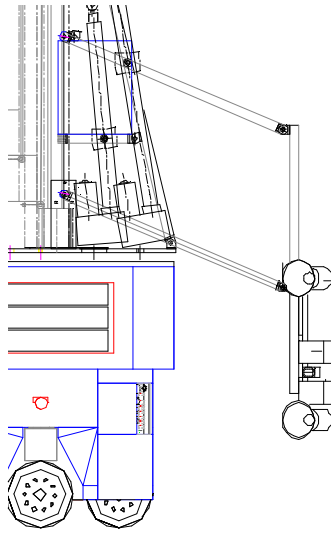


FIGURE 5: CPS-I Fourbar Mechanism

The parallelogram geometry of the fourbar is a significant factor in placing the DAP at a desired location. A parallelogram fourbar has a unique coupler movement; that is, the coupler undergoes pure translation. The initial posture of the coupler will remain throughout the fourbar's translation. In this case, the DAP is the coupler. The preferred posture of the DAP is vertical. Thus if the DAP is initially vertical, then it will remain vertical throughout the fourbar's motion. This is useful because the package will have the same posture with respect to the drums at a lower as well as an upper position for each drum. This posture also helps to compact the fourbar into a stowed position when the vehicle is in motion.

D.3.4.2 Geometry

The development of this mechanism was primarily based on simple kinematic equations of motion. To create the geometry, a dimensional-synthesis was performed to determine the coupler output position as the mechanism actuates through the prescribed motion. The lengths of the identical driver and follower links are determined by this synthesis. The coupler, or DAP in this case, must clear the K3A along its movement. It must be positioned far enough away from the K3A in its lower position such that the field of view is not blocked. It also must be properly stowed within the platform boundary. Using these constraints, the driver and follower links were calculated to be 49 in. (hinge to hinge). These links must be identical in length to maintain the parallelogram geometry. The hinge to hinge distance on the ground and coupler links also had to be identical but their distance is not a critical design constraint. This distance, as assembled, is approximately 17 in.

D.3.4.3 Components

To manufacture this fourbar mechanism, the Item aluminum components were again used. 28 x 28 mm. aluminum extrusions were used for the driver, follower, and

coupler links. The ground link is constructed of a single 40 x 40 mm. mast. To allow the single rotational DOF of each joint, 28 x 28 mm. hinge joints were used.

D.3.4.4 Actuation

To provide motion to the driving link, the amount of torque necessary to drive the fourbar was calculated. The camera package and all accessories weigh approximately 5.5 lb. This requires a maximum torque of 270 in.-lb. created at the driver/ground joint when the driver is parallel to the floor. A 0.25Hp 1.8 Amp DC motor, attached to the base of the system, supplies rotary motion to the driver. Position control of the input is achieved from an optical encoder attached to the motor drive to feedback position. A limit switch is positioned to halt the motion once the mechanism reaches a stowed configuration.

D.4 PHASE III

The scope of Phase III was to redesign and enhance CPS-I such that a ‘productized’ system was developed and a working prototype was delivered. This prototype was to be demonstrated onsite at a DOE storage facility to ensure the feasibility of the design. What actually surfaced from Phase III, however were two systems: CPS-IE and CPS-II. CPS-IE was an enhanced version of CPS-I and CPS-II was a less complex productized version. Both systems are capable of the same task, however each has its own distinct qualities.

D.4.1 CPS-IE Lift Mechanism

CPS-IE (See Figure 6), an *Enhanced* version of CPS-I, was constructed for demonstration at the DOE Fernald Environmental Management site near Cincinnati, Ohio. To improve overall efficiency and reliability of the first system several necessary enhancements were made as follows: lift mechanism revisions; weight reduction of the CPS; fourbar mechanism revisions; and the addition of a rotational DOF to the DAP to allow for dent detection.

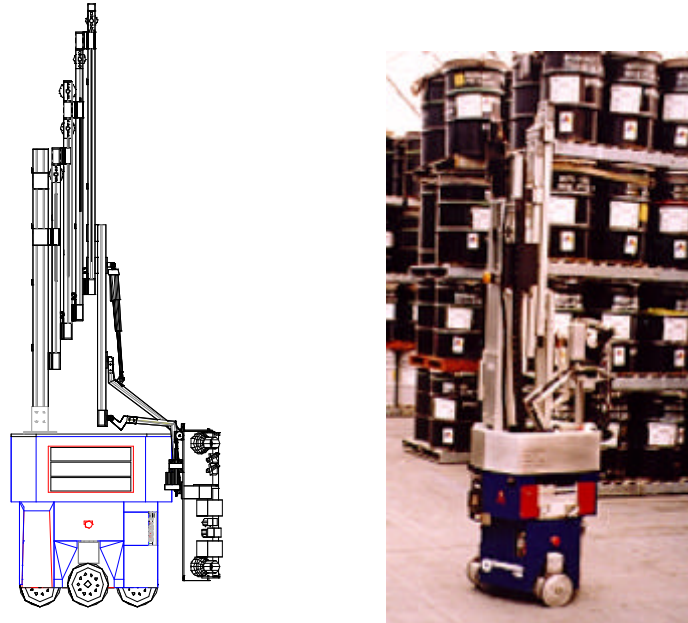


FIGURE 6: CPS-IE

D.4.1.1 Concepts

Several lift concepts were considered as follows: using the same lift mechanism as in CPS-I; designing an extendable mast which would extend and retract like a hinged truss; designing a single mast system consisting of a single fixed element. The hinged truss concept was not used because this required a redesign of the entire system instead of an enhanced version. The single fixed mast concept was not used because DOE wanted the system to be capable of compacting enough to clear a ten-foot bay door in its stowed configuration. A fixed mast that could reach 16 ft. would extend past the ten-foot height in its stowed configuration. Thus, the concept chosen was to enhance the pre-existing lift mechanism.

D.4.1.2 Bearing Configurations

One important factor in the design stage of the enhanced lift mechanism was the elements' freedom of relative motion with respect to one another. Due to the location of the bearing units on CPS-I, the elements were limited as to how much travel was available before the bearings from two separate elements collided. This was a result of two bearing units riding along the same shaft. Because elements move at different rates with respect to one another, the contact is eminent. To avoid this, the design of the lift on CPS-IE had to have one bearing per shaft. This would eliminate any contact during the prescribed motion. To achieve this, several bearing configurations were studied (See Figure 7). Concepts were considered using the 14, 10, and 6 mm. bearing units. As can be seen by these concepts, each bearing has its own independent shaft to translate on.

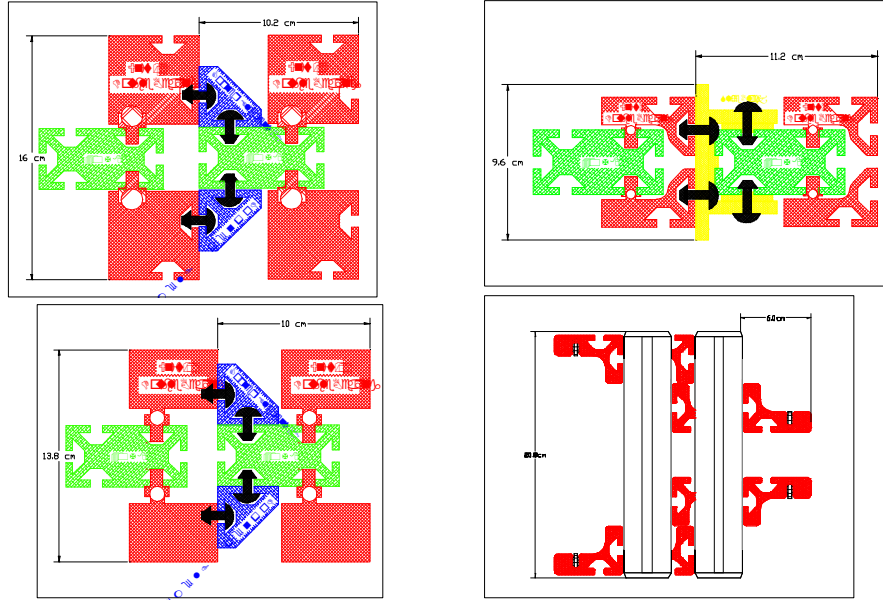


FIGURE 7: CPS-IE Bearing Configuration Concepts

The 6 mm. bearing unit configuration was chosen, and is assembled using rectangular box elements on which bearing units ride along shafts placed on the inner and outer faces of these elements (See Figure 8). This allowed one bearing unit per shaft; thus the outer and inner bearings could translate along the same plane, but never come into contact with one another. This design reduced the overhead footprint of the system allowing the number of elements to be increased to six if desired. The addition of a sixth element to this enhanced version was desired to not only increase the positioning speed, but to also decrease the stowed height.

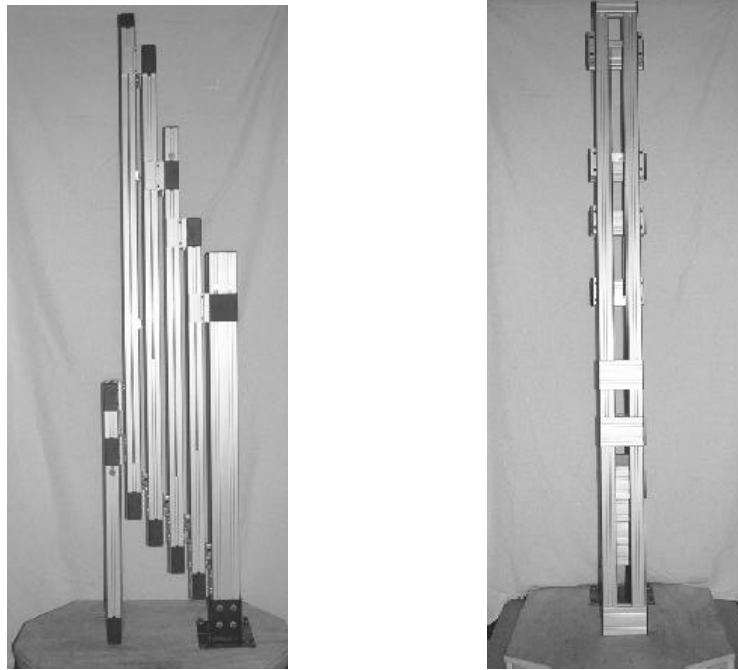


FIGURE 8: CPS-IE Lift Mechanism Element Construction

D.4.1.3 Length of Contact

One concern of this new lift design was the maximum lateral load applied to the 6mm bearing units. According to Item Product's Inc., manufacturer of these 6mm double bearing units, they can only withstand a 28 lb. lateral load on each roller [3]. This lateral load is a result of a couple created by the system's inertia under acceleration. The couple has a direct correlation to a parameter defined as length of contact (LOC) (See Figure 9). LOC is the vertical distance between the point of contact of the outer bearings of one element and the point of contact of the inner bearings of the next element. As the LOC decreases, the moment arm of the couple increases; thus, the lateral load increases. A safe minimum LOC was established for each element pair as follows: elements 1 & 2 - 15 in.; elements 2 & 3 - 12 in.; elements 3 & 4 - 12 in.; elements 4 & 5 - 6 in.; and elements 5 & 6 - 3 in. This minimum LOC increases the system's stowed height, however is vital to the overall safety of the system.

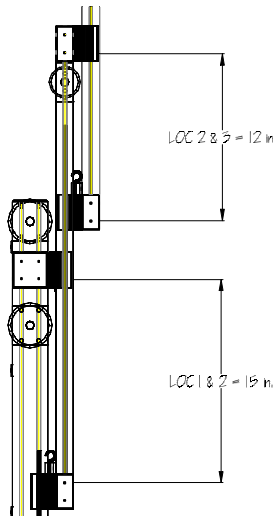


FIGURE 9: Length of Contact (LOC)

D.4.1.4 Pulley Configurations

Once the element component assembly had been established, the pulley cable associations were defined. Recall that one can visually determine how these associations will effect the motion and magnification of the elements. The procedure used to determine the element ratios was to visually inspect the magnifications created as different pulley configurations were studied. Several pulley/cable configurations were evaluated to arrive at desired speed amplifications. Each configuration was defined by the number and location of pulleys on each element starting with the first moving element (element-2) to the last moving element (element-6). For example a two-element system in which the first element contained two pulleys and the second element had one pulley would be defined as a 2-1 configuration. Thus, CPS-I would be a five-element system with a pulley configuration 2-1-1-1. The number of pulleys on the fifth element is not included simply because there are no pulleys on element five. Three configuration concepts and their predicted speed amplifications are as follows: 0-1-2-1-1, 7x; 2-1-2-1-1, 13x; and 2-1-2-2-1, 15x. These three configurations will be defined by their final element speed amplification; thus, the three pulley configurations are PC7, PC13, and PC15, respectively.

Configuration PC13 was the first of three 'six element' configurations studied. With a pulley configuration of 2-1-2-1-1, the theoretical element magnifications were as follows: element 2- 1:1; element 3- 3:1; element 4- 5:1; element 5- 9:1; and finally element 6- 13:1. Analysis shown in Chapter 4 will prove that approximately 12 in. of actuation are required to lift element six to a height of 16 ft. This configuration requires an 867 lb. load on the actuator.

Configuration PC15 goes one step further and adds another pulley to element 4. This configuration, 2-1-2-2-1, results in the following theoretical element magnifications: element 2- 1:1; element 3- 3:1; element 4- 5:1; element 5- 9:1; element 6- 15:1. Although this configuration had initially been considered, it was decided that this magnification was too great for the prototype phase. Thus this configuration was merely a concept.

After studying configuration PC13 for some time, this seemed the best overall configuration to use. In fact, the design was finalized and the lift constructed. At this time, the selection of the linear actuator was studied. The selection process began as a decision

based on dynamic loading of the cabling as well as duty-cycle and power consumption of the actuator. The dynamic loading, as shown in Chapter 4, is based on 15% over the maximum static load calculated. This maximum value is 867 lb., thus the dynamic load is calculated as approximately 997 lb. Considering the actuator would experience more of the constant velocity loads, the dynamic loading was not a consideration in the duty-cycle calculation. Based on the 997 lb. maximum capacity necessary, the decision to use a 1000 lb. actuator was chosen.

D.4.1.5 Actuator Selection

Due to procurement issues, the decision was made to use the same linear actuator used in CPS-I on CPS-IE. To accommodate this actuator, a final revision was made to the pulley configuration. A 7x configuration was chosen to accommodate the 24 in. - 500 lb. actuator stripped from CPS-I. The pulley configuration (PC7) of this six-element mechanism is 0-1-2-1-1. This configuration results in the following theoretical element magnifications: element 2- 1:1; element 3- 2:1; element 4- 3:1; element 5- 5:1; element 6- 7:1. After geometric evaluation of the angles of the cables on this system, the actual magnification of the overall system is 6.9x, which is slightly higher than CPS-I. A simple change in the PC13 configuration resulted in a major amplification reduction. This reduction decreased the theoretical actuator static load capacity to 488 lb. which falls within the 500 lb. continuous-load rating. Recalling MV however, the stroke increased to over 21 in.

The dynamic loads on the actuator were then calculated for the mechanism. As previously noted, the maximum static load was calculated as 488lb. Under a determined 5 ft./sec² acceleration the additional theoretical dynamic load is calculated as 75.8 lb.; thus, the total actuator dynamic loading condition is 563.8 lb. This exceeds the actuator's continuous-load rating; however, is still under its maximum load-capacity rating. Life expectancy of this actuator under these conditions was reduced to 10,000 cycles.

D.4.2 CPS-IE Weight Reduction

The biggest enhancement made to CPS-IE was a reduction in the overall weight of the system. CPS I weighed 200% over the original estimated value. This excess weight is detrimental to the energy stores required to properly position the robot and to actuate a DAP into the necessary configuration. This untethered system requires a charging period which takes away from time that could be spent acquiring data. A reduction in system weight results in more infrequent charge times as well as less power consumption during positioning.

D.4.2.1 DAP Reduction

Several steps were taken to reduce the energy-depleting weight of CPS-I. The biggest reduction in weight came from decreasing the number of DAP from four on CPS-I to one on CPS-IE. Along with reducing weight, this change decreased system complexity, minimized system-recharging time, and reduced manufacturing cost. Element component weight was also reduced. The overall weight of CPS-IE was 261 lb., a 34% reduction over CPS-I.

D.4.2.2 Bearing Changes

Another significant weight reduction was obtained by reducing the bearing unit size on each of the elements. Recall CPS-I used Item's 14 mm. double bearing units. Each bearing unit consists of two rollers, two eccentric adjustment bolts, two lubricating end caps, an extruded aluminum housing, a variable length steel shaft on which the roller rides, and a variable length clamp which holds the shaft to an element. The 14 mm. size refers to the diameter of the shaft. Item also offers these units in a 25, 10, or 6 mm. (See Figure 10) option. The overall weight of one 14 mm. double bearing unit is 2.04 lb. The respective shaft and clamp is 3.24 lb. per meter. The 10 mm. unit is 2.5 lb. and its respective shaft and clamp is 1.91 lb. per meter. The 6 mm. unit is 0.57 lb. and its respective shaft and clamp is 0.75 lb. per meter. Thus, per meter the 10 mm. and 6 mm. units are a 16.5% and 75% reduction in weight, respectively. Considering CPS-I had 16 bearing units and over 6 meters of shaft, this yields a large overall reduction in CPS weight. Several concepts were considered using both the 10 and 6 mm. units.



FIGURE 10: Item Products', Inc. 6mm Double Bearing Unit

D.4.3 CPS-IE Fourbar Mechanism

Similarly to CPS-I, the single DAP had to be extended down and away from the lift mechanism. Several concepts were considered for this design as follows: a sliding mechanism to push the package away from the lift; a scissor-type expandable truss that could also push the package away from the lift; a hinged mechanism that could swing the package out from the lift; a roller-positioning device which would push the package away from the lift using the lift's actuation; actuating the entire CPS forward; and finally an enhanced version of the fourbar used on CPS-I.

The sliding mechanism was a simple concept, however because the lift mechanism had to be lowered after the slider had been actuated, the stowed height of the lift would be increased. The scissor type expandable truss, although very lightweight and sturdy, was deemed too complex for an enhancement. The hinged mechanism was also a simple concept, however as with the slider, the stowed height would have to be increased. The stability of the roller-positioning device was questionable, and its complexity along with the stowed height being increased were three negative aspects of a mechanism that could be actuated using the lift mechanism's motion. Translating the entire CPS forward so that the camera could be lowered required a large amount of power as well as a change in the

center of gravity (CG) of the system. Finally, an enhanced version of the fourbar used on CPS-I was considered. After considering the experience gained from the fourbar on CPS-I, as well as the additional height advantage given by the symmetric nature of the parallelogram mechanism, the fourbar seemed the mechanism of choice for the enhanced version.

D.4.3.1 *Synthesis*

To design the enhanced fourbar, a step by step procedure was followed which began with a qualitative synthesis. According to Norton [4], the qualitative development of a fourbar mechanism is an iterative process between the synthesis and analysis during the design process. Through this process, one is able to geometrically alter a design according to inherent constraints, then test the results kinematically and dynamically to verify that it is mechanically acceptable. If not, this qualitative synthesis procedure is once again followed to make necessary changes to the previous iteration.

The first step to the fourbar synthesis is the selection of the mechanism type, or type synthesis [4]. Part of this synthesis occurred during the conceptual stage of CPS-IE; that is, the decision to use a fourbar mechanism to translate the camera package down and away from the mobile platform was already made. To complete the type synthesis, the only remaining step is to determine what type of fourbar linkage to use. One of the biggest benefits of the fourbar used in CPS-I was its ability to move a camera package to two locations, an upper and a lower, along a single line of action. This is beneficial because not only does the mechanism's lower position move the camera down and away from the mobile platform, but the upper position adds an additional height advantage, thus reducing the height requirements of the lift mechanism. Because of the fourbar's parallelogram geometry, a calculation of its Grashof condition yields the type 'special-case-Grashof' linkage.

Once the type of linkage is established, the next step in Norton's design process is the dimensional synthesis. This procedure allows the designer a method of determining the lengths or proportions of the fourbar's links to provide the necessary motion. Because of the fourbar's parallelogram nature, this process is simplified because the driver and follower links, as well as the ground and coupler links must be equal in length. Thus the number of decisions is reduced from four to two. The easiest method of establishing the geometry of the fourbar is by graphically plotting out the motion of the links, knowing the initial spatial constraints of the system. A compass, a protractor, and a rule, along with knowledge of Euclidian geometry are the only tools necessary to graphically synthesize the mechanism.

D.4.3.2 *Constraints*

The initial spatial constraints considered when synthesizing this mechanism were the limiting factors of the design process. In this application, there were several constraining factors imposed. The lowest position of the camera's focal point must be 12 in. above the ground to capture the lowest drum position. The line of action of the camera's focal point had to be 8.3 in. from the K3A's outer most edge to achieve the proper clearance of the camera's field of view. When the fourbar on CPS-I was stowed, the entire camera package had to fall within the outer perimeter of the mobile platform, thus allowing the system to rotate within a three-foot aisle. The ground joints of the fourbar were constrained 11.5 in. from the mobile platform's outer edge because of the prescribed location of the lift's sixth element. Finally, the mechanism was designed

to minimize link length thus reducing weight, torque requirements, and joint loads. All constraints were considered during the graphical synthesis of the mechanism.

Several designer-imposed constraints were then made to aid the graphical synthesis. The coupler joints were constrained at 6 in. from the camera's focal point line of action to account for the package to rotate for dent detection. The bottom of the DAP would remain 2 in. off the ground to avoid any contact. Finally the outermost edge of the DAP was determined as 3 in. from the camera's focal point line of action, which allowed the package to be stowed within the mobile platform's perimeter. With all constraints known at this point, the link lengths were determined graphically to be 20 in. The follower link would be located 2.5 in. off the mobile platform while in the stowed configuration to allow for any linkage rotation. The ground and coupler links were then determined as 10.5 in. The motion can then be plotted graphically to determine the validity of the mechanism's positioning (See Figure 11).

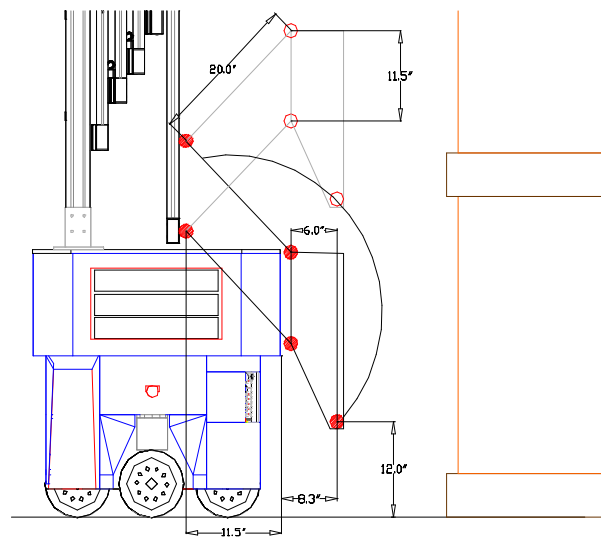


FIGURE 11: Constraints and Plot of the Enhanced Fourbar Mechanism

D.4.3.3 Quality

The fourbar mechanism's quality is determined by analyzing its position throughout the motion while checking for interference, toggle positions, and the transmission angle. There are several areas of interference along the path of this enhanced fourbar. Defining the upper of the two rotating links as the driver, then a straight-line follower link will interfere with the mobile platform. To avoid this, the binary link can be reshaped into any form as long as its two hinge locations do not change. Thus this link may be angled such that the interference with the mobile platform is eliminated (See Figure 12).

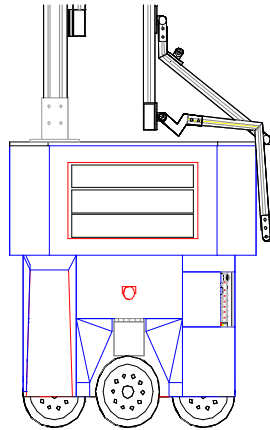


FIGURE 12: Fourbar Links

The toggle position of this mechanism will occur if the angle between the coupler and either of the two rotating links becomes 180 degrees (See Figure 13). There are two configurations in which a toggle position is reached on this type of mechanism. In this case, the only configuration of concern is the stowed position. The parallelogram geometry of this fourbar will yield colinearity between all links at the toggle position. An evaluation of the mechanism's quality shows that it will reach a toggle position if rotated approximately 5° past its stowed configuration. This position, however, is impossible to reach without experiencing interference with the lift-mechanism. Toggle position is undesirable in this case because it can cause the prescribed motion to drastically change creating interference or dangerously unstable motion of the mechanism.

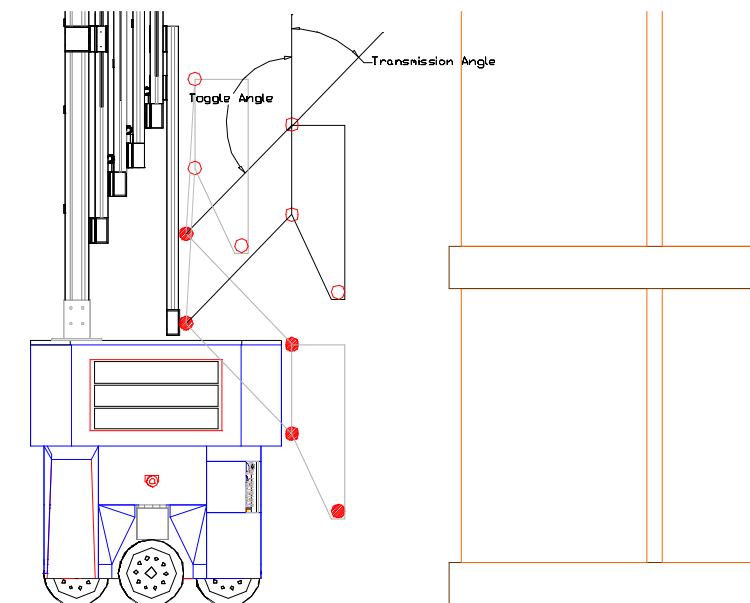


FIGURE 13: Toggle Position Angle and Transmission Angle

Norton [4] defines the transmission angle as the acute angle between the coupler and the

output link. This angle is important because it defines the direction of the load on the mechanism's joint. At any particular instant along the mechanism's motion, each joint has a radial and a tangential component. The tangential component of the follower link is the load that is responsible for all torque created at that link. Thus it is desired to keep this load maximum. The radial component, on the other hand, simply directs load onto the joint itself; thus, this load creates unnecessary stress as well as undesired friction on the joint's bearings. As the transmission angle decreases, the radial load increases while the output torque decreases. Norton recommends a transmission angle no less than 35 degrees, unless there is a very small external load on the follower link. There is very little external load on this link in this application. Thus, the small transmission angle encountered in the stowed configuration was neglected, and its resulting radial load was later analyzed. This will be shown in Chapter 4.

D.4.3.4 *Dynamic Results*

As was stated earlier, the design process of a fourbar mechanism is an iterative process. Properly designing this mechanism required synthesis followed by analysis. The analysis of this mechanism was split into two parts. To calculate the position, velocity, and acceleration of each link at any instant of time, a kinematic analysis was done. Once these values were obtained, a dynamic analysis was done. The dynamic analysis determined the forces and torque throughout the mechanism, including the information at the joints. The maximum torque capacity is 523 in.-lb. (59.093 N-m) when the driver link is parallel to the floor. The maximum radial load at a pin joint is 32.6 lb. (145 N), and occurred at 60° between the ground and the input crank. This is considerably lower than the maximum dynamic load rating of the bearing (288 lb.) used at the pin joints. The life expectancy of these roller bearings, calculated using an L₁₀ life rating, is over 2 billion revolutions. Both analyses will be shown in detail in Chapter 4.

D.4.3.5 *Actuation Concepts*

Actuator selection for the new fourbar was complicated because any weight applied to the sixth element would be magnified 6.9 times at the lift actuator. This weight would also affect the system's stability as the overall center-of-gravity (CG) changes when the system is raised. Two designs were considered when determining the actuation of the enhanced fourbar. One concept to reduce weight was to use the motion of the lift mechanism to actuate the driver link. Attaching an extended lever arm to the existing driver, the lever could be positioned into a cycloid-shaped slot that smoothly rotates the driver about its pivot point. The lift's motion would control not only the camera's height, but also its lateral translations. The other concept was to hinge a linear actuator on the sixth element attached to another hinge on the driver link of the fourbar. This linear actuator would crank the link down and back up as it extended and retracted, respectively.

D.4.3.6 *Cycloid-Slot*

The first concept, the cycloid-slot, was studied using basic cam design. According to Rocheleau and Moore [6], the concept requires a cam that has a rotating follower, as well as a form-closed joint [4] that would push and pull on the follower. The lever attached to the driving link of the fourbar would be the follower in this case. As the lift mechanism would rise, the lever would be positioned into the groove. A continued vertical translation would cause the lever to rotate around the groove thus the follower, or lever, would rotate and in turn actuate the fourbar

(See Figure 14). At any other time, the fourbar would remain in a stowed position. The design of the cycloid is calculated such that the full range of motion would occur along the track. The concept would require a brake to hold the fourbar position in place after being rotated, as well as some type of retractable pin, such as a solenoid, that could be positioned in the groove when the fourbar needed to be actuated.

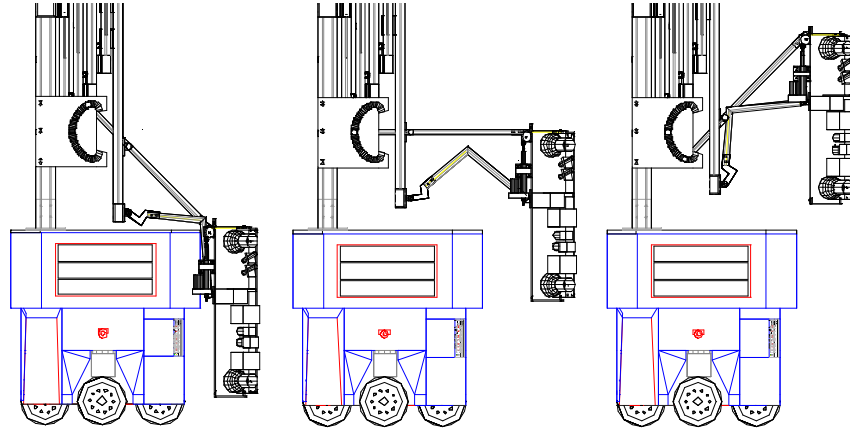


FIGURE 14: Fourbar Using the Cycloid-Slot

A kinematic analysis of this cam mechanism was performed to determine its equations of motion. The angular position, velocity, and acceleration of the follower were determined using simple harmonic motion functions along with the basic constraints of the system at hand. Several difficulties developed during the study. After developing the equations of motion, the motion was modeled physically as well as in Working Model [9]. Both models demonstrated the complexity of the motion. As the lever reaches half way through the slot, it reaches a toggle position. Without an external load applied, the lever's motion becomes unpredictable at this point. This could physically be felt on the prototype, and was seen using the Working Model software package. To avoid this toggle, the design would have to incorporate another type of force. This concept, unique and challenging, became too time consuming to pursue. A detailed analysis of the cycloid-slot, however, is shown in Chapter 4.

D.4.3.7 Linear Actuator

The second concept for actuating the fourbar was proposed to crank the driver through a pivoting linear actuator attached to Element 6 (See Figure 15). Using the torque requirements calculated by the dynamic force analysis, the actuator selection could be made. Input torque was calculated at 523 in.-lb. The goal was to keep the actuator as close as possible to Element 6 so the weight of the actuator did not overly affect the CG of the system. The actuator, however, must be mounted so that the mechanism could collapse into a compact stowed position. To determine the mounting locations, a simple graphical synthesis was performed to track the motion of a linear actuator attached to the mechanism. The pivot location on the driver link was

imperative in determining the load capacity and the stroke of the actuator. A choice to use a 200 lb. actuator approximately 3 in. from the pivot point provided a continuous torque rating of 600 in.-lb. To mount this along the driver link, at 3 inches from the joint, required a stroke of approximately 7 in. The actuator selected is the 8 in. Bug series linear actuator manufactured by Ultra Motion. The actuator weighs approximately 3 lb., and has a maximum speed of 2.5 in./sec. The actuator was mounted using the same hinge joints as the fourbar.

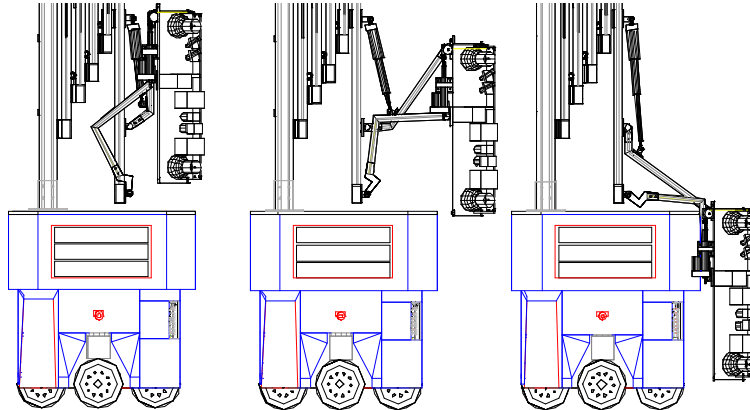


FIGURE 15: CPS-IE Fourbar

D.4.3.8 **Manufacture**

After the iterative design process was complete, the construction of the fourbar began. The design of this mechanism was much more complex than the fourbar on CPS-I. Both driver and follower links on this mechanism were angled to allow unobstructed motion. The links were constructed from Item's 28 x 28 mm. aluminum extrusions. The angles were machined and the extrusions were held together using aluminum braces. Because of the dynamic loads calculated at the joints, special hinges were made to act as the mechanism's pin joints. The hinge housings were manufactured using a Computer Numerically Controlled (CNC) milling machine. Each housing holds two light series ball bearings rated at a dynamic load of 288 lb. As previously stated, the maximum dynamic load on the joint by a pin bolted to the fourbar links was calculated as 32.6 lb.

D.4.3.9 **Controlling the Motion**

Once installed, the fourbar mechanism worked well during retraction. The actuator did, however, experience difficulty when extending. The fourbar is dropped down during the actuator's extension. Thus the controller must be programmed to account for the change in the load on the actuator due to the change in torque as the fourbar rotates out and the CG changes position. The closed loop gain on the linear actuator was set to produce smooth motion when the actuator had substantial and a near constant amount of load on it. Conversely, when the actuator has a light load on it, when it is close to the stowed position, an unsmooth, "jerky" motion is experienced. Controlling the actuator under a varying load condition is the root of the problem. To solve this problem, a compression spring was placed inline over the actuator rod. When the fourbar nears the stowed position, the housing of the actuator butts up against the spring, and the spring starts to compress. Without the spring, the load on the actuator nearly goes to zero when

the fourbar approaches the stowed position. With the spring, as the actuator approaches the stowed position, the compression of the spring increases the load on the actuator. The end result is a near constant force seen by the actuator throughout its entire stroke. The closed loop gain is tuned to this near constant force to produce a very smooth motion throughout the entire operating range of the fourbar mechanism.

D.4.4 CPS-IE Panning Mechanism

A new constraint on the DAP was added during the design of CPS-IE. The DOE wanted the system to detect dents in the storage drums. Dent-detection requires a laser mounted on the DAP to sweep approximately 93° while scanning across the drum’s surface. The rotation occurs along a vertical center of rotation of the DAP. The immediate concept envisioned was to simply place a motor and gearhead on top of this axis and allow it to directly drive the motion. A linear actuator was chosen to perform the motion because the PID controller code for the linear actuator was well established and field-tested by the controls-group. Using a linear actuator to perform rotary motion was a challenge the mechanical-design-group faced.

D.4.4.1 Torque Requirements

The first step in designing the panning mechanism was to calculate the amount of torque necessary to rotate all components of the DAP using a prescribed angular acceleration. Each of the nine components along the axis of rotation has its own individual mass (m) and center of gravity (CG), located a perpendicular distance (d) from this axis of rotation. The mass moment of inertia (I) of each of these components was calculated according to its definition. According to Norton [4], This second moment of mass is defined as the product of the mass and the square of the distance to the axis of rotation:

$$I = md^2 \tag{5}$$

Knowing the CG coordinates (x, y, z) of each of the camera package’s components with respect to the axis of rotation simplifies the calculation of the distance d:

$$d = \sqrt{x^2 + y^2 + z^2} \tag{6}$$

Using this calculation along with its mass, a total inertia is determined by summing the component inertias. Chapter 4 details the values calculated and also displays the dynamic analysis of this rotation. In summary, the total moment of inertia calculated for the camera package is 0.857 lb.-in.-sec². According to the dynamic analysis, the torque produced by the rotation of the camera package can range from 0.857 in.-lb. for a 0.5 rad/sec² angular acceleration to 1.3 in.-lb. for 1.5 rad/sec² acceleration. Using these calculations, the linear actuator can be specified.

D.4.4.2 Actuation

To acquire rotation from a purely translational source, a cabling system was developed. The initial system simply attached a 0.0625 in. diameter cable to a linear actuator which was mounted parallel to the DAP's axis of rotation. The cable then runs to a 1 in. radius pulley which simply changes the direction of the cable by 90 degrees. The cable is then wound around and attached to a 2in-diameter pulley, approximately 1.75 in. from the center. This offset creates a moment about the pulley’s axis of rotation. The center of the pulley is attached to the axis of

rotation of the camera package. Thus, as the pulley rotates, the camera package rotates through the same angle. This setup is only viable when the linear actuator is retracting. Upon extension, the cable cannot be pushed. To create a torque in the opposite direction, a linear extension spring is attached 180 degrees around the pulley from the cable attachment point. The spring creates an opposing force to the actuator as the pulley is rotated. In essence a couple is established. The spring is then capable of returning the mechanism back to its original position due to the potential energy stored within the spring.

One undesirable effect of this concept was the linearity of the spring load. As the actuator retracted, the spring was extended. Because the load is linearly dependent on the extension, the cable load increased linearly. This change in load directly effected the controls of the linear actuator. As was seen with the fourbar actuator, the actuation is smoother when the load is constant. As this load changes, the actuator must be controlled to maintain a smooth sweeping motion. It was vital to the camera package to have a very smooth and uninterrupted, constant angular velocity while data was being taken. With the changing load, the actuator was constantly undergoing control changes that result in accelerations placed on the camera package. These results were unacceptable, thus a new concept was developed.

D.4.4.3 Dual-Pulley Concept

To maintain a smooth, uninterrupted rotation the torque measured along the axis of rotation of the camera mechanism must be constant. Reconsidering the first concept, the torque created on the pulley by the spring increased as the spring was extended, and decreased as the spring was retracted. The goal was to maintain a constant torque during this rotation. One method of achieving this was to vary the spring's attachment location on the pulley as the rotation occurred. An effective method of obtaining this is to create a cam-like effect that would change the radius of the spring's attachment location while keeping the actuator's attachment location radius constant. Recall, the torque generated along the DAP's axis of rotation was a result of the distance or radius from the axis of rotation to the attachment location. Since torque is the product of the load created by the spring and its respective radius from the axis of rotation, the idea was to decrease this radius as the load increased. A unique dual-pulley concept was developed to perform the necessary radius change (See Figure 16).

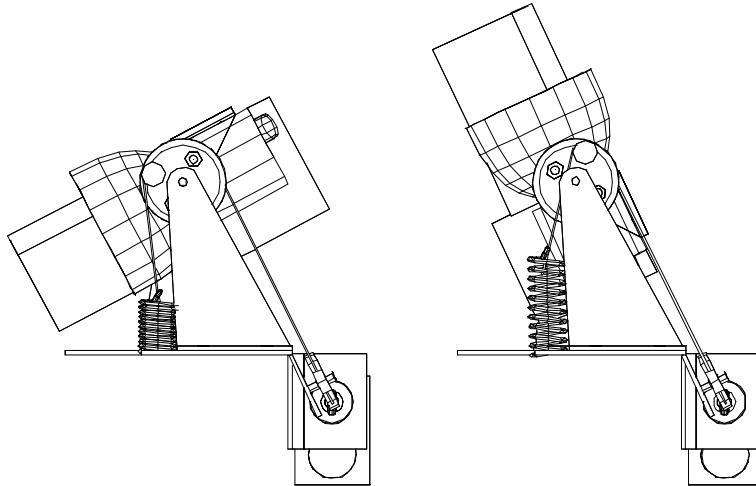


FIGURE 16: CPS-IE Panning Mechanism (top)

The connection point on the 2 in.-radius pulley that the actuator rotates remains the same. A second smaller radius pulley, attached to the extension spring, is then positioned on the larger pulley, but offset to the axis of rotation. As the DAP rotates, the actuator radius stays the same, however the spring's radius decreases. Thus as the spring gains potential energy, the lever arm used through the pulley to create the torque decreases. The result is a constant torque about the axis of rotation. The linear actuator virtually sees a constant torque applied by the spring.

D.4.4.4 *Manufacture*

Choosing the components of the mechanism (See Figure17) involved selecting a linear actuator, an extension spring, and the housing that the camera package pivots within. The linear actuator was chosen based on the selection of the fourbar actuator. The same actuator used in that mechanism was also used for the panning mechanism. This actuator is a 2 in. model of the Bug series by Ultramotion. An in depth review was done to select the proper spring. Initially the concept was to use a torsion spring, however the immediate availability was limited compared to that of extension springs. Several factors were studied when searching for the correct spring: free length; maximum extension; preload extension; initial tension; spring rate; and overstressing extension. The housing created to hold the package was a simple cutout of 6061 T6 Aluminum plate with a top and bottom bracket each containing a small roller bearing. The entire assembly weighs approximately 22 lb.

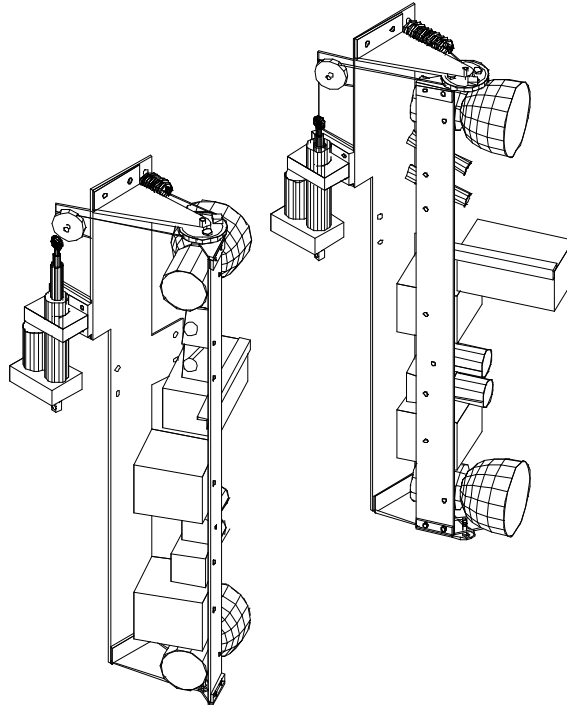


FIGURE 17: CPS-IE Panning Mechanism (isometric)

D.4.5 CPS-II Concepts

D.4.5.1 Fixed Mast Concept

CPS-II was designed for the second-generation system, ARIES-II, which will be deployed at Los Alamos National Labs (LANL) in late Fall 1997. LANL does not need the vertical reach capability of the first generation CPS-I and CPS-IE, because they do not stack 55-gallon drums over 3 high and do not stack 85-gallon drums over 2 high. LANL safety requirements dictate these conditions. To save cost and reduce maintenance, it was decided to use a fixed mast on the CPS deployed at Los Alamos. A 2.8 m single element mast with a 100 mm. by 100 mm. extruded aluminum profile is used (See Figure 18). Parker Hannifin Corporation, Hauser Division, produces this model, HLE100C. The profile houses a rolling carriage that is attached to a timing belt. A pulley connected to a 25:1 spur gear reducer and a 12 VDC brushless servomotor on the bottom drives the belt. The carriage rides along a slot outside the housing, allowing attachment of a plate to lift the payload. A fourbar mechanism and a DAP are each mounted to the plate, and the entire assembly is then mounted on top of the K3A.

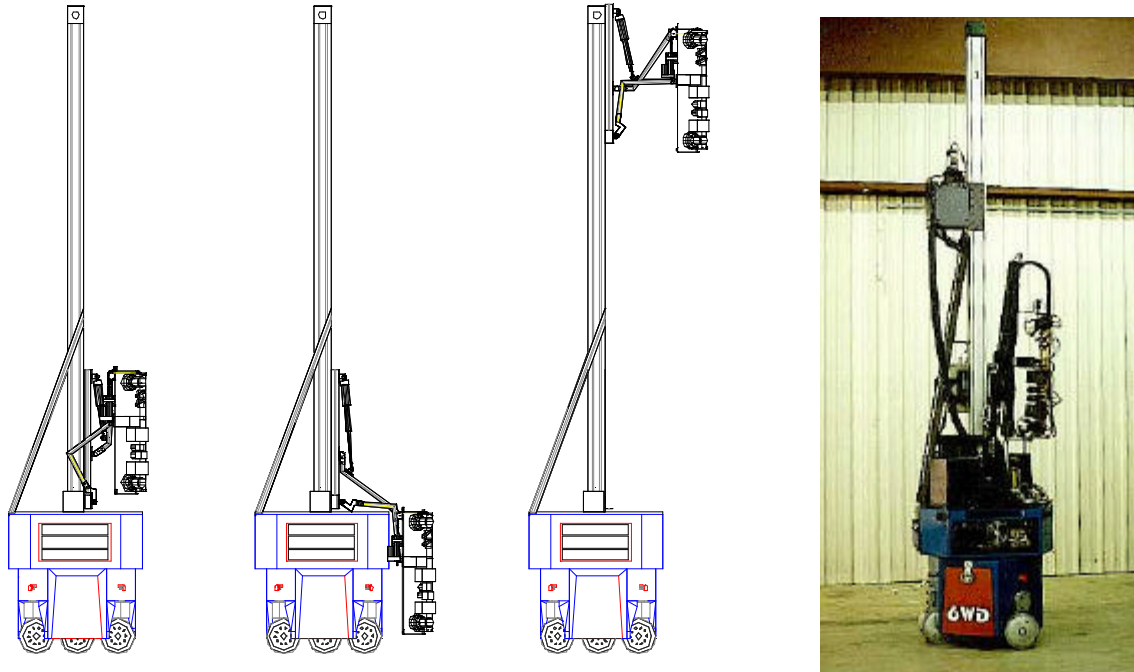


FIGURE 18: ARIES-II

D.4.5.2 Mast Selection

The dimensions of the mast depend on the type of mast chosen and the length of travel necessary. At LANL, the maximum height the DAP needs to reach is 120 in. The fourbar mechanism attached to the carriage, moves the camera from 12 in. to 50 in. off the ground. The lift raises the camera package an additional 70 in. The model chosen contained an extended carriage; that is, the carriage itself is extended in length to provide a larger bending moment capacity as well as a larger mounting platform. For a 70 in. length of travel the extrusion length was 100 in. and the total mast length was 110 in. The excess length was due to the pulley locations as well as a safety region in case the unit is overdriven. Thus at 110 in. mounted atop the 32 in. high K3A gives a stowed height of 142 in., or just less than 12 ft. Recall that the LANL system was required to reach 10 ft., which was 6 ft. less than CPS-IE.

D.4.5.3 Actuation

Motor and gearhead selection for this unit was based, once again, on the 30 lb. camera package CG located 3 ft. from the carriage CG. One consideration in this selection is the inertia mismatch of the system. This is a ratio of the inertia of the load and gearhead to the inertia of the motor's rotor. This value must be less than 5:1, but is preferred to be as low as a 1:1 ratio. As the inertia mismatch decreases to this 1:1 ratio, this decreases the chance of regeneration. Regeneration occurs if the mass of the load causes the motor to spin, thus acting as a generator. The gearhead recommended by Hauser is a 25:1 spur gear reducer, and the motor combination chosen was a servo amplifier and brushless motor at 12 VDC. The drive system was chosen such that the drive shaft runs through the drive pulley and out the other side. This shaft was then attached to a braking system as well as an encoder allowing the system to feedback position information.

The 30 lb. DAP, when extended out horizontally, was 3 ft. away from the mast, resulting in a moment on the carriage of 90 ft.-lb. The carriage bearings on the mast are capable of loads up to 250 ft.-lb. The 10 in./sec. anticipated vertical speed of the DAP is well under the 30 lb. rated 150 in./sec. speed for the linear-carriage unit.

D.4.5.4 Fourbar Mechanism

The only changes made to the fourbar on CPS-II are the link lengths. The driver and follower links had to increase to 25 in. This results in a 750 in.-lb. torque capacity, or a 24% increase from CPS-IE. A 500 lb. linear actuator similar to the one used on the previous fourbar was used to drive the mechanism. The panning mechanism was changed on CPS-II. The panning action is now achieved with a rotary actuator to reduce the complexity of the linear actuator, cable-pulley, and cam-like panning mechanism used in CPS-IE. A gearhead between the rotary actuator and the DAP output shaft, allows the user to change the ratio between the motor and the camera package, thus giving more angular velocity options.

D.4.5.5 CPS-II Results

CPS-II represents a fixed mast system and weighs approximately 223 lb., a 44% decrease from the original prototype. This unit is capable of inspecting a stack of three 55-gallon drums and two 85-gallon drums. The system maintains 3 DOFs, however there are much fewer moving parts on this system. Power consumption is calculated as 0.099kW. Expected throughput is up to 100 drums/hour with dent detection and over 300 drums/hour without drum detection. These throughput numbers are expected to increase significantly upon further enhancements to the system. The system is set to be deployed at Los Alamos National Laboratories in late 1997.

D.5 ANALYSIS

D.5.1 Center of Gravity Analysis

The CG analysis of each system is important because it yields the total weight of the mechanism, and it also gives the theoretical location of an assumed point mass of each component which aids in determining the stability of the system. A composite method of determining the CG of the overall system is used. Beer and Johnston's method defines the system's overall moment about a point as being equal to the sum of the individual component moments. Using this definition, the three components of the overall CG can be computed [1]. The equations are as follows:

$$\bar{X} \sum W = \sum \bar{x}W \quad (7a)$$

$$\bar{Y} \sum W = \sum \bar{y}W \quad (7b)$$

$$\bar{Z} \sum W = \sum \bar{z}W \quad (7c)$$

Using these equations requires that the weight and CG of each individual component of the composite system be determined with respect to a global coordinate system. This information was determined and placed into a spreadsheet, where the calculations were performed. An overall CG was determined for each element so the position changes could be modeled, thus allowing the user to determine worst-case scenarios. This information is vital to the stability analysis of each system.

The mass of each component on CPS-I was determined by either removing the component and weighing it, or by calculating weight knowing the volume and density of the component. The component's CG was either calculated or defined by balancing it on three axes. This information was then input into a spreadsheet where the overall CG calculation was performed. The individual component weights were used again in the CG analysis of both CPS-IE and -II. To find the CG locations on CPS-IE and -II, 3-dimensional models were created using AutoCAD. The software could calculate the x, y, and z components of a line drawn from a global origin to the CG of each component. The global origin is placed at the middle of the top edge on the back of the baseplate. With all the information input, the results were tabulated (See Table 1):

| POSITION | WEIGHT(lb.) | HEIGHT(in.) | CG-X(in.) | CG-Y(in.) | CG-Z(in.) |
|----------------|-------------|-------------|-----------|-----------|-----------|
| CPS-I stowed | 395.7 | 116(top) | -1.32 | 16.58 | 30.48 |
| CPS-I maximum | 395.7 | 156(camera) | -1.32 | 16.58 | 54.47 |
| CPS-IE stowed | 261.41 | 115(top) | 0.45 | 17.85 | 29.89 |
| CPS-IE maximum | 261.41 | 192(camera) | 0.45 | 17.64 | 75.88 |
| CPS-II stowed | 203.35 | 133(top) | -1.61 | 14.35 | 27.0 |
| CPS-II maximum | 203.35 | 120(camera) | -1.61 | 13.06 | 35.75 |

TABLE 1: CPS CG Locations

The height of the cameras in each system show that each CPS was designed to reach different values. Comparing a percentage of the CG-Z coordinate to the camera height, CPS-I is 35% the total height, CPS-IE is 39.5% the total height, and CPS-II is 30% the total height. This simply states that CPS-II has the lowest CG for its total height. The important factor here, however is the location of the CG when it is stowed since the robotic platform will be in motion when the DAP is in the stowed position. Looking at a percentage decrease in the CG-Z coordinate, CPS-IE's CG was lowered 1.94%, and CPS-II was lowered 11.42%. These values will be used to calculate the stability of the system on an incline as well as during a sudden deceleration.

D.5.2 Lift Mechanism Actuator Load Analysis

D.5.2.1 Newtonian Force Analysis

To determine the applied load necessary at the actuator of these systems, static and dynamic force analyses were performed on the cables connecting the elements to the actuator (Cables 1 and 2). The dynamic loads associated with this system seemed negligible considering the small accelerations experienced during normal operation. The maximum speed attained by the Electrac 100 actuators is 2.5 in./sec. It is estimated that these actuators reach this speed in approximately 0.04 sec. Thus the acceleration is 5 ft./sec.². The force on the actuator is this acceleration multiplied by the mass lifted by the cable attached to the actuator. The static tension on each cable was calculated using static equilibrium. The combined load on the actuators from Cables 1 and 2 as a function of their actuation was also determined (See Figure 19). Cable 2 of CPS-I had a theoretical static load of 153 lb. This yields a dynamic load of 177.3 lb at 5 ft./sec.². To verify this, a load cell was placed inline with cable 2 on CPS-I. The system was then raised and lowered, as it would be under a normal operating environment. The load cell data was compared with the calculated static force. The average load determined statically was 151.7 lb., and the average load obtained with the load cell was 146.9 lb. These values fall within 3% of one another. As far as the dynamic loads, a maximum of 175.9 lb. was reached when accelerating and decelerating the lift. This is approximately 13% above the maximum calculated static value, and within 1% of the theoretical dynamic assumption of 177.3 lb.

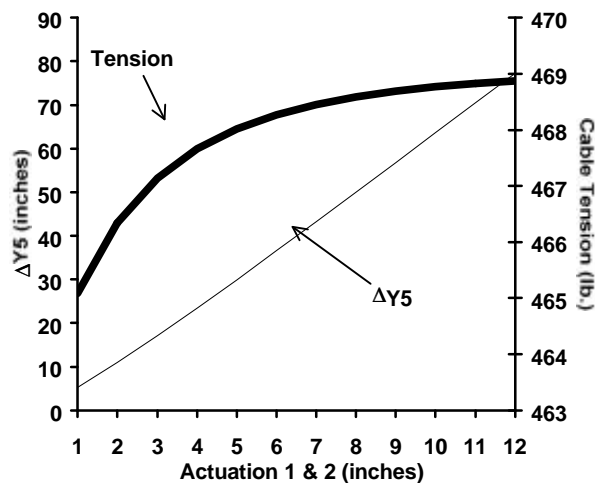


FIGURE 19: CPS-I Actuator Load

The same procedure was followed on CPS-IE, however at the time of testing the 13X pulley configuration was assembled. To validate the static-load analysis that was performed on CPS-IE (See Figure 20), a load cell was again attached inline with the cable being actuated. This time the total load capacity of the system was tested; that is, the full load applied to the single actuator was determined from Cables 1 and 2 spliced together to yield the maximum static cable tension. The data was taken using three different loads on element six. Thus the values could be compared to three distinct static force analyses.

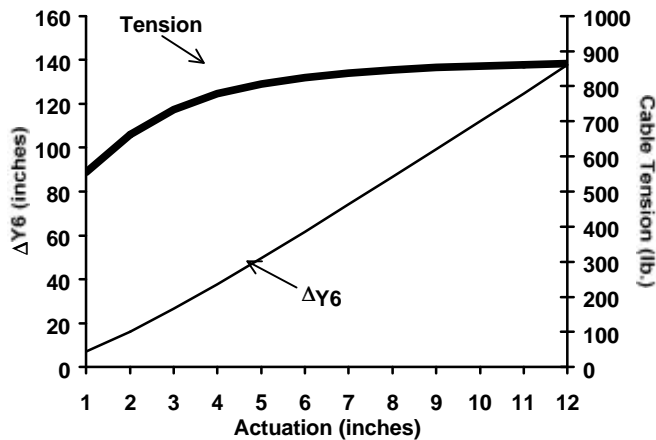


FIGURE 20: CPS-IE 13X Actuator Load

The dynamic load characteristics encountered with this configuration were very similar to the characteristics on CPS-I. To calculate these loads an acceleration of 5 ft./sec.² is used along with a maximum static cable load of 859 lb. This results in a maximum dynamic load of 992 lb. During testing the largest load measured with a load cell was 957 lb., which is approximately 9% higher than the highest calculated static load of 859 lb., and within 2% of the estimated theoretical value. This 9% compared to the 13% of CPS-IE is then used as the basis when selecting the actuator and the cables. Because these dynamic loads were very similar, this 13% value was considered as a reasonable assumption when performing future dynamic load calculations on a similar CPS.

The loads on the redesigned 7x system were calculated using the same approach as used on CPS-I and the 13x system discussed above. Using this information, a static analysis was done. The resulting actuator static load under this configuration was 488 lb. Using the information gained from the previous two examples, the expected dynamic load should be near 13% of this value. A quick look at Newton's second law shows that the 488 lb. converted to a system mass accelerated at 5 ft./sec.² will yield a maximum dynamic load of 564 lb.

D.5.2.2 Mechanical Advantage Force Analysis

To simplify the tedious process of performing a static load analysis, a method was developed by viewing the mechanical advantage of each element and their respective weights. Recalling the definition of MV, the ratio of output load to input load is equal to the ratio of input distance to output distance.

$$MV = \frac{F_{OUT}}{F_{IN}} = \frac{d_{IN}}{d_{OUT}} \quad (8)$$

Thus the input force is equal to the product of the output force with the distance ratio (DR), where the distance ratio is the output distance over the input distance. The DR is obtained by inspecting the kinematics of the system. Once the DR of each element is known, along with their respective weights, the input force can be calculated for each element. The sum of these respective forces will yield a total input force that determines the actuator load capacity.

$$F_{IN} = \sum \left(\frac{F_{OUT}}{MV} \right) \quad (9)$$

Recall, however, that the DR of each element must be exact to obtain a realistic value. Theoretical DR will yield a theoretical actuator load, which is not a proper value to base the actuator selection on. One must acquire the actual DR of the particular element. This is done through a kinematic analysis of the mechanism's motion. This offers another method to calculate actuator load capacity, and is verified on the 7x system as shown in Table 2.

| Element # | Local MV | Weight (lb.) | Weight/MV (lb.) |
|-----------|----------|--------------|-----------------|
| 2 | 1/1 | 11.1 | 11.1 |
| 3 | 1/1 | 11.6 | 11.6 |
| 4 | 1/3 | 12.0 | 36 |
| 5 | 1/5 | 13.8 | 69.0 |
| 6 | 1/7 | 50.0 | 350.0 |
| | | Total: | 477.7 |

TABLE 2: CPS-IE PC7 Static Load Calculation (MV Method)

Recall that PC7's theoretical static load was determined as 488 lb. This estimation is within 2% of the actual value. The method was used as a fast response method of determining actuator load capacity for different configurations.

D.5.3 Actuator Life Cycle Analysis

An analysis of the Electrak 100 linear actuator used in both CPS-I and -IE, defines relevant life expectancy. The Warner Electric catalog displays a performance curve for the Electrak 100. The plot displays the life (number of cycles) vs. the rated load (percentage). The actuator load capacity for each system was 500 lb. CPS-I actuator-1 required a maximum load of 280 lb., which according to the chart will last a minimum of 27,000 cycles. Actuator 2 on this system with a maximum load of 150 lb. should last 33,000 cycles. CPS-IE's actuator received up to its maximum load capacity. At this maximum, the actuator is expected to perform 10,000 cycles.

D.5.4 Cable Selection Process

During cable selection for the lift mechanisms a cable property-study was performed. Cables having a minimum breaking strength of 8-12 times the working load were selected. CPS-I had a 3/16 in. diameter, 7 x 19-strand nylon coated stainless steel cable with a minimum breaking strength of 3700 lb. CPS-IE had a 5/16 in. diameter, 7 x 19-strand nylon coated stainless steel cable with a minimum breaking strength of 9000 lb. The pulley contact radii of each CPS was

accounted for, and was designed to keep 1/3 of the cable circumference in contact with the pulley circumference. Also the pulleys' minimum diameters were designed to be 25 times the cable diameter to minimize bend radius stresses on the cables. Minimum fatigue life for the cables was calculated at 300,000 cycles running on a 3.25 in. diameter pulley. Cable stretch was also calculated. For CPS-I, the 3/16 in.-diameter cable stretches 0.05in per 10 in. cable length under a 1000 lb. load. The 5/16 in.-diameter cable on CPS-IE stretches 0.025 in. per 10 in. cable length under the 1000 lb. load. These factors were taken into account as cables were selected.

D.5.5 Fourbar Actuator Load Analysis

To obtain the loads associated with the fourbar mechanism, a dynamic force analysis was performed. To perform a dynamic analysis a kinematic analysis was first done. An assumption was made for an initial angular velocity, allowing the kinematic state of each link to be determined. A dynamic analysis was then completed using these kinematic values. This dynamic analysis gave the resultant loads on each joint, as well as the required input torque, which aids in selecting the fourbar's actuator. Here, the analysis is broken down into four distinct steps, each important to the solution. The first step was to set the mechanism's initial conditions. Step 2 calculated the second mass moments (moments of inertia) of each moving link. Step 3 was the kinematic analysis, and finally step 4 was the dynamic force analysis. Each step is detailed in the following paragraphs.

Step 1 was important because it initialized the process. The inputs included the geometry as well as the mass of each link. The goal was to minimize the pin joint load while changing the shape of the links. The initial positions, angular velocities, and angular accelerations were also input during this step. Actuator specifications can be input to determine the most efficient model for the job at hand. Inputs for the fourbar on CPS-IE are shown in Table 3.

| Initial Condition Inputs | Link 1 | Link 2 | Link 3 | Link 4 |
|---|---------------|---------------|---------------|---------------|
| Mass(kg.) | N/A | 1.41 | 10.17 | 1.84 |
| Effective Lengths(m.) | 0.508 | 0.508 | 0.28 | 0.508 |
| Angular Position(deg.) | 0 | 60 | 0 | 60 |
| Angular Velocity(rad./sec.) | 0 | 0 | 0 | 0 |
| Angular Acceleration(rad./sec. ²) | 0.4 | 0 | 0.4 | 0 |

TABLE 3: CPS-IE Fourbar Initial Conditions

Step 2 of the analysis was to calculate the second moments of mass, or moments of inertia of each of the three moving links. The geometry of each link, including the location of its center of gravity (CG) must be known. The CG locations are later used as the point at which the link's linear acceleration occurs. The total CG of each link was found by summing up the mass moments of each known segment of the link and setting this sum equal to the overall mass moment of that particular link. The moments of inertia were then calculated for each element using the parallel axis theorem to account for each individual moment of inertia within the link.

The moments of inertia were used in calculating the amount of torque necessary to provide the desired motion. An attempt was made to minimize the second mass moments because of their tendency to resist angular acceleration [4]. The moments of inertia calculated for each of the three moving links were found using the parallel axis theorem along with the concept that each link may be broken into a composite of several items. This theorem and its results are shown below:

$$I_{ZZ} = I_{GG} + md^2 \quad (10)$$

This equation is applied to the individual links' geometry. The hinges for each link are approximated as cylinders with a 14 mm. radius. The links are 28 x 28 mm. extrusions and are assumed to be rectangular bars. Using equation 10 along with the links' geometry and CG locations, the moments of inertia are shown in Table 4.

| Link Number | 2 | 3 | 4 |
|---------------------------------------|-------|-------|-------|
| Moment of Inertia(kg-m ²) | 0.058 | 0.546 | 0.087 |

TABLE 4: CPS-IE Fourbar Links' Moments of Inertia

There is a calculated value for Link 3, however recall that moment of inertia is the resistance to angular acceleration, and the moment of inertia only effects a rotating body. There is no effect on a translating body. Recall that Link 3 is the DAP which undergoes pure translation; thus, this calculation is unnecessary. It is included so that the document could be used at another time using a fourbar mechanism that requires a coupler rotation.

Step 3 included a partial kinematic analysis of the mechanism. The position and the linear acceleration of the link's CG were calculated. Each was found analytically using kinematic equations of a fourbar mechanism. The following equations are used:

$$CG \text{ position vector} : R_{PA} = pe^{jq} \quad (11)$$

$$CG \text{ Linear Acceleration} : a = \mathbf{ra}je^{jq} - \mathbf{rv}^2e^{jq} \quad (12)$$

To find the position vectors on Link 2 from the CG to the hinges, the CG properties were found to be 172.1 degrees and 0.194 m. from hinge 1-2, and 4.8 degrees and 0.316 m. from point hinge 3-2 (See Figure 21).

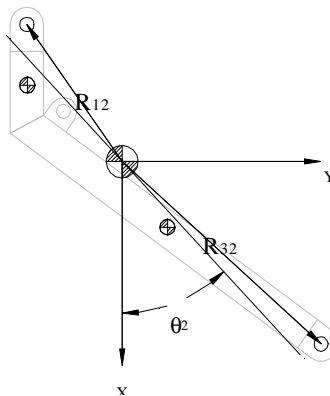


FIGURE 21: Fourbar Link 2

$$R_{12} = 0.194e^{j(q_2 + 172.1^\circ)} \quad (13)$$

$$R_{32} = 0.316e^{j(q_2 + 4.8^\circ)} \quad (14)$$

To find the position vectors on Link 3, the CG was determined to be 191.8 degrees and 0.373 m. from hinge 2-3, and 208 degrees and 0.111 m. from hinge 4-3 (See Figure 22).

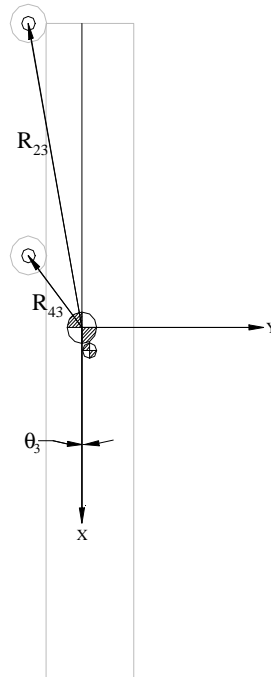


FIGURE 22: Fourbar Link 3

$$R_{23} = 0.373e^{j(q_3 + 191.8^\circ)} \quad (15)$$

$$R_{43} = 0.111e^{j(q_3 + 208^\circ)} \quad (16)$$

To find the position vectors on Link 4, the CG was determined to be 192.3 degrees and 0.279 m. from hinge 1-4, and -14.2degrees and 0.243 m. from hinge 3-4 (See Figure 23).

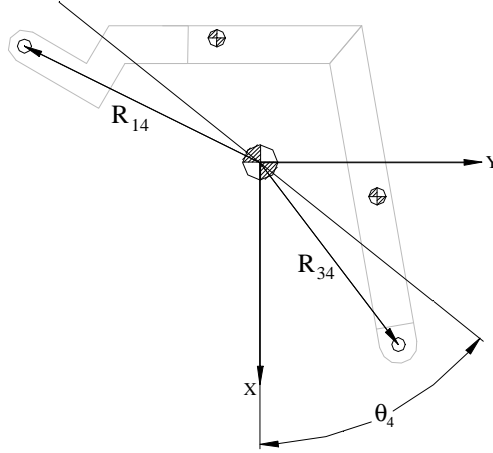


FIGURE 23: Fourbar Link 4

$$R_{14} = 0.279e^{j(q_4+192.3^\circ)} \quad (17)$$

$$R_{34} = 0.243e^{j(q_4-12.8^\circ)} \quad (18)$$

The linear accelerations of each CG with respect to a local XY coordinate system were then found. The CG on link 2 is -7.9 degrees from the imaginary straight line that joins the two hinges and is 0.194 m. from hinge 1-2. Thus the linear acceleration of the CG on link 2 is:

$$a_{CG2} = 0.194\mathbf{a}_2 j e^{j(q_2-7.9^\circ)} - 0.194\mathbf{v}_2^2 e^{j(q_2-7.9^\circ)} \quad (19)$$

Recall there is no angular acceleration of link 3. The linear acceleration for every point on this body is the same, which is the linear acceleration of hinge 2-3 where the input and coupler join. Thus the linear acceleration of the CG of link 3 is the linear acceleration of its hinge 2-3 location:

$$a_{CG3} = a\mathbf{a}_2 j e^{jq_2} - a\mathbf{v}_2^2 e^{jq_2} \quad (20)$$

The linear acceleration of the CG on link 4 was then found. The CG of this link is 12.3 degrees from the imaginary straight line that joins the link's two hinge points and is 0.279 m. from the hinge 1-4.

$$a_{CG4} = 0.279\mathbf{a}_4 j e^{j(q_4+12.3^\circ)} - 0.279\mathbf{v}_4^2 e^{j(q_4+12.3^\circ)} \quad (21)$$

A similar set of analytical equations exists for all angular velocities and accelerations, however this analysis is testing for maximum joint forces that occur dynamically during acceleration. The CG position and its respective linear acceleration are the only calculations necessary to proceed with the dynamic analysis, which is the goal of this study.

Step 4 was the dynamic analysis, to find the reaction forces at the pin joints along with the required input torque placed on the driving link to provide the required kinematic conditions. The dynamic analysis used in this document is based on Newtonian mechanics. To perform a dynamic

analysis of the fourbar, force and torque equilibrium equations were found for each link. A total of nine equations and nine unknowns were produced, which are easily input into a 9x9 matrix and simultaneously solved using a matrix solver.

The results of the dynamic force analysis on the fourbar on CPS-IE are shown in Table 5. These results were found by setting the initial position to a specific location, then accelerating the driving link, thus modeling the worst case of starting the fourbar in motion from different locations along its path. The maximum torque-input necessary was calculated as 523 in.-lb. (59.093 N.-m.) when the driver link is parallel to the floor. This value was then used to determine the actuator specifications for the mechanism.

| θ_2 (deg.) | F_{12x} (N.) | F_{12y} (N.) | F_{32x} (N.) | F_{32y} (N.) | F_{43x} (N.) | F_{43y} (N.) | F_{14x} (N.) | F_{14y} (N.) | T_{12} (N.-m.) |
|----------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|---------------------|
| 60 | - | -31.963 | 131.06 | 32.03 | 29.607 | 33.063 | 11.379 | 33.125 | 51.571 |
| 70 | - | -31.196 | 123.40 | 31.247 | 21.796 | 31.954 | 3.56 | 31.981 | 55.777 |
| 80 | - | -30.523 | 116.98 | 30.556 | 15.283 | 30.915 | -2.955 | 30.907 | 58.309 |
| 90 | - | -29.895 | 111.23 | 29.91 | 9.498 | 29.91 | -8.734 | 29.867 | 59.093 |
| 100 | - | -29.281 | 105.75 | 29.277 | 4.05 | 28.918 | - | 28.84 | 58.105 |
| 110 | - | -28.652 | 100.20 | 28.629 | -1.402 | 27.923 | - | 27.813 | 55.377 |
| 120 | -108.14 | -27.98 | 94.221 | 27.938 | -7.235 | 26.905 | - | 26.767 | 50.991 |
| 130 | - | -27.22 | 87.277 | 27.162 | - | 25.833 | -32.13 | 25.671 | 45.076 |
| 140 | -92.384 | -26.296 | 78.485 | 26.223 | -22.51 | 24.64 | - | 24.458 | 37.809 |

TABLE 5 Fourbar Dynamic Force Analysis Results

The maximum radial load at the pin joints is -32.6 lb. (-145 N.) on F12x at 60 degrees. This is considerably lower than the maximum dynamic load rating of the bearings (288 lb.) used at the pin joints. The life expectancy of these roller bearings calculated using an L_{10} life rating, is over 2 billion revolutions. The General Bearing Corporation (GBC) predicts the life of their bearings in several ways. This method, common to most manufacturers through the Anti-Friction Bearing Manufacturers Association (AFBMA), is known as L_{10} , or rating life [8]. This value represents the number of revolutions that 90% of a similar group of bearings will complete before failure. GBC's calculation is shown below:

$$L_{10} = 3\left(\frac{C}{P}\right)^3 \times 10^6 \quad (22)$$

In this equation, C is the dynamic capacity of the bearing, and P is the radial load on the bearing.

D.5.6 Cycloid-Slot Analysis

The kinematic analysis of the cycloid-slot actuation concept was based on the cycloidal displacement of a follower in a slotted cam mechanism. The equations of motion were developed with a cam-follower approach using harmonic functions to represent the driver's angular-position,

-velocity, and -acceleration [5]. Harmonic functions are used because they are continuous through the first and second derivatives of displacement [4]. The jerk function is also continuous across its differentiation. The variables used in the analysis are as follows:

- θ : driver angle
- β : total range of angular travel
- D : total travel period
- d/D : normalized travel period
- α : driver angular acceleration
- C : amplitude of the sine wave
- ω : driver angular velocity

In this mechanism, zero acceleration is desired at the beginning and end of the motion. Thus a sinusoidal function is used for the angular acceleration. The harmonic function for angular acceleration is:

$$\mathbf{a}(d) = C \sin\left(2\mathbf{p} \frac{d}{D}\right) \quad (23)$$

Integrate to find the angular velocity w :

$$\mathbf{w}(d) = -C \frac{D}{2\mathbf{p}} \cos\left(2\mathbf{p} \frac{d}{D}\right) + k_1 \quad (24)$$

Applying the boundary condition $w = 0$ at $d = 0$ gives $k_1 = C \frac{D}{2\mathbf{p}}$. The end boundary condition of $w = 0$ at $d = D$ produces the same k_1 . Equation (26) can be written:

$$\mathbf{w}(d) = C \frac{D}{2\mathbf{p}} \left[1 - \cos\left(2\mathbf{p} \frac{d}{D}\right) \right] \quad (25)$$

Integrate to find the displacement:

$$\mathbf{q}(d) = C \frac{D}{2\mathbf{p}} d - C \frac{D^2}{4\mathbf{p}^2} \sin\left(2\mathbf{p} \frac{d}{D}\right) + k_2 \quad (26)$$

Apply the boundary condition that $\mathbf{q} = 0$ at $d = 0$ to yield $k_2 = 0$. To find the amplitude of the acceleration sine wave C , apply the boundary condition $\mathbf{q} = \mathbf{b}$ at $d = D$ to get:

$$C = 2\mathbf{p} \frac{\mathbf{b}}{D^2} \quad (27)$$

With all the constants determined, the position, velocity, acceleration, and jerk equations (S-V-A-J) can be written as follows:

$$\mathbf{q}(d) = \frac{\mathbf{b}}{D} d - \frac{\mathbf{b}}{2p} \sin\left(2p \frac{d}{D}\right) \quad (28)$$

$$\mathbf{w}(d) = \frac{\mathbf{b}}{D} \left[1 - \cos\left(2p \frac{d}{D}\right) \right] \quad (29)$$

$$\mathbf{a}(d) = 2p \frac{\mathbf{b}}{D^2} \sin\left(2p \frac{d}{D}\right) \quad (30)$$

$$\mathbf{J}(d) = 4p^2 \frac{\mathbf{b}}{D^3} \cos\left(2p \frac{d}{D}\right) \quad (31)$$

Note that equation (30) is the expression of a cycloid. The S-V-A-J plots for this cycloidal displacement function with sinusoidal acceleration are shown below (See Figure 24).

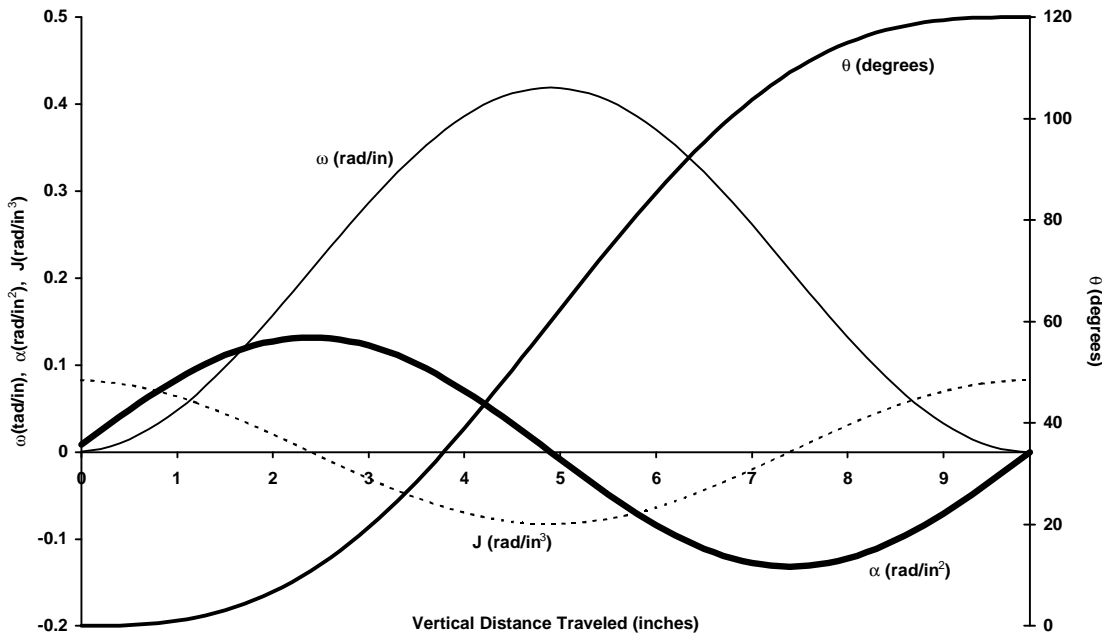


FIGURE 24: S-V-A-J Plots for Cycloidal Displacement and Sinusoidal Acceleration

A Mathcad document that plots the path of the cycloid-groove dependent on the total sweep angle, the vertical travel distance, the lever length, and a prescribed origin was developed. With this information available, the geometry can be optimized between lever length and the vertical distance traveled to obtain the optimum motion. Each input results in a unique cycloid-groove based on the equation of motion of the lever displacement. This equation can also be used to supply data to a CNC milling machine, simplifying the slot manufacturing process. One of the plots is shown on Figure 25.

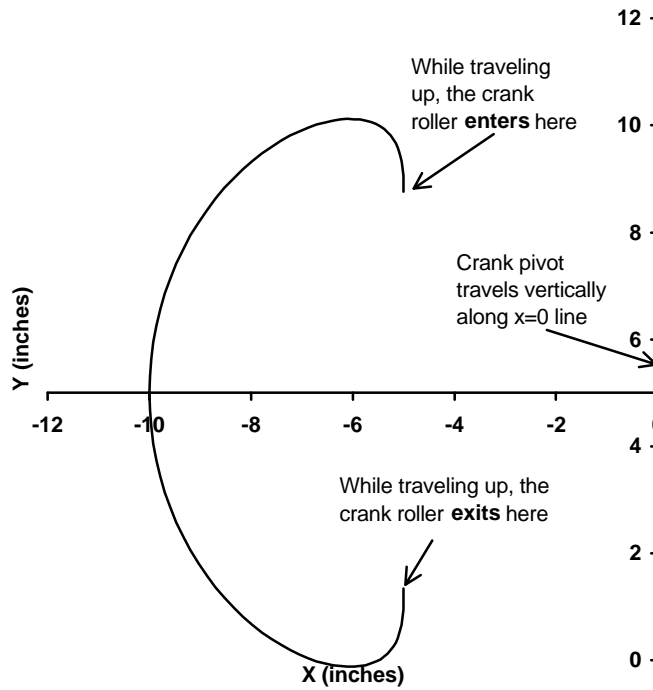


FIGURE 25: Slot Profile for Cycloidal Displacement and Sinusoidal Acceleration; D=10", L=10"

D.5.7 Panning Mechanism Analysis

To model the Panning Mechanism on CPS-IE a kinematic and dynamic study of the rotation was performed. The resulting equations were input into a Mathcad document, and a model was established enabling a user to input different geometry and components to obtain the optimum results. To develop the equations of motion of this system, the overhead geometry along with the associated variables are shown in Figure 26.

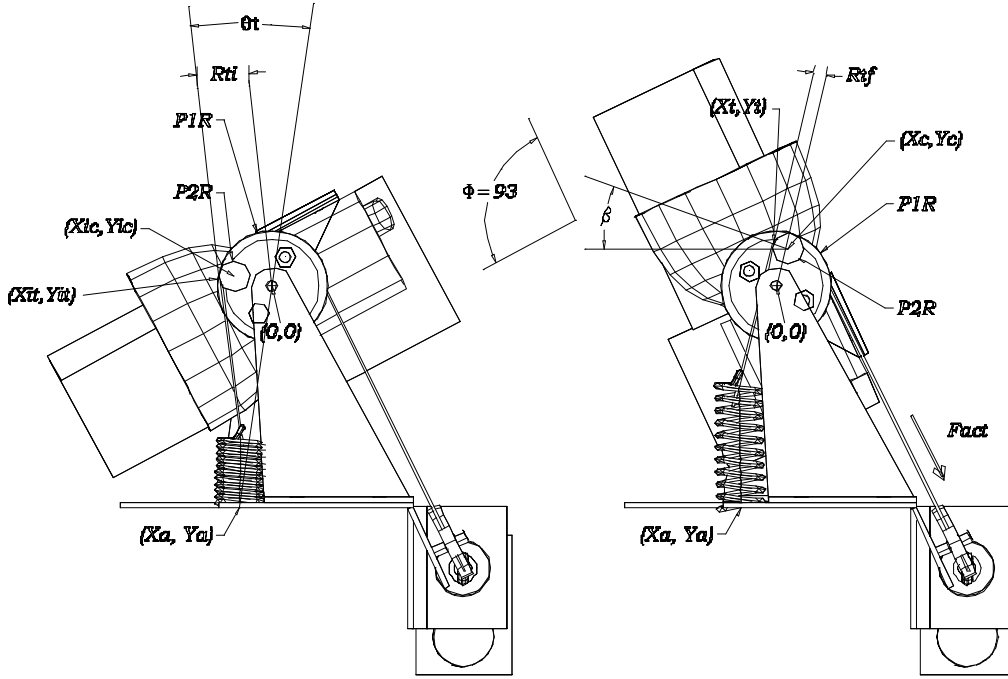


FIGURE 26: Panning Mechanism Variables on CPS-IE

The input variables for this analysis are as follows:

- ϕ \equiv angle of rotation of the data acquisition package;
- P1R \equiv radius of the large pulley;
- P2R \equiv radius of the small pulley;
- (x_a, y_a) \equiv spring attachment point to ground;
- (x_i, y_i) \equiv initial center of pulley-2 when the spring is relaxed;
- θ_t \equiv the angle formed between the tangent line of the spring and the line defined from (x_a, y_a) to the center of pulley-1.
- L_{free} \equiv free length of the spring
- L_{max} \equiv maximum extension of the spring
- $L_{preload}$ \equiv preload extension of the spring
- IT \equiv initial tension in spring
- k \equiv spring rate

The point of interest in developing the equations is the center of P2. P2 is attached to P1 so there is no translation or rotation of P2 with respect to P1. Thus the center of P2 is simply a point rotating about an axis (the axis of rotation of P1). To calculate x-y coordinates, a rotational transformation is performed.

$$x_c(\mathbf{f}) = x_{i_c} \cos(\mathbf{f}) - y_{i_c} \sin(\mathbf{f}) \quad (32)$$

$$y_c(\mathbf{f}) = x_{i_c} \sin(\mathbf{f}) + y_{i_c} \cos(\mathbf{f}) \quad (33)$$

Referring to Figure 28, several angles and distances were found using simple Euclidean Geometry:

$$D(\mathbf{f}) = \sqrt{(x_a - x_c(\mathbf{f}))^2 + (y_a - y_c(\mathbf{f}))^2} \quad (34)$$

$$\mathbf{q}(\mathbf{f}) = a \sin\left(\frac{P2R}{D(\mathbf{f})}\right) \quad (35)$$

$$\mathbf{a}(\mathbf{f}) = a \tan\left(\frac{y_a - y_c(\mathbf{f})}{x_a - x_c(\mathbf{f})}\right) \quad (36)$$

$$\mathbf{b}(\mathbf{f}) = \frac{\mathbf{p}}{2} - \mathbf{q}(\mathbf{f}) - \mathbf{a}(\mathbf{f}) \quad (37)$$

The coordinates of the connection point of cable 2 and P2 (x_{da}, y_{da}) will be tracked using a rotational transformation similar to P2's center point. Also the initial and angle-dependent tangential points of contact between cable 2 and P2 (x_t, y_t) are defined:

$$x_{da}(\mathbf{f}) = x_{i_t} \cos(\mathbf{f}) - y_{i_t} \sin(\mathbf{f}) \quad (38)$$

$$y_{da}(\mathbf{f}) = x_{i_t} \sin(\mathbf{f}) + y_{i_t} \cos(\mathbf{f}) \quad (39)$$

$$x_{i_t} = x_{i_c} + P2R \cos(\mathbf{b}(0)) \quad (40)$$

$$y_{i_t} = y_{i_c} - P2R \sin(\mathbf{b}(0)) \quad (41)$$

$$x_t(\mathbf{f}) = x_c(\mathbf{f}) + P2R \cos(\mathbf{b}(\mathbf{f})) \quad (42)$$

$$y_t(\mathbf{f}) = y_c(\mathbf{f}) - P2R \sin(\mathbf{b}(\mathbf{f})) \quad (43)$$

θ_t , the tangent line angle is:

$$\mathbf{q}_L(\mathbf{f}) = \frac{\mathbf{p}}{2} - \mathbf{b}(\mathbf{f}) - a \tan\left(\frac{y_a}{x_a}\right) \quad (44)$$

The effective radius of transmitted torque (R_t) is:

$$R_L(\mathbf{f}) = (x_a^2 + y_a^2)^{\frac{1}{2}} \sin(\mathbf{q}_L(\mathbf{f})) \quad (45)$$

Using spring information initially defined, the amount of torque(T) placed at the center of P1 can be calculated.

$$T(\mathbf{f}) = (k\Delta x(\mathbf{f}) + IT)rt(\mathbf{f}) \quad (46)$$

From this torque, the load on the actuator (F_{act}) was found:

$$F_{act}(\mathbf{f}) = \frac{T(\mathbf{f})}{P1R} \quad (47)$$

From these equations developed, the torque was plotted versus the rotation of P1 (See Figure 27). It was desired to keep a constant torque and the results show a change in torque of approximately 3 in.-lb. during the rotation with the current setup. This was as close to a constant torque as was possible with the geometry configuration and the available spring selection.

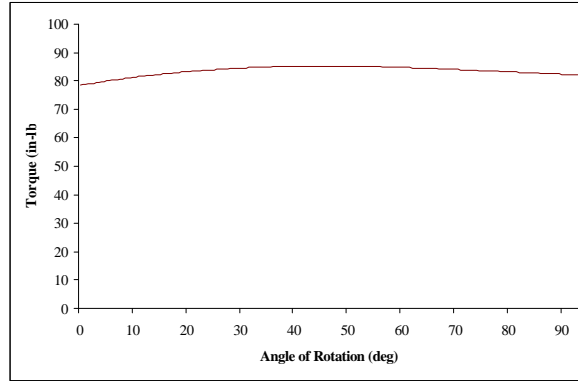


FIGURE 27: DAP Torque-Plot

The torque on the mechanism due to inertia of each component was determined to test for dynamic force effects. The individual components and their mass and CG distances were determined (See Table 6).

| Component | Mass(slug) | Distance, d (in.) | Inertia (lb.-in.- sec.²) |
|------------------|-------------------|------------------------------|--|
| Strobe(top) | 0.0187 | 0.7575 | 0.0107 |
| Laser(top) | 0.0078 | 1.647 | 0.0212 |
| Laser(bottom) | 0.0078 | 1.647 | 0.0212 |
| Bar Code Reader | 0.0622 | 2.4821 | 0.3832 |
| ACM(top) | 0.0276 | 1.9139 | 0.1011 |
| Camera(top) | 0.0137 | 0.5489 | 0.0047 |
| Camera(bottom) | 0.016 | 0.6733 | 0.0073 |
| ACM(bottom) | 0.0276 | 1.9139 | 0.1011 |
| Strobe(bottom) | 0.0187 | 0.7575 | 0.0107 |
| Mounting Bracket | 0.07 | 1.673 | 0.1959 |
| | | TOTAL | 0.8570 |

TABLE 6: Mass, CG Locations, and Inertia of DAP Components

A range of angular accelerations was determined to aid in the selection of the actuator (See Table 7). These accelerations were used to determine the torque requirement thus the necessary load capacity of the selected actuator. The results in Table 6 display very small inertia values for each of the components. This is a result of the axis of rotation being positioned near the center of mass of the camera package. This is not coincidental. One of the concerns during the design process was the location of the axis of rotation. Placing this axis close to the components' CG locations decreased these values of inertia, thus decreasing the torque requirements of the mechanism.

| Angular | Inertia(lb.-in.- sec.²) | Torque(in.-lb.) |
|----------------|---|------------------------|
| 0.5 | 0.8570 | 0.4285 |
| 0.6 | 0.8570 | 0.5142 |
| 0.7 | 0.8570 | 0.5999 |
| 0.8 | 0.8570 | 0.6856 |
| 0.9 | 0.8570 | 0.7713 |
| 1.0 | 0.8570 | 0.8570 |
| 1.1 | 0.8570 | 0.9427 |
| 1.2 | 0.8570 | 1.0285 |
| 1.3 | 0.8570 | 1.1142 |
| 1.4 | 0.8570 | 1.1999 |
| 1.5 | 0.8570 | 1.2856 |

TABLE 7: Panning Mechanism Torque Requirements

D.5.8 Power Consumption Analysis

One important factor considered during the design of these mechanisms was power consumption. In an autonomous vehicle, it is imperative to minimize power consumption. The energy consumed for each system was analyzed (See Table 8). The analysis is based on a simple calculation of the work performed by each of the linear actuators. This work is simply the load applied multiplied by the distance this load was applied across. CPS-I's actuator 1 carries a 280 lb. load at 1.6 in./sec. Actuator 2 carries 153 lb. at 1.7 in./sec. The total power to perform this actuator work is 0.056 and 0.0294 kW. for every second it moves it's respective distance. The fourbar consumes 0.03 kW. Thus CPS-I consumes a total of 0.115 kW. of power. CPS-IE is measured in the same manner. It's single lift actuator moves 500 lb. at a rate of 1.25 in./sec. consuming 0.071kW. The fourbar moves 200 lb. at a rate of 1.25 in./sec. consuming 0.03 kW./sec. The panning mechanism moves 42 lb. at a rate of 2 in./sec. consuming 0.009 kW. Thus CPS-IE consumes a total of 0.11 kW. CPS-II has a rotary servomotor that must lift a 53 lb. load at a rate of 10 in./sec. consuming 0.06 kW. The fourbar and the panning mechanism are assumed the same as CPS-IE. Thus CPS-II consumes a total of 0.099 kW. CPS-IE is 4.35% more efficient than CPS-I, thus CPS-II is 14% more efficient.

| System | CPS-I | CPS-IE | CPS-II |
|-----------------------|--------------|---------------|---------------|
| Power Consumed | 0.115 | 0.110 | 0.099 |

TABLE 8: CPS Power Consumption

D.5.9 Stress Analysis

A stress analysis was performed on the actuator load transfer bracket (ALTB) of CPS-IE (See Figure 28). Because this bracket receives all loads transferred from the actuator to the lift mechanism and it acts as a cantilevered beam, the failure of this component was a concern. The ALTB was machined from 6061-T6 aluminum with a yield strength of 40 kpsi. [4]. A 0.5 in. diameter pin transfers the load from the actuator to the bracket. The pin is made from 4340 quenched and tempered steel with a yield strength of 124 kpsi. [8].

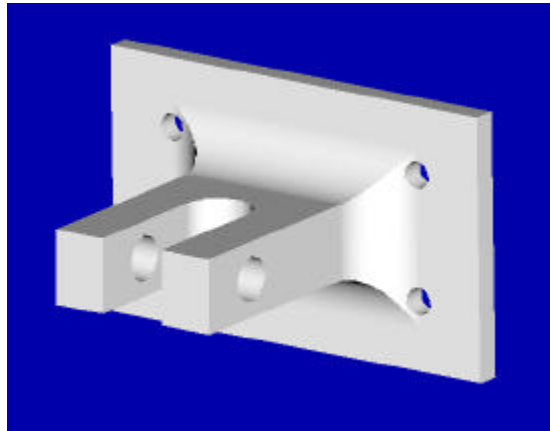


FIGURE 28: CPS-IE Actuator Load Transfer Bracket (ALTB)

The biggest concern of this component was the tensile stresses created in the upper and lower regions along the cantilever portion. The point of highest stress concentration was analyzed to determine the maximum stress encountered in the material. The stress was calculated using the assumption of a 2-D cantilever beam under a vertical load at the pin location. Stress elements (Points A and B) were studied as locations of interest (See Figure 29), then the results were compared to a finite element analysis (FEA).

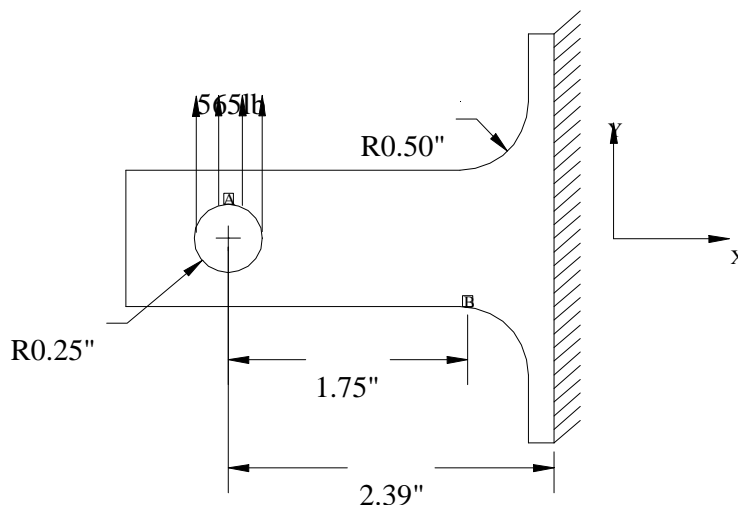


FIGURE 29: ALTB 2D Load Assumption

The first step in this analysis was a static force analysis to determine the component's reaction forces. The reactions at the fixed end revealed a -565 lb. force in the y-direction and a 1350 in.-lb. moment in the z-direction. From these reactions and the applied uniform load, the stress element values were determined (See Figure 30).

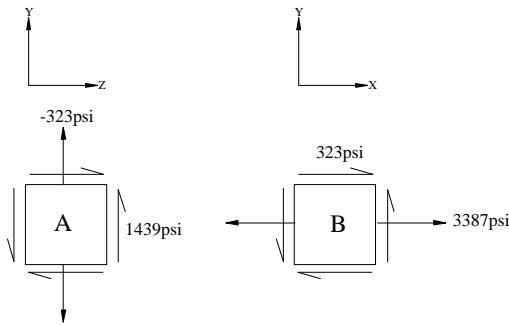


FIGURE 30: Stresses at Points A and B

Mohr's circle was used to determine the maximum principle normal stress at each of the point elements. Point A was subjected to a stress concentration of 2.15 due to the pin-hole, thus the maximum principle normal stress was 2.767 kpsi. Point B was subjected to a stress concentration factor of 1.4 based on the assumption of a rectangular filleted bar [8], thus the maximum principle normal stress was 5.369 kpsi. Parametric Technology Corporation's Pro/MECHANICA Version 18.0 was used to calculate the stresses at these locations using FEA (See Figure 30). The principle normal stress at point A was estimated as 5.394 kpsi., and point B was 2.683 kpsi. These values fall within 1% and 3% respectively. The critical stress location (point A) results were within a safety factor of 7 with respect to the material's yield strength.

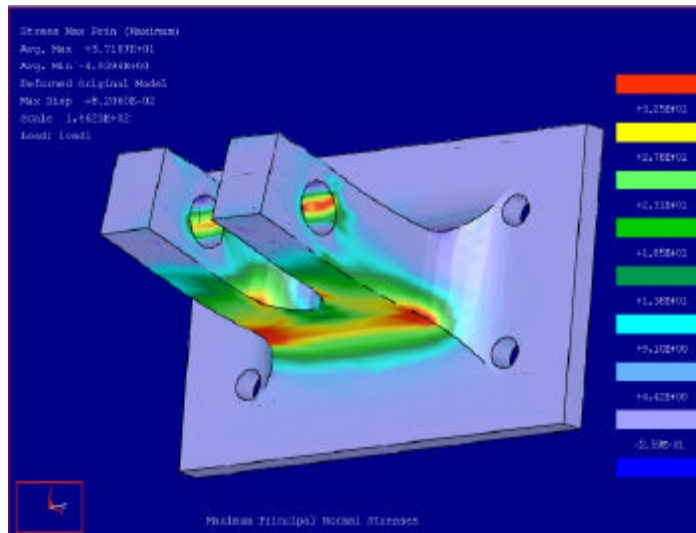


FIGURE 31: Stress Analysis Results of ALTB on Pro/MECHANICA

D.5.10 Stability Analysis

The location of the center of gravity (CG) and its effect on the stability of the overall system was studied for each CPS. One of the largest safety concerns inherent of this mechanism is the system's stability. The CG height on all three systems results in a large amount of stored potential energy. As the system accelerates and decelerates, inertia forces become a factor. These forces are even more profound as the system proceeds along an incline. To determine the safety limitations of this system a stability analysis was performed. The first step in determining the stability of this system is to define what is considered 'stable'. The biggest concern is the possibility of the system falling over as a result of some sudden deceleration. Thus the system's ability to pivot about some point due to its momentum will demonstrate the stability of the mechanism.

The obvious pivot location that the rotation could occur would be at the K3A's wheels. The three sets of wheels are in an equilateral triangular configuration. Each set rotates through the same angle that the CPS rotates. The hexagonal base of the K3A, however, stays in one position. The result is that the location of the wheels' point of contact tripod changes as the system turns. Because of its equilateral design, there are several wheel configurations that the system may be traveling on (See Figure 32).

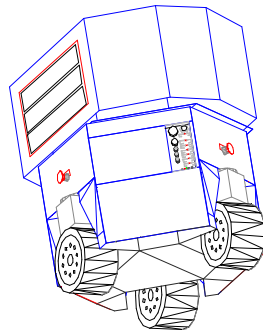


FIGURE 32: K3A Undercarriage

The analysis is performed for the 'worst case' wheel configuration. This occurs when the horizontal distance between the forward wheel(s) and the y-component of the system's CG is smallest (See Figure 33). This configuration has a contact line of action parallel to the CPS's x-axis. The two sets of wheels that are inline are located in the forward section of the system.

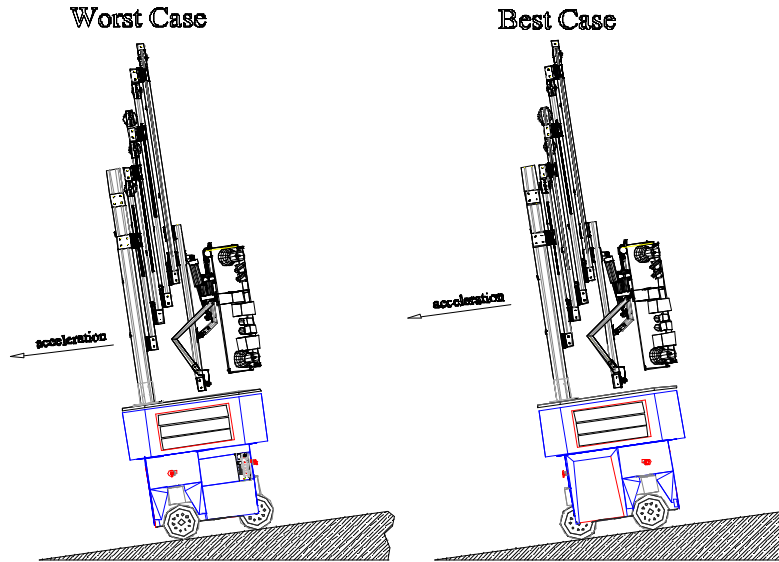


FIGURE 33: ARIES K3A Wheel Configurations

The scenarios considered in this analysis are based on several operating conditions. The first scenario is a simple test of an acceleration applied to the system on a level surface. Next the stability is tested as the system is inclined, modeling the behavior experienced on the ramps encountered during operation. The accelerations will be varied, and for each the resultant moment due to the system's mass and inertia forces encountered during an acceleration will be calculated. The point of instability will then be defined as the acceleration or deceleration which results in a zero moment at the wheel's line of contact. When this occurs, the system is in a state of balance or equilibrium. This state is extremely unstable because any increased acceleration will cause a change of sign in the resultant moment, which in turn may rotate the system about this line of action. This test is performed for each system, and for a large range of inclinations. The geometry used to derive the necessary equations is shown in Figure 34.

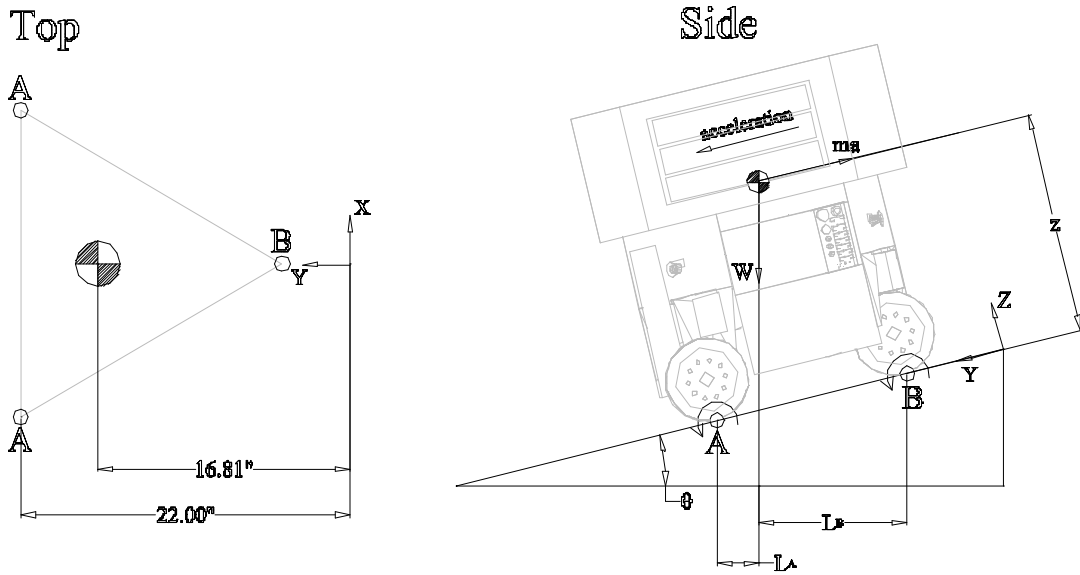


FIGURE 34: Stability Geometry

There were several inputs that were necessary before the derivation of stability equations was possible. The variables needed were as follows: system weight; ramp inclination; acceleration; y-z components of the system's CG; and the y-component of the wheel line of contact. The equations used to determine a resultant moment about this point are based on Newton's second law. This method is used in place of the method of dynamic equilibrium. According to Beer and Johnston, one method of viewing inertia forces is to treat them as any other force. Some think that "...inertia forces and actual forces, such as gravitational forces, affect our senses in the same way and cannot be distinguished by physical measurements." [1] Thus the inertia forces, with a magnitude of the product of the system's mass and the acceleration it undergoes, were placed in the opposite direction of this acceleration. The system's mass was treated as a point mass, and the location of this point was the CG location of the entire system (CPS and K3A).

The first step in performing the force analysis was to define a sign convention. The worst case scenario is used. The positive acceleration is in the forward direction, and is pointed down the slope of the inclination. The line of action of the two points of contact was the location where the moments were taken about. Point A is located in the front, which consists of two of the three sets of wheels, and Point B is in the rear. The first step was to sum moments about each of these points.

$$MA(a) = -WL_A - zma \quad (48)$$

$$MB(a) = WL_B - zma \quad (49)$$

MA and MB represent the resultant moment about points A and B, respectively. To determine the maximum accelerations and decelerations that are acceptable, the resultant moments are set to zero, and the acceleration is solved for:

$$Decel_{\max} = \frac{-gL_A}{z} \quad (50)$$

$$Accel_{max} = \frac{gL_B}{z} \quad (51)$$

These two calculations are the most important because they represent the safe range of acceleration which the system may undergo in its specified configuration. If the acceleration falls over the maximum in either direction the system is defined as unstable, and there is a good chance the mechanism will tip over.

Plotting the resultant moments, MA and MB, versus a range of accelerations, shows a linear relationship. Figure 35 is a plot with the K3A on level ground. The plot shows that the resultant moment is zero when the deceleration is -9-ft./sec^2 and the acceleration is $+21\text{-ft./sec}^2$. The difference in the absolute value of the two accelerations is the safe range magnitude. The maximum negative acceleration defines the maximum deceleration that the system can withstand, while the maximum positive value defines the maximum acceleration. This range can easily be found on this plot by reading the 'safe' values between the two x-axis intercepts.

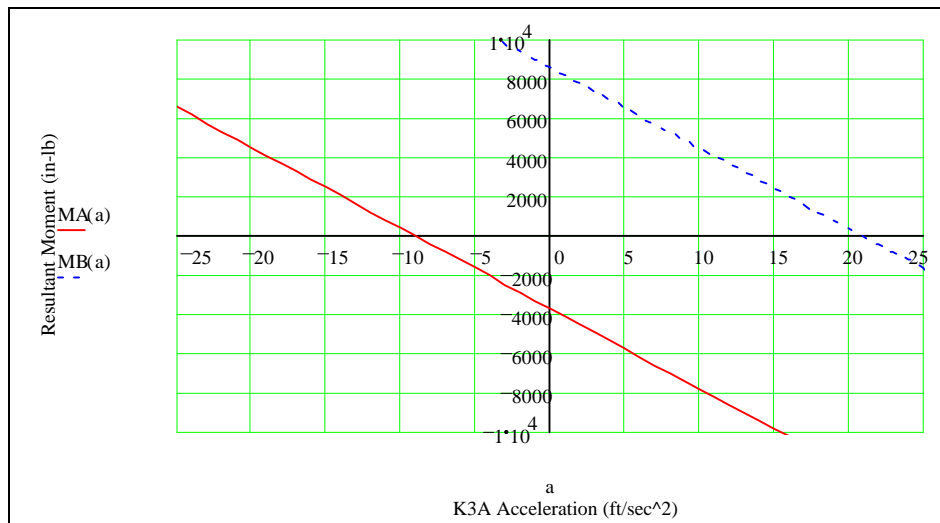


FIGURE 35: CPS-IE Stability Plot at 0 degrees Inclination

This analysis was performed for each of the three systems. The results are summarized below, however a range of results for CPS-IE is shown first. These results indicate the maximum accelerations and decelerations for CPS-IE in a large range of inclinations:

| Inclination (degrees) | Max Deceleration (ft./sec ²) | Max Acceleration (ft./sec ²) |
|-----------------------|--|--|
| 0 | -9.00 | 21.00 |
| 1 | -8.44 | 21.56 |
| 2 | -7.87 | 22.11 |
| 3 | -7.30 | 22.66 |
| 4 | -6.73 | 23.19 |
| 5 | -6.16 | 23.73 |
| 6 | -5.58 | 24.25 |
| 7 | -5.01 | 24.77 |
| 8 | -4.43 | 25.58 |
| 9 | -3.85 | 25.78 |
| 10 | -3.27 | 26.27 |
| 11 | -2.69 | 26.76 |
| 12 | -2.11 | 27.23 |
| 13 | -1.53 | 27.70 |
| 14 | -0.94 | 28.17 |
| 15 | -0.36 | 28.62 |
| 16 | 0.23 | 29.06 |

TABLE 9: CPS-IE Safe Range Accelerations as a function of Inclination

As the inclination increases, the maximum deceleration allowable decreases. On level ground the system can accelerate up to 21.00 ft./sec.² and can decelerate up to -9.00 ft./sec.² As the inclination increases this deceleration decreases while the acceleration increases. At the 9-degree inclination, which is the maximum expected of this system, the maximum deceleration allowed is -3.85 ft./sec.² with a maximum acceleration of 25.78 ft./sec.² The maximum commanded deceleration of the system is -1.3 ft./sec.² and the maximum commanded acceleration is 0.65 ft./sec.² These values fall within the safe range, however an unexpected abrupt stop is a concern. When an inclination of 15 degrees is reached, any deceleration will cause the system to be unstable. This position is perfectly balanced, or is in a state of equilibrium. This is obviously a very unsafe position, and is not recommended.

The same analysis is performed for negative angles. In this configuration, the system is more stable because the CG is not as close to the line of action. The results of this analysis show that at the 9-degree inclination, the maximum deceleration is -13.93 ft./sec.² and the maximum acceleration is 15.70 ft./sec.² This is a much safer range than the positive angles. This range allows an acceleration or deceleration in either direction of about the same magnitude. This is due to the CG being in a midpoint location between the two points of action. A static system becomes unstable at a negative angle of -32 degrees, where the maximum deceleration is -25.09 ft./sec.² and the maximum acceleration has changed signs to 0.073 ft./sec.² The lesson here seems to be that the system will travel down a ramp more safely in a backward direction. This indeed is what was done during testing at the Fernald facility. Similar analyses were performed for CPS-I and CPS-II (See Table 10).

| D.5.11 ystem | Inclinatio n (degrees) | Max. Deceleration (ft./sec.²) | Max Acceleration (ft./sec.²) |
|-------------------------|---------------------------------------|---|--|
| CPS-I | 0 | -8.76 | 18.21 |
| | 9 | -3.62 | 23.03 |
| CPS-IE | 0 | -9.00 | 21.00 |
| | 9 | -3.85 | 25.78 |
| CPS-II | 0 | -12.98 | 20.54 |
| | 9 | -7.78 | 25.32 |

TABLE 10: CPS Safe Range Accelerations

A trend is shown in this data. As the CG's are lowered and centered on the K3A, the range of safe acceleration is increased. The increase in range of acceleration on the 9-degree inclination from CPS-I to CPS-II is approximately a 20% increase. This range adds to the safety of the vehicle during its mobility.

D.6 ATTAINMENTS

Each of the three systems constructed during this project was unique in its own way. As each design became reality and each system was put to the test, many lessons were learned in the quest to automate the monitoring process. Each design grew out of its predecessor, and became a better more efficient machine. Each of the three designs now has it's own accomplishments and abilities, a few of which will be presented here.

D.6.1 CPS-I

CPS-I was the first prototype system. The constructed mechanism weighed 395 lb. It is capable of acquiring data from a column of four 85-gallon drums, while maintaining a stowed height of just under 10 ft. Overall power consumption was calculated as 0.11 kW. This system performed successfully during a demonstration for the DOE at USC's testing facility on November 30, 1995 with a throughput of 240 drums/hour.

D.6.2 CPS-IE

CPS-IE was constructed as an enhanced version of CPS-I. Enhancements include a reduced overall weight to 261 lb., a 34% decrease. The mechanism as designed is capable of acquiring data from a stack of four 110-gallon drums, while maintaining a stowed height under 10 ft. The CG of this stowed configuration is reduced, which increases the stability of the system. The capability of dent-detection was added. Power consumption on this system is 0.11 kW. For nearly 3 weeks in August 1996, ARIES performed its inspection tasks for DOE management at the DOE Fernald Site north of Cincinnati, Ohio. The testing was performed in the TS-4 building onsite. This testing included the monitoring of various drum columns, and the successful navigation within the prescribed confines of each aisle. The robot was proven stable when it traversed over spill isolation berms on a 9°

ramp. The overall throughput of ARIES-IE during testing was 60 drums/hour, down from ARIES-I, due to the addition of dent-detection on the new system. CPS-IE is scheduled to be tested, re-evaluated, and put into service at the Idaho National Engineering and Environmental Laboratory (INEEL) in early 1998.

D.6.3 CPS-II

CPS-II represents a fixed mast system and weighs approximately 223 lb., a 44% decrease from the original prototype. This unit is capable of inspecting a stack of three 55-gallon drums and two 85-gallon drums. The system maintains 3 DOFs, however there are fewer moving parts on this system. Power consumption is calculated as 0.099 kW. Expected throughput is up to 100 drums/hour with dent detection and over 300 drums/hour without drum detection. These throughput numbers are expected to increase significantly upon further enhancements to the system. The system is set to be deployed at Los Alamos National Laboratories in late 1997.

D.7 DISCUSSION

To support the need of an autonomous inspection system for DOE's low-level nuclear waste facilities, three systems have been designed, analyzed, and constructed. The iterative design process has transformed a concept into proven product. The project's success is supported by the successful completion of many arduous tasks and demonstrations. From drawing board, to prototype, to final product, these three designs (CPS-I, CPS-IE, and CPS-II) shown in Figure 36, have met and in some cases surpassed the demands of the customer.

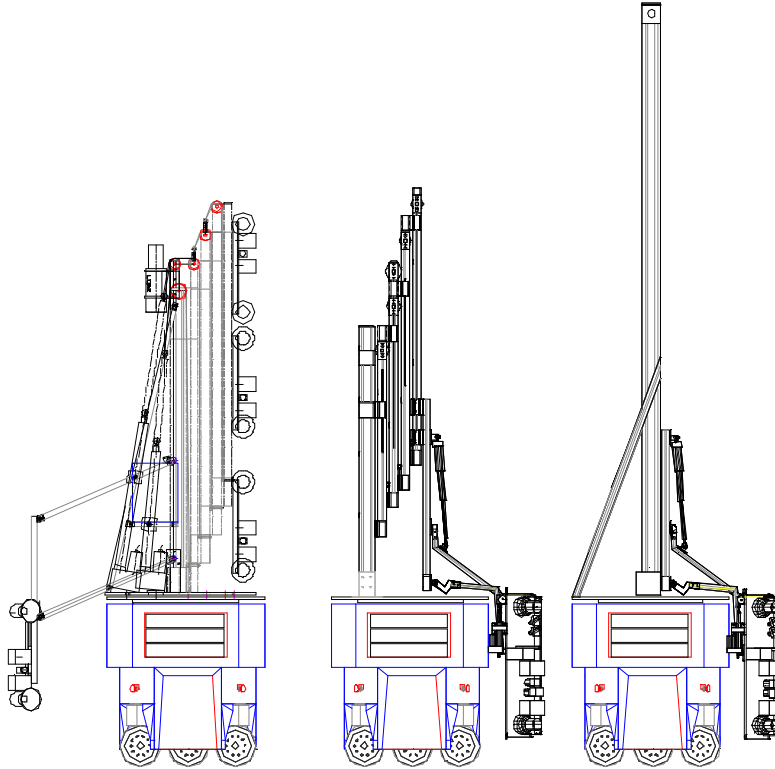


FIGURE 36: CPS-I, CPS-IE, and CPS-II

The objective of this project is to produce commercial mobile robot systems for the Department of Energy for use in drum storage inspection. When ARIES-IE is deployed at INEEL and ARIES-II at LANL, the objective of the project will have been met. It is anticipated that the systems will be tuned, updated and improved. It is also anticipated that the systems will be enhanced and modified to tackle other applications such as decontamination and decommissioning at other DOE nuclear sites.

D.8 REFERENCES

1. Beer, Ferdinand P., and Johnston, E. Russell, Jr. Vector Mechanics for Engineers: Statics and Dynamics. 5th ed. New York: McGraw-Hill, 1988.
2. Bostick, Judith L., et al. An Intelligent Inspection and Survey Robot. Final Report. South Carolina Universities Research and Education Foundation. Clemson University, 27 January, 1994.
3. Item Products, Inc. Catalog. Houston, 1995
4. Norton, Robert L. Design of Machinery. New York: McGraw-Hill, 1992.
5. Rocheleau, David N. "Mechanical Deployment System on ARIES an Autonomous Mobile Robot." Proceedings of the 4th National Applied Mechanisms & Robotics Conference. Volume 1, AMR95-020. Cincinnati, 10-13 December, 1995.
6. Rocheleau, David N. and Moore, Matthew M. "Camera Positioning Mechanism for ARIES an Autonomous Mobile Robot." Proceedings of the 1996 ASME Design Engineering Technical Conference. 96-DETC/MECH-1154. Irvine, 18-22 August 1996.
7. Sahag, Leon Marr. Kinematics of Machines. New York: The Ronald Press Company, 1952.
8. Shigley, Edward Joseph, and Mishcke, Charles R. Mechanical Engineering Design. 5th ed. New York: McGraw-Hill, 1989.
9. Working Model. Computer Software. Version 3.0.3 for Windows. Knowledge Revolution, 1995.

Appendix E

ARIES: An Intelligent Inspection and Survey Robot

ROBOTIC VEHICLE SYSTEMS

Cybermotion Incorporated

E. ROBOTIC VEHICLE SYSTEMS

WARNING ! THE K2A IS A HEAVY AND EXTREMELY POWERFUL VEHICLE !

Because of its relatively demure appearance, there is a tendency for people to underestimate its potential for doing damage and bodily harm. Do not attempt to operate the vehicle until you have read this and other appropriate manuals, and are thoroughly familiar with its operation and safety features !

DO NOT OPERATE THE VEHICLE IN THE PRESENCE OF PERSONS WHO ARE NOT AWARE OF ITS POTENTIAL FOR INJURY AND ITS SAFETY FEATURES.

E.1 UNPACKING AND INSPECTION

E.1.1 Removal from shipping crate

Before removing the K2A from its crate, remove the top of the crate and make a quick visual inspection of the unit. If the crate has sustained substantial damage, or if the vehicle has shifted inside of the crate, do not proceed with the unpacking process until the carrier has been contacted and has inspected the damage.

Instructions for the removal of the K2A from its shipping crate are stenciled on the crate. After removing the top of the crate, the access side, and the restraining yoke, remove the chock blocks behind each wheel. The vehicle may now be rolled slowly out of the crate.

E.1.2 Removal of cover and internal inspection

The vehicle is shipped with its E-STOP circuit open and will not operate until it has been reconnected. **DO NOT SWITCH THE UNIT ON UNTIL THE FOLLOWING INSPECTION HAS BEEN COMPLETED!**

Unsnap the 6 cover retaining fasteners and gently lift the cover off the platform. If the cover appears to be stuck, check to make sure you are keeping it level. A slight amount of jiggling may be required to ease the cover past the rubber battery tie down straps. Note: Although the E-STOP circuit is left unplugged when the K2A is shipped , it will be necessary to unplug the connector at this point during subsequent removals of the cover.

With the cover removed, make the following visual inspection:

Check for loose screws or obvious damage, especially to the cover snaps.

Check the electronics assemblies to assure that all boards are securely in place and that the control assembly is not loose.

Check for loose wires or battery cables. Check the lower half of the sheet metal body for damage, loose parts, or the presence of oil.

If loose components are found, tighten them, being careful not to short any electrical circuits with your tools.

THE BATTERIES USED IN THE K2A ARE CAPABLE OF PRODUCING ENORMOUS SHORT CIRCUIT CURRENTS. ALWAYS DISCONNECT BOTH THE (+) AND (-) TERMINALS OF THE BATTERIES BEFORE ATTEMPTING TO WORK ON ANY ELECTRICAL SYSTEM OF THE VEHICLE!

E.1.3 Preliminary power-up tests

During the following tests, the E-STOP circuit should remain disconnected. If any anomalies are noticed during the following check out, turn the unit off and contact the factory. After inspection of the platform, perform the following tests.

Place a voltmeter between the test points marked BC (Battery Common) and +12 Volts. The voltage should measure between 12.0 and 14.5 Vdc.

Move the positive lead of the meter to the +24 Volt test terminal. This voltage should read 0 Volts. Next, place the key into the key switch and rotate it to the "on" position just long enough to take a voltage reading and then turn it back off. The voltage should rise to between 24.0 and 29.0 Vdc.

Place the negative lead of the voltmeter in the DC (Digital Common) test point. Repeat the on-off test for the +5, +12, and -12 Volt test points.

The acceptable limits are:

Test Point Range

+5 Vdc +4.85 to +5.10 Vdc.

+12 Vdc +11.50 to +12.50 Vdc

-12 Vdc -11.50 to -12.50 Vdc.

If all supplies measure properly the key may be left in the on position without danger of damage. When the key is turned on, the internal computer will run a RAM test program which takes about a second. During this time, only the DRIVE PWM and STEER PWM LEDs should light. If all RAM memory checks good, the two PWM lights will turn off, and the FWD/REV LED will light. If a RAM failure is detected, the LEFT/RIGHT LED will flash, and the vehicle will not operate or communicate over the supervisory link. If the program down load jumper is installed on the DC-1 Computer card (see K2SRV manual), the DRIVE PWM and STEER PWM LEDs will light, and all others should be off.

Turn the key switch off and remove the key. Place the cover back on the K2A, being sure to connect the E-STOP circuit. Snap the cover down, and pull out all three E-STOP buttons. Turn on the key switch. No motor activity should occur.

E.1.4 Manual Drive Test

The vehicle is now ready for direct control. Place the test umbilical into the "Tether" connector on the K2A control panel and plug the other end into the LINK connector of the JA-01 box. Plug an RS-232 cable between the HOST A connector of the JA-01 and the serial port of the HOST computer. Place the OPERATING DISK in drive A and enter:

A:<ENTER>

K2SRV<ENTER>

NOTE: See K2SRV manual for software requirements and loading instructions.

When the program has booted up and placed the OPERATING MENU on the screen, check for a robot status of "NORMAL OPERATION". If a "SUPERLINK FAIL" status message is seen, check your cabling, the power on the JA-01, and the power switch at the vehicle. If any other warnings are received, consult the K2SRV Software Manual.

If everything is normal, press the MANUAL/TEACH mode button (PF-6) on the HOST. Be prepared to hit the HALT button (PF-10) should any unexpected action occur. The ENABLE safety interlock LED on the K2A's control panel should begin to flash, there should be a click from the main drive relay, and you should hear a low level, 2500 Hz tone. This tone is the PWM signal to the motors, and was intentionally placed in the audible range to reduce the possibility of the vehicle operating without being noticed. If the tone is present, it should be possible to drive the vehicle using the joystick.

If the click or tone is absent, check the E-STOP circuit. This series circuit requires that all E-STOP switches be closed (pulled out), and that the circuit be closed at the turret connector (pins 5 and 6). The

circuit is intended to pass through turret mounted interlocks and E-STOP switches. If no turret is mounted, the supplied dummy connector must be plugged into the turret connector to complete this circuit (pin 5 to 6). (For K2A's originally supplied with PROM revision 3.01 and newer, an open E-STOP line will generate an "E-STOP ACTIVATED" status message in the K2SRV status window.)

If the problem persists, check the control and motor fuses on the control panel of the K2A platform.

E.2 THEORY OF OPERATION

The K2A platform was designed to provide unsurpassed maneuverability to autonomous and tele-operated systems in industrial environments. Do not be fooled by its demure appearance, **THE K2A IS A HEAVY DUTY PIECE OF EQUIPMENT (APPROX. 300 lb.), AND CAN CAUSE SERIOUS INJURY OR DAMAGE IF USED CARELESSLY.** Because of its unique three wheel drive system, the platform can easily develop in excess of 240 lb. of tractive force on a relatively smooth surface. The K2A has sufficient power to push an adult human about like a rag doll!

E.2.1 Synchro-Drive

The drive system of the K2A represents a relatively new concept in mobility, in which all the wheels of the vehicle are locked together in both steer and drive. Thus, when a Synchro-Drive vehicle executes a turn, all three wheels turn in unison and trace parallel paths to each other. The result of this geometry is that the platform itself does not rotate as a turn is executed. For this reason, a turret flange is provided at the top center of the vehicle, which rotates in unison with the steering. Systems mounted to this flange will face in the direction of forward motion of the vehicle. A Synchro-Drive vehicle can thus follow any path geometry. Since all wheel driving forces are perpetually parallel, these vehicles also have excellent tractive properties, and can accurately determine their relative motion.

Early (first generation) implementations of Synchro-Drive used a continuous means such as belts or chains to tie the wheel assemblies together. These configurations suffered from alignment problems due to the uneven distribution of slack between the driven pulleys. The result of this problem was that the first generation platforms tended to be relatively inaccurate and they required a significant level of maintenance. The K2A represents a second generation of Synchro-Drive vehicles and uses a patented concentric shaft drive system to accomplish the required functions in a much more accurate and reliable manner. There are no alignment adjustments in the K2A as all gears are permanently keyed to their respective drive shafts.

E.2.2 Mechanical description

The following description refers to the cut-away in Fig. 1: The K2A has two motors (steering and drive) and associated power trains. The steering motor (A) drives the vertical steering shaft (C) through a spiral gear reducer (B) which gives a reduction of 106:1. The vertical steering shaft is coupled to the turret mounting flange at the top. This flange uses an expanding locking collet to assure backlash free engagement with the turret.

An optical angular pulse encoder (J) is located on the vertical steering shaft just above the spiral reducer in the upper column housing (M). The ribbon cable connector for the steering encoder is mounted on a plate in a boss at the bottom of the column, and the round connector of the slip ring is similarly mounted in a second boss. The slip ring is mounted above the encoder, with its rotating connector (N) being in the center of the top of the steering shaft.

The lower end of the vertical steering shaft terminates in a miter gear which engages three like gears on the three leg steering shafts (E). These shafts are hollow and a drive shaft is suspended in the center of each of them. On the outside end, each leg steering shaft has an identical miter gear that engages a like gear on the foot housing (F), thus affecting the steering of the wheel.

The drive motor (H) has an optical pulse encoder (O) mounted on top of it. The motor drives an oil filled gear box (I) providing a 24:1 reduction through two levels of spur gears. The output of the drive gear box (J) has a miter gear which is smaller than those in the steering chain. This gear drives three like miter gears attached to the three horizontal leg drive shafts. These gears are located in the hollow centers of the 3 steering gears. Each of these shafts is terminated on its outer end by an identical miter gear (K) which drives the respective foot vertical drive shaft.

The vertical drive shaft for each foot powers its respective wheel through a bevel gear set (L). This gear set has a reduction ratio that exactly matches the ratio of the wheel diameter and its steering circle (as traced on the floor). Thus, if the steering is actuated while the drive motor is held fixed, the foot rolls around the steering circle by reflex action. This action reduces floor damage and makes for a smooth, efficient turn even while the vehicle is stationary.

E.2.3 Electrical description

The K2A contains two electronic assemblies, the computer and the motor power amplifier. The computer is based on the Z-80 CPU and is implemented in CMOS to conserve power. The computer card is powered by a chopper power supply for efficiency and to isolate it from transients produced by the motors. All control signals from the computer to the motor power amplifier are optically isolated to prevent glitches and to serve as a protective barrier in the event of catastrophic failure of the motor power amplifier.

Early K2A platforms contained a four quadrant steering and a two quadrant drive motor amplifier. These units used a relay to accomplish reversing of the drive

motor. Later units use two 4- quadrant amplifiers. The primary reason for this change was the fact that as heavier loads are placed on the K2A an increasing coupling occurs between the steering and the drive. This coupling is the direct result of the increased frictional torque at the point of contact of the tire and it has been found to degrade the accuracy of position determination. The use of a 4 quadrant amplifier on the drive eliminates this effect. The newer amplifier uses only power FETs in its H-bridges. Additionally, the new amplifier has improved interlocks. There are slight differences between the signals required by these two amplifiers (see table below). Cybermotion can upgrade two quadrant systems to four quadrant drives (consult factory for pricing).

E.2.4 Panel LEDs

As a trouble-shooting aid, the actual optoisolator signals, running from the computer to the motor power amplifier, pass through LEDs on the K2A control panel. These signals are:

NAME SIGNAL DESCRIPTION

DRIVE PWM Drive pulse width modulation (power) command.

Lights brighter with increasing drive command.

FWD/REV This signal controls the direction of the drive motor.

ENABLE This signal must be pumped (toggled) by the control computer.

This pumping assures that a failure of the control computer will halt the vehicle.

STEER PWM Steering pulse width modulation (power) command. Lights brighter for an increasing steering power command.

LEFT/RIGHT Steering direction command. On power-up or after a reset, this LED will flash if the computer self test detects a RAM memory failure.

E.2.5 Power Circuits

The K2A contains two, 12 Volt, high capacity batteries which are connected in series to form a 24 Volt supply. Although the tap between the batteries is brought to the charger connector on the control panel of the K2A, the batteries are always charged in series. For this reason, **IT IS IMPORTANT NEVER TO DRAW CURRENT FROM THE +12 VOLT BATTERY TAP** as this will cause an imbalance of charge in the two batteries and rapid loss of capacity.

The on-board electronics of the K2A are powered by an encapsulated, isolated switching power supply. To serve as a trouble shooting aid, the voltages produced by this supply may be measured on test points on the K2A's control panel. (For details on the acceptable limits for these voltages, see section 1.3).

E.2.6 Torque Limit Adjustments

At the lower right of the K2A's control panel there are two small screw driver adjustments. These adjustments set the torque fold-over points for the steering and drive servos. Although it is relatively safe to leave steering torque limit at maximum (fully CW), **IT IS IMPORTANT TO LIMIT THE DRIVE TORQUE TO THE LOWEST LEVEL THAT IS NEEDED**. This may be done by adjusting the Drive Torque Limit adjustment CCW until fold over results during the most demanding operation for which the vehicle will be used. At fold-over, the pitch of the PWM will change and the thrust of the platform will be reduced to approximately 15 percent of maximum. In the AUTOMATIC mode, torque fold over will cause a program to be aborted with a "STALL ABORT" status. In the manual mode, the vehicle will simply lose power, and the operator will have to reduce the command to zero before the fold over circuit will reset.

Circuit diagrams for the slip rings, the front panel "tether" connector, and other circuits can be found in Appendix A of this manual.

E.2.7 Communications and control

There are a wide variety of networking protocols in use, however, most were designed for data processing applications and not for real time control. Of those protocols that were designed for real time control, most are prohibitively expensive and complex. For these reasons, Cybermotion developed its own protocol based on the most readily available communications hardware. This protocol uses a standard RS-232 asynchronous transmission and may be implemented on virtually any computer from a main frame to a micro.

E.2.8 Data Transmissions

The Cybermotion protocol is based on a flexible addressing concept rather than on a rigid set of commands (such as drive, turn, etc.). The system is master/slave in nature and has only two message formats - a request for data from a slave computer and a transmission of data to a slave. The message for transmission of data to a slave is structured as follows:

:NNAAAACCDD...DDSS<CR><LF>

Where:

: Indicates a data transmission.

NN Number of data bytes (2 ASCII hex numerals i.e., 0-F)

AAAA Beginning destination address (4 ASCII hex numerals)

The high byte is sent first.

CC Slave computer number (2 ASCII hex numerals)

DD...DD Data (2 ASCII hex numerals / byte)

SS Check sum for all digit pairs, calculated by subtraction starting with zero, and represented as 2 ASCII hex numerals

<CR> Terminator (required)

<LF> Optional for ease of monitoring.

For example: :020100030102F7<CR><LF>

would transmit two bytes, 01 into address 0100 Hex and 02 into the address 0101 Hex in computer 03 Hex. If slave number 1 received this message properly and calculated the proper checksum (00-02-01-00-03-01-02=F7H), after the <CR> was received, it would place the two bytes in memory and transmit the checksum back to the master as a single 8 bit byte (i.e., binary, not in ASCII hex).

E.2.9 Data Requests

The message format for a request for data from a slave computer is as follows:

;NNAAAACC<CR><LF>

Where: ; Indicates a data request

NN Number of data bytes requested (2 ASCII hex numerals i.e., 0-F)

AAAA Beginning address of data (4 ASCII hex numerals) The high byte is sent first.

CC Slave computer number (2 ASCII hex numerals)

<CR> Terminator

<LF> Display aid for monitoring

During reception of this message a slave will calculate the checksum for the message as it is received. After the slave receives the message it will immediately transmit the requested data in raw 8 bit binary bytes, subtracting each from the checksum of the received request. After the last byte of requested data has been transmitted, the slave will append the combined checksum in the form of an 8 bit binary byte (no borrow to a higher byte is calculated). Action is taken by the slave immediately after the receipt of the computer number (not on receipt of the <CR>).

For example, if slave 03 received: <CR><LF>;01010003

It would calculate the checksum as; 00 -01 -01 -00 -03 = FB (Hex). Slave 3 would then send the value found at address 0100 in its memory (say for example the value is 10H), and subtract it from the previous checksum; FBH - 10H = EBH, and it would then send the combined checksum [0ECH] as a single byte (binary.)

Slave computer numbers have been assigned as follows:

E.2.10 Quick Drive Message

The Quick drive message protocol will operate only with K2A's that have the K2ASS4 prom (Version 3.03 or later) installed. This dual slave version of the prom is for users that wish to control the K2A from another on board computer. The Quick message requires a much less time for applications requiring tight control, as it combines a data transmission and a request. The message can be used on either the supervisory or control link of K2ASS4 equipped platforms. Quick drive messages may be mixed with conventional messages. Standard Cybermotion systems do not use this message.

NOTE: ALTHOUGH the quick drive command provides tight coupling, it makes no use of the safety and navigation subsystems. For this reason, CRUISE is recommended. The message format for a quick drive message to the K2A is;

*DDDDSSSSCC<CR>

Where: * Indicates a quick drive command.

DDDD Is the drive command (-250 to +250). 4 ASCII Hex numerals representing a 2 byte word (LOW BYTE FIRST).

SSSS Is the steer command (-350 to +350). 4 ASCII Hex numerals representing a 2 byte word (LOW BYTE FIRST).

CC Is the checksum for DDDDSSSS as described for other messages.

<CR> Terminator

The K2A will ignore quick drive "*" messages unless it has been placed in either the MANUAL or MANUAL CLOSED LOOP Modes by transmitting the proper value to MODE (See K2COM.DEF). When the "*" message is received, the K2A will test the checksum. If this is correct it will reset the deadman timer to .5 seconds, and execute the steering and drive commands. If more than .5 seconds elapses between "*" messages, the K2A will go into halt with a "DEADMAN KICKOUT" status (See section 3).

After a proper "*" message, the K2A will return the following response;

xyyaac

Where;

xx Is the current X Position (In binary, with low byte first).

yy Is the current Y Position (In binary, with low byte first).

aa Is the current Azimuth (In binary, with low byte first).

c Is the checksum in binary, for both the received message and the response.

This is the negative sum of DD, DD, SS, SS, x, x, y, y, a, and a starting with 00 (as in other messages).

E.2.11 Timing considerations

After making a transmission of data or a request for it, the master must calculate how long to wait for a reply. If a reply is not received in a reasonable time, the message should be repeated by the master. The K2SRV programs, for example, will repeat a message three times before signaling an error. Calculating the time to wait for a reply is normally done by counting the number of characters transmitted in the message and allowing the time it takes to send a single character multiplied by the number of characters plus 2 (for response time).

NOTE: If the FDX radio modem is in the path of the data, its baud rate is 2,560 baud standard and there are 12 bits/byte. This must be used to calculate the time.

For ASCII, the time required to send a character is calculated as:

Xmission Time = (Bytes Sent + 2) * Time/Bit * Bits/Byte

Bits/Byte= 1 start bit + 8 data bits + 2 stop bits = 11

Time/Bit = 1/Baud rate (in bits/second)

E.2.12 Control philosophy

Using the two message protocols above, a master computer can read and write data in a 64 K area of each slave computer. The actual effect of data transmissions to a slave will be determined by the software running in the slave. There are two general types of data, control and parametric. For example, writing the MODE value of the K2A platform is a control function, and determines the algorithms it will execute. Parametric data is then used by the operating algorithm to do the desired function.

The actual modes available in the K2A are subject to frequent supplementation. A complete description of the modes available in your platform is contained in the printed listing of K2COM.PRN in the appendix. Among the operating modes are MANUAL (Torque commanded), AUTOMATIC (Execution of up to 250 program commands in sequence), and CLOSED LOOP (Velocity commanded). (For an explanation of the automatic mode, see also the K2SRV/K2AAV Software User's Manual.)

E.2.13 Adding computers

On board the vehicle there is often more than one slave computer on a network. Since RS-232 was not designed for bus operation, it is necessary to OR tie the transmitters of any slaves that must report on a

single channel. To implement this the K2A has pull-down resistors (to -12 Vdc) on both the transmit and receive lines of each channel. Since there is only one receiver (the master's) on the slave-to-master line, any number of slave transmitters may be added to this line by simply diode OR tying them. The K2A uses a transistor buffered output to allow it to drive up to 10 slave receivers on the internal link if it is programmed as a master (see next section).

There are four ways to connect multiple transmitters to a line;

- 1) Diode OR tying (Cathodes connected together to Rx).
- 2) Buffered diode OR tying
- 3) Daisy chaining
- 4) The turret interface panel (TIP-01 or TIP-02)

If three or fewer receivers are present on a line, it may be driven by a standard RS-232 driver through a diode (with the cathode connected to the line). If between three and ten receivers are present, the line must be driven by a transistor buffered output. Finally, if more than ten receivers are present, the line should pass through an RS-232 receiver, be combined with the signal to be added, and be transmitted out in a daisy chain fashion to the next slave. As lines become loaded the immunity to noise is reduced, so they should be kept away from any wires carrying noisy high current signals (such as motor currents). It is also advisable to use shielded cable for all communications. Since the vehicle is so compact, lines are intrinsically short and noise problems are relatively rare.

The TIP-01 Turret Interface Panel provides for the serial interconnection of up to eight slave processors without the user having to worry about buffering problems. The TIP-01 also provides many other functions such as selecting the highest priority channel for communications with the HOST computer when multiple channels are available (tether, radio, beacons, etc.). The TIP-01 is highly recommended for vehicles with multiple slave processors on board. For more information see the TIP-01 manual.

SINCE ALL COMMUNICATIONS BETWEEN THE K2A AND THE TURRET PASS THROUGH THE SLIP RING, IT IS IMPORTANT TO KEEP NOISE OFF ITS OTHER LINES. For this reason, if the application turret is to contain motors or other noise sources, it is recommended that the power for these be supplied by a modest battery in the turret. This battery may then be trickle charged from the main batteries through the slip ring. An alternate to a battery is a bank of filter capacitors (if the total turret peak current does not exceed the slip ring current rating). IN EITHER CASE, IT IS IMPORTANT TO LIMIT INRUSH CURRENTS TO THE TURRET DURING POWER-UP TO AVOID DAMAGE TO THE SLIP RING OR BLOWING OF THE TURRET FUSE.

E.3 OPERATION OF THE PLATFORM

There are two serial channels on the K2A platform. The first channel is the "Supervisory Link" and is intended for communication and monitoring with the HOST computer (HOST) as the link master. On this link the K2A and other on board computers always act as slaves. When the vehicle is operating in an autonomous mode, this link is used only for monitoring the vehicle's performance. The term "autonomous" is used here to indicate that the vehicle is navigating on its own, without the need for control signals from outside. The vehicle may also be controlled in a semiautonomous manner by an alternate host computer. In this case, the alternate host would control the vehicle through the Supervisory Link via the "Host B" input of the JA-01 Joystick / Link Arbitrator. This mode is referred to as semiautonomous because, although there are no humans involved in the process, the vehicle is incapable of operating without a remote computer's constant input.

The second link is the "Internal Control Link" (or just Control Link for short) and is used to control the vehicle during autonomous operation. While the K2A is always a slave on the Supervisory Link, it may be supplied as either a slave or master on the Control Link. Most integrated vehicles supplied by Cybermotion

(such as the Nexus) are configured with the K2A as the Control Link master. When the customer intends to control the vehicle from an on board computer, the K2A is supplied with a different PROM. Thus the K2A PROM will be marked either K2ASS4 (SLAVE/SLAVE), or K2ASM4 (SLAVE/MASTER).

The two serial communications modules present in the K2A (whether two slaves or a slave and master) are run as independent background tasks. These modules may be thought of as functioning in much the same way as a data bus operates within a computer. The K2A software is constantly being upgraded and supplemented, so for details on the operation of your version consult the K2COM.PRN listing in Appendix B of this manual. What follows is a brief description of the most common modes of operation.

E.3.1 Control from HOST Computer

The K2A is under the direct control of the HOST computer when the HOST is in either the Halt, Manual, or Manual Closed Loop Mode. In the Halt mode, the HOST continually transmits halt commands to the platform, thus over-riding any movement commands issued by another computer.

In the Manual and Closed Loop modes, the HOST continually transmits the values of the JA-01's joystick position along with a deadman time to the vehicle. The deadman time resets a down counter in the vehicle. The K2A automatically decrements this time value every 0.1 seconds. If the timer reaches zero, the K2A will place itself in the Halt mode.

In between the transmissions of this joystick data are interlaced requests for data to fill the menus selected at the HOST, and transmissions of data entered on the screen by the operator. Because of the heavy communications overhead required for responsive control, screen update is slowest in these modes. In the Manual and manual reverse modes, the K2A uses the joystick values directly as PWM commands to the motors. This means of control is most natural to operators as it closely simulates the response of most ordinary vehicles.

In the Closed Loop mode (Z Mode in K2SRV software) the joystick commands are used as velocity commands for the closed loop operation of the steering and drive servos. This mode feels less natural to the driver and is somewhat more dangerous than the normal Manual modes. As the vehicle approaches a grade, for instance, no increase in joystick command is required to maintain velocity. By the same token, if the vehicle hits an obstruction, such as a hapless pedestrian, it will command full power in an attempt to maintain speed. The proper setting of the Torque Limit Adjustments will minimize these problems (see section 2.3.3). Driving the vehicle in the Closed Loop mode has the advantage that the act of turning (especially under heavy loads) cannot back-drive the drive motor. For this reason, driving in this mode provides better dead reckoning accuracy. This feature makes the Closed Loop Manual mode attractive for teaching paths to heavy vehicles. The closed loop mode is also preferable on steep grades since the vehicle will not roll if the Joystick is released.

It is important to distinguish between the mode of operation of the HOST and that of the K2A platform itself. Indeed, in any mode except the Dual Host mode, the HOST interrogates the mode of the K2A constantly. In these modes, if the mode of the K2A is different from that last set by the HOST, the HOST will change itself and the K2A to the Halt mode and announce "MODE TERMINATED". In the Dual Host Mode, the HOST simply becomes an observer. The following table shows the correlation between these modes.

E.3.2 Control from alternate host computer

With the HOST in the Dual Host Mode, it is possible for a second host to control the K2A over the Supervisory Link. This is accomplished by plugging the serial line from the second host into the HOST B input at the rear of the JA-01 Joystick/Link Arbitrator. In this configuration, the HOST will maintain control of the vehicle in any mode except the Dual Host Mode. In the Dual Host Mode, the HOST will

interrogate the JA-01 for its status using the "S" command (see the JA-01 User's Manual). Bit 0 of the status byte returned by the JA-01 is used to flag a link request by the B host.

When either host computer is preparing to use the link and does not presently have control of it, it must repeatedly issue "S" commands. The first "S" command will set the link request bit in the status register. If the other computer releases the link (using an "R" command), any subsequent "S" commands will receive a status byte in response. The K2SRV software has the good manners to release the link on any request after it has accomplished a single communications transaction. If the B Host is not polite enough to release the link in a reasonable time the HOST will display "LINK BEING HOGGED", but will otherwise do nothing. This message informs the operator that data appearing on the screen may be stale or totally invalid. It should also be noted that the "S" command starts conversions on both the X and Y joystick A/D converters. If the alternate host is to use joystick values in its program, it must issue "S" commands before reading the joystick registers (see JA-01 manual).

The A HOST input of the JA-01 has a special capability. If any byte is sent into this port with bit 7 set true, the link is returned to the A Host. This function is used by the HOST if the Halt Mode is selected by the operator during Dual Host operations. Thus, it is possible to monitor the operation of the platform while controlling it with another computer. This capability is not only an aid to program development but also adds a measure of safety to the debug process.

E.3.3 Control from an on board computer

The K2A platform can also be controlled by another on board computer over the Control link provided that the K2A has been supplied with the K2ASS4 version of its PROM. In this configuration, as in the arbitrated configuration, the HOST must be placed in the Dual Host Mode. Although the HOST will continue to check for link requests, it will not receive any and it will therefore never release the link. Since the Control Link, through which the platform is being controlled, is independent of the Supervisory Link in this configuration communications throughput is double or better than that of the arbitrated mode.

Although it is not generally preferable, it is also possible to control the K2A on the Supervisory link from an on board processor. This may be done by interrupting the normal host's communications on the Supervisory link, and is possible through the use of the PORT REQUEST feature of the TIP-01. The actual control sequence is the same in either case, but if the Supervisory link is used, all control and monitoring from the fixed host is lost.

The Halt mode of the HOST continues to function if the Control link is used by the on board host, provided that the on board master does not repeatedly reset the MODE value. In both this and the arbitrated configuration, the alternate host computer should frequently request the STATUS of the K2A. If this status is found to have changed to Halt, the alternate host should become dormant. It is suggested that any user supplied on board computer be equipped with a slave interface to the Supervisory Link. The K2SRV software package has diagnostics menus through which the user will be able to view data in the computer as byte, signed word, or plot values.

E.3.4 Continuous Control

When an on board or remote host is to control the K2A, it may do so in two basic ways; continuous control or by down-loaded path control.

The K2A may be controlled continuously in either the MANUAL MODE (torque commanded), or in the CLOSED LOOP MANUAL MODE (velocity commanded). The closed loop mode is generally recommended for its accuracy, but the driving computer must send smoothed commands. If a step function is commanded in the closed loop mode, the K2A program will drive the servos violently in order to comply. When the alternate host takes control of the K2A, it must issue a mode value to the base. This is a single byte value, written to the MODE address 2400H (See the listing K2COM.DEF). This and all the messages that are discussed here follow the format described in section 2.4.1. To place the K2A in the torque

commanded mode, a 1 is written to this location; for the velocity commanded mode, a 5 is written to 2400H. The MODE value should be written ONLY at the beginning of the control sequence and NOT repeatedly.

After the MODE has been set to 1 or 5, the host must continuously send a 5 byte message to control the drive and steering. Between these messages the host can request data using the format described in section 2.4.2, or write other data, but no more than one message should be inserted between the 5 byte control messages. Alternatively, if the K2A has the K2ASS4 prom installed, the "Quick Drive" message may be used (See 2.4.3).

If the quick message is not used, the 5 byte "transmit" message is sent to address 2402H (DEADMAN) and writes the values DEADMAN, DVEL, and SVEL (See K2COM.PRN). The conventional drive format is;

BYTE RANGE MEANING

1 2-30 DEADMAN TIME SET- The vehicle will stop in this many 10ths of a second if another message is not received).

2,3 -250 to 250 DRIVE COMMAND- This is a signed 16 bit word, low byte first, and is either the torque command, or the velocity command for the drive servo depending on the MODE.

250= 2.50 ft/sec in mode 5.

4,5 -350 to 350 STEER COMMAND- This is a signed 16 bit word, low byte first, and is either the torque command, or the velocity command for the drive servo depending on the MODE

350= 350 Binary Deg/sec in mode 5.

These three values (5 bytes) were placed together in the K2A's memory so that they could be sent in a single message. The message will appear as follows:

```
:05240201140000FFFFC2<CR><LF>
```

This means write 5 bytes starting at address 2402H in computer number 01, the deadman being 14H (2.0) seconds, the drive being 0 and the steering being 0FFFFH (-1 decimal). The negative checksum is C2H. This would result in an extremely slow turn to the left in closed loop, but would probably do nothing in mode 1 (since it wouldn't be much torque).

The controlling host should check the status of the K2A by reading the STATUS byte (2401H). The status should be 0 (normal). Various status values are given in K2COM.DEF in the appendix. When the host is finished with its move, it should set the MODE back to 0 (HALT).

NOTE: Collision avoidance does not function in this method of control, but it is possible to interrogate the CA-01 for target range and perform collision avoidance in the controlling software.

E.3.5 Path Program Down-Loading

The K2A may also be controlled by down-loading a path program from an alternate host, and then putting the K2A into the AUTOMATIC mode by sending a 2 to the mode byte. The path program normally includes 255 6 byte instructions, a 2 byte positive 16 bit Drive Acceleration value (1 to 5), and a 2 byte positive 16 bit Steer Acceleration value (1 to 15). The Automatic Mode is used by Cybermotion in all of its autonomous operations.

IMPORTANT: With K2A software carrying revisions 3.06 and beyond, it is not necessary to send all 255 instructions. Instead, only the needed instructions may be loaded. However, it is now necessary to load the memory location LASTINS (See K2COM.DEF) with the highest valid program instruction number. When the K2A is turned on, this value is set to 255, however downloads of programs from the K2SRV or

Dispatcher programs may change this value. This value is intended as a safety limit to prevent accidental execution of invalid program segments. If any instruction above this value is encountered, the vehicle will halt with a ILLEGAL FUNCTION status.

Once this path program data has been sent, a 1 may be written to PLOADED (2407H) to indicate to the K2A that it has a valid program loaded. The MODE may then be set to AUTOMATIC (2). When the K2A is first placed in AUTO there is a slight delay while it calls on the Control link (K2ASM4 only) to try to find out what standard hardware is connected (such as collision avoidance). After about 2 seconds, the K2A will begin executing the program. After the first automatic operation, the 2 second delay will not occur, as the K2A keeps a log of the systems on board. NOTE: IN EARLIER VERSIONS OF THE K2A SOFTWARE, THE X AND Y POSITION WERE SET TO ZERO AT THE BEGINNING OF AN AUTOMATIC PROGRAM EXECUTION. THIS IS NO LONGER THE CASE, AND IT IS IMPORTANT THAT THE USER ASSURE THESE VALUES ARE VALID BEFORE MOVEMENT COMMANDS ARE EXECUTED! When an END or HALT instruction is encountered, the K2A will change its MODE back to HALT (0).

It is possible to "fudge" in the AUTO mode, by starting a program which contains a first step of JUMP to step 0. Thus the K2A will do nothing while the program loops. The host may then write the next few steps (instructions) which end with another jump to itself. To release the K2A from a loop, the host may rewrite the jump address. The advantage of using such techniques is that all of the power of the AUTO mode is available (such as docking instructions and collision avoidance operation).

As of version 3.06, the USE instruction may also be used for modified automatic control. In this case, a simple program may be down loaded and made to execute with its arguments taken from RAM so as to continuously change its destination. For example:

Step Instr. S X Y

0 WRITEW 1 16000 0 ; Zero X target.
1 WRITEW 1 16002 0 ; Zero Y target.
2 WRITEB 1 16004 0 ; Zero Speed.
3 SET XY - 0 0 ; Zero position.
4 READB 1 16004 - ; New Target?
5 JUMP= 4 - 0 ; IF not loop.
6 USE 004 16000 16002 ; Addr of targets
7 RUN - - - ; Run to target at
8 JUMP 0 - -

Note that the controlling computer would drive the platform by simply putting relative destination coordinates in addresses 16000 and 16002 and the speed in 16004 (Base 10), and then reading 16004 waiting for it to go to zero to indicate that the move had been executed. If absolute position commands were used, steps 0, 1, and 3 could be eliminated. NOTE that since the S argument is a single byte, the address it represents in the USE instruction is 16000 (3E80H) plus its value. The 256 bytes starting at 16000 (3E80H) are reserved as user scratch pad.

E.3.6 Resuming a Path Program

A path program may be interrupted by the K2A, or by an operator at the host terminal. Provided that the vehicle is not relocated or disoriented (i.e powered down), the interrupted path program may be continued at the last instruction by sending a mode command of RESUME. When RESUME is executed, the K2A will head for its next destination from wherever it is. Thus if the vehicle encounters an obstruction, the

operator can manually drive it around the obstacle and RESUME the program. The vehicle will head directly for the next path vector end point from its new location (it will not try to get back on the original path).

E.3.7 Protected continuous control (Cruising)

The MANUAL, and ZMODE methods of control were designed primarily for direct K2A operation. When the K2A is used in the Navmaster configuration, these modes do not take advantage of any of the vehicle's powerful collision avoidance or navigation capabilities. The CRUISE instruction is a unique AUTO mode instruction in that it causes the vehicle to move while the program goes on to the next instruction.

CRUISE loads the variables DCR.VEL (Drive) and SCRVEL (Steer) with its X and Y arguments, and sets a flag that causes a background program to attempt to reach these drive and steer velocity targets using DACCEL and SACCEL, and taking into account data from the collision avoidance system. This allows a host computer or operator to send velocity requests to the vehicle by writing them into these variables. A master computer may also change the auto program to invoke DOCKing and WALL navigation behavior. The X,Y position and azimuth are read by the controlling host during CRUISING just as in the simpler modes.

E.4 MAINTENANCE

The K2A was designed for minimum maintenance. If your application does not exercise the vehicle a substantial percentage of the time, except for battery charging, maintenance may be done on 24 month basis. Tire replacement may occasionally be necessary if the vehicle is operated on harsh surfaces. Tires are matched for diameter under load, and should be ordered and changed as a set.

E.4.1 Battery charging and maintenance

The K2A uses two, Sonnenschein, 12 Volt, 85 Ah batteries. These batteries may be charged at a rate of up to 18 Amps without degradation of their performance. Unlike some battery types, these batteries provide the maximum number of cycles when they are maintained at high levels of charge. If the batteries are discharged to the end of their capacity they will give fewer cycles. On the other hand, OVER-CHARGING WILL CAUSE SERIOUS DEGRADATION OF CAPACITY.

If the batteries are to be charged with a standard automotive charger it is important to discontinue charging as soon as the charge current drops to between 2 and 3 amperes. If unattended charging is to be done overnight a timer should be placed on the charger and should be set to turn the unit off after 3 to 5 hours (depending on the state of depletion). AN AUTOMOTIVE CHARGER SHOULD NEVER BE USED WITH AUTO DOCKING CHARGING STATIONS as contact pitting will occur.

For applications that require maximum battery life, minimum charge times, and/or auto docking, the Cybermotion BC-01 Battery Charger is recommended. The BC-01 charges at rates up to 16 Amperes and supplies filtered DC. The charger also contains control circuitry that holds off current until a short time after contact is made with the battery circuit. This feature allows vehicles to auto dock without contact ring pitting. The BC-01 may be left on the vehicle indefinitely without harming the batteries. Finally, the BC-01 provides optoisolated output signals that indicate the state of charging to the vehicle (through the DB-01 Docking Beacon System).

E.4.2 Lubrication

The basic lubricant for the K2A's grease points should be extreme pressure grease that meets or exceeds M.I. G23827(B). One brand that meets this standard is Pennzoil Bearing Grease No. 706. Lubrication should be done on all grease points every 2000 hours or every two years, whichever comes first.

Lubrication points are shown in Figure 2. The grease may be applied to the gears with a rubber knife or stiff bristle brush. Rotate the respective servo between applications to assure complete coverage. **DO NOT GET THE APPLICATION TOOL CAUGHT IN THE GEAR TEETH AS DAMAGE MAY RESULT.** The drive gear box is filled with oil but should never require maintenance. If an oil leak is experienced or if the box is drained for service, it should be refilled with Penzoil EP Gear Lubricant No. 4096 or equivalent.

Appendix F

ARIES: An Intelligent Inspection and Survey Robot

FIELD TRIALS

**Clemson University, Cybermotion Incorporated,
and The University of South Carolina**

F. FIELD TRIALS

F.1 FERNALD

ARIES was tested, evaluated, and demonstrated at Fernald Site during 5-22 August 1996 according to requirements of the project's Phase 3 proposal. The ARIES tests were conducted in Building TS-4 at the Fernald Site. This building has translucent plastic walls and sodium vapor interior lights. TS-4 has capacity for as many as 12,000 drums and is partitioned into separate halves by short berms.

The time spent at Fernald was both necessary and very beneficial to the completion of Phase 3. The goal of this demonstration was to characterize productization issues. The experience of operating the system in the "real environment" was invaluable and identified areas for further investigation and enhancements (throughput, power efficiency, exception handling onboard, etc.). During the fourth day at Fernald, the system was fully operational and navigating drum aisles in the TS-4 facility.

Overall, the system set up easily and worked beyond our expectations. We feel that all major items from the test plan were at least addressed informally during the 17-day period. Specific items concerning the Site Manager and operation of the system were covered in detail with FERMCO personnel. According to feedback from customers at the formal demonstration on 21 August, the demonstration was successful.

The system now resides at the Cybermotion facility in Salem, VA where it is undergoing final enhancements for productization. An additional system will be fabricated and tested. A test aisle facility will be set up in Salem for evaluations and demonstrations. The University of South Carolina and Clemson University both continue in their efforts to port the on-board software from VxWorks to the Windows NT platform.

F.2 IUOE

7/14

- * Arrived at IUOE facility in late afternoon due to flat tire on truck.

7/15

- * Set up system.
- * Laser scan was not working.
- * Recalibrated vision system.
- * Batteries dead.

7/16

- * Batteries still dead (breaker tripped on charger).
- * Laser scan fixed.
- * Fore-bar actuator broke belt.
- * Received replacement batteries from Cybermotion.

7/17

- * Brecht arrives to work on vision system.
- * Vision system properly calibrated.
- * Arranged overnight shipment for new actuator belt.
- * Worked on AutoCAD drawings of test plan layouts.
- * Began calibration of color camera.

7/18

- * Observed that facility floors are very uneven in places.
- * Belt NOT received (they sent it USPS).
- * Experimented with methods for automated color calibration.
- * Began laying out some paths.
- * Brought in some drums for training the vision system.

7/19

- * Began training of vision system.
- * Received replacement belt(s) for the actuator.
- * Actuator still needed other repairs (which were made).
- * Having installation problems with Site Manager.
- * Vision system auto-calibration routines finally complete.

7/20

- * ARIES had some navigation problems.
- * Laser scans did not seem to be working properly.

7/21

- * ARIES ran pretty much all day.
- * Collision avoidance portions of the test plan were completed.
- * Navigation problems seem to be resolved.
- * A bug was discovered in the the database portion of Mission Handler.

7/22

- * Ran "square-10" navigation precision test.
- * Navigated "long aisle."
- * Worked on Database bug (and workaround).

7/23

- * Ran "long aisle" test.
- * Ran "missing pallet long aisle" test.
- * Ran "staggared pallet long aisle" test.
- * Ran "crooked pallet long aisle" test.
- * Set up "chevron" aisles.
- * Navigated chevron aisle configuration.
- * Determined that corrupted lasercal file was to blame for poor laser scans.

7/24/97

- * Determined that INEEL barcodes (here) probably would not be suitable. They appear to have been printed with a 9-pin dot matrix printer with a bad ribbon.

Barcodes should be laser-printed with dark, well-defined lines.

- * Fore-bar actuator broke belt AGAIN!
- * Work continues on the vision system.

7/25/97

- * Repaired actuator (one belt left).
- * Test aisle configuration set up (minus 85-gallon drum aisle).
- * Test aisle navigated.
- * Read USC barcodes on drums.
- * Continued work on vision system.

7/26/97

- * Continued debug on database problem.
- * Continued work on vision system.
- * Trained vision system.

7/27/97

- * Killed database bug.

F.3 LOS ALAMOS NATIONAL LABORATORY

11/10

- * Arrived at LANL and checked in.
- * Made preliminary measurements in the dome.
- * Truck carrying system did not arrive until late afternoon and delivered system to a central receiving point. Will not be able to unpack until Wednesday (Tuesday is a holiday).

11/12

- * Learned that two (of the eight) crates were heavily damaged in shipping. Fortunately, they were the crates containing the LIDAR fiducials. The status of the system as a whole will not be verified until tomorrow when we put the CPS on the robot (Although it looks like everything else is okay).
- * Crates were delivered to the storage facility this afternoon.
- * Computers were set up, and the majority of the crates were unpacked.
- * The cold seems to be having minor effects on the offboard computers.

11/13

- * Put CPS on robot.
- * Discovered that Barcode reader was damaged in shipping. Have replaced it with spare.
- * Discovered that one of the strobe modules was damaged in shipping and are trying to repair it (no spare here).

- * Condition of the LIDAR will be checked tomorrow.
- * The rest of the CPS appears intact.
- * Have finished unpacking (the escort restriction is seriously curtailing the amount of time we have to work).

11/14

- * Received replacement strobe component.
- * Continued preparing AutoCAD map of facility.
- * Began placing the fiducials.
- * Plan to operate the system (a minimal subset) tomorrow.

11/15

- * Completed site measurements and AutoCAD map.
- * Set up the inspection site.
- * Executed nine inspection runs.
- * May have to do some work on front sonar transducers.
- * May have a problem with the panning mechanism.
- * Initial performance of the vision system has been poor (hope to address that next week).

11/17

- * Made adjustments to ultrasonic transducers (apparently) adversely affected by the cold.
- * Increased number of aisles in test area.
- * Covered sensitive electronic components in plastic to protect them from condensed water falling from the roof of the tent.
- * Began to address turning in a 36-inch aisle.
- * Calibrated LIDAR.

11/18

- * Batteries may be weakened by the cold resulting in less efficient charging and shorter run times.
- * Repaired (shipping) damage to pan mechanism.
- * Replaced front ultrasonic transducers.
- * Had trouble with E-Stop circuits (maybe due to water from the tent).
- * Will attempt to assess the condition of the vision system tomorrow.

11/19

- * John left today.

- * Found loose battery connections.
- * Vision system is in poor condition (needs calibration, adjustments, &etc).
- * Are navigating the 36" (really closer to 34") aisles.
- * Observed noise from strobes (?) affecting barcode reader performance.
- * Began repairing crates for return shipment.

11/20

- * Repaired damaged crates (except for one).
- * Cleared equipment for free-release.
- * Boxed fiducials and CPS.
- * Plan to complete packing by noon tomorrow.
- * Shipping destination is unclear right now (at Vijay's request), although it would seem wise to return the system to Cybermotion for continued testing and preparation for INEEL demo.

F.4 IDAHO NATIONAL ENVIRONMENTAL ENGINEERING LABORATORY

7/12/98

- * Arrived in Idaho Falls, ID.

7/13/98

- * Arrived at INEEL North Boulevard Robotics Center facility.
- * Assembled ARIES II.
- * Unpacked equipment for ARIES II.
- * Designed test/demo site for coexistence with IMSS.

7/14/98

- * Set up and mapped test "Site."
- * Completed check out of ARIES II.
- * Began test and debug of "Plugin" instruction (for docking the robot).

7/15/98

- * John returns to VA.
- * Replaced class IIIb laser with class IIIa (eye-safe) laser and de-activated dent detection system (class IIIb laser).
- * Laser calibration training video (Lasercal.exe - for physical calibration of vision system).

7/16/98

- * Site Manager training.
- * Brief KXA training.
- * ARIES shutdown and startup training and video.

7/17/98

- * Completed Site Manager training.
- * Training on "training" the vision system (train.exe).
- * Training on Test application (test.exe).

- * Review of documentation.
- * Review of delivered equipment.

7/18/98

- * No activity.

7/19/98

- * Return to Columbia, SC.