

To the Graduate Council:

I am submitting herewith a thesis written by Thomas E. Swift entitled "Basic Vision Experiments with the Mobile Robot HERMIES-II." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.



Robert W. Rochelle, Major Professor

We have read this thesis
and recommend its acceptance:



Accepted for the Council:



Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Thomas E. Swift
Date Nov. 15, 1985

Basic Vision Experiments with
the Mobile Robot HERMIES-II

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Thomas E. Swift
December 1985

ACKNOWLEDGMENTS

The work reported here was performed under U.S. Department of Energy contract #DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc. I wish to express my gratitude to numerous people who have assisted me in the research and preparation of this thesis. In particular, I am indebted to Professors Robert Rochelle and Robert Bodenheimer who served on my committee, and Dr. Bill Hamel who both served on my committee and directed my work at Oak Ridge National Laboratory. I also wish to thank Dr. Chuck Weisbin and the rest of the CESAR team for providing the facilities, advice and encouragement throughout the period of my research assistantship without which this work would not have been possible.

Finally, I wish to especially thank my wife, Darla, whose help in the editing and preparation of the final copy of this thesis proved invaluable.

ABSTRACT

The research team of the Center for Engineering Systems Advanced Research (CESAR) uses the mobile robot HERMIES-II (Hostile Environment Robotic Machine Intelligence Experiment, Series II) for experiments in autonomous obstacle avoidance, path planning and navigation. Sensor feedback from the robot comes from several sources, including sonar and a binary vision system. It is this vision system with which this work will primarily be concerned.

The vision system implemented on the robot is based on the Micron-Eye OpticRam camera from Micron Technology Inc. The camera was interfaced to the robot and several image processing routines were written in the language FORTH. The general purpose assembly language routines provided with the MicronEye were rewritten and condensed for this specific application. "Primitives" were written to allow external computers such as CESAR's LISP computer to access information provided by the camera. Routines were written to do basic tasks such as calculating the angle from the center of the robot to the edge of an obstacle. These routines were used to develop navigation routines. The LISP computer previously had used only sonar feedback to determine the locations of obstacles and controlled the robot by passing commands over an RS-232 serial link. The new routines allowed more processing to be done in the robot and decreased the time needed to accomplish an overall task.

TABLE OF CONTENTS

CHAPTER	PAGE
I. INTRODUCTION	1
II. THE MICRONEYE SYSTEM	8
Hardware	8
Software	9
III. MODIFICATION OF ASSEMBLY LANGUAGE ROUTINES	11
Routines Size Reduction	11
Interfacing Routines to FORTH Environment	12
IV. IMAGE PREPROCESSING ROUTINES	15
The Routines	15
Optimizing Routines as CODE Words	16
V. NAVIGATION ROUTINES	19
The Robot's World	19
Incorporating Vision with Navigation	25
Other Vision Primitives	39
VI. FUTURE VISION SYSTEMS	47
VII. CONCLUSIONS	50
LIST OF REFERENCES	53
APPENDICES	55
Appendix A	56
Appendix B	68
VITA	78

LIST OF FIGURES

FIGURE	PAGE
1.1. The author with the mobile robot HERMIES-II in the Center for Engineering Systems Advanced Research (CESAR) Laboratory	2
1.2. Block Diagram of Robot	4
5.1. Diagram and Coordinate System of Navigation Area	20
5.2. Diagram for CORRIDOR Routine	24
5.3. Robot's Eye View of Right Edge of Obstacle	28
5.4. MicronEye Image of Right Edge of Obstacle	28
5.5. Flowchart of GO Routine	30
5.6. Geometry for Movement in DODGE2LEFT	32
5.7. Lines Used as a Marker	36
5.8. MicronEye Image of Lines	36
5.9. Diagram for GETPERP Routine	37
5.10. Diagram for BOX-ORIENT Routine	40
5.11. Sobel Templates	43
5.12. MicronEye Image of Block on Light Table	45
5.13. Image of Block After Sobel Edge Detection	45

CHAPTER I

INTRODUCTION

The Center for Engineering Systems Advanced Research (CESAR) was established in 1983 at the Oak Ridge National Laboratory (ORNL) to address long-range, energy-related research in intelligent control systems [1]. The Center was established by the U.S. Department of Energy (DOE), Office of Basic Energy Sciences, Division of Engineering and Geosciences to study and develop intelligent machines that will operate in unstructured hostile environments. In addition to intelligent machines, the CESAR team is studying areas that include mathematical modeling, real-time control, sensor integration and strategy planning. The aim of this research is to develop an autonomous machine that can make intelligent decisions based on its perception of its environment which will be unknown and possibly changing. The first step toward this long-range goal was the construction of the mobile robot HERMIES-I in early 1984.

HERMIES-I stands for Hostile Environment Robotic Machine Intelligence Experiment, Series-1. As described in [2] and shown in Figure 1.1, HERMIES is a self-contained mobile robot with manipulators. It is driven by two independently controlled D.C. motors which are powered, along with the rest of the robot, by two 12-volt, 15-amp-hour, rechargeable batteries. Connections are provided for external power which can run the robot or recharge the batteries when the robot is off. The robot is controlled by an on-board distributed processor architecture. The main computer is an IBM-XT with a single floppy disk drive. One satellite processor is a Z8 microcontroller which is programmed in assembly language and is dedicated to controlling the



Figure 1.1. The author with the mobile robot HERMIES-II in the Center for Engineering Systems Advanced Research (CESAR) Laboratory.

arms. The other processor is a dedicated stepper motor controller produced by Cybernetic Micro Systems and directs the stepper motors that move the pan-tilt sensor platform. A block diagram of the robot is given in Figure 1.2.

The sensor platform is located on top of the robot and is able to pan through 350 degrees and tilt plus or minus 45 degrees from horizontal. Located on this platform are the sonar sensor and the OpticRAM binary camera. These two sensors were used heavily in the research for this thesis. The sensor platform steering allows a side area of the environment to be examined without moving the entire robot.

HERMIES was constructed for CESAR in the first half of 1984 by Prism Robotics Inc. The robot was designed as a low-cost testbed for ideas and algorithms developed by the CESAR team. Although its capabilities are somewhat limited (i.e., the Heathkit arms used as manipulators can only lift about 16 oz.), the mobility and sensors on the robot allow the CESAR team to test algorithms such as a path planning algorithm in a real world, albeit controlled, situation. HERMIES gives researchers a better idea of the limitations of hardware interacting with its environment than can be achieved through computer simulation. In this role as a learning tool, the robot has been very useful to the CESAR team.

HERMIES-I is the first step in a series of increasingly complex and capable machines planned for CESAR. Many modifications have been made to HERMIES to increase its capabilities and reliability including a new sensor platform mounted in late October. This platform is driven by servo motors and contains five sonar units rather than one. The modifications made to HERMIES have been extensive enough that the robot has been designated HERMIES-II, which is the designation to be used throughout this thesis. Fu-

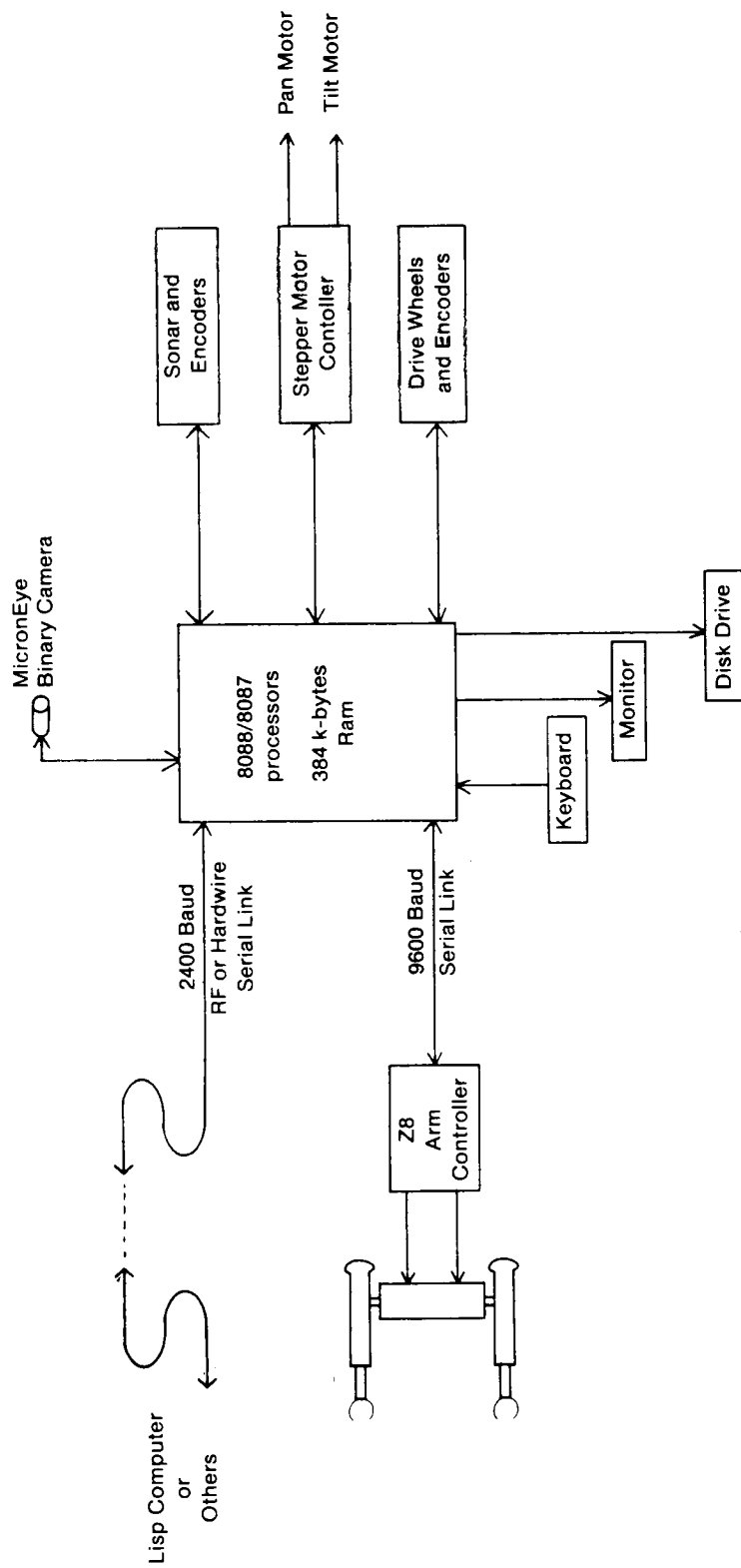


Figure 1.2. Block Diagram of Robot.

ture versions of HERMIES will be much larger and will have much more powerful computers on board as well as arms capable of lifting up to 35 pounds.

All on-board programming for the robot is done in the language FORTH. FORTH is a flexible, extensible, programming language that was designed especially for control purposes. It is a high-level language that operates at speeds approaching that of assembly language. FORTH is an interactive environment and allows the user to define new commands, called "words," based on previously defined words. The version of FORTH used on the robot is polyFORTH™ by FORTH Inc. [3]. FORTH is a stack-oriented language and operates in Reverse Polish Notation (RPN). This is one of the factors that leads to FORTH's speed. FORTH also allows time-critical tasks to be written in assembly language through the definition of CODE words. Defining a CODE word calls up FORTH's simple one-pass assembler. Assembly language mnemonics are joined in a manner similar to the way high-level words are joined to define a new word. The routine is then executed the same way high-level routines are executed. The text by Brodie [4] is a very good introduction to the language for anyone not familiar with FORTH.

The polyFORTH system for the IBM-PC/XT has several nice features, including a color graphics package, full-screen editor and 8087 numeric co-processor support. The numeric co-processor package was used extensively in this work. Especially useful were the trigonometric functions and floating point arithmetic.

The majority of the programs written for this thesis were written in FORTH. It should be noted that whenever FORTH commands or programs are referred to in this text they will be written in all upper-case (i.e., PCALC,

GO, or DUP). The program listings, which are given in Appendix A, are also in a format peculiar to FORTH. Programs are saved on "screens" which are 16 lines of 64 characters in size. Documentation for the programs is saved on "shadowblocks" which are the corresponding screens on the next disk drive in the system. In a two-drive system it is possible to change from the program listing to the shadowblock by typing "Q." In Appendix A the shadowblocks are listed to the left of the program blocks.

One goal of CESAR research is to develop a robot that can intelligently react to its environment. Obtaining this goal depends heavily on the amount and quality of sensor data that the robot can obtain. This thesis research moves toward this goal by providing the robot with more advanced sensor data. Previously, only sonar data had been used to determine where obstacles were located. It soon became apparent that this presented some drawbacks. The cone-shaped beam path of the sonar created some ambiguity as to exactly where edges and corners were located. This problem was partially relieved by using four sonar transducers hooked together as an array. This reduced the beam path from 32 degrees to 12 degrees. However, a need for more accurate data was still noted. This need prompted the attempt to use the simple camera provided with the robot. The idea was to use the camera to determine as closely as possible where the edge of an obstacle was located. By knowing more accurately where the edge is, it is possible to plan a better path around the obstacle. What was needed was a set of primitives that would access and manipulate visual data, pick out prominent features, and make decisions based on the information obtained.

The work undertaken to develop these primitives can be broken down into two main phases. Phase one involved interfacing the camera and its

software to the FORTH environment, and developing some basic routines to manipulate the data. Chapter II describes the hardware and software of the MicronEye System. The phase one work is covered in Chapters III and IV. Chapter V covers phase two of the work. This involved developing and coding the image processing and navigation routines that make up the primitives. These primitives can be used stand-alone or in conjunction with another computer. Chapter VI discusses plans for the future with the MicronEye and with the new vision system that CESAR has recently acquired. Finally, VII covers the conclusions.

CHAPTER II

THE MICRONEYE SYSTEM

2.1 Hardware

The MicronEye is based on the IS32 OpticRam developed by Micron Technology, Inc. It is a 64k dynamic RAM in a 16-pin ceramic DIP package with a clear glass lid to allow light to fall on the elements of the RAM. The IS32 is composed of 65,536 individual light sensitive elements that are arranged into two arrays of 128 rows and 256 columns. Each element or pixel is a light sensitive capacitor that is charged to five volts and then allowed to discharge toward zero. The rate of discharge is proportional to the intensity of the light falling on the array and to the length of time the array is exposed. The time of exposure is controlled from software allowing greater flexibility in controlling the system while eliminating the need for a mechanical shutter. After an image is obtained, the OpticRAM is refreshed, charging the capacitors back to their initial voltage level. The two arrays are separated by a non-light sensitive zone that is approximately 25 elements in width.

In a typical application, the OpticRAM would be exposed or "soaked" for a specific amount of time and then read. A particular pixel would be read by addressing the appropriate row and column for that pixel. The IS32 would then compare the voltage of the capacitor with a fixed threshold voltage. If the capacitor voltage is higher than the threshold voltage, a "1" is output for that pixel. If the capacitor voltage is lower, a "0" is output. Due to the fact that the capacitor discharge rate is proportional to light intensity, a logic level of "0" is associated with a white pixel and a logic level of "1" is associated with a black pixel. The MicronEye Operator's Manual [5] describes a method by

which grey scale images can be achieved. Multiple images taken with different length exposure times can be averaged together to determine the relative greyness of the pixels. This method was not needed, however, since the binary image was adequate for the low resolution, high contrast scenes used in the work described in this thesis. Full technical specifications of the IS32 OpticRAM are given in Appendix D of the MicronEye Operator's Manual.

Electronics are provided as an interface between the computer and the OpticRam. A board that plugs into a standard slot on an IBM-PC or PC-compatible computer is provided with the system. This board contains all of the circuitry necessary to drive the IS32. The circuits include a timing generation circuit for baud rate, read/write, and refresh signals; address descramble to provide proper addressing to the IS32; command receiver and transmitter circuits to interpret commands from the computer and receive data from the OpticRAM; and a soak/refresh circuit that acts as an electronic shutter. A full description of these circuits is given in Appendix G of reference [5].

2.2 Software

The software provided with the MicronEye system consists of a demonstration BASIC program named MEYE.BAS, a source listing of the assembly language subroutines that are called from BASIC, and the assembled code of the subroutines. The MEYE program is menu driven and allows the user to display pictures taken by the MicronEye camera, select different picture modes, change the exposure parameters, and display the current status of the camera. This program introduces the new user to the MicronEye and demonstrates the features of the system.

The real heart of the system, however, is the set of assembly language

subroutines that are called by the MEYE program. These do the work of sending the proper commands to the camera hardware, receiving the image from the camera and storing it in memory, and moving the image to the screen buffer for display. The routines take care of all details necessary to properly display the images in the various sizes and modes which include 128×64 , 512×64 , 512×128 and 640×128 with binary and limited grey scale images available in the last three sizes. Like the BASIC program MEYE, the subroutines are general purpose and serve to demonstrate the features of the system. This complexity was not needed for this project and the subroutines were eventually reduced to one-third of their original length. As described in the next chapter, only three basic routines were needed. They were PCALC which obtains the image, MOVESCR which displays the image, and CLEARW which initializes the image buffer with known values.

CHAPTER III

MODIFICATION OF ASSEMBLY LANGUAGE ROUTINES

3.1 Routines Size Reduction

The assembly language routines provided with the MicronEye system are about 1500 bytes long. They were written to be general purpose and demonstrate the features of the camera. The demonstration program provided with the camera is written in BASIC. It calls the assembly language routines to set up the camera and display the picture in various modes. As can be seen on page 4-5 of the MicronEye Operator's Manual, there are six different modes in which pictures can be obtained and displayed, with one or two pictures being displayed on the screen at once. It was apparent at the onset of the project that the complexity and flexibility of these routines were not necessary. Since the user is encouraged to customize the system to his needs by the inclusion of the source listings for the routines, one of the first tasks was to study the listings to learn exactly how the routines functioned.

The first decision to be made was which picture mode to use. The 128 × 64 mode was chosen for several reasons, the first and foremost of which was speed of execution. For an intelligent mobile robot to react effectively to its environment, it is necessary to process sensor data as quickly as possible. The 128 × 64 mode of the camera produces a picture of the smallest size and therefore with the least amount of data to be processed. This mode also produces the lowest resolution, which is an acceptable trade-off since the objects to be recognized are large and contrast sharply with the background. The 1024-byte binary image provided can be processed faster and with simpler

algorithms than the larger, more complex modes available. The smaller size is also the quickest to update with a new image.

Other modifications made to the routines included deleting routines, such as the print screen routine, which were not relevant to the planned robot navigation routines. Along with all parts of the routines dealing with different size pictures, all parts dealing with images taken from the lower array were deleted. The presence of the lower array, coupled with the gap of several pixels between the arrays, only served to make processing more cumbersome without adding more information. The final version of the assembly language routines is about 500 bytes, or one-third the length of the original routines.

3.2 Interfacing Routines to FORTH Environment

The condensed versions of the assembly language routines were still too long and complex to be rewritten as FORTH code words. It was simpler to edit and reassemble the routines using the IBM Macro Assembler. The poly-FORTH DOS routines were then used to locate the file containing the assembled opcodes on the DOS diskette. The data from this file was transferred to a FORTH screen buffer and saved on the FORTH diskette as a screen. A FORTH word (see DOWNCODE in listing) was then written to move the opcode from the screen buffer to normal memory for execution. Appendix B contains the listing generated by the Macro Assembler.

As new FORTH words were defined, the pointer to free memory moves toward high memory. It was necessary to place the assembly language routines, which are relocatable, in memory at a place where they would not be affected by changes in FORTH routines. It was first decided to place the assembly language routines below the FORTH applications code in memory.

This proved inefficient since occasional changes in the basic FORTH system would change the starting address of the routines. It was then decided to locate the routines high in memory where FORTH seldom operates. The polyFORTH operating system for the IBM PC is designed to operate only in the first 64k segment of memory. Because of this, it was decided to locate the assembly language at the start of the second 64k segment of memory. This allowed the starting addresses of the routines and buffers to always remain constant. Another advantage of placing the routines in a separate segment stems from the size of the buffers used to manipulate images. A 1k-byte buffer is used to hold the original image and two 8k-byte buffers are used for manipulation. These buffers would have taken too much memory away from FORTH applications in the lower 64k-byte segment of memory.

The utilities provided with the polyFORTH system include intersegment operators that allow manipulation of data in segments other than the first 64k-byte. The starting addresses of the image buffers and key variables used by the assembly language routines were assigned as constants in the FORTH applications. The basic mechanism for calling an assembly language routine from FORTH consists of defining a CODE word which executes a subroutine CALL. This FORTH system, however, does not support intersegment calls (far calls). Therefore the opcodes for a far call were obtained from reference [6] and the calls to the subroutines were hand assembled. FORTH words were written to execute these subroutine calls. Three routines are called from FORTH in this manner. PCALC is a FORTH word which calls the routine that actually obtains the image from the camera and stores it in the 1k-byte buffer labeled WORKMAP. MOVESCR is a FORTH word which calls the routine that moves the image from WORKMAP to the screen buffer for

display. Finally CLEARW sets all bytes in WORKMAP to a specific value. This routine aids in debugging image manipulation routines by providing a test pattern of known characteristics.

•

CHAPTER IV

IMAGE PREPROCESSING ROUTINES

4.1 The Routines

The image produced by the MicronEye camera is a binary image and is assembled in the WORKMAP buffer by the assembly language routines. The image is 128×64 pixels and therefore contains 8192 bits of information. The WORKMAP buffer is arranged in memory as a 16×64 array of bytes where each byte contains eight pixels. When processing an image it is frequently easier to operate on each pixel individually. The 8192-byte buffers TEMPBUFF and BIGBUFF were set up to contain an expanded version of the image where each pixel is assigned a separate byte. A convenient method was needed to transfer an image from buffer to buffer. FORTH words were written to handle this transfer. The words are described here and the code is given in Appendix A.

The word IMAGEEXPAND is the most frequently used image manipulation word. It expands the contents of the WORKMAP buffer so that each bit in WORKMAP (or each pixel in the image) is assigned a separate byte in the buffer TEMPBUFF. This allows pixels to be manipulated with byte instructions.

IMAGE-CONTRACT is the antithesis of IMAGEEXPAND. Either BIGBUFF or TEMPBUFF can be specified as the buffer containing the data to be compressed into WORKMAP. This word is used when it is desired to see the results of processing on an image. The modified image is compressed into WORKMAP from which it can be displayed using the word MOVESCR.

Sometimes it is necessary to move an image from one of the large buffers to another. For this, the words TEMP>BIG and BIG>TEMP were

written. They move the contents of TEMPBUFF to BIGBUFF and the contents of BIGBUFF to TEMPBUFF respectively.

A word used frequently when working with objects on a light table is OBJLOCATE. This word tries to establish a rectangle around an object to measure its location. This word scans first from left to right on increasing rows of the image to establish the upper left-hand corner of the object. It then scans from top to bottom on decreasing columns starting at the right edge. This establishes the upper right corner of the object. The lower right corner is established by scanning from right to left and from the bottom row to the top. Finally, the lower left corner is found by scanning from bottom to top starting at the left edge of the image. Once the rectangle containing the object has been defined, processing can be limited to this rectangle and the rest of the image ignored. OBJLOCATE has disadvantages, however, in that it is only suited for "ideal" situations, such as a single object on a light table. In this situation, the object is backlit and there is no background clutter or "noise." The presence of background clutter would nullify the effectiveness of OBJLOCATE by making the area of interest artificially large.

4.2 Optimizing Routines as CODE Words

The words IMAGEEXPAND, IMAGE-CONTRACT and OBJLOCATE are used frequently in applications that involve image processing routines such as edge detection or smoothing. It was soon apparent that the amount of time needed to execute these three words added an unacceptable amount of overhead time to the execution of a given image processing routine. The word IMAGEEXPAND took approximately 17 seconds to execute, IMAGE-CONTRACT took approximately 8 seconds and OBJLOCATE took about 21

seconds to execute. With the long-range goal of real-time reaction on the part of the robot to a given situation and the frequency with which these words are used, it was decided that the effort involved in rewriting these words in assembly language was justified.

To achieve the increase in speed but retain the flexibility and ease of understanding of high level FORTH, only parts of these words were written as CODE words. The most time consuming parts of the routines were those that did the actual bit manipulation. For example, in the word IMAGEEXPAND the CODE word IMAGEXP takes a byte from WORKMAP, masks off each individual bit and stores it in TEMPBUFF. The byte from WORKMAP and the address in TEMPBUFF, however, are generated in the high level IMAGEEXPAND. A similar situation occurs in IMAGE-CONTRACT. Here, the CODE word IMAGECONT takes an address in TEMPBUFF or BIGBUFF and generates a byte using the lowest bit of the next eight bytes. This byte is then stored in WORKMAP by the higher level word. In OBJLOCATE, the CODE words ULC, URC, LRC and LLC scan the rows and columns of the image in TEMPBUFF until a pixel of the object in question is found. The numbers returned by these CODE words represent the byte offset into the TEMPBUFF buffer. These numbers are then converted into the coordinates of the corners of the rectangle containing the object. As can be seen in the listing for OBJLOCATE in Appendix A, the coordinates are stored as variables. The listing in the appendix contains both the strictly high level and the mixed high- and low-level versions of these words.

The optimization of these words greatly improved the performance of the image processing routines. Each of the three words ran in less than a

second after being optimized. This enhanced the viability of using image processing with the MicronEye system on the robot HERMIES-II.

CHAPTER V

NAVIGATION ROUTINES

5.1 The Robot's World

The robot's world consists of a 12×24 foot navigation area in the CESAR Laboratory at ORNL. A grid of 2×2 foot squares is laid out on the floor with black paint. Numbers for each foot of the axes are marked along the right and front walls. A diagram of the navigation area is shown in Figure 5.1. A coordinate system was adopted such that the 270 degree direction is along the main length of the room. This coordinate system is also shown in Figure 5.1. Within this area are four boxes on wheels that can be moved about to form various obstacles. Two of the boxes are $2 \times 2 \times 4$ feet high, one box is $1 \times 1 \times 4$ feet high and the last is $2 \times 2 \times 2$ feet high. These are the only obstacles that are allowed to obstruct the robot. While this is not a "real world" situation, it provides valuable information on the performance of the robot and its sensors.

A typical application currently being carried out in the laboratory has the robot simply executing commands being sent to it by an external computer. The external computer is a Lambda model produced by LISP Machines Inc. (LMI). This computer is based on a LISP processor developed at the Massachusetts Institute of Technology and is used by the CESAR team to implement algorithms for path planning, obstacle avoidance, and other artificial intelligence projects. A program has been written to select a path through a set of known obstacles and direct the robot to follow this path. Another program uses the robot's sonar to detect where unknown obstacles are and then attempts to plan a path to the given goal. In both programs, little or no

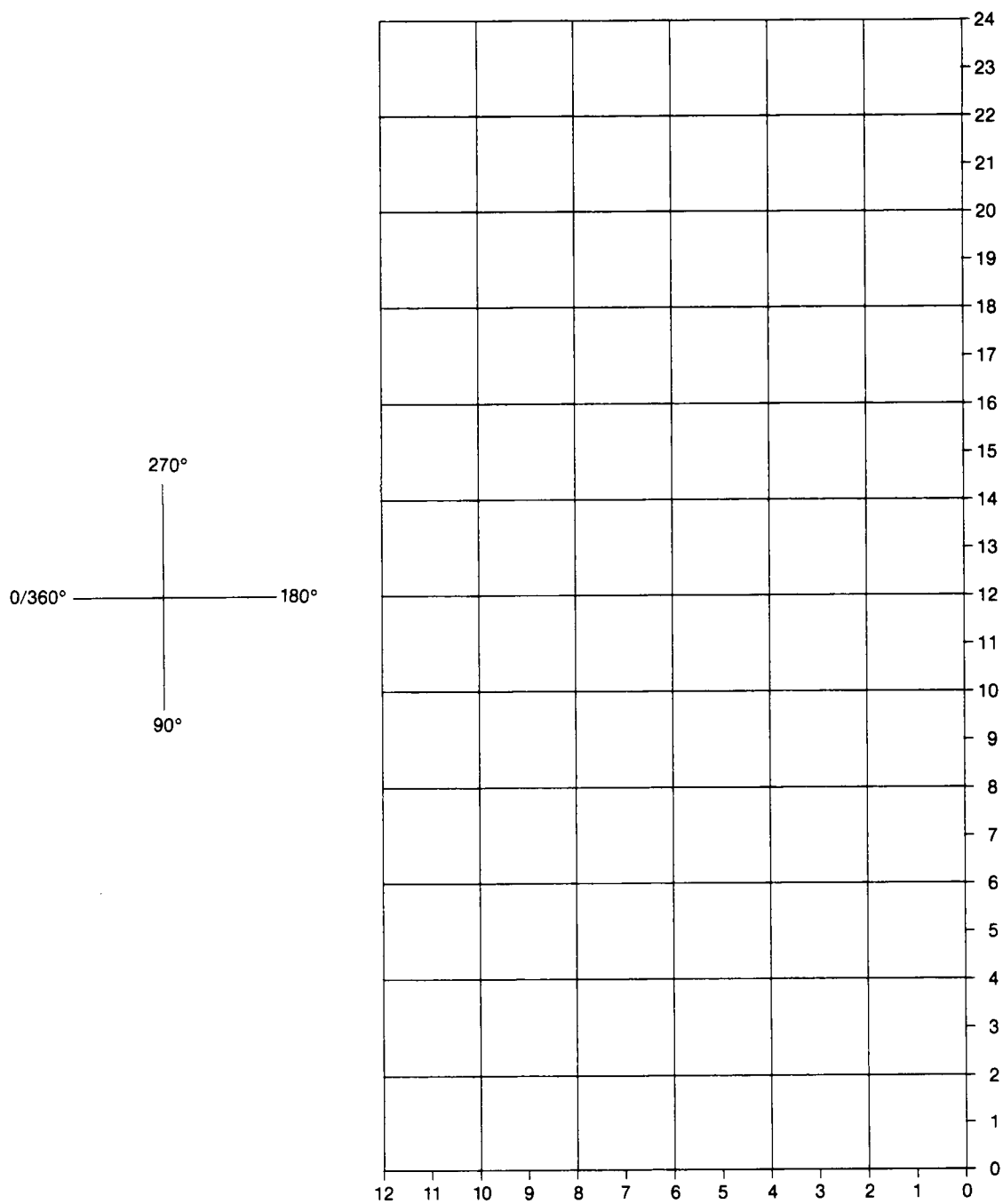


Figure 5.1. Diagram and Coordinate System of Navigation Area.

information processing is done on board the robot. The robot simply executes the commands passed to it by the Lambda computer. The Lambda does all of the work of processing the sonar data from the robot, keeping up with the robot's current position and orientation, and deciding which direction the robot should move next.

In order to develop vision routines that would provide useful information to the Lambda, it was decided to implement some navigation routines on board the robot that would test the effectiveness of the camera. The first step toward this was to implement routines that would allow the robot to constantly update its current position. To maintain as much accuracy as possible, poly-FORTH's math package with its floating point arithmetic routines was integrated with the robot routines. The words SFMOVE, SBMOVE, SLTURN, AND SRTURN (see screens 250 and 251 of the listing in Appendix A) were developed as floating point analogues of FMOVE, BMOVE, LTURN, and RTURN. These words constantly update the floating point variables X-POS, Y-POS and ORIANG. X-POS and Y-POS contain the current x and y coordinates of the robot relative to the room coordinate system shown in Figure 5.1. ORIANG contains the robot's angular orientation. These variables must be initially set with the word CURRENT-POSITION and the values of the robot's position. After the variables are set, they will be constantly updated if the floating point movement words are used.

When HERMIES is being directed to move to a given position or goal, the most direct method is to turn the robot to face the goal and then move straight to it. To accomplish this the word ORIENT was written (see screen 252 of the listings). ORIENT assumes that the robot's current position is contained in the variables X-POS, Y-POS, and ORIANG and that the x and y

coordinates of the goal position are on the stack. It first calculates the x and y offsets and uses the Pythagorean theorem to calculate the hypotenuse of that triangle which is the straight line distance to the goal and stores this value in the variable LENGTH. ORIENT then calculates the magnitude and direction of the angle to turn to face the goal. The magnitude is determined with the ATAN function and the direction is determined from which value of the x and y offsets are negative. The robot then turns the necessary amount and direction to face the goal. Another variable, DIRECTION, is set to indicate which direction the robot turned. DIRECTION is set to one if the robot turns left and zero if the robot turns right. This variable is used by other words that need to determine which direction the robot last turned.

Once the robot is prepared to move, it is necessary to determine whether it is possible to move to the goal or if there is some obstacle in the way. This can be accomplished with the sonar. In early experiments with HERMIES, the situation occasionally arose where the sonar sensor, when pointed straight ahead, would not detect an obstacle that was within the robot's path but outside of the sonar beam. This led to collisions between the robot and the obstacle. To avoid this, it was necessary to check for a clear path or corridor down which the robot could travel. This was done on the Lambda computer by testing the sonar data obtained from an AUTOSCAN which performs a sonar sweep through 180 degrees at three degree increments. This same feature is incorporated into on-board programming by the word CORRIDOR. CORRIDOR takes sonar readings at five degree increments from the center to 70 degrees to the left and then 70 degrees to the right. As each sonar reading is taken, the value is compared to the value in the variable LENGTH and to a value calculated from the angle of the head and the width

of the robot (see Figure 5.2). If at any point the value of the sonar falls within the rectangle shown in Figure 5.2, the scan is stopped and the pan angle of the head is stored in the variable LTHETA or RTHETA. The head then returns to the center position and a value of zero, one, two or three is left on the stack. The binary representations of these values represent physically the orientation of an obstacle in front of the robot. A value of zero (binary 00) indicates that no obstacle is within the path of the robot and it is safe to move forward. A value of one (binary 01) indicates that an obstacle is to the right of the robot and the robot should move to the left to avoid it. A value of two (binary 10) indicates that an obstacle is to the left of the robot and it should be avoided by moving to the right. Finally, a value of three (binary 11) indicates that an obstacle is directly in front of HERMIES and a further decision must be made to decide how to move around the obstacle. The algorithms that make use of this data to avoid obstacles are described later in this chapter.

The word CORRIDOR executes faster than the similar algorithm that runs on the LISP computer. There are three main reasons for this. First, a smaller area is scanned and larger increments are used, therefore fewer sonar readings are needed. This takes advantage of the "blocks world" nature of the robot's environment and the few large obstacles. Second, data is processed as the word is executed, and scanning is stopped once an obstacle is detected, whereas in the LISP algorithm, a full 180 degree scan is always taken. Finally, the data is processed on board the robot and not shipped across the serial link to the external computer which slows execution. The disadvantage of the word CORRIDOR, however, is that it is not effective in the building of a world model where it is useful to have as much data as possible. CORRIDOR only gives information on an obstacle immediately in the robot's path. Used in

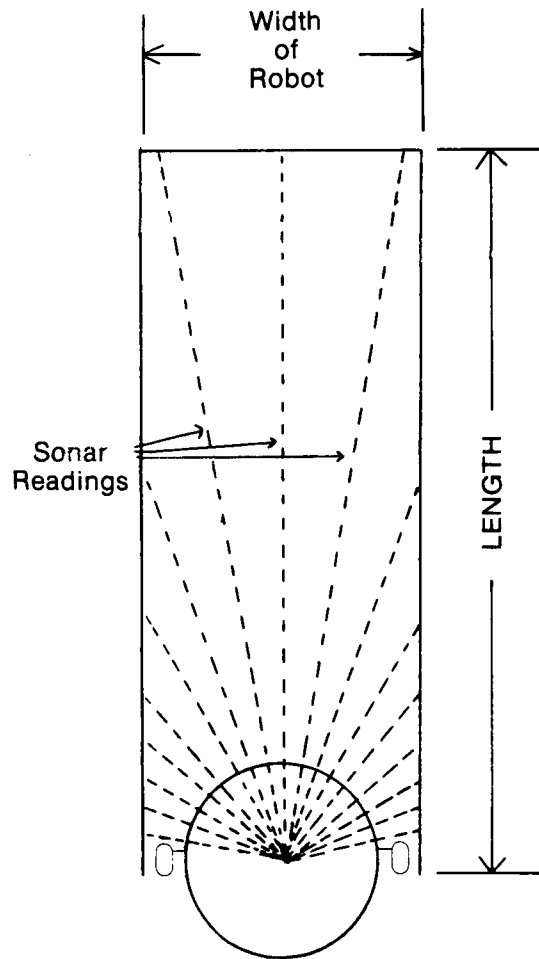


Figure 5.2. Diagram for CORRIDOR Routine.

conjunction with the vision algorithms described next, however, it can improve the robot's ability to maneuver around an obstacle.

5.2 Incorporating Vision with Navigation

Sonar data is adequate for roughly determining the distances to obstacles and walls but leaves some ambiguity as to the position of edges. In contrast, the MicronEye camera can be used to detect the edge of an obstacle but cannot give distance information. The combination of the two devices, however, can provide more useful information about the robot's world than either can alone. This type of sensor integration is one area of study in which CESAR personnel are interested.

To integrate the vision system into navigation routines, it was decided to use the camera to detect the edge of an obstacle that is in the robot's path. From the work done with the camera in a light-table situation, it was clear that the camera derived the most useful information from a high contrast, low noise image. Initial tests with the camera on board the robot demonstrated that this situation could be approximated in the laboratory. The walls of the laboratory were already painted a light color and when the boxes were assembled, they had been painted dark colors. By opening the aperture of the camera to a wide setting, any light shadows or irregularities in the walls were washed out while the dark obstacles would still show up. This provided the high contrast, backlit situation that was desired. Variations in lighting and objects in the background occasionally caused problems, but, in general, this method worked acceptably well. These problems are discussed later in this chapter.

The first edge-detection algorithm developed for the MicronEye sys-

tem was based on an assumption of a noise-free background and a sharp distinction between obstacle and background. This method scanned each column from the object side toward the edge in question, searching for the first column that was completely white or "empty." Experiments with this algorithm quickly exposed its weaknesses. First, the background was seldom completely white or noise free. Even though most of the background was washed out, there were scattered black specks and dark patches from shadows and objects such as electrical outlets. These patches were small but they were enough to disrupt the algorithm. Second, the algorithm located edges wherever an empty column was located, regardless of which side of the obstacle was in view. If the right edge was being sought, the algorithm scanned from left to right. If the left edge was in view, however, the algorithm would place the right edge at the far left side of the image. It was quickly obvious that a more sophisticated algorithm was needed.

The word HISTOGRAM (block 247 of listing) was written in response to this need. HISTOGRAM first obtains an image and expands it into TEMPBUFF. It then sums the number of black or "object" pixels in each column of the image. The sum from each column was stored in the corresponding column in the buffer BIGBUFF. This word provides a simple but effective means to detect the transistion from an area of mostly black (object) to mostly white (background). The words ?LEFTEDGE and ?RIGHTEDGE accomplish the task of determining in which column the transistion occurs. For example, ?RIGHTEDGE initially set the variable RCOL to 127. It then scans the bytes of the first row of BIGBUFF comparing the sums in each byte to the threshold value (normally 32). If the byte represents a background column, the next byte is checked for an object column. If the next byte is an

object column, that column number is stored in the variable RCOL. This method insures that only the transistions from background to object are recorded. ?LEFTEDGE works in the same manner except it scans from left to right in search of the transition.

?LEFTEDGE and ?RIGHTEDGE are used in the words LEFTEDGE and RIGHTEDGE respectively (blocks 253 and 254). LEFTEDGE and RIGHTEDGE are used to search for their respective edges and calculate the angle from the center to that edge. RIGHTEDGE uses ?RIGHTEDGE to determine if there is an edge present in the current field of view. If there is not, the head is moved five degrees to the right and another image is obtained. This process is continued until an edge is found or until the head has moved 90 degrees. If an edge is found, the angle to that edge is calculated using the head angle and the word RRHO. RRHO calculates the angle from the center of the field of view to the edge. If the head has moved 90 degrees, the exposure time is increased, the head is centered and the entire process is repeated. This algorithm will not accept the possibility of the robot being surrounded on three sides by obstacles. It assumes that the edges can be located within the 180 degree plane in front of the robot. LEFTEDGE is the left analogue of RIGHTEDGE. Figure 5.3 shows a robot's eye view of the right edge of one of the boxes in the lab, and Figure 5.4 shows the terminal screen which displays the MicronEye camera's view of the same scene.

The navigation routines discussed so far in this chapter are all brought together in the routine named GO (block 259 in the listings). This routine attempts to move the robot from an initial position to a final goal position. Nothing is known beforehand about the position of any obstacles, and all that is input to the program is the robot's initial position and orientation and the

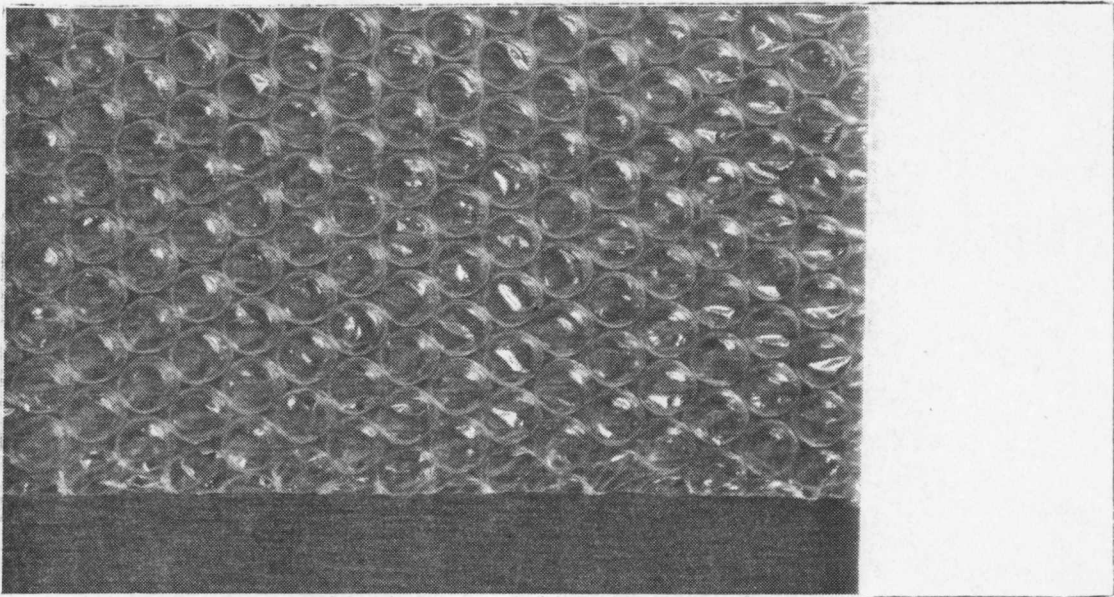


Figure 5.3. Robot's Eye View of Right Edge of Obstacle.

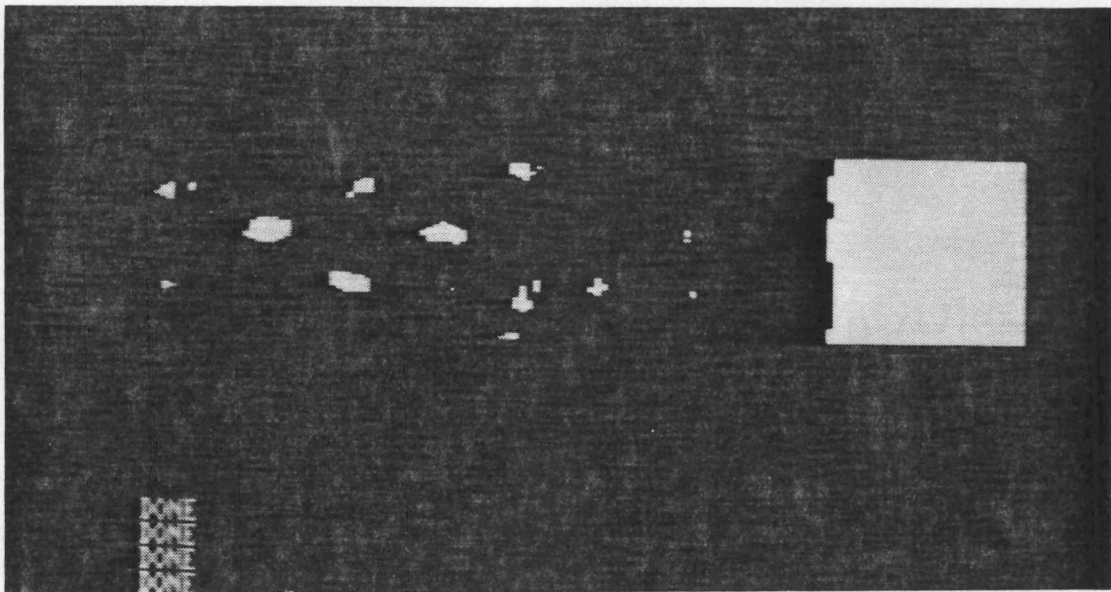


Figure 5.4. MicronEye Image of Right Edge of Obstacle.

coordinates of the goal. GO assumes that the robot's initial position has been set by CURRENT-POSITION and that the floating point coordinates of the goal are currently on the stack. It then sets the goal position and uses ORIENT to orient the robot toward the goal. GO then attempts to move the robot to the goal using the vision system to maneuver around obstacles in the robot's path. The flowchart for the GO routine is given in Figure 5.5, and the components of this flowchart will be discussed in detail.

The routine first sets the coordinates of the goal using the word SETGOAL. Variables are assigned for the goal coordinates because they could be called for several times during the course of moving to the goal. The robot is then oriented toward the goal with ORIENT as described earlier in this chapter. The word CORRIDOR is then used to determine the placement of any obstacles which might be in the robot's path. If no obstacles are between the robot and the goal, the robot moves straight to the goal and the word is finished. If, however, an obstacle is obstructing the robot's path, GO must determine how to maneuver around it. If CORRIDOR returns a value of "1," there is an object to the right of center of the robot and it should pass to the left of the obstacle. This is done by the routine DODGE2LEFT. If CORRIDOR returns a "2," the object is to the left of the robot and the routine DODGE2RIGHT is used. Finally, if CORRIDOR returns a value of "3," the obstacle is directly in front of the robot and the routine ENDAROUND is used to decide which side of the obstacle the robot should maneuver around.

DODGE2LEFT (block 258) turns the head to the point that the CORRIDOR routine detected the obstacle and then uses LEFTEDGE to find the angle to the left edge of the obstacle. The head of the robot is then centered on the edge and a sonar reading is taken to determine the distance

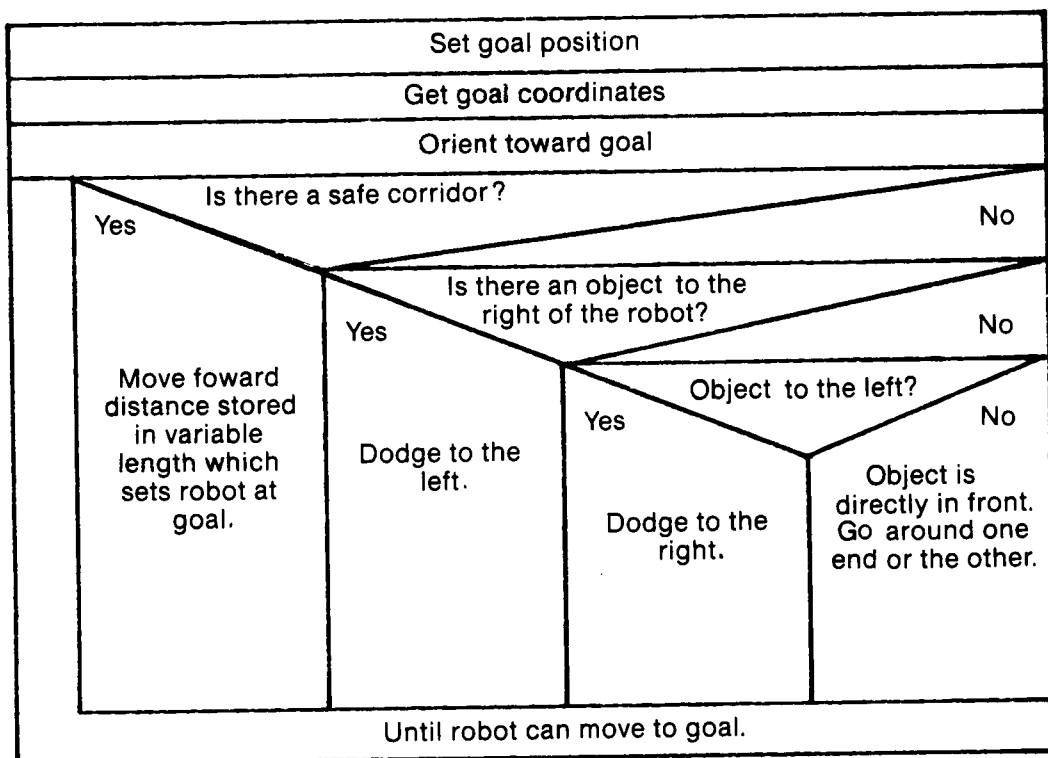


Figure 5.5. Flowchart of GO Routine.

to the edge. The geometry of Figure 5.6 is used to determine the angle to turn and distance to move to reach the next point. A direction flag is set and the robot is turned this angle by GETTURN. Another sonar reading is taken to insure that the robot can move this distance. If not, an ABORT is called and all execution stops. If nothing is in the way, the robot moves forward to the next position, reverses the last turn made, and then reorients toward the goal. DODGE2RIGHT is the right analogue of DODGE2LEFT.

The routine ENDAROUND (block 257) is executed if there is an obstacle directly in front of the robot. In this situation, the robot must make a decision about which direction to go to attempt to move around the obstacle. The first step in making this decision is to locate the edges. ENDAROUND does this with LEFTEDGE and then RIGHTEDGE. The routine MAXSONAR then takes a sonar reading at 10 degrees beyond each edge and compares the two readings. Based on this comparison, a direction flag is set to indicate which side has more open space and therefore which direction to turn. The angle to the edge is stored in the variable THETA. The direction flag and THETA are used by GETTURN to orient the robot directly at the edge. Another sonar reading is then taken to determine the distance to the edge. The distance and angle to the next point are calculated as described for DODGE2LEFT and the robot is moved to this point. Then, again as described above, the robot is reoriented toward the goal in preparation for another CORRIDOR scan.

In a broad sense, the goal of CESAR research is to develop a robot that can autonomously and intelligently accomplish some task. This involves more than navigating through an unknown and possibly hostile environment. Once the robot reaches a given location, it will have to perform some

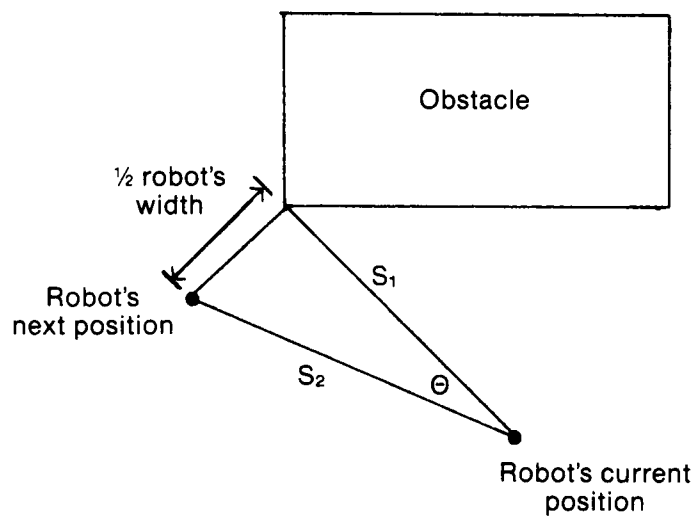


Figure 5.6. Geometry for Movement in DODGE2LEFT.

function—such as turning a valve, moving a load, or flipping a switch. As a first step toward this, HERMIES was equipped with two Heathkit Hero arms for manipulation. These arms are limited in the amount of weight they can pick up and have been used only for simulation purposes to date.

The FLIPSWITCH routine (block 265) was written to explore the possibility of using the MicronEye camera to recognize a marked location at which some task is to be accomplished. A typical scenario would have the robot navigate to some location using a routine such as the GO routine. At this point, the robot may not be at the exact location expected due to positioning error or may be oriented oddly. The robot would then search for a marker of some kind and upon finding it, would reposition itself to a known location. Assured of being at the right place, the robot would use its manipulators to accomplish the assigned task. The FLIPSWITCH routine is the implementation of one algorithm for finding and repositioning on a marked location.

The marker that the FLIPSWITCH routine searches for takes the form of two vertical, black lines, $\frac{3}{8}$ inch wide, and eight inches apart, drawn on a sheet of white paper. The choice of two lines was made originally to take advantage of the parallax principle. Theoretically, the apparent angle between the two lines should become less as the viewer moves away from the perpendicular to the plane of the lines, keeping a constant distance. It was expected that, by obtaining the distance to the lines from a sonar reading, this principle could be used to determine the robot's angle from the plane of the wall when oriented directly toward the lines. Experimentally, however, this algorithm did not work as well as expected. Tests with the MicronEye camera and the lines showed that the difference in apparent angles between extremes of orientation was typically only a few degrees. The resolution of the overall robotic system

was not fine enough to allow accurate positioning of the robot based on this information. Another limiting factor was the narrow field of view of the Micron-Eye camera itself. The field of view of the camera was experimentally determined to be 9.5 degrees wide by 1.8 degrees high. This field of view was adequate for selectively viewing the edges of boxes but prevented the viewing of widely separated lines.

The two lines were useful, however, as a unique marker. If only one line were used, other features in the lab, such as a conduit running up the wall, could be confused for the line. With two lines, when one was found, the other could be searched for and the position correctly identified. This was the approach taken in the FLIPSWITCH routine.

The first step that FLIPSWITCH takes is to face the end wall. This would have the robot directly facing the lines if it were in the correct position. The routine FINDLINES (block 263) is then called to search for the lines. An integral part of FINDLINES is the routine COUNTLINES (block 249). COUNTLINES obtains an image, takes a histogram of it and counts the number of light to dark transistions. This number, representing the number of lines, is left on the stack. FINDLINES first scans to the left for 45 degrees. If no lines are found, the head is returned to the center and a 45-degree scan to the right is performed. If at any time a line is detected, the scanning process stops and the head is left at that position. Once a line has been found, FLIPSWITCH positions the head so that both lines are in the field of view. If there are not two lines, the FLIPSWITCH routine is aborted with the message "NOT GOAL." If both lines are present, however, the angle from the center of the robot to the center of the lines is calculated, and the robot is turned to face the lines.

Figure 5.7 shows a robot's eye view of the line and 5.8 shows the MicronEye image of the lines.

In order to accomplish a repeatable task, it was necessary to position the robot at the same location and orientation each time the task was executed. The location and orientation chosen was directly in front of the lines and as close as possible to perpendicular to the wall. The routine GETPERP (block 264) was written to accomplish this. GETPERP first takes three sonar readings along the wall to set up the triangles shown in Figure 5.9. The first reading is taken at 20 degrees to the left of center, the second at center position, and the third at 20 degrees to the right of center. The small triangle in Figure 5.9 represents the robot facing the wall at some unknown angle. The line segments, S1, S2, and S3, represent the three sonar readings. After the sonar readings have been obtained, GETPERP uses simple geometry, including the laws of sines and cosines (see LOCOS and LAWOSIN in the listings), to compute successively θ_3 , θ_4 , θ_5 and S4. The angles θ_1 and θ_2 are given and are equal to 20 degrees. The key calculations for GETPERP are the angle θ_5 and the distance S4. Once these calculations are made, the robot is turned the angle θ_5 , moved the distance S4, and turned back 90 degrees. This series of movements puts the robot directly in front of the lines and perpendicular to the wall.

The robot is now in a known location with a known orientation. This method could be used to reinitialize the robot if positional error occurred while the robot was moving. With the lines at a previously known location, the robot's current position could be updated to prevent a collective buildup of error. In the FLIPSWITCH word, the robot is moved forward until it is a point two feet from the wall and the word WAVEARM is called. WAVEARM is a

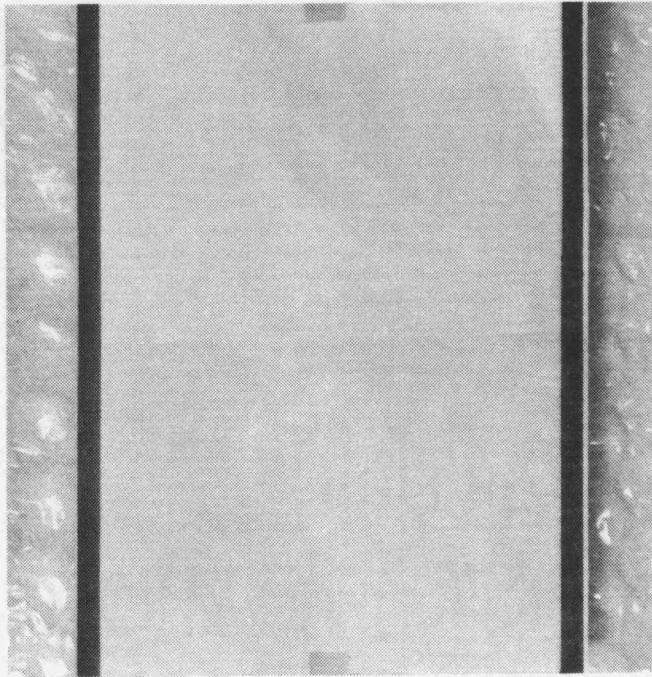


Figure 5.7. Lines Used as a Marker.

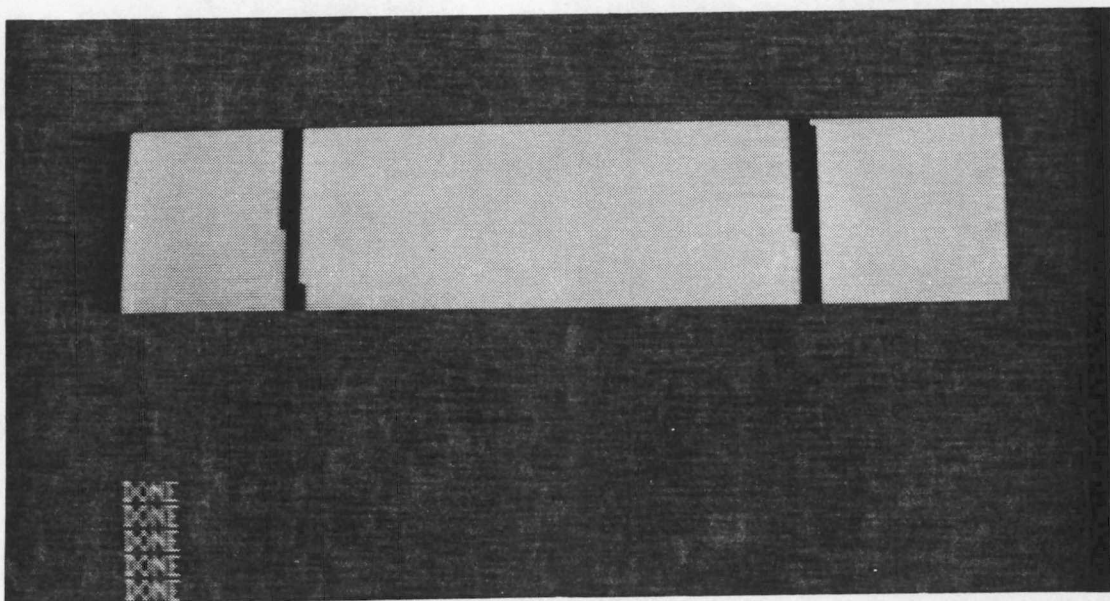


Figure 5.8. MicronEye Image of Lines.

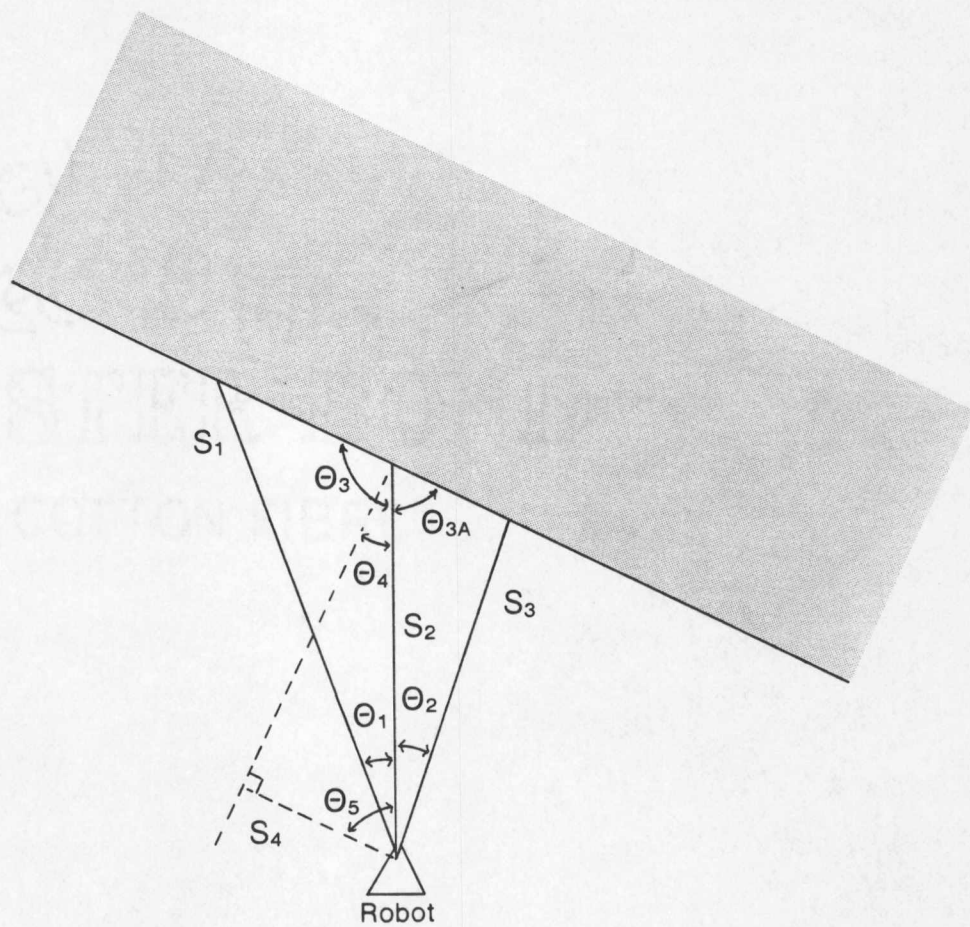


Figure 5.9. Diagram for GETPERP Routine.

simple routine that moves the arm out and then back in, but it simulates the execution of a manipulation task once the robot has reached its destination. It is expected that in the future a mock control panel will be used to experiment with manipulation tasks for HERMIES.

All of the routines described in this thesis have been executed only in the CESAR Laboratory. In this situation the walls are light in color and the obstacles are dark in order to simulate a backlit situation. Only a few obstacles are allowed and they are of a regular shape. Even in this controlled situation, some problems have been experienced which affect the performance of the MicronEye system. One end of the laboratory is used for navigation experiments with the robot, but the the other end contains computers, desks, workbenches and people working on various projects. When the robot is facing away from this end, as is usually the case, there is no problem. However, when the robot is facing this end, the cluttered background sometimes confuses the vision algorithms and prevents the accurate determination of obstacle edges. There are also features in the navigation area which add background clutter, such as the door to the laboratory, which is darker than the walls. The obstacles normally can be set up in such a manner that the view of the door is obscured.

Another problem arises from the use of "bubble wrap" packaging material to cover the surfaces of the obstacles. The obstacles were covered with this packing material to produce better sonar reflections. Previously, the sonar pulses would, at certain angles, glance off of the obstacles instead of being reflected back to the transducer, rendering part of the obstacles invisible to the sonar. The rough texture of the packing material eliminated this problem but created new problems for the vision system. The "bubbles" would reflect

the overhead lights toward the camera, producing white dots in the black obstacles. When the majority of the surface being viewed was still black, the vision routines were not affected. Sometimes, however, certain combinations of obstacle orientation and overhead lighting produced a large amount of reflected light which was interpreted as a false edge. The position of the boxes in the room also produced different amounts of reflectance. Despite these problems, successful navigation experiments were executed in the laboratory and much was learned about the possibilities of using vision for mobile robots. These experiments demonstrated that vision is a viable sensor for navigation, but also that much more sophisticated algorithms are needed for real world situations.

5.3 Other Vision Primitives

The navigation routines described above have immediate applications in the experiments being carried out with the robot, HERMIES-II. Other primitives have been written which may have applications in experiments such as manipulation algorithms. The first of these routines, BOX-ORIENT (block 261), tries to determine the orientation of one of the large boxes in the robot's world. Figure 5.10 shows the two extremes of orientation that a box can have. As a box is rotated through 90 degrees, the angle of orientation will appear to increase to 45 degrees and then decrease to zero. For this reason, BOX-ORIENT will always return a value between zero and 45 degrees. BOX-ORIENT first moves the robot to a set distance of two feet from the box and then searches for the right edge. A sonar reading is taken at the right edge and this is line segment "C." Line segment "B" is given and equals two feet. The angle to the edge is designated as "a." The law of sines is used to calculate

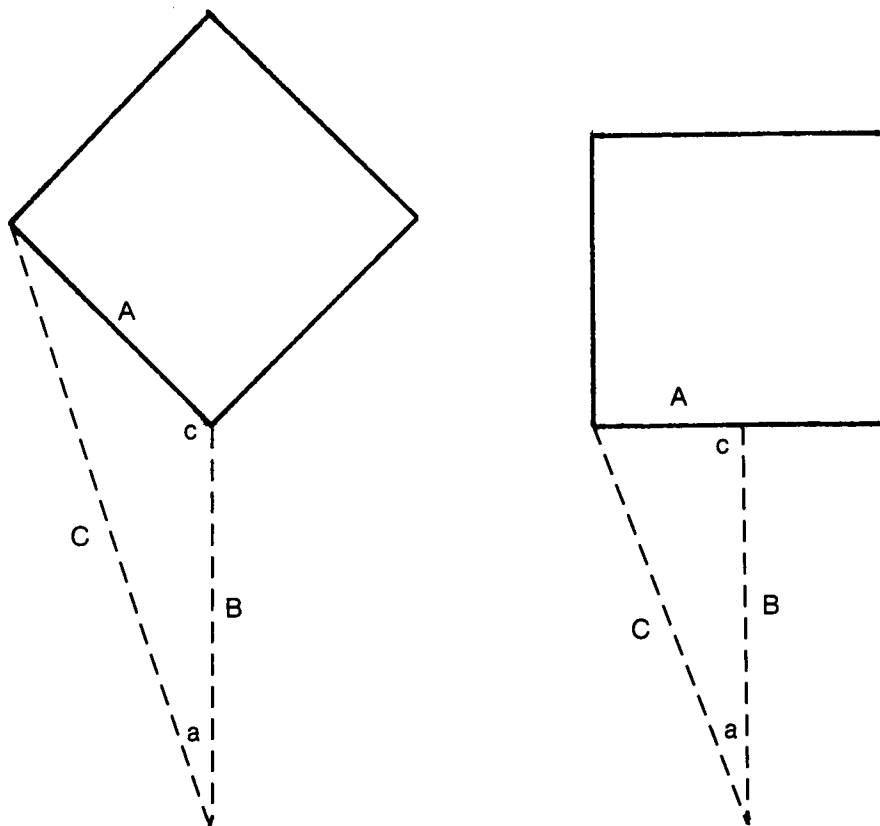


Figure 5.10. Diagram for BOX-ORIENT Routine.

"A" and then the law of cosines is used to calculate the angle "c." The absolute value of this angle subtracted from 90 degrees gives the angle of orientation. This routine has been tested and found to have an accuracy of plus or minus five degrees. The error results in part from the shape of the sonar beam. The routine assumes the sonar reading is a line drawn from the sensor to the exact edge of the box. In reality, the sonar beam is conical in shape and, therefore, it is not possible to determine the precise point of contact. Another cause of error is in the positioning of the head. Due to the integer nature of FORTH routines, angular accuracy is limited to plus or minus one degree.

The other vision primitives were developed for use with a light table. When the initial work to interface the MicronEye system to FORTH was finished, the system was being used with an IBM XT computer and a light table. The camera was mounted on a ring stand and directed down toward the light table. The first high-level image-processing words, therefore, were written for a situation such as using the robot's manipulators to pick up an object from the light table. These routines were a smoothing routine to decrease spurious effects such as noise, an edge detection routine, and a routine to determine the size and centroid of an object. This is not an exhaustive list of image processing routines, and has no immediate applications to the work done in this thesis other than to explore possible uses for the MicronEye system. These routines would be useful, however, to manipulation experiments carried out at some point in the future.

The smoothing routine was based on the neighborhood averaging technique described in section 4.3 of Gonzalez and Wintz [7]. This is a simple and straightforward method which averages the grey level values of a 3×3 image region centered on a pixel. The grey level of the center pixel is then

replaced with the average value. This method was adapted for a binary image in the word NEIGHBOR (block 242). In NEIGHBOR, the number of black or object pixels is summed and this value is compared to a given threshold value. If the number of object pixels is greater than or equal to the threshold value, the center pixel is set as an object pixel. The threshold value can be set to a value between five and nine.

The edge detection routine also comes from reference [7] and is based on the use of Sobel operators for edge detection. As described in section 7.1 of [7] for a given 3×3 image region

a	b	c
d	e	f
g	h	i

with G_x and G_y defined as

$$G_x = (g + 2h + i) - (a + 2b + c) \quad 5.1$$

$$G_y = (c + 2f + i) - (a + 2d + g) \quad 5.2$$

the gradient at point e is defined as

$$G = (G_x^2 + G_y^2)^{1/2} \quad 5.3$$

or alternately as

$$G = |G_x| + |G_y| \quad 5.4$$

This algorithm can be implemented on a computer using the templates shown in Figure 5.11 which are often referred to as Sobel operators. Using

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Figure 5.11. Sobel Templates.

these templates, the gradient can then be written and computed in vector notation by the following equations.

$$G_x = \mathbf{w}'_1 \mathbf{x} \quad 5.5$$

$$G_y = \mathbf{w}'_2 \mathbf{x} \quad 5.6$$

where $\mathbf{x}$ is the image region and $\mathbf{w}_1$ and $\mathbf{w}_2$ are the Sobel templates. The gradient formula is then given by

$$\mathbf{G} = \mathbf{w}'_1 \mathbf{x} + \mathbf{w}'_2 \mathbf{x} \quad 5.7$$

The gradient at each point of the binary image of the MicronEye system is computed in this manner by the FORTH word SOBEL (block 246). The result is scaled back to a binary image and stored in the buffer BIGBUFF. The word GETEDGE obtains an image, displays it, performs the edge detection and then redisplay it. Figures 5.12 and 5.13 show an image before and after edge detection.

The last primitives are to determine the size and centroid of an object in the image and are based on the 0-th and 1-st order moments of an image. As described in reference [8], the 0-th order moment of an image represents the “weight” or area of the object in question and is simply a summation of the object pixels. The 0-th moment is given by the equation

$$M_0 = \sum_{x=0}^{63} \sum_{y=0}^{127} f(x,y) \quad 5.8$$

This equation is implemented in the word 0MOM (block 266) in which the final area is contained in the variable M0.

The first-order moment produces the coordinates of the centroid of

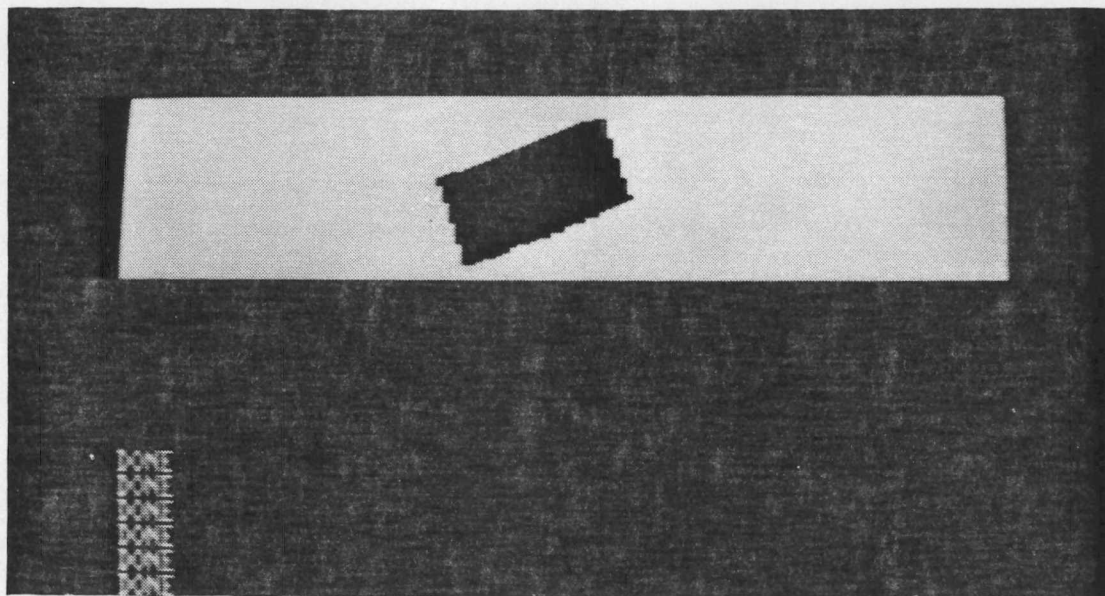


Figure 5.12. MicronEye Image of Block on Light Table.

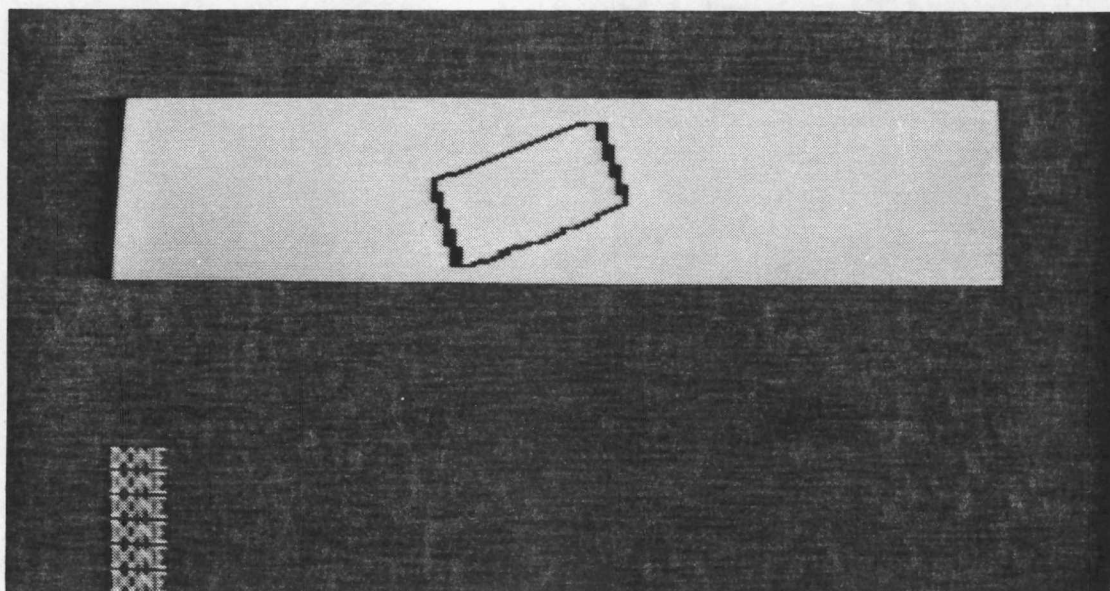


Figure 5.13. Image of Block After Sobel Edge Detection.

the object in question. The equations for the coordinates are

$$MX1 = \sum_{x=0}^{63} \sum_{y=0}^{127} xf(x,y)/M0 \quad 5.9$$

and

$$MY1 = \sum_{x=0}^{63} \sum_{y=0}^{127} yf(x,y)/M0 \quad 5.10$$

These equations are implemented in the word 1MOM and the coordinates are left in the variables MX1 and MY1. Both 0-th and 1-st order moment algorithms are designed for use on binary images, which made them easy to implement for the MicronEye system. These two primitives, together with GETEDGE and NEIGHBOR, would be useful in manipulation routines where it is important to know the size, shape and position of an object so that a manipulator can be directed to pick it up.

CHAPTER VI

FUTURE VISION SYSTEMS

In September, 1985, CESAR received shipment of an IRI P256 vision system from International Robomation/Intelligence. This is an advanced vision system that provides images with a resolution of 256×256 pixels and 256 grey levels. This system is intended for use with future versions of HERMIES and is presently being used in a stand-alone mode.

The hardware of the IRI system consists of the host computer (SPC), video digitizer (VDG), camera interface (CIF), co-processor (COP), two disk drives, and a floppy disk controller (FDI) that formats diskettes in quad density (640k-byte per disk) [9]. The host computer is based on a 12 MHz MC68000 microprocessor, with 64k-bytes of static RAM and 32k-bytes of EPROM, that contains the monitor and system programs. This computer controls all peripheral devices, executes system and I/O routines, executes applications programs and co-processor functions if the co-processor is not present in the system. The video digitizer acquires and digitizes an image from the camera and stores the image in one of the frame buffers. There are four frame buffers, each capable of holding an image of 256×256 bytes. Images can be moved from buffer to buffer and it is on these areas of memory that most processing routines work. Preprocessing hardware is provided with the IRI system to do point transformations and histogram calculations. This allows the user to perform functions such as contrast enhancement, range expansion, thresholding, and grey level inversion. All operations performed by the preprocessing hardware are performed in a maximum of 20 milliseconds. The co-processor is a systolic array processor operating on the pixels of a frame

in the Single Instruction-Multiple Data (SIMD) mode. The internal clock rate is 20 MHz which results in a performance of up to 40 million operations per second (MOPS). Many functions such as convolutions and gradients are microprogrammed into the co-processor and can be executed at a rate of 30 or more frames per second [9].

The operating system of the IRI host computer is based on the polyFORTH operating system and handles functions such as I/O and editing of applications programs. IRI has extended the basic polyFORTH system into what they refer to as the Vision Command Language of VCL. This includes commands that handle functions specific to the vision system. VCL commands can be executed interactively and include commands to input and digitize images, move images among buffers, perform preprocessing functions and array processor functions. The list of commands is extensive and covers the features of the vision system very well. A full description of the commands can be found in the IRI P256 Programmer's Manual [10].

The IRI vision system is so much more advanced in features and capabilities than the MicronEye vision system that it is difficult to compare the two. The most impressive difference is the speed of execution of the IRI system. The IRI system provides functions similar to the SMOOTH and GET-EDGE routines described in the last chapter. The SMOOTH routine for the MicronEye system took 29.45 seconds to execute on the 128×64 pixel image. The most similar routine in the IRI system to the neighborhood averaging algorithm of SMOOTH was the MED filter. This filter selected the median grey scale value for each 3×3 neighborhood and replaced the center pixel with this value. This routine executed in 20 seconds, which is 1/3 less time, while operating on an image that is eight times the size of the MicronEye image.

The Sobel edge detection routine, GETEDGE, took 12.5 seconds to execute. The Sobel edge detection routine in the IRI system is microcoded into the co-processor and executes in 1/30 of a second. The co-processor also contains a convolution function that can execute a low-pass filter which achieves the same smoothing effect as neighborhood averaging. This routine executes in 38 milliseconds. The time estimates were obtained from reference [11].

These two vision systems are radically different in features and capabilities but each serves its unique purpose well. While the IRI system is a flexible, advanced vision system for serious image processing applications, the MicronEye system serves well as a low-cost system for users just entering the field of vision processing. It was provided with the original HERMIES robot as just such an introductory system. The decision by CESAR personnel to purchase the IRI system was based on an understanding of the importance that vision will play in the sensor systems of future robotic systems. In the short term, the IRI system will be used to explore the possibilities connected with using vision as part of an overall sensor system. It is expected that the IRI camera will be mounted on HERMIES-II part of the time to experiment with vision in a controlled but real world situation. This will necessitate the tethering of the robot to the IRI system to transmit pictures and receive commands but should provide valuable information for the integration of vision into future versions of HERMIES.

CHAPTER VII

CONCLUSIONS

A MicronEye OpticRAM vision system was interfaced to the mobile robot HERMIES-II. The integration of the MicronEye software to the robot's software involved rewriting and condensing the general purpose assembly language subroutines to meet the specific needs of the proposed application. The final subroutines were 1/3 of their original size. These subroutines were interfaced to the FORTH programming environment to be called by high level FORTH programs. Once the basics of the system were in place, high level FORTH routines, called "words," were written to perform preprocessing on images, explore the robot's world with sonar, and detect features of objects using the vision system. Words were written to detect the edges of obstacles in the robot's path and to detect a pair of vertical black lines used to mark a location on the wall of the laboratory. The navigation routine GO was written to demonstrate the effectiveness of using the MicronEye vision system to maneuver around obstacles to a preselected goal. Another routine, FLIP-SWITCH, was written to seek out the marked location on the wall and move the robot directly in front of the lines and perpendicular to the wall.

One area of CESAR research is sensor fusion for mobile robots. This is the integration of data from many sensors to provide relevant information about the robot's environment. The object of this thesis research was to use existing hardware to provide the HERMIES-II robot with more information as a first step toward sensor fusion. Previously, only sonar, which provides distance feedback, was used as a sensor for the robot. There were disadvantages to using sonar only in that it left ambiguity as to the exact position of edges

due to the conical shape of the sonar beam. The vision system was used in an attempt to locate the edges of the obstacle more accurately.

This research demonstrated that the low cost MicronEye system could be used to locate edges and, when used in conjunction with sonar, lead to successful navigation routines. Also demonstrated was the ability to identify a location designated by a unique marker.

Limitations of the MicronEye system were brought out by this research. The OpticRam camera responds best to backlit situations and is susceptible to variations in available lighting. The low resolution of the camera prevents the processing of detailed images. Finally, the length of time necessary to execute processing routines detracts from its viability as a real time system.

Despite its limitations, the MicronEye vision system, coupled with the sonar sensor, has been shown to provide better quality information than the sonar alone. It has also been shown that this information can be provided in a comparable and sometimes shorter length of time. In general, the addition of the vision system has improved the sensor capabilities of the HERMIES robot. This will, in turn, lead to more effective navigation algorithms such as those running on the LISP computer which should be able to plan a more efficient path if the edges of obstacles are more clearly defined. Finally, this work demonstrated that the potential exists to process more data on board the robot than has been attempted in the past.

It is suggested that the next research step be to integrate the vision primitives described in this thesis into presently running path planning algorithms to further test the effectiveness of using vision as a sensor. Preliminary work with the IRI system has shown it to be a powerful and flexible vision tool.

It is suggested that more sophisticated algorithms be developed to make use of this system in anticipation of its integration onto future versions of HERMIES.

LIST OF REFERENCES

LIST OF REFERENCES

1. Weisbin, C.R. et al., "Strategy Planning by an Intelligent Machine," First World Conference on Robotics Research: The Next Five Years and Beyond, MS84-501, Bethlehem, PA (August 14-16, 1984).
2. Weisbin, C.R. et al., "HERMIES-I: A Mobile Robot for Navigation and Manipulation Experiments," Robots 9 Conference, Detroit, MI (June 3-6, 1985).
3. Van De Walker, Ray, Elizabeth D. Rather et al., *polyFORTH II Reference Manual*, FORTH, Inc., Hermosa Beach, CA (1984).
4. Brodie, Leo, *Starting FORTH*, Prentice-Hall, (1981).
5. *MicronEye Operator's Manual*, Micron Technology Inc., Boise, ID (1984).
6. *iAPX 88 Book*, Intel Corporation, Santa Clara, CA (July 1981).
7. Gonzalez, R.C., and Paul Wintz, *Digital Image Processing*, Addison-Wesley Publishing Co. (1977).
8. Rosenfeld, A. and A.C. Kak, *Digital Picture Processing*, Academic Press (1982).
9. *P256 Technical Manual*, International Robomation/Intelligence, Carlsbad, CA (1985).
10. *IRI P256 Vision System Programmer's Manual*, Version 2.1A, International Robomation/Intelligence, Carlsbad, CA (1985).
11. *A Primer on Machine Vision and the IRI Vision System*, International Robomation/Intelligence, Carlsbad, CA (1985).

APPENDICES

APPENDIX A

Annotated FORTH Listing

558

This screen contains the code for the assembly language routines that drive the Microneye camera. They are approximately 400 bytes long. They are loaded directly from this block to memory at location 1,4400 hex by the word DOWNCODE. The first 4400 hex bytes are buffers for the storage and manipulation of images taken in by the camera.

237

```

0 (
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

559

Microneye utilities load screen
WIDTH is set to 31

WORKMAP, BIGBUFF, AND TEMPBUFF are buffers for the storage and manipulation of images. EBASE is the beginning of assembly language code and EXPOSTIME contains the time in msec. that an image is exposed for.

DOWNCODE load the assembly routines from block 243 to memory.

GRAFINIT initializes the graphics screen.

PCALC takes a picture and loads it into WORKMAP.

MOVESCR moves the image from WORKMAP to the screen of the PC.

CLEARX initializes WORKMAP to a specific value.

238

```

0 ( MICRONEYE UTILITIES )
1 : HELP 239 HELPS ; HELP
2 MATH LOAD HEX 1F1F WIDTH : DECIMAL GRAPHICS LOAD
3 HERE U. HEX
4 1,4400 2CONSTANT EBASE 1,0400 2CONSTANT BIGBUFF
5 1,0000 2CONSTANT WORKMAP 1,2400 2CONSTANT TEMPBUFF
6 1,441A 2CONSTANT EXPOSTIME
7
8 : DOWNCODE OED BLOCK 0 EBASE 400 EMOVE ;
9 DOWNCODE
10 : GRAFINIT GINIT HR DARK *ADAPTER OR #LINES ;
11 CODE PCALC 9A C, 442C , 1000 , NEXT
12 CODE MOVESCR 9A C, 4450 , 1000 , NEXT
13 CODE MCLEAR 9A C, 44F5 , 1000 , NEXT
14 : FO PCALC MOVESCR ;
15 DECIMAL 240 265 THRU ( MICRONEYE APP. )

```

560

239

561

IMAGEXP takes a byte and a double-length address from the stack and expands the byte so that each bit is placed in a consecutive address starting at the double address.

IMAGEXPAND takes a byte from WORKMAP and calculates the necessary address in TEMPBUFF for the expanded bits. All of 1k WORKMAP is expanded to 8k TEMPBUFF.

IMASCONT is the opposite of IMAGEXP. It takes 8 consecutive bytes beginning at double-address and compresses them into one byte left on the stack.

240

```

0 ( IMAGE-EXPANSION ) HEX
1
2 CODE IMAGEXP ( n da - ) 0 POP 2 POP )SEG CALL 1 DS SSG
3     2 DS LSG 0 POP 1 PUSH 8 # 1 MOV 80 # 2 MOV
4     BEGIN 0 PUSH 2 0 AND 1 PUSH 1 DEC 0 SHR V 1 POP
5     2 SHR 0 W ) MOV B W INC 0 POP LOOP
6     1 POP 1 DS LSG NEXT
7 DECIMAL
8 : IMAGEXPAND 16 0 DO 64 0 DO 1 16 * J + WORKMAP ROT M+ EC!
9     TEMPBUFF 1 16 * J + 8 * M+ IMAGEXP LOOP LOOP ;
10
11 CODE IMASCONT ( da - n ) 0 POP 2 POP )SEG CALL 1 DS SSG
12     2 DS LSG 1 PUSH 0 # 2 MOV 8 # 1 MOV BEGIN 2 SHL
13     W ) 0 MOV B 01 # 0 AND 0 2 OR W INC LOOP
14     1 POP 1 DS LSG 2 PUSH NEXT
15

```

562

IMAGE-CONTRACT compresses the data in the 8 kbytes TEMPBUFF or BIGBUFF into the 1 Kbytes WORKMAP for display.

TEMP>BIS and BIG>TEMP transfer an image from one buffer to another.

241

```

0 ( IMAGE CONTRACTION ) DECIMAL
1
2 : IMAGE-CONTRACT ( TEMPBUFF or BIGBUFF - ) 16 0 DO 64 0 DO
3     2DUP 1 16 * J + 8 * M+ IMASCONT
4     WORKMAP 1 16 * J + M+ EC! LOOP LOOP 2DROP ;
5
6 VARIABLE TEMP
7 : TEMP>BIS TEMPBUFF BIGBUFF 8192 EMOVE ;
8 : BIG>TEMP BIGBUFF TEMPBUFF 8192 EMOVE ;
9
10
11
12
13
14
15

```

563

GETTEMP gets the nth byte from TEMPBUFF and adds it to TEMP.

SUMPPIXELS sums the J-1, J, and J+1 bytes of the Ith row of TEMPBUFF and stores the value in the variable TEMP.

NEIGHBOR is a routine that performs image smoothing on an image by using the technique of neighborhood averaging. A 3x3 array of pixels surrounding each pixel is considered. If the majority of these are 1's the pixel in question is set to 1. If not, it is set to 0. The majority threshold can be changed by dividing by a different value. (from 5 to 9)

SMOOTH does everything necessary to get an image, smooth it with NEIGHBOR and redisplay it.

242

```

0 ( IMAGE MANIPULATION CONT. ) DECIMAL
1
2 : GETTEMP ( n - ) TEMPBUFF ROT M+ EC! TEMP +! ;
3
4 : SUMPPIXELS ( I,J - ) 2DUP 1- + GETTEMP 2DUP + GETTEMP
5     1+ + GETTEMP ;
6
7 : NEIGHBOR TEMP>BIS 127 1 DO 63 1 DO 0 TEMP ! 1 128 * DUP
8     128 - J SUMPPIXELS
9     DUP J SUMPPIXELS
10    128 + J SUMPPIXELS
11    TEMP @ 6 / 1 128 * J + BIGBUFF ROT M+ EC! LOOP LOOP ;
12
13 : SMOOTH PCALC MOVESCR IMAGEXPAND NEIGHBOR
14    BIGBUFF IMAGE-CONTRACT MOVESCR ;
15

```

564

These words are the code words used to optimize the object location routines. ULC finds the upper left corner and LRC finds the upper right corner. Both words start with the buffer address (TEMPBUFF or BISBUFF) and leave the offset into the buffer of the located corner on the stack.

ULC searches from left to right and from top to bottom.
LRC searches from top to bottom and from right to left.

565

LRC finds the lower right corner and searches from right to left and from bottom to top.

LLC finds the lower left corner and searches from bottom to top and from left to right.

S(I,J) returns the byte from TEMPBUFF corresponding to the I J indexes left on the stack. The NOT is to reverse the logic for an backlit image.

566

These variables give the corners of a viewed object.

COLIMITS and ROWLIMITS give the limits of a located object in the proper form for DO LOOPS.

OBJLOCATE uses ULC URC LRC and LLC to locate the corners of a viewed object and stores the proper row and column coordinates in the proper variable.

SDX is used in the word SOBEL. It is an approximation of the X derivative of the image.

243

```
0 ( FINDING CORNERS IN ASSEMBLY ) DECIMAL
1
2 CODE ULC ( TEMPBUFF - n ) 0 POP 2 POP >SEG CALL 1 DS SSG
3     2 DS LSG 1 PUSH 8192 # 1 MOV BEGIN W ) 0 MOV B
4     W INC 0 #B 0 CMP -LOOP W DEC 1 POP 1 DS LSG
5     W PUSH NEXT
6
7 CODE URC ( TEMPBUFF - n ) 0 POP 2 POP >SEG CALL 1 DS SSG
8     2 DS LSG 1 PUSH 127 # 2 MOV 128 # 1 MOV BEGIN
9     1 PUSH 64 # 1 MOV BEGIN 64 # 0 MOV 1 0 SUB 1 PUSH
10    128 # 1 MOV 1 MUL B 1 POP 2 0 ADD 0 W MOV
11    W ) 0 MOV B 0 #B 0 CMP -LOOP 2 DEC 1 POP 0 #B 0 CMP
12    -LOOP 0 POP 0 DS LSG W PUSH NEXT
13
14
15
```

244

```
0 ( CORNERS CONT. ) DECIMAL
1
2 CODE LRC ( TEMPBUFF - n ) 0 POP 2 POP >SEG CALL 1 DS SSG
3     2 DS LSG 1 PUSH 8192 # 1 MOV BEGIN 1 W MOV
4     W DEC W ) 0 MOV B 0 #B 0 CMP -LOOP 1 POP 1 DS LSG
5     W PUSH NEXT
6
7 CODE LLC ( TEMPBUFF - n ) 0 POP 2 POP >SEG CALL 1 DS SSG
8     2 DS LSG 1 PUSH 0 # 2 MOV 128 # 1 MOV BEGIN
9     1 PUSH 64 # 1 MOV BEGIN 1 0 MOV 0 DEC 1 PUSH
10    128 # 1 MOV 1 MUL B 1 POP 2 0 ADD 0 W MOV
11    W ) 0 MOV B 0 #B 0 CMP -LOOP 2 INC 1 POP 0 #B 0 CMP
12    -LOOP 0 POP 0 DS LSG W PUSH NEXT
13
14 : S(I,J) ( I,J - n ) SWAP 128 # + TEMPBUFF ROT #+ ED# NOT ;
15
```

245

```
0 ( LOCATE OBJECT AND SOBEL OPERATORS ) DECIMAL
1 VARIABLE UPROW VARIABLE UPCOL VARIABLE RROW VARIABLE RCOL
2 VARIABLE DOWNROW VARIABLE DOWNCOL VARIABLE LROW VARIABLE LCOL
3
4 : COLIMITS RCOL @ 1+ LCOL @ ; ROWLIMITS DOWNROW @ 1+ UPROW @ ;
5
6 : OBJLOCATE IMAGEXPAND
7     TEMPBUFF ULC 128 /MOD UPROW ! UPCOL !
8     TEMPBUFF URC 128 /MOD RROW ! RCOL !
9     TEMPBUFF LRC 128 /MOD DOWNROW ! DOWNCOL !
10    TEMPBUFF LLC 128 /MOD LROW ! LCOL ! ;
11
12 : SDX ( I,J - n ) 2DUP 1+ S(I,J) 2* )N 2DUP 1- S(I,J) 2* )N
13     F- 2DUP 1+ SWAP 1- SWAP S(I,J) )N F+ 2DUP 1- SWAP 1-
14     SWAP S(I,J) )N F- 2DUP 1+ SWAP 1+ SWAP S(I,J) )N F+
15     1- SWAP 1+ SWAP S(I,J) )N F- FABS N ) ;
```

567

SDY is used in the word SOBEL. It is an approximation of the Y derivative of the image.

SOBEL computes the SOBEL gradient of the image in TEMPBUFF for edge detection. It computes the X-derivative and the Y derivative and sums them. Dividing by 4 scales the image back to a binary image.

GETEDGE combines all words necessary to get an image, perform edge detection and display the new image.

246

```
0 ( MOMENTS OF IMAGES CONT. )    DECIMAL
1
2 : SDY ( I,J - n ) 2DUP SWAP 1- SWAP S1(I,J) 2* )N 2DUP SWAP 1+
3   SWAP S1(I,J) 2* )N F- 2DUP 1- SWAP 1- SWAP S1(I,J) )N F+
4   2DUP 1- SWAP 1+ SWAP S1(I,J) )N F- 2DUP 1+ SWAP 1-
5   S1(I,J) )N F+ 1+ SWAP 1+ SWAP S1(I,J) )N F- FABS N) ;
6
7 : SOBEL ( OBJ. LOCATE IMAGE X PAND ) COLIMITS DO ROWLIMITS DO
8   I J SDX I J SDY + 4 / NOT I 128 * J + BIGBUFF ROT
9   M+ EC! LOOP LOOP ;
10
11 : GETEDGE OBJ. LOCATE IMAGE X PAND TEMP BIG SOBEL BIGBUFF
12   IMAGE-CONTRACT MOVESCR ;
13
14
15
```

568

HISTOGRAM gets an image, displays and expands it into TEMPBUFF. Then the pixels in each column of TEMPBUFF are added and stored in that column's position in the 1st row of BIGBUFF.

?LEFTEDGE takes a HISTOGRAM and then scans the 1st row of BIGBUFF from left to right looking for transitions from white to dark. The last transition is stored in LCOL.

?RIGHTEDGE takes a HISTOGRAM and scans the 1st row of BIGBUFF from right to left looking for transition from white to dark. The last transition is stored in RCOL.

247

```
0 ( HISTOGRAM THRESHOLDING FOR EDGE DETECTION )    DECIMAL
1
2 : HISTOGRAM PCALC MOVESCR IMAGE X PAND
3   128 0 DO 0 64 0 DO
4     TEMPBUFF I 128 * J + M+ EC! +
5     LOOP BIGBUFF I M+ EC! LOOP ;
6
7 : ?LEFTEDGE HISTOGRAM 0 LCOL !
8   127 0 DO
9     BIGBUFF I M+ EC! 32 ) IF BIGBUFF I 1+ M+ EC! 32 (
10      IF I 1+ LCOL ! THEN THEN LOOP ;
11
12 : ?RIGHTEDGE HISTOGRAM 127 RCOL !
13   1 127 DO
14     BIGBUFF I M+ EC! 32 ) IF BIGBUFF I 1- M+ EC! 32 (
15      IF I 1- RCOL ! THEN THEN -1 +LOOP ;
```

569

These words are used in navigation.

RHO calculates the angle from the center of the image to the edge of a visible object.

HYP and OPP calculate the sides of a triangle given by the angle to the edge and the distance to an object.

ZETA and TAU calculate angles necessary to reorient the robot toward the goal.

SRHO AND SRRHO calculate the left and right angle from center of the edges given in LCOL and RCOL and leave them on the numeric stack.

LCONVERT takes a real number & converts it to feet and inches format for FMOVE. UNCONVERT goes the other way.

248

```
0 ( MICRONEYE APPLICATIONS CONT. )    DECIMAL
1 ( TRIANGULAR CALCULATIONS ) VARIABLE LENGTH VARIABLE DIRECTION
2 VARIABLE LTHETA VARIABLE RTHETA VARIABLE THETA
3
4 : RHO ( - a ) 64 LCOL @ - ABS )N .336 F* 128.0 F/ ATAN N) ;
5 : RRHO ( - a ) 64 RCOL @ - ABS )N .336 F* 128.0 F/ ATAN N) ;
6 : HYP ( - ) ( adj - hyp ) THETA @ )N COS F/ ;
7 : OPP ( - ) ( hyp - opp ) THETA @ )N SIN F* ;
8 : ZETA ( - a ) ( opp,d - ) FMAP F/ ATAN N) ;
9 : TAU ( zeta - tau ) 90 THETA @ + SWAP - ;
10
11 : SRHO ( ) ( - a ) 64 LCOL @ - )N .336 F* 128.0 F/ ATAN ;
12 : SRRHO ( ) ( - a ) RCOL @ 64 - )N .336 F* 128.0 F/ ATAN ;
13 : LCONVERT ( - feet,inches ) ( r - ) 10.0 F* N) 10 /MOD
14   SWAP 12 10 +/ ;
15 : UNCONVERT ( ft,in - ) ( - r ) N 12.0 F/ )N F+ ;
```

COUNTLINES takes a histogram and scans it looking for transitions from light to dark leaving the number on the stack.

ANGTEST assumes both lines are in view and calculates the apparent angle between them.

1STQUAD takes a vector pointing into the 1st quadrant, calculates the X and Y offsets, and updates X-POS and Y-POS.

2NDQUAD does the same for a vector into the 2nd quadrant.

all angles are assumed from the X-axis.

0 (CONSTANT POSITION NAVIGATION ROUTINES) DECIMAL

```

1
2 : COUNTLINES ( - n ) HISTOGRAM 0 127 0 DO
3   BIGBUFF I M+ ED 32 ) IF BIGBUFF I 1+ M+ ED 32 (
4     IF 1+ THEN THEN LOOP ;
5
6 : ANGTEST ( ) ( - a ) ?LEFTEDGE ?RIGHTEDGE SRHO SRHO F+ FABS ;
7
8 VARIABLE X-POS    VARIABLE Y-POS    VARIABLE ORIANG
9 VARIABLE X-GOAL   VARIABLE Y-GOAL
10
11 : 1STQUAD ( ) ( n a - ) F2DUP SIN F+ Y-POS S+!
12                               COS F+ FNEGATE X-POS S+! ;
13
14 : 2NDQUAD ( ) ( n a - ) F2DUP SIN F+ Y-POS S+!
15                               COS F+ X-POS S+! ;

```

3RDQUAD and 4THQUAD update X-POS and Y-POS for vectors into the 3rd and 4th quadrants.

SFMOVE performs an FMOVE but updates the X and Y positions as it does so.

CURRENT-POSITION sets the variables X-POS, Y-POS and ORIANG to the values on the stack.

SETGOAL sets the variables X-GOAL and Y-GOAL to the values on the stack.

0 (CONSTANT POSITION NAVIGATION CONT.) DECIMAL

```

1
2 : 3RDQUAD ( ) ( n a - ) F2DUP COS F+ X-POS S+!
3                               SIN F+ FNEGATE Y-POS S+! ;
4 : 4THQUAD ( ) ( n a - ) F2DUP COS F+ FNEGATE X-POS
5                               S+! SIN F+ FNEGATE Y-POS S+! ;
6
7 : SFMOVE ( ) ( n - ) F DUP LCONVERT 180.0 ORIANG S  F-  F DUP
8   FO( IF FABS F DUP 90.0 F)
9       IF 180.0 FSWAP F- 2NDQUAD ELSE 1STQUAD THEN
10      ELSE F DUP 90.0 F)
11      IF 180.0 FSWAP F- 3RDQUAD ELSE 4THQUAD THEN
12      THEN  FMOVE ;
13
14 : CURRENT-POSITION ( ) ( x,y,a - ) ORIANG S! Y-POS S! X-POS S! ;
15 : SETGOAL ( ) ( x,y - ) Y-GOAL S! X-GOAL S! ;

```

SBMOVE executes a BMOVE and update the current position of the robot.

SLTURN and SRTURN execute left and right turns while updating the variable ORIANG. They also check for turning past 360 degrees or 0 degrees.

NOTE: all of these words assume an origin in the lower right hand corner of the room as looking toward the bookshelves. Pointing toward the bookshelves is the angle 270 degrees.

OCCHECK checks to see if the X or Y offset are zero and if they are adds a very small amount. This prevents division by zero.

0 (CONSTANT POSITION NAVIGATION CONT.) DECIMAL

```

1
2 : SBMOVE ( ) ( n - ) F DUP LCONVERT 180.0 ORIANG S  F-  F DUP
3   FO( IF FABS F DUP 90.0 F)
4       IF 180.0 FSWAP F- 4THQUAD ELSE 3RDQUAD THEN
5       ELSE F DUP 90.0 F)
6       IF 180.0 FSWAP F- 1STQUAD ELSE 2NDQUAD THEN
7       THEN  BMOVE ;
8
9 : SLTURN ( ) ( a - ) F DUP ORIANG S+! N) LTURN ORIANG S 360.0
10   F) IF -360.0 ORIANG S+! THEN ;
11 : SRTURN ( ) ( a - ) F DUP FNEGATE ORIANG S+! N) RTURN
12   ORIANG S  FO( IF 360.0 ORIANG S+! THEN ;
13
14 : OCCHECK ( ) ( x,y - ) F DUP FO= IF .001 F+ THEN FSWAP F DUP FO=
15   IF .001 F+ THEN FSWAP ;

```

573

ORIENT takes the real X and Y coordinates of a point in the room and turns the robot to face that point. The X and Y offsets are first calculated and checked for zero. Then the direction and angle to turn are calculated. The straight line distance to the point is calculated and saved in the variable LENGTH.

252

```

0 ( CONSTANT NAVIGATION CONT. ) DECIMAL
1
2 : ORIENT ( ) ( x y - ) Y-POS S@ F- FSWAP X-POS S@ F- 0CHECK
3 F2DUP FDUP F@ FSWAP FDUP F@ F+ FSQRT LENGTH S! FDUP F0(
4 IF FSWAP FDUP F0(
5 IF FSWAP F/ ATAN 90.0 F+ 270.0 FSWAP F-
6 ELSE FSWAP F/ ATAN 90.0 F+ 270.0 FSWAP F- THEN
7 ELSE FSWAP FDUP F0(
8 IF FABS FSWAP F/ ATAN
9 ELSE FSWAP F/ ATAN 90.0 FSWAP F- 270.0 F+
10 THEN THEN
11 ORIANG S@ F- FDUP F0( IF 1 DIRECTION !
12 ELSE 0 DIRECTION ! THEN FABS FDUP 181.0 F) IF 360.0
13 FSWAP F- DIRECTION @ NOT DIRECTION ! THEN
14 DIRECTION @ IF SLTURN ELSE SRTURN THEN ;
15

```

574

LEFTEDGE uses the Microneye to find the left edge of an object in the field of view. If a left edge is not found, the head is turned 10 degrees to the left and another picture taken. If an edge is not found within 90 degrees, a longer exposure time is specified and the process is repeated.

253

```

0 ( MICRONEYE NAVIGATION ) DECIMAL
1
2
3
4 : LEFTEDGE BEGIN ?LEFTEDGE LCOL @ 0 )
5 IF HEADLOOK 270 - DUP LCOL @ 64 )
6 IF RHO - ELSE RHO + THEN LTHETA ! 1
7 ELSE 5 LHEAD 1 STEPPER-WAIT HEADLOOK 270 - DUP 90 )
8 IF EXPOSETIME E@ 100 + EXPOSETIME E!
9 RHEAD 1 STEPPER-WAIT
10 ELSE DROP THEN 0
11 THEN UNTIL RHEAD 1 STEPPER-WAIT ;
12
13
14
15

```

575

RIGHTEDGE is the right analogue of leftedge.

254

```

0 ( MICRONEYE NAVIGATION CONT. ) DECIMAL
1
2 : RIGHTEDGE BEGIN ?RIGHTEDGE RCOL @ 127 (
3 IF 270 HEADLOOK - DUP RCOL @ 64 )
4 IF RRHO + ELSE RRHO - THEN RTHETA ! 1
5 ELSE 5 RHEAD 1 STEPPER-WAIT 270 HEADLOOK - DUP 90 )
6 IF EXPOSETIME E@ 100 + EXPOSETIME E!
7 LHEAD 1 STEPPER-WAIT
8 ELSE DROP THEN 0
9 THEN UNTIL LHEAD 1 STEPPER-WAIT ;
10
11
12 : GETSONAR ( - ) ( - rSONAR ) 0 SONAR.B )N 10.0 F/ ;
13
14
15

```

GETSONAR takes a sonar reading and leaves it on the numeric stack as a real number.

576

MAXSONAR moves the head to the angles found by LEFTEDGE and RIGHTEDGE and takes a sonar reading. A direction flag is set according to which sonar value is higher.

GETSONAR takes a sonar reading and converts it to floating point.

GETTURN turns the robot in the direction of the flag set by MAXSONAR. TURNBACK reverses the actions of GETTURN.

MOVEOBST? expects a sonar reading on the stack and moves the robot to within 4 feet of an obstacle if not already.

AXISCOR moves the robot back 4 inches so that the center of rot. of the wheels is the same as the head.

255

0 (MICRONEYE NAVIGATION CONT.) DECIMAL

```

1
2 : MAXSONAR LTHETA @ 10 + DUP LHEAD 1 STEPPER-WAIT 500 MS
3   0 SONAR.B SWAP RHEAD 1 STEPPER-WAIT RTHETA @ 10 + DUP
4   RHEAD 1 STEPPER-WAIT 500 MS 0 SONAR.B SWAP LHEAD
5   1 STEPPER-WAIT
6   ) IF 1 DIRECTION ! LTHETA @ THETA !
7   ELSE 0 DIRECTION ! RTHETA @ THETA ! THEN ;
8
9 : GETTURN THETA @ )N DIRECTION @ IF SLTURN ELSE SRTURN THEN ;
10 : TURNBACK THETA @ )N DIRECTION @ IF SRTURN ELSE SLTURN THEN ;
11
12 : MOVEOBST? ( ) ( threshold - ) FDUP GETSONAR FDUP FROT F)
13   IF FSWAP F- SMOVE ELSE FDROP FDROP THEN ;
14
15 : AXISCOR 0.33 SMOVE ; : UNCOMP 0.33 SMOVE ;

```

577

LSMALLSCAN scans from the center point to 70 degrees to the left at 5 deg. increments taking sonar readings. It takes into account the width of the robot and the distance in LENGTH and checks to see if anything is within that rectangle. If it is the scan stops there and the head angle is stored in LTHETA.

RSMALLSCAN is the right analog of LSMALLSCAN.

CORRIDOR uses LSMALLSCAN and RSMALLSCAN to make sure that there is a safe passage for the robot to move down.

256

0 (DETECT CORRIDOR) DECIMAL

```

1
2 : LSMALLSCAN BEGIN 5 LHEAD 1 STEPPER-WAIT 1.0 90
3   HEADLOOK 270 - - )N COS F/ FDUP LENGTH S@ F(
4   IF GETSONAR F) ELSE FDROP LENGTH S@ GETSONAR F) THEN
5   HEADLOOK 270 - 69 ) OR UNTIL
6   HEADLOOK 270 - LTHETA ! CENTER ;
7
8 : RSMALLSCAN BEGIN 5 RHEAD 1 STEPPER-WAIT 1.0 90
9   270 HEADLOOK - - )N COS F/ FDUP LENGTH S@ F(
10  IF GETSONAR F) ELSE FDROP LENGTH S@ GETSONAR F) THEN
11  270 HEADLOOK - 69 ) OR UNTIL
12  270 HEADLOOK - RTHETA ! CENTER ;
13
14 : CORRIDOR ( - 00,01,10 or 11 ) LSMALLSCAN LTHETA @ 70 ( 2*
15   RSMALLSCAN RTHETA @ 70 ( + ;

```

578

BLOCKED? takes a sonar reading and compares it to the distance on the stack. If the sonar reading is less it aborts. The distance is increased by 1.4 feet to account for the arms.

GETCORDS leaves the X and Y coordinates of the goal on the numeric stack. Used with ORIENT.

ENDAROUND is an algorithm used when an obstacle is directly in front of the robot. It uses the camera to detect the left and right edges then takes a sonar reading beyond each edge and turns toward the largest. The distance and angle to a point clear of the edge is calculated and the robot moves there. It then reverses the turns made and ORIENT's toward the goal again.

257

0 (MICRONEYE NAVIGATION CONT.) DECIMAL

```

1
2 : BLOCKED? ( ) ( hyp - ) 1.4 F+ GETSONAR F)
3   ABORT" BLOCKED! " ;
4
5 : GETCORDS ( ( - x,y ) X-GOAL S@ Y-GOAL S@ ;
6
7
8 : ENDAROUND LEFTEDGE RIGHTEDGE MAXSONAR AXISCOR 500 MS
9   GETTURN 2.0 GETSONAR F/ ATAN FDUP DIRECTION @
10  IF SLTURN ELSE SRTURN THEN
11  FDUP N) THETA +! SIN 2.0 FSWAP F/
12  FDUP BLOCKED? SMOVE
13  TURNBACK UNCOMP GETCORDS ORIENT ;
14
15

```

579

DODGE2RIGHT assumes that an obstacle has been located and is in front of the robot to the left of center. The head is turned to face the box and the right edge is found. A sonar reading is taken at the edge and an angle and distance is calculated so that the robot can move beyond the edge. The robot moves to a point beyond the edge, reverses the turns made and re-ORIENT's toward the goal.

DODGE2LEFT is the same as dodge to right except the obstacle is to the right of center and the robot needs to dodge to the left. Both of these words assume that the obstacle has been located. Both of these words assume that the obstacle has been located and that the location has been saved in LTHETA or RTHETA.

258

0 (MICROEYE NAVIGATION CONT.) DECIMAL

```

1
2 : DODGE2RIGHT LTHETA @ 5 + LHEAD 1 STEPPER-WAIT RIGHTEDGE
3 RTHETA @ DUP 0( IF ABS LHEAD ELSE RHEAD THEN
4 1 STEPPER-WAIT 1.5 GETSONAR F/ ATAN FDUP
5 N) THETA ! 0 DIRECTION ! AXISCOR GETTURN CENTER
6 SIN 1.5 FSWAP F/ FDUP BLOCKED? SFMOVE
7 TURNBACK UNCOMP GETCORDS ORIENT ;
8

```

```

9 : DODGE2LEFT RTHETA @ 5 + RHEAD 1 STEPPER-WAIT LEFTEDGE
10 LTHETA @ DUP 0( IF ABS RHEAD ELSE LHEAD THEN
11 1 STEPPER-WAIT 1.5 GETSONAR F/ ATAN FDUP
12 N) THETA ! 1 DIRECTION ! AXISCOR GETTURN CENTER
13 SIN 1.5 FSWAP F/ FDUP BLOCKED? SFMOVE
14 TURNBACK UNCOMP GETCORDS ORIENT ;
15

```

580

GO takes floating point X and Y coordinates of a goal relative to the robot's center and causes the robot to move to the goal. If any obstacles are in the way the robot uses vision to find to edges and then moves around the obstacle.

259

0 (MICROEYE NAVIGATION CONT.) DECIMAL

```

1
2 : GO ( - ) ( Xreal , Yreal - )
3 SETGOAL GETCORDS ORIENT
4 BEGIN CORRIDOR DUP
5 IF DUP 1 =
6 IF DODGE2LEFT
7 ELSE 2 =
8 IF DODGE2RIGHT
9 ELSE ENDAROUND THEN
10 THEN 0
11 ELSE DROP LENGTH 90 SFMOVE 1
12 THEN
13 UNTIL ;
14
15

```

581

LOCOS implements the law of cosines to find the side oppsite of a known angle. It expects 2 sides and the angle between.

LAWOSIN implements the law of sines to find an angle. It expects a known side, the side opposite the unknown angle and the angle opposite the known side.

260

0 (LAW OF SINES AND COSINES) DECIMAL

```

1
2 : LOCOS ( ) ( B,C,a - A ) COS FOVER F* FROT FDUP FROT F*
3 2.0 F* FNEGATE FSWAP FDUP F* F* FSWAP FDUP F*
4 F* FSQRT ;
5
6 : LAWOSIN ( ) ( A B a - b ) SIN F* FSWAP F/ ASIN ;
7
8
9
10
11
12
13
14
15

```

582

BOXORIENT moves the robot to a predetermined distance from the box. It then uses the vision to find the angles to the edge and the sonar to find the distance to the edges and center. From this information it calculates the orientation of the box.

261

```

0 ( VISION NAVIGATION PRIMITIVES ) DECIMAL
1
2 : BOX-ORIENT ( ) ( - angle) 2.0 MOVEOBST? PCALC
3   RIGHTEDGE RTHETA @ RHEAD 1 STEPPER-WAIT
4   2.0 GETSONAR RTHETA @ )N LOCOS ( A)
5   GETSONAR RTHETA @ )N LAMOSIN ( c)
6   90.0 F- FABS CENTER ;
7
8
9
10
11
12
13
14
15

```

583

POSINIT is for starting at an often used position.
 FACEWALL causes the robot to turn and face the 270 degree point
 HEXTEST moves the robot thru a hexagon to test the positional accuracy of the SFMOVE command.
 WAVEARM simulates the flipping of a switch by moving the arm out and then back in.

262

```

0 ( NAVIGATION PRIMITIVES CONT. ) DECIMAL
1
2 : POSINIT 6.0 8.0 270.0 CURRENT-POSITION ;
3 : FACEWALL CENTER 270.0 DRIANG S@ F- FDUP F@
4   IF FABS S@TURN ELSE SLTURN THEN ;
5
6 : HEXTEST POSINIT 2.0 12.0 ORIENT LENGTH S@ SFMOVE
7   2.0 16.0 ORIENT LENGTH S@ SFMOVE 6.0 20.0 ORIENT LENGTH S@
8   SFMOVE 10.0 16.0 ORIENT LENGTH S@ SFMOVE 10.0 12.0 ORIENT
9   LENGTH S@ SFMOVE 6.0 8.0 ORIENT LENGTH S@ SFMOVE FACEWALL ;
10
11 : WAVEARM 40 LLPIVOT 4000 MS 40 RLPIVOT ." WAVEARM " ;
12
13
14
15

```

584

FINDLINES searches for the lines with the camera and COUNTLINES The word scans from center to 45 degrees left in 9 degree increments and if no lines are found, it scan from center to 45 degrees right. The head is left pointing at the lines.

LOCOS implements the law of cosines to find the side oppsite of a known angle. It expects 2 sides and the angle between.

LAMOSIN implements the law of sines to find an angle. It expects 11 a known side , the side opposite the unknown angle and the angle opposite the known side.

263

```

0 ( NAVIGATION ROUTINES CONT. ) DECIMAL
1
2 : FINDLINES ( - n) 0 5 0 DO DROP COUNTLINES DUP 0) IF LEAVE
3   ELSE 9 LHEAD 1 STEPPER-WAIT THEN LOOP
4   DUP 0= IF CENTER 5 0 DO DROP COUNTLINES DUP 0)
5   IF LEAVE ELSE 9 RHEAD 1 STEPPER-WAIT THEN
6   LOOP THEN ;
7
8
9
10
11
12
13
14
15

```

585

SONARSET takes a sonar reading at 20 degrees to the left and stores it in S1; at the center position and stores it in S2 and at 20 degrees right and stores it in S3.

GETPERP uses SONARSET then the law of sines and law of cosines with the known angle to calculate the robot's angle relative to the wall. The robot then turns the angle calculated to be parallel to the wall, moves to a point directly in front of the place in question and then turns 90 degrees to face the wall.

NOTE: SIN and COS only return angles from 0 to 90 degrees so LAMOSIN will return an angle which should be subtracted from 180.

264

```

0 ( NAVIGATION ROUTINES CONT. ) DECIMAL
1
2 VARIABLE S1 VARIABLE S2 VARIABLE S3
3
4 : SONARSET 20 LHEAD 1 STEPPER-WAIT 300 MS GETSONAR S1 S!
5 CENTER 300 MS GETSONAR S2 S! 20 RHEAD 1 STEPPER-WAIT
6 300 MS GETSONAR S3 S! CENTER ;
7
8 : GETPERP SONARSET S1 S@ S3 S@ F) IF S1 S@ ELSE S3 S@
9 THEN S2 S@ 20.0 LOCOS ( C ) S1 S@ 20.0 LAMOSIN
10 ( T3A ) 90.0 FSWAP F- ( T4 ) FDUPI SIN S2 S@ F+ ( S4 )
11 FSWAP 90.0 FSWAP F- ( T5 ) S1 S@ S3 S@ F)
12 IF SLTURN SFMOVE 90.0 SRTURN
13 ELSE SRTURN SFMOVE 90.0 SLTURN THEN ;
14
15

```

586

FLIPSWITCH is a task that causes the robot to seek out and move to a place on the wall marked with two lines and simulate flipping a switch there. It first faces the wall and finds the lines. It turns the robot to face the lines and then uses GETPERP to move the robot till it is perpendicular to the wall in front of the lines. The robot then moves forward till it is 2 feet from the wall and waves its arms.

265

```

0 ( NAVIGATION ROUTINES CONT. ) DECIMAL
1
2 : FLIPSWITCH FACEWALL FINDLINES 1 =
3 IF ?LEFTEDGE SRMO FDUPI N) ABS FO(
4 IF RHEAD ELSE LHEAD THEN 1 STEPPER-WAIT
5 THEN COUNTLINES 2 = NOT ABORT* NOT GOAL ! "
6 HEADLOOK 270 - )N
7 ANGSTEST SRMO FSWAP 2.0 F/ F+ F+ FDUPI FO(
8 IF ABS SRTURN ELSE SLTURN THEN CENTER FO
9 GETPERP 2.0 MOVEDBST? WAVEARM ;
10
11
12
13
14
15

```

587

OMOM determines the 0th moment of the image which gives the area by summing the pixels that are on.

1MOM calculates the centroid of the image, leaving them in the variables MX1 MY1.

266

```

0 ( MOMENTS OF THE IMAGE ) DECIMAL
1
2 VARIABLE MO VARIABLE MX1 VARIABLE MY1 VARIABLE MXY
3 VARIABLE MX2 VARIABLE MY2
4
5
6 : OMOM ( - ) OBLLOCATE IMAGEEXPAND 0 MO ! COLIMITS DO
7 ROWLIMITS DO I J S1(I,J) MO +! LOOP LOOP ;
8
9 : 1MOM OMOM 0.0 MX1 S! 0.0 MY1 S! COLIMITS DO ROWLIMITS DO
10 I J S1(I,J) DUP I * )N MO @ )N F/ MX1 S+!
11 J * )N MO @ )N F/ MY1 S+! LOOP LOOP ;
12
13
14
15

```

588

This is the high level version of OBJLOCATE.

267

```

0 ( HIGH LEVEL OBJLOCATE )   DECIMAL
1
2 VARIABLE UPROW  VARIABLE DOWNROW  VARIABLE LCOL  VARIABLE RCOL
3
4 : GETBYTE ( I J - n ) WORKMAP 4 ROLL 16 * 4 ROLL 8 / + M+ EC! ;
5
6 : MASK ( I - n ) 1 SWAP 8 /MOD DROP 8 SWAP - 0 DO 2* LOOP 2/ ;
7
8 : OBJLOCATE 63 UPROW ! 0 DOWNROW ! 127 LCOL ! 0 RCOL !
9       128 0 DO 64 0 DO
10         I J GETBYTE J MASK AND 0=
11         IF UPROW @ I ) IF I UPROW ! THEN
12           DOWNROW @ I ( IF I DOWNROW ! THEN
13             LCOL @ J IF J LCOL ! THEN
14             RCOL @ J ( IF J RCOL ! THEN THEN
15       LOOP LOOP ;

```

589

These are the high level versions of IMAGEEXPAND and
IMAGE-CONTRACT.

268

```

0 ( HIGH LEVEL IMAGE EXPAND AND CONTRACT )   DECIMAL
1
2 : IMAGEEXPAND 128 0 DO 64 0 DO
3       I J GETBYTE J MASK AND 0= IF 0 ELSE 1 THEN
4       TEMPBUFF J I 128 * + M+ EC!
5       LOOP LOOP ;
6
7 VARIABLE TEMP
8
9 : IMAGE-CONTRACT ( tempbuff or bigbuff - )
10      1024 0 DO 0 TEMP !
11      8 0 DO 2DROP 7 I - J 8 * + M+ EC! -1 AND
12      I )N 2** N) * TEMP +!
13      LOOP TEMP @ WORKMAP I M+ EC!
14      LOOP 2DROP ;
15

```

590

269

APPENDIX B

Annotated Assembly Language Listing

```
TITLE   MICRONEYE ASSEMBLER ROUTINES (MEYEDVR)
PAGE    66,132
.SALL
COMMENT :
```

MEYEDVR -- MODULE DESCRIPTION

This driver module takes care of all the necessary arrangements for getting an image from the MicronEye to the graphics page in memory of the IBM PC. The routine is assembled to be relocatable.

There are 2 assembly routines available to the user--
MOVESCR and GETPIC. GETPIC takes the setup parameters selected at the main menu and calculates the commands that GETCALC will send the MicronEye and the number of bytes the MicronEye will return. MOVESCR moves the picture in the WORKMAP area to the screen.

All variables in the argument list are assumed to be of type INTEGER. The variables are defined as follows:

PIC_TYPE -- determines the format of the image transmitted from the MicronEye. The following are valid commands:

0 - 128 x 64 picture (black & white)

The command byte sent this routine is not the same as the control byte sent to the MicronEye. For a complete description of the MicronEye control byte refer to the Operator's manual.

EXPOSE_TIME -- the number of milliseconds for which the image should be exposed.

WHITE_PCT -- A value between 0 and 100 that indicates the approximate percent of the current picture that is white.

BUFFER SIZE REQUIREMENTS FOR EACH PICTURE TYPE

PICTURE TYPE	BUFFER NAME	BYTES WHEN USING	
		1 ARRAY	2 ARRAYS
128 x 64 (B/W)	WORKMAP	1024	2048

SUBTTL MAIN SUBROUTINES AND DATA DECLARATIONS

MAIN SUBROUTINES AND DATA DECLARATIONS

```

0000                                PAGE
                                CSEG  SEGMENT BYTE
                                ASSUME CS:CSEG,DS:CSEG,ES:CSEG

;
;
0000 AA      WORKMAP      DB      0AAH      ;use 128 x 64 only
0400                                ORG      400H
0400 AB      BIGBUFF     DB      0ABH      ;buffer for display
2400                                ORG      2400H
2400 AC      TEMPBUFF    DB      0ACH      ;temp buffer for manipulation
4400                                ORG      4400H

;
4400 0400     MAPBYTES     DW      1024
4402 0000     KEY_VALUE    DW      0
4404 0000     PICTYPE      DW      0
4406 0000     SCREEN_START DW      0
4408 0040     SCR_ROWCT    DW      64
440A 0010     SCR_COLCT    DW      16
440C 0001     ARRAYCT     DW      1
440E 0000                                DW      0
4410 0318     CONTROL     DW      318H      ;camera control port
4412 0318     STATUS      DW      318H      ;camera status port
4414 0319     DATAIN     DW      319H      ;camera data-from port
4416 0319     DATAOUT    DW      319H      ;camera data-to port
4418 0000     EXITEDF     DW      0
441A 012C     EXPOSE_TIME  DW      300
441C 0000     WHITE_PCT   DW      0
441E 1B      COMMAND     DB      1BH
441F ????     ROWCTR      DW      ?
4421 ????     COLCTR      DW      ?
4423 00      BITCT       DB      0
4424 0000     WORDCT      DW      0
4426 ????     WHITECT     DW      ?
4428 0400     TOTBYTES    DW      1024
442A ????     MAPADR      DW      ?

;
;

```

SUBTTL PARMCALC — Determine picture characteristics

PARMCALC — Determine picture characteristics

```

                                PAGE
                                ;
                                PCALC  PROC  FAR
442C                                PUSH  DS
442C 1E                            PUSH  ES
442D 06                            PUSH  BX      ;SAVE CALLING ENVIRONMENT
442E 53                            PUSH  SI
442F 56                            MOV   AX,CS      ;set up ES to be same as routines
4430 8C C8                        MOV   ES,AX      ; code segment
4432 8E C0                        MOV   AX,CS      ;set up DS to be the same as this routine'
4434 8C C8                        MOV   DS,AX      ; code segment
4436 8E D8                        MOV   BX,0       ;PICTYPE offset
4438 BB 0000                      MOV   PICTYPE,BX  ;128 X 64
443B 89 1E 4404 R                 MOV   ARRAYCT,1  ;use only lower array
443F C7 06 440C R 0001           CALL  ACIACLR   ;initialize comm link
4445 E8 4550 R                   CALL  FRAMEGRAB ;get picture from MicronEye
4448 E8 4492 R                   POP    SI
444B 5E                           POP    BX
444C 5B                           POP    ES      ;restore environment
444D 07                           POP    DS
444E 1F                           RET           ;RETURN TO FORTH
444F CB
4450                                PCALC  ENDP
                                ;
                                ;
                                SUBTTL MOVESCR — Move picture to graphics screen

```

MOVESCR -- Move picture to graphics screen

```

                                PAGE      +
MOVESCR PROC FAR
4450      1E      PUSH      DS      ;save calling environment
4451      06      PUSH      ES      ;
4452      53      PUSH      BX
4453      56      PUSH      SI
4454      8C 08      MOV      AX,CS      ;
4455      AE D6      MOV      DS,AX      ;set up proper data segment ptr
4456      6E C0      MOV      ES,AX      ;set extra segment pointer
4457      E8 4511 R   CALL     MOVEBUF   ;subroutine to expand image
4458      E2 0400 R   MOV      SI,OFFSET BIGBUF
4459      FC      CLD      ;make sure direction flag is forward
445A      8B 3E 4456 R MOV      DI,SCREEN_START ;point destination register
445B      B8 E800      MOV      AX,0B800H      ; at desired offset on the
445C      BE D0      MOV      ES,AX      ; graphics screen
445D      B9 0040      MOV      DX,64      ;DX is set to the number of rows/2
445E      D1 E4      SHR      DX,1      ; because we handle two rows at a time
445F      BB 0040      MOV      BX,64      ;BX is set to the number of bytes/row
4460      8B C5      nxtrow: MOV     CX,BX      ;CX is the number of words per row
4461      D1 E9      SHR      CX,1
4462      F3/ AE      REP      MOVSW      ;move the entire even row
4463      2B FB      SUB      DI,BX      ;set up for odd row which is 2000H away
4464      81 C7 2000      ADD      DI,2000H
4465      BE CB      MOV      CX,BX
4466      D1 E9      SHR      CX,1
4467      F3/ AE      REP      MOVSW      ;move the entire odd row
4468      81 FF 1FB0      SUB      DI,1FB0H      ;set up for next even row
4469      2B FB      SUB      DI,BX
446A      4A      DEC      DX      ;continue until we've done every row
446B      75 E5      JNZ      NXTROW
446C      5E      POP      SI
446D      5F      POP      BX      ;restore environment
446E      C7      POP      ES
446F      CF      POP      DS
4470      1F      RET
4471      CB
MOVESCR ENDP

```

SUBTTL FRAMEGRABBER ROUTINES -- gets image from MicroEye

FRAMEGRABBER ROUTINES — gets image from MicronEye

```

PAGE +
ASSUME ES:CSEG
FRAMEGRAB PROC NEAR
4492 C7 06 4402 R 0000      MOV     KEY_VALUE,0      ;zero keyvalue
4498 BF 0000 R              MOV     DI,OFFSET WORKMAP
449E BA 26 441E R          MOV     AH,COMMAND      ;tell MicronEye to soak w/o send
449F B9 3E 442A R          MOV     MAPADR,DI
44A3 B0 CC C0              OR      AH,0COH
44A6 EB 455B R              CALL    SENDCMD
44A9 EB 4572 R              CALL    SOAK              ;soak for specified expose time
44AC BB DF                  MOV     BX,DI              ;save start address of buffer for compare
44AE EB 4589 R              CALL    INT0FF           ;disable interrupts during grab
44B1 BA 26 441E R          MOV     AH,COMMAND      ;tell MicronEye to send picture (w/o soak)
44B5 EB 455B R              CALL    SENDCMD
44B8 BB 16 4412 R          MOV     DX,STATUS
44BC AC D8                  MOV     AX,DS              ;equate extra segment and data segment
44BE BE C0                  MOV     ES,AX
44C0 C7 06 4426 R 0000      MOV     WHITECT, 0      ;zero whitect
44C6 FC                      CLD              ;set direction reg to forward movement
44C7 B9 000F                nfbyt: MOV     CX,15      ;set up timeout register for byte receipt
44CA EC                      rechck: IN      AL,DX    ;if byte not available after 15
44CB D0 E0                  SHL      AL,1          ; checks then we assume the MicronEye is
44CD 73 0D                  JNC     DNCHK      ; done sending
44CF 42                      INC     DX              ;when byte has come we point DX to
44D0 EC                      IN      AL,DX          ; DATAIN and get the byte and then
44D1 4A                      DEC     DX              ; repoint DX at the status register
44D2 AA                      STOSB      ;put the byte in the buffer
44D3 D0 E0                  SHL      AL,1          ;increment whitect if high bit of byte
44D5 B3 16 4426 R 00        ADC     WHITECT, 0      ; was white
44DA EB EB                  JMP     NFBYT      ;go back and try and get another byte
44DC E2 EC                      dnchk: LOOP    RECHK      ;this is the timeout decrementer
44DE BB C7                  MOV     AX,DI              ;if we have timed out then we check and
44E0 2B C3                  SUB     AX,BX          ; see if we got as many bytes as we had
44E2 3D 0014                CMP     AX,20          ; hoped to get
44E5 7F 00                  JG      NTO              ;if not enough bytes received then we
44E7 EB 4582 R              nto:  CALL    INT0N      ;reenable interrupts
44EA BA 26 441E R          MOV     AH,COMMAND      ;tell MicronEye to refresh w/o send
44EE B0 CC 80              OR      AH,80H
44F1 EB 455B R              CALL    SENDCMD
44F4 C3                      nta:  RET
;
;
44F5 FRAMEGRAB ENDP
;
;
;

```

FRAMEGRABBER ROUTINES -- gets image from MicronEye

```

                                PAGE  +
44F5      CLEARW  PROC  FAR          ;the CLEARW routine zeroes the WORKMAP buffer
44F5 1E      PUSH  DS
44F6 06      PUSH  ES
44F7 8C C8    MOV   AX,CS           ;get the code segment
44F9 8E D8    MOV   DS,AX          ;DS the same as code segment
44FB BF 0000 R  MOV   DI,OFFSET WORKMAP
44FE FC      CW:   CLD              ;note that only an area corresponding
44FF 8B 0E 4400 R  MOV   CX,MAPBYTES ; in size to the current picture type
4503 D1 E9    SHR   CX,1           ; is cleared
4505 8C D8    MOV   AX,DS          ;set up the ES register to point to the
4507 8E C0    MOV   ES,AX          ; data segment
4509 B8 00AA   MOV   AX,0AAH       ;select the value to be replicated thru memory
450C F3/ AB   REP   STOSW          ;this command singlehandedly zeroes the desire
450E 07      POP   ES              ; memory area
450F 1F      POP   DS
4510 CB      RET
4511      CLEARW  ENDP
;
4511      MOVEBUF PROC  NEAR
4511 B9 0400    MOV   CX,1024       ;1024 bytes in image
4514 FC      CLD                   ;direction bit forward
4515 BE 0000 R  MOV   SI,OFFSET WORKMAP ;present image
4518 BF 0400 R  MOV   DI,OFFSET BIGBUFF ;expanded image will be here
451B B8 0000   HERE:  MOV   AX,00    ;clear ax
451E AC      LODSB                 ;get byte from present image
451F 8A D8    MOV   BL,AL          ;save byte in bl
4521 BA 0003   MOV   DX,03         ;shift and mask counter
4524 56      PUSH  SI
4525 51      PUSH  CX
4526 8B F2   HERE2: MOV   SI,DX     ;setup index reg.
4528 BA B4 4548 R  MOV   AL,[BITMASK+SI] ;get mask value
452C 22 C3    AND   AL,BL          ;mask extra bits
452E 8B CA    MOV   CX,DX          ;get present count
4530 D1 E1    SHL   CX,1           ;multiply by 2
4532 D2 E8    SHR   AL,CL          ;shift bits to low 2 bits
4534 8B F0    MOV   SI,AX          ;setup index register
4536 BA BC 454C R  MOV   CL,[TABLE+SI] ;get value for new byte
453A 8A C1    MOV   AL,CL
453C AA      STOSB                 ;put byte in bigbuff
453D 4A      DEC   DX              ;increment count
453E 83 FA FF   CMP   DX,-1        ;check for last two bits
4541 75 E3     JNE  HERE2
4543 59      POP   CX              ;get large counts
4544 5E      POP   SI
4545 E2 D4     LOOP  HERE          ;decrements cx
4547 C3      RET
4548      MOVEBUF ENDP
;
4548 03 0C 30 C0  BITMASK DB  003H,00CH,030H,0C0H
454C 00 0F F0 FF  TABLE DB  000H,00FH,0F0H,0FFH

```

FRAMEGRABBER ROUTINES — gets image from MicronEye

```

                                PAGE
4550      ACIACLR PROC          NEAR
4550 8B 16 4410 R              MOV     DX,CONTROL
4554 B0 C0                    MOV     AL,0C0H           ;send master reset to camera
4556 EE                        OUT     DX,AL
4557 B0 28                    MOV     AL,28H           ;send camera protocol of
4559 EE                        OUT     DX,AL           ; 1 start, 8 data, 1 stop bits
455A C3                        RET
455B      ACIACLR ENDP
;
435B      SENDCMD PROC        NEAR
455B B9 03E8                  MOV     CX,1000
455E 8B 16 4412 R              scmd:  MOV     DX,STATUS
4562 EC                        IN      AL,DX           ;get status of camera
4563 A8 40                    TEST    AL,40H           ;see if command can be sent
4565 75 03                    JNZ     sok
4567 E2 F5                    LOOP    scmd           ;loop until ready or timeout
4569 C3                        RET
456A 8A C4                    sok:   MOV     AL,AH           ;set up command
456C 8B 16 4416 R              MOV     DX,DATAOUT
4570 EE                        OUT     DX,AL           ;send camera command
4571 C3                        RET
4572      SENDCMD ENDP
;
4572      SOAK PROC           NEAR
4572 8B 0E 441A R              MOV     CX,EXPOSE_TIME ;soaktime = number msec delay
4576 E3 09                    JCXZ    NOSOAK
4578 51                        S1:   PUSH    CX
4579 B9 0106                  MOV     CX,262           ;set up loop for 1 msec
457C E2 FE                    S2:   LOOP    S2
457E 59                        POP     CX
457F E2 F7                    LOOP    S1
4581 C3                        NOSOAK: RET
4582      SOAK ENDP
4582      INTON PROC          NEAR           ;enable kbd/timer interrupts
4582 E4 21                    IN      AL,21H
4584 24 FC                    AND     AL,0FCH
4586 E6 21                    OUT     21H,AL
4588 C3                        RET
4589      INTON ENDP
;
4589      INTOFF PROC         NEAR           ;disable kbd/timer interrupts
4589 E4 21                    IN      AL,21H
458B 0C 03                    OR     AL,3
458D E6 21                    OUT     21H,AL
458F C3                        RET
4590      INTOFF ENDP
;
4590      CSEG ENDS
                                END

```

Segments and groups:

Name	Size	align	combine class
CSEG	4590	BYTE	NONE

Symbols:

Name	Type	Value	Attr
ACIACLR.	N PROC	4550	CSEG Length =000B
ARRAYCT.	L WORD	440C	CSEG
BIGBUFF.	L BYTE	0400	CSEG
BITCT.	L BYTE	4423	CSEG
BITMASK.	L BYTE	4548	CSEG
CLEARW.	F PROC	44F5	CSEG Length =001C
COLCTR.	L WORD	4421	CSEG
COMMAND.	L BYTE	441E	CSEG
CONTRL.	L WORD	4410	CSEG
CW.	L NEAR	44FE	CSEG
DATIN.	L WORD	4414	CSEG
DATOUT.	L WORD	4416	CSEG
DNCHK.	L NEAR	44DC	CSEG
EXITEDF.	L WORD	4418	CSEG
EXPOSE_TIME.	L WORD	441A	CSEG
FRAMEGRAB.	N PROC	4492	CSEG Length =0063
HERE.	L NEAR	451B	CSEG
HERE2.	L NEAR	4526	CSEG
INTOFF.	N PROC	4589	CSEG Length =0007
INTON.	N PROC	4582	CSEG Length =0007
KEY_VALUE.	L WORD	4402	CSEG
MAPADR.	L WORD	442A	CSEG
MAPBYTES.	L WORD	4400	CSEG
MOVEBUF.	N PROC	4511	CSEG Length =0037
MOVESCR.	F PROC	4450	CSEG Length =0042
NFBYT.	L NEAR	44C7	CSEG
NOSDAX.	L NEAR	4581	CSEG
NTA.	L NEAR	44F4	CSEG
NTD.	L NEAR	44E7	CSEG
NXTROW.	L NEAR	4472	CSEG
PCALC.	F PROC	442C	CSEG Length =0024
PICTYPE.	L WORD	4404	CSEG
RECHK.	L NEAR	44CA	CSEG
ROWCTR.	L WORD	441F	CSEG
S1.	L NEAR	4578	CSEG
S2.	L NEAR	457C	CSEG
SCMD.	L NEAR	455E	CSEG
SCREEN_START.	L WORD	4406	CSEG

SCR_COLCT.	L WORD	440A	CSEG	
SCR_ROWCT.	L WORD	440B	CSEG	
SENDCMD.	N PROC	455B	CSEG	Length =0017
SOAK.	N PROC	4572	CSEG	Length =0010
SOK.	L NEAR	456A	CSEG	
STATUS.	L WORD	4412	CSEG	
TABLE.	L BYTE	454C	CSEG	
TEMPBUFF.	L BYTE	2400	CSEG	
TOTBYTES.	L WORD	442B	CSEG	
WHITECT.	L WORD	4426	CSEG	
WHITE_PCT.	L WORD	441C	CSEG	
WORDCT.	L WORD	4424	CSEG	
WORKMAP.	L BYTE	0000	CSEG	

The IBM Personal Computer MACRO Assembler 06-19-85 PAGE Symbols-2
MICRONEYE ASSEMBLER ROUTINES (MEYEDVR)

Warning Severe
Errors Errors
0 0

VITA

Thomas Earl Swift was born in Clarksville, Tennessee on April 7, 1960. He attended elementary and junior high schools in Charlotte, Tennessee and was graduated from Dickson County Senior High School in May 1978. He began his undergraduate studies that fall and received his Bachelor of Science degree in Electrical Engineering in June 1983. The following fall he began graduate studies toward a Master's of Science degree in Electrical Engineering. He worked on a research assistantship from January 1984 until December 1985 at Oak Ridge National Laboratory. His Master's degree was awarded in December 1985.