December 1981

Report. No. STAN-CS-81-889

Also numbered:
AIM-323A

AL Users' Manual

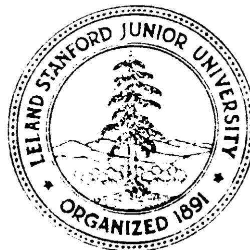
by

Shahid Mujtaba
Ron Goldman

Third Edition

Department of Computer Science

Stanford University
Stanford, CA 94305



AL Users' Manual

Third Edition

by

Shahid Mujtaba & Ron Goldman

Stanford Artificial Intelligence Laboratory
Stanford University, California 94305.

December 1981.

ABSTRACT

AL is a high-level programming language for manipulator control useful in industrial assembly research. This document describes the current state of the AL system now in operation at the Stanford Artificial Intelligence Laboratory, and teaches the reader how to use it. The system consists of the AL compiler and runtime system and the source code interpreter, POINTY, which facilitates specifying represent ation of parts, and interactive execution of AL statements.

This work was supported by the National Science Foundation through the following grants: NSF-APR-74-01390-A04, NSF-DAR-78-15914 and NSF-MEA-80-19628.

dedicated to

yarm

barm

garm

rarm

TABLE OF CONTENTS

1. INTRODUCTION 1
   1.1 How to use this manual 1
   1.2 Other implementations of the AL system 2

2. THE AL SYSTEM AT SAIL 3
   2.1 Design philosophy of AL 3
      2.1.1 Introduction and history 3
      2.1.2 Plantime and runtime systems 4
      2.1.3 Data and control structure 5
      2.1.4 Motion of objects 6
      2.1.5 Sensory information 7
      2.1.6 Programming aids 7
         2.1.6.1 AL compiler 7
         2.1.6.2 Interactive model building 8
         2.1.6.3 Debuggers 8
   2.2 AL system hardware 8
   2.3 Software 11
   2.4 Programming in AL 13

3. AL TUTORIAL 14
   3.1 Basic constructs 14
      3.1.1 Data types 14
         3.1.1.1 *SCALARS* 15
         3.1.1.2 *VECTORS* 17
         3.1.1.3 *ROTATIONS* 18
         3.1.1.4 *FRAMES* 20
         3.1.1.5 *TRANSFORMS* 21
      3.1.2 Block structure - i.e. "what's a program" 22
      3.1.3 A simple program 23
   3.2 Simple *MOVE* statement 24
      3.2.1 More about *barm* and *bpark* 25
   3.3 Using the fingers: *OPEN, CLOSE & CENTER* 28
   3.4 Intermediate points - *VIA, APPROACH* and *DEPARTURE* 29
   3.5 Modelling objects - affixment & indirect moves 30
   3.6 Sensing forces - simple condition monitors 34
   3.7 Force and stiffness control 35
   3.8 Control structures: *IF, FOR & WHILE* statements 36
   3.9 Control structures (cont): *CASE & UNTIL* statements 38
   3.10 Simultaneous motion: *COBEGIN-COEND, SIGNAL-WAIT* 41
   3.11 Arrays 44

3.12 Procedures 45
3.13 Hints to the Programmer 48
3.13.1 Upward pointing grasping positions 48
3.13.2 Initialization and program end 48
3.13.3 Slowing down movements 48

4. THE AL LANGUAGE 49
4.1 Basic constructs 49
4.1.1 Programs 49
4.1.2 Variables 49
4.1.3 Comments 49
4.2 Data types and expressions 50
4.2.1 Algebraic data types: *SCALAR, VECTOR, ROT, FRAME, TRANS* 50
4.2.2 Labels, Events and Strings 50
4.2.3 Arrays 50
4.2.4 Dimensions 51
4.2.5 Declarations 51
4.2.6 Arithmetic expressions 52
4.2.7 Predeclared constants and variables 54
4.2.8 Some examples 55
4.3 Affixment: *AFFIX & UNFIX* 55
4.4 Motions and Device operation 56
4.4.1 The basic *MOVE* statement 56
4.4.2 Intermediate points: *VIA*, *DEPARTURE & APPROACH* 56
4.4.3 Force and Stiffness control 58
4.4.3.1 Spring force application - stiffness 58
4.4.3.2 Constant force application 59
4.4.3.3 Zeroing the force wrist 60
4.4.3.4 Collecting force components 60
4.4.4 Condition monitors 61
4.4.4.1 Types: force, duration, event, boolean, arrival and departing 61
4.4.4.2 *ENABLE* and *DISABLE* - labelled condition monitors 63
4.4.5 User error handler *-ERROR* and *RETRY* 64
4.4.6 Other clauses: *DURATION, SPEED-FACTOR,*
NULLING & WOBBLE 64
4.4.7 Controlling the fingers: *OPEN, CLOSE & CENTER* 65
4.4.8 *STOP & ABORT* 66
4.4.9 Other devices 66
4.4.9.1 The *OPERATE* statement 66
4.4.9.2 The ADAC interface 67
4.4.9.3 The Vision Module 67
4.4.9.4 The VAL controllers 67
4.5 Non-motion statements 68

4.5.1 Assignment statement 68
4.5.2 Traditional control structures: *IF, FOR,* ***WHILE,*** *UNTIL,* ***CASE*** . 68
4.5.3 Procedures 70
4.5.4 Parallel control: ***COBEGIN-COENDD,*** *SIGNAL & WAIT* 70
4.5.5 Statement condition monitors 71
4.5.6 *PAUSE* statement 71
4.5.7 I/O 72
4.5.8 Macros 72
4.5.9 *REQUIRE* statement 73

5. USING AL 75
5.1 Compilation of user programs 75
5.1.1 Compilation with switches 76
5.2 Loading and executing the AL program 76
5.3 Complete runtime execution sequence 78
5.4 Error Corrections and Recovery 80
5.4.1 Parsing errors 80
5.4.2 Compiler errors 83
5.4.3 PALX errors 83
5.4.4 Loading errors 83
5.4.5 Runtime errors 84
5.4.5.1 Mot ion associated errors 84
5.4.5.2 Non-Motion errors 86
5.4.5.3 Continuation from runtime errors 87
5.5 Hints on debugging 87
5.5.1 Parse time debugging aids 87
5.5.2 Runtime debugging aids 88
5.6 The GAL program module for graphing force data 88

6. POINTY 90
6.1 Description of POINTY 90
6.1.1 Introduction 90
6.1.2 Pointing with a manipulator 91
6.1.2.1 Implicit specification of frames 91
6.1.2.2 Pointer 91
6.1.3 System hierarchy 92
6.2 Executing POINTY 93
6.3 POINTY instructions 94
6.3.1 Basic constructs 95
6.3.2 Data types, expressions, declarat ions and dimensions 95
6.3.2.1 Explicit and implicit declarations 95
6.3.2.2 Predeclared variables and constants 95
6.3.2.3 Implicit data types 96

6.3.2.4 Expressions 96
6.3.3 Affixment tree operations - *AFFIX & UNFIX* 97
6.3.4 Motion and Device operations 98
6.3.4.1 MOVE statement 98
6.3.4.2 Hand motion - *CENTER, OPEN, CLOSE* 99
6.3.4.3 *DRIVE* command 100
6.3.4.4 PUMA simulated joystick commands 100
6.3.5 Non-motion AL statements 102
6.3.5.1 Assignment statements 102
6.3.5.2 Control structures 102
6.3.5.3 Procedures and Macros 102
6.3.5.4 Condition monitors 104
6.3.5.5 I/O 104
6.3.5.6 *REQUIRE* statement 104
6.3.6 Deletion statement 104
6.3.7 Display routines 105
6.3.7.1 Data Disk screen 105
6.3.7.2 VT05 screen 106
6.3.7.3 Force Graphics 106
6.3.8 File input/output 106
6.3.8.1 Saving current state - *WRITE* 106
6.3.8.2 Getting a given world state - *READ & QREAD* 107
6.3.8.3 Saving a terminal session - *PHOTO* 108
6.3.9 HELP module 108
6.3.10 Interactive debugger 111
6.3.10.1 Setting breakpoints 111
6.3.10.2 Permanent breakpoints 112
6.3.10.3 Abort execution 112
6.3.10.4 Restarting a program 112
6.3.10.5 Displaying source level code 112
6.3.10.6 Listing the existing break points 112
6.3.10.7 Removing a break point 113
6.3.10.8 Resuming execution 113
6.3.11 Miscellaneous commands 114
6.4 Hints on using POINTY 115
6.4.1 Suggested sequence for using pointer 115
6.4.2 Hints 116
6.4.3 Accurate Part Relocation 116

8. APPENDICES 118
Appendix I. AL reserved words, predeclared constants and macros . . 118
Appendix II. POINTY reserved words 120
Appendix III. AL command summary 121

Appendix IV. POINTY command summary 124
Appendix V. AL execution summary 125
Appendix VI. AL examples 126
Appendix VII. POINTY examples 135
Appendix VIII. Vision Module Routines 139
Appendix IX. Generating a new system 145
Appendix X. References 153
Appendix XI. Acknowledgements 155

9. INDEX 156

TABLE OF FIGURES

Fig. 2.1 Hardware setup for AL at SAIL 9
Fig. 2.2 PUMA arm 10
Fig. 2.3 Stanford Scheinman arm 10
Fig. 2.4 Software organization 12

Fig. 3.1 Diagram of hand showing coordinate axes 26
Fig. 3.2 Diagram of four arms on table 27

Fig. 5.1 Sample Output of force graphics 89

Fig. 6.1 Sample of display when using help command at menu node 109
Fig. 6.2 Typical path through the help module 110

I. INTRODUCTION

1.1 How to use this manual

This document attempts to gather in one place all the information that a user needs to program a manipulator in the AL programming language at the Stanford Artificial Intelligence Laboratory (SAIL). In addition to meeting the requests made by other research organizations for detailed information on the current status and configuration of the system at SAIL, it updates the original AL document which was a design specification of the language. The AL system has been growing and continuously evolving as new features were implemented and used. This third edition of the manual includes features not present in the editions of November 1977 and January 1979. The current system is in a stable state while work is currently going on towards rewriting a new system in PASCAL to eventually supersede this system.

Chapter 2 describes the AL programming system and the related hardware and software configuration at SAIL. It is an overview and description for the general reader.

Chapter 3 and the succeeding chapters are for the AL user. Chapter 3 is an example based tutorial illustrating the use of simple AL instructions. This chapter assumes that the reader is familiar with interactive computer programming in high level languages such as FORTRAN. Previous knowledge of manipulator programming or programming in ALGOL is not essential. After completing this chapter, the user should have at his command a subset of AL instructions which will enable him to write simple programs.

Chapter 4 describes the AL language, and gives the complete set of currently implemented AL instructions in a concise manner. This chapter should be sufficient for an experienced programmer to use as a reference manual.

Chapter 5 describes how to execute AL programs, the errors that might occur in the different stages of compiling and executing the AL program, and how to cope with them.

Chapter 6 describes the interactive AL system called POINTY, which allows the user to generate the frame tree data structure for AL programs, and to try out segments of AL programs.

The Appendices include reserved words, brief summaries of commands and instructions, extended examples and instructions on building up systems of AL and POINTY, references and acknowledgements.

1.2 Other implementations of the AL system

Ron Goldman is currently re-writing the AL system to produce a standalone version on a minicomputer. The system is being written in OMSI PASCAL and fully resident on a PDP-11 system. User programs may be written, compiled and executed on a PDP-11/45, and a separate processor, a PDP-11/60, will be used for servoing the arms.

The system will be interactive, and include desirable features from the current system in addition to other features that users have requested from time to time.

A version of the AL system has been implemented at the University of Karlsruhe in Germany by C. Blume who visited at the Stanford Artificial Intelligence Laboratory in the summer of 1979.

A high level robot language which resembles AL was designed and implemented in LISP at the University of Tokyo in 1981.

2. THE AL SYSTEM AT SAIL

2.1 Design philosophy of AL

2.1.1 Introduction and history

The WAVE system for manipulator control was designed and implemented by Lou Paul in 1973 on the Scheinman Stanford model arm and was used extensively by him and Bob Bolles.

The experience with WAVE led to the initial specifications of AL in 1974 by both of them and Jerry Feldman, Ray Finkel, and Russ Taylor.

The initial implementation of the compiler and runtime system for AL was done by Finkel and Taylor, and subsequently taken over by Ron Goldman, who is now developing a more highly interactive version of AL.

Vic Scheinman designed the two Stanford model arms in use at SAIL, while Tom Gafford and Ted Panofsky were responsible for the computer interface to the manipulator. Ken Salisbury, Gene Salamin, Lee Winnick and Jim Maples maintained the hardware at various times. Later work of Vic Scheinman led to the design of the PUMA 600 arms and the wrist force sensor currently in use. The first fingers for our PUMA arms were designed by Ken Salisbury. Currently, Jeff Kerr is designing hands with modular fingers suitable for interfacing to the PUMA arms while Ken Salisbury is designing a nine degree of freedom three fingered general purpose hand. Rick Vistnes wrote the software to interface the PUMA arms to AL. Salisbury and Shahid Mujtaba developed the tabletop flexible fixturing for accurate and repeatable relocation of workpieces and baseplates.

The work of Paul and Bruce Shimano was responsible for developing the kinematics of manipulation and the arm servo code. Shimano subsequently implemented force compliance, while Tatsuzo Ishida has done a theoretical analysis of two arm cooperative manipulation. Salisbury maintained the arm code and developed active force and stiffness control using the wrist force sensor and implemented automatic wrist calibration. Goldman implemented runtime trajectory computation. Oussama Khatib and John Craig are currently developing software for arm control in cartesian space.

The first AL parser was written by Bill Laaser and Pitts Jarvis, and subsequently taken over by Mujtaba.

POINTY, a related system, was conceived in 1975 by Dave Grossman and Taylor, and initially implemented by Taylor. Maria Gini, Pina Gini and Mujtaba have

subsequently implemented a newer version. Enrico Pagello has also contributed to it. Mujtaba has expanded it to execute AL source code interactively.

Barry Soroka has produced models of the arms and assembly station parts using the ACRONYM system for geometric modelling.

The design of AL has continually been modified and updated on the basis of new experience and information by Grossman, Shimano, Goldman, Mujtaba, Salisbury, Vistnes and Soroka under the overall guidance of Tom Binford.

The AL system is geared towards batch manufacturing where setup time is a key factor. To minimize programming time we rely on a symbolic data base and previously defined assembly primitives, and a quick and simple means of putting into a program the things we want to tell the manipulator to do. By testing out the system on undergraduate industrial engineering students with minimal experience in manipulators and robotics, we have found that learning to use the AL system is relatively simple, and that it is unnecessary to learn the complete system before putting it to use. Team programming sessions by researchers in manipulation at the Workshop on Software for Assembly held in November 1977 at SAIL showed the possibility of learning to program AL in a short time. The AL system has also been used for term projects in a Robotics course given in the Fall quarters of 1978, 1979 and 1980.

2.1.2 Plantime and runtime systems

Experience with WAVE (the predecessor to AL) had shown that calculating trajectories for manipulators was a desirable feature. It was thus decided that trajectory calculations, together with all other calculations which need only be performed once, should be done at compile time on the assumption that this allocation of effort would reduce the computing load at execution time and eliminate recomputation every time a sequence of actions is executed.

This sequence of planning and execution led to the existence of two systems - the plantime system and the runtime system in the initial implementation of AL. The plantime system consisted of the AL compiler whose function was to take the user written AL program, simulate it, point out errors to the user, and output instructions to the runtime system. The compiler performed a simulation of the program (called world modelling) to verify that it was indeed possible to do what the user asked within the limits of what AL was capable of doing, and to warn the user about unexpected consequences (e.g. if the user accidentally asked that the arm be moved through the table). The runtime system took the output of the plantime system, and proceeded to perform the motions.

This approach was changed because of subsequent developments. Computation costs have dropped dramatically and this makes possible the future

use of multiple processors in distributed computation. Better arm servo software, faster arm solution and more sophisticated path calculation algorithms tend to reduce the computation load, thereby permitting more decisions to be made at runtime. It was also realized that certain trajectories are best computed during runtime (e.g. force compliant motion, moving belt, when the workplace is highly unstructured). (See "Discussion of Trajectory Calculation Methods" by Mujtaba in Progress Report 4.). The sophisticated control structures of AL and the increased use of sensory information and runtime data increased the capability of AL, but at the same time presented increased problems to the world modeller. (See description in section 4.6 of the January 1979 edition of the AL USERS' MANUAL.) It was also found that computing trajectories at compile time generated very large user program object code files, with the result that user program size was limited. The current AL system no longer does world modelling; trajectory calculation is now done during runtime.

2.1.3 Data and control structure

The principal mode of input to AL is textual. The use of symbolic programming means that for parts in a pallet, for instance, there is no need to define the position of all the parts, if the distance between parts (which is usually constant) is known. "Once the corner of a pallet is taught and the part separation is known, laborious record-playback programming is no longer necessary given proper software in an associated minicomputer." [Engelberger, J.F. in "A Robotics Prognostication", Joint Automatic Control Conference Proceedings 1977, p 198.] Symbolic programming simplifies the interfacing of AL with other means of generating world models, like interactive graphics and computer aided design. It permits the setting up of library programs which may be called by supplying the relevant parameters. Symbolic programming eases the job of specifying complex motions if such motions can be parametrized or described algebraically - for example, it is easier to tell the hand to move a certain distance along an arbitrary direction than it is to move it manually when multiple joints have to be adjusted simultaneously. Teaching by doing, on the other hand, requires the recording of a very large number of points (tape recorder mode) unless only the end points of motions are of interest and the nature of the paths between these end points are unimportant.

There are levels of complexity which are much more readily transmitted from man to machine through an interface of symbolic text. Simultaneous motions of two arms, specifications for termination, and error conditions are more likely to be unambiguously stated through the medium of text, since these may require multiple logical relationships to be satisfied. Non-textual forms of input can be very useful for defining target locations, suggesting arm trajectories designed to avoid collisions, initial setup of a workstation, and other purposes of this nature.

AL has more data types than other conventional high level languages. In

addition to *SCALAR* numbers, it allows the specification of *VECTOR*, *ROTATION*, *FRAME*, *TRANS*, *STRING* and *EVENT* data types. A *VECTOR* consists of a triple of three real numbers. A *ROT* consists of a direction vector and an angle to indicate the amount of rotation. A *FRAME* describes the position and orientation of an object coordinate system, while a *TRANS* describes the relationships between *FRAMES*. In addition, arrays of all these data types may be defined. Arithmetic operators are available not only for the standard scalar operations but also for such operations as rotation and translation.

We want to write programs in a natural manner. The machine-language like aspect of previous manipulation languages made it cumbersome to write long programs in any structured way. We want a language which lends itself to a more systematic and easily understood programming style. To this end, the use of ALGOL-like control structures are an improvement over linear machine-language code with jumps. The block structure of ALGOL is also present in AL.

Experience with languages like SAIL and WAVE has shown that text macros are a useful feature; they reduce the amount of repetitive typing, and allow symbolic definition of constants and variables in a way which would be otherwise impossible. AL has a general-purpose text macro system. Procedures are provided, as in other languages, to reduce the amount of code when similar computations or operations need to be done at numerous places in a program.

AL also permits the control of parallel processes by allowing the flow of control of the program to be divided up, which allows certain operations to be performed simultaneously (e.g. simultaneous movement of different manipulators), after which the various processes merge back together. Synchronization primitives are also provided.

2.1.4 Motion of objects

AL has a mechanism to keep track of the location of a component piece of an assembly automatically as the assembly is moved; this mechanism is called affixment and used extensively with the concept of *FRAME* to describe objects. Frames may be affixed to each other, so that after affixing an object to the manipulator, the user can forget about the manipulator completely, and think in terms only of where objects have to interface with other objects. Instead of having to worry about how to move the arm, the user can specify the final orientation and position of the object, and AL will take care of working out what the arm has to do in order to accomplish the stated objective. The user can think of movement of the objects rather than the movement of the manipulator. This is significantly different from other programming schemes where the program consists of a series of arm motions whose relationship to objects in the real world is known only to the user, and where the user effectively has to provide explicitly the distance and angular relationship of the object to the manipulator for each motion

statement.

2.1.5 Sensory information

AL allows alternative actions on the basis of sensory input during runtime by checking whether certain conditions have exceeded a specified threshold, and if so executing a predetermined action. This is called condition monitoring. When error conditions are encountered, it is possible to set a sequence of actions into motion that will try to allow recovery. This is not possible in the case of linear control where program execution has to be aborted.

2.1.6 Programming aids

AL has several features that help the user during different phases of compilation and execution of his program to ensure that errors are caught as early as possible, and to simplify programming.

2.1.6.1 AL compiler

The AL parser takes the user-written AL program and checks that it is syntactically correct, generating error messages where necessary. It also makes use of the AL declarations generated by POINTY if told to do so. It enables programs to be input through disk files written by means of text editors, or through the teletype.

The AL parser tries to catch and correct errors early, so that less time is wasted on a compilation if it needs to be aborted. Also, by catching errors early, it is possible to generate error messages in the context of the user source program. Two main checks are used to eliminate an important class of errors. Dimension checking across assignments and expressions is done by the parser to ensure that units have been correctly specified by the user and are compatible with what is expected. Type checking across assignment statements and across the terms and factors of an expression ensure that operations are performed on arguments of the right data type, and that assignment of an expression is done to a variable of the same data type.

The output of the AL parser is used by the AL compiler to generate the binary file of pcode for the runtime system.

AL allows interactive error correction by permitting the user to ask for a standard fixup, or to change (patch) the offending source code for minor errors and continue from there without having to resort to the system text editor and a recompilation. Error recovery is local, and permits backing up to the beginning of the innermost current statement. To back up any further, it is necessary to resort to the text editor. At the user's option, a corrected copy of the source file is

made.

2.1.6.2 Interactive model building

The interactive AL system or source code interpreter, POINTY, to be described in detail in Chapter 6, allows the user to create interactively the frame tree for AL programs with the aid of the manipulator as well as to try out AL statements. The interactive nature of POINTY is also helpful in testing out small segments of programs before incorporating them in a larger AL program.

2.1.6.3 Debuggers

Several debuggers are available during execution of the program to enable the user to correct his mistakes by allowing him to patch his programs, and to let him examine and change the values of variables.

Debugging an AL program during execution involves examining and modifying variables, and single stepping through a program. This can be done in the interactive AL system by means of the source level debugger.

11DDT is the PDP-11 machine language symbolic debugger used by AL wizards to debug the runtime system, and by the user to continue or restart execution of his program.

2.2 AL system hardware

The hardware for the AL system consists basically of a PDP KL10 computer (hereafter referred to as PDP-10) for compiling and loading the AL program, a PDP-11/45 computer (hereafter referred to as PDP-11) for executing the AL program, two STANFORD model Scheinman arms, two UNIMATE PUMA 600 arms, an electric socketdriver, a Machine Intelligence Corporation VS-100 Vision Module, and an ADAC interface with 64 A/D channels and 4 D/A channels in addition to various peripherals such as terminals and disks.

The relationships between the various components are shown below, and diagrams of the PUMA and STANFORD arms are included. The PDP-11 system is interfaced to the PDP-10 system, the manipulators, ADAC interface and the vision module. The PDP-11 controls each joint of the STANFORD arms directly. Each joint of the PUMA arms is controlled by an individual microprocessor (a 6502) which is in turn controlled directly by the PDP-11. (It is possible to control the PUMAs through VAL by re-connecting the LSI-11 that comes with the PUMA controller.) Any communication between the PDP-10 and the manipulators must go through the PDP-11 runtime system, since there is no direct interface between the PDP-10 and the arms.

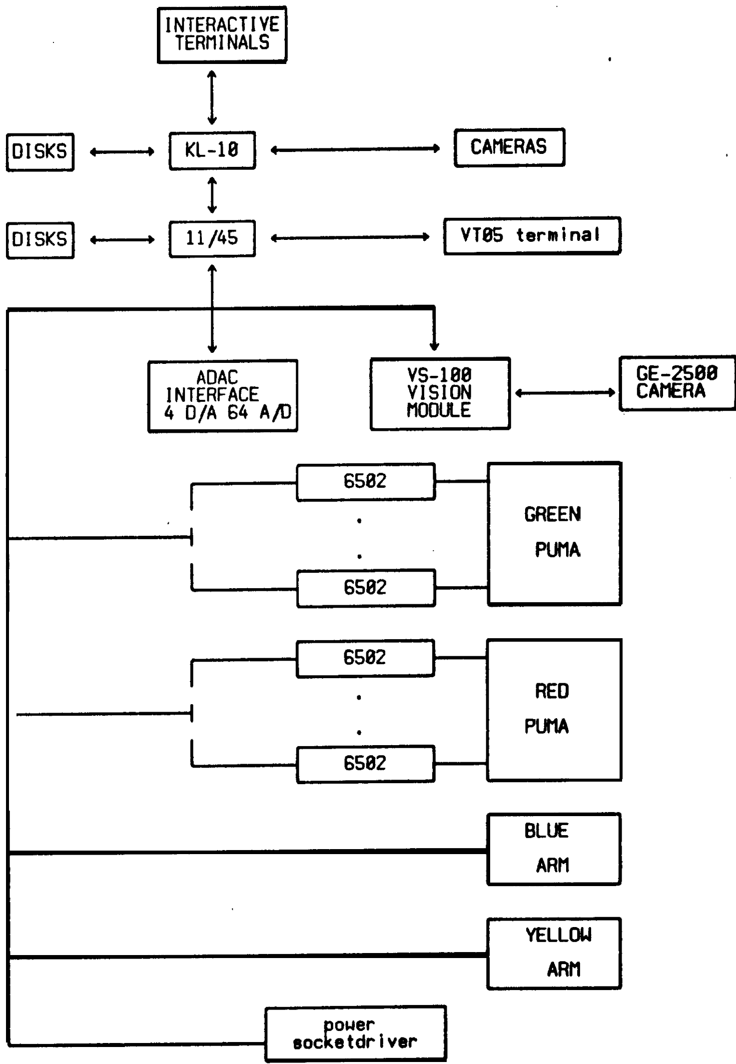


Fig. 2.1 Hardware setup for AL at SAIL

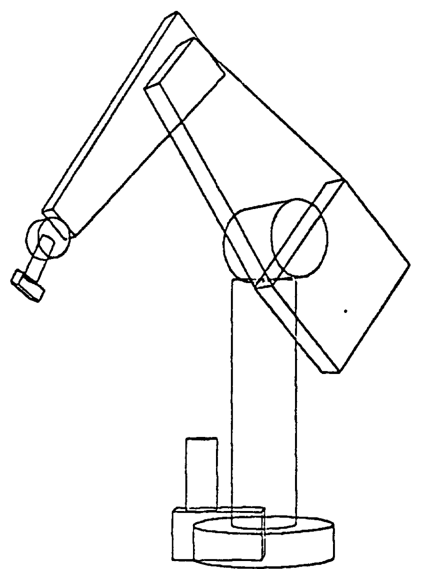


Fig. 2.2 PUMA arm

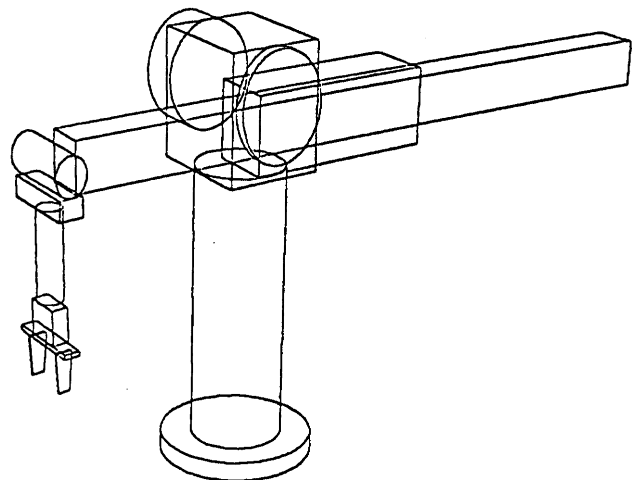


Fig. 2.3 Stanford Scheinman arm

2.3 Software

The software organization of the current AL system at SAIL is shown in Fig. 2.4. Each of the blocks indicates a module of programs that can be in core at one time, and the files that each module needs and generates are written alongside the lines connecting the modules.

Data and programs are stored in files which have names of the form ABCDEF.XYZ where ABCDEF is a combination of one to six alphanumeric characters making up the name, and XYZ is any combination of zero (in which case the dot is omitted) to three characters, making up the extension. The extension serves to distinguish different files in a family of files of the same name.

Affixment information can be generated in AL statement form using the interactive AL system, POINTY and saved in a declaration file. The program and data can be prepared and saved on a disk file FOO.AL (where FOO is the name, and .AL is the extension which serves to identify an AL source program) using the text editor. It can also be input directly to the AL compiler through the teletype.

The AL parser takes the AL program written by the user, and checks that it is syntactically correct, generating error messages where necessary. It generates an intermediate file (using S-expressions), with extension .SEX, that is passed to the AL compiler. At the option of the user, the AL parser will generate a logging file with extension .LOG with all the error messages, and a corrected copy of the source file with extension .NEW. For programs input directly through the teletype, a disk file copy of the program will be generated with extension .TTY.

The AL compiler reads in the S-expression file generated by the AL parser, changes it into an internal form, and deletes the S-expression file. It then does code generation, and generates a file with extension .ALP which contains information on the pcode and numerical constants. If information on the symbols is required as well, a file with extension .ALS is generated.

The file with extension .ALP is used by PALX, the PDP-11 cross assembler to assemble a binary load module having extension .BIN for the runtime system. The intermediate file with extension .ALP is typically deleted by the ALSOAP program, which is run automatically after the AL compilation.

The binary file with extension .BIN is loaded together with the AL pcode interpreter and the run time system by a program called 11TTY. If desired, a force graphics program, GAL, can be run on the PDP-10 to display forces encountered by the arms.

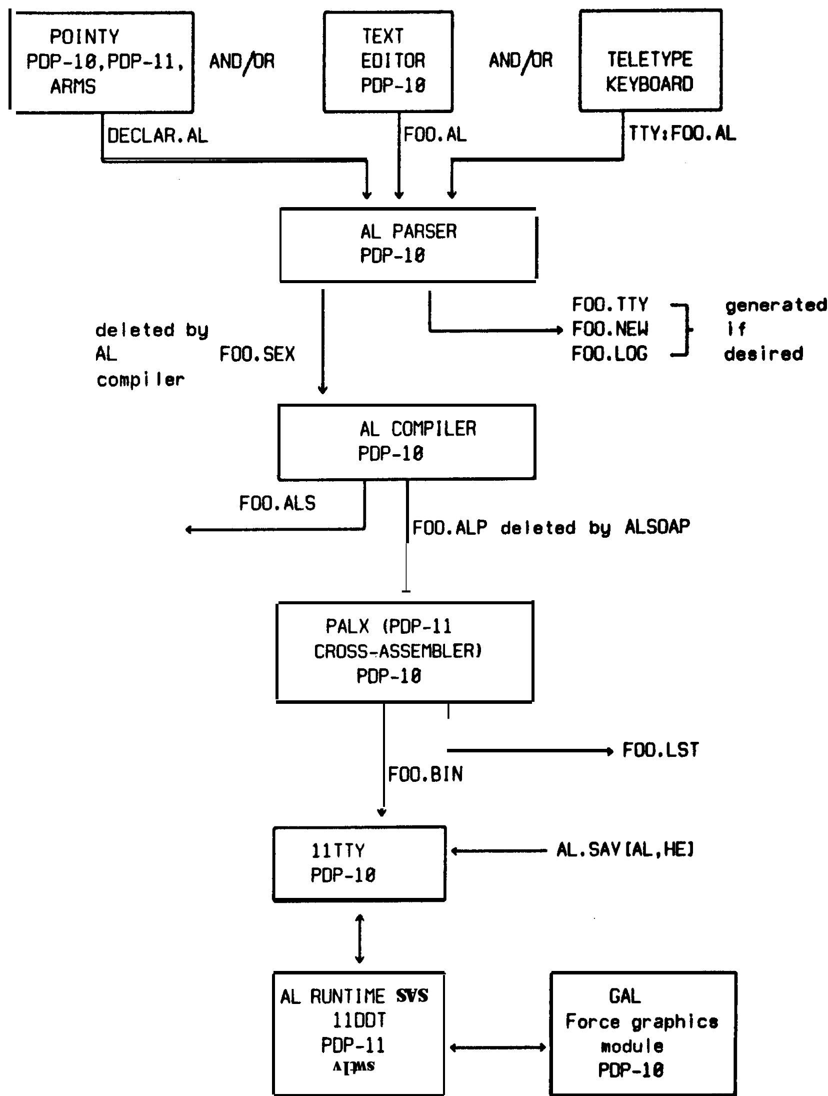


Fig. 2.4 Software organization

2.4 Programming in AL

In order to program an assembly in AL, an assembly plan should first be worked out which includes a rough layout of the parts and the sequence of motions to accomplish the assembly.

The parts and fixtures should be laid out in the work place in the desired physical locations. AL has to be given the information of the object layout, and this can be done either by direct measurement using a ruler and other measuring equipment, or with the aid of manipulators and POINTY, the interactive AL system using the manipulator (c.f. Chap 6). The data must be incorporated into a file which has AL statements which specify how to move the parts to accomplish the desired assembly.

Having obtained the program, the user gets it into the computer system by some means (at SAIL this means through one of the interactive terminals).

The program is compiled, loaded, and executed and debugged much like any other program.

3. AL TUTORIAL

3.1 Basic constructs

The purpose of this chapter is to introduce the reader to the AL language, and through a series of examples, show its use in the programming of manipulator motions. The basic constructs of the AL language are described in this section. Other instructions will be described in the the following sections of this chapter.

The notation will be as follows: Within the programs and examples reserved words will be shown in upper case, while variables and predefined constants will be shown in lower case. In all other places, they will be represented in upper and lower case italics respectively.

3.1.1 Data types

At the heart of each computer language are the types of data that can be handled. For example, FORTRAN has INTEGER and REAL numbers; other languages can handle strings of alphabetic characters. The data types in AL were chosen to handle the special problems that arise in controlling manipulators, and in working with three-dimensional objects in the real world which have directed distances, locations and orientations.

A variable is an identifier that can take on various values. Identifiers consist of a string of alphanumeric characters (letters and numbers) and underscore "_". Some examples: *pump_base*, *handle*, *screw_hole_2*, and *P132*. Note that all identifiers must start with a letter (*3inch_screw* is no good). Upper and lower case are equivalent, i.e. *ABC*, *abc*, and *aBc* all refer to the same variable.

Variables can be given a value by means of an assignment statement, which consists of the variable name, a left arrow ("←"), and an expression of the correct type. When an assignment statement is executed, the expression on the right hand side is evaluated, and the result becomes the new value of the variable on the left hand side.

AL, like ALGOL, requires each variable to be declared, that is, one must state what data type a variable is before it is used. AL also uses ALGOL type block structure which means that all variables declared between a particular *BEGIN* and *END* are accessible only to code which appears between the same *BEGIN-END* pair. It is also possible for the same variable name to be used in different blocks without conflict. Block structure will be explained more fully later (3.1.2). We shall now look at the data types available in the AL language.

3.1.1.1 *SCALARS*

The most elementary data type in AL is the *SCALAR*, which is internally represented as a floating-point number. Scalars can be used for dimensionless quantities, such as the number of times some operation is to be repeated, or for dimensioned ones like the length of an object or the angle between two parts. The arithmetic operations available on scalars are addition, subtraction, multiplication, division and exponentiation, represented by the normal arithmetic operators: "+", "-", "*", "/", and "↑". Exponentiation has precedence over multiplication and division which in turn have precedence over addition and subtraction, as in other algebraic languages. Several commonly used functions are also available: the square root function, *SQRT*; the trigonometric functions *SIN*, *COS*, *TAN*, *ASIN*, *ACOS* taking one argument, and *ATAN2* taking two arguments; the natural logarithm *LOG*; and the exponential function *EXP*.

Scalar constants are written as (base ten) numbers, possibly with a decimal point or fractional part; for example 2, 1., 3.14159, -123.45 are all scalar constants.

Below is an example showing the declaration and use of scalar variables. In the examples in this section we will use a mnemonic scheme for naming variables to clarify the type of each entity. Note that AL statements are separated by semicolons. Also curly brackets "{}" are used to enclose comments.

```
SCALAR s1, s2;          {A declaration consists of a data type followed by
                         a list of variable names separated by commas, and
                         ending with a semicolon.}

s1 ← 2;
s2 ← 3.50;              {s1 has the value 2.0, and s2 is 3.50}
s1 ← s2 * (s1 - 3.2);   {Now s1 = -4.20}
```

It is often desirable to associate a physical dimension with a variable. AL provides for scalars with the dimensions of *TIME*, *DISTANCE*, *ANGLE*, and *FORCE*. It should be noted that *ANGLE* is generally considered dimensionless, but that for our purposes, the definition has been made a little flexible to allow for an entity which is useful for defining rotations and for checking that angular quantities are handled correctly. Dimensioned variables are just like regular scalar variables, except that they are associated with an appropriate dimensional unit: *sec*, *inches*, *deg* or *ounces*, which have the obvious meanings. AL can also handle *cm*, *oz*, *lbs*, *gm*, *rpm* and *radians*.

Dimensioned variables are used exactly in the same way as simple variables, except that AL checks for consistent usage. Dimension checking is done for each arithmetic operation and each assignment. Addition, subtraction and assignment

require exact dimensional match, though if the match fails and one of the two arguments is simple (dimensionless), it will be coerced to the right type, after an appropriate message to the user. Multiplication and division do not require dimensional match; they produce a result of a dimension different from that of the arguments which is then propagated through the expression. In this way intermediate results can be of dimensions not declared. This causes no problems unless such results are used in an assignment. The square root function may be used on scalars of arbitrary physical dimensions, and the dimensions of the result will be the square root of that of the argument. The *SIN*, *COS* and *TAN* functions are applied to scalars having dimensions of *ANGLE* and assumed to have units of degrees. The result is dimensionless. The inverse functions *ASIN*, *ACOS*, and *ATAN2* take dimensionless arguments; the resulting value has dimensions of *ANGLE* and units of *DEGREES*. The exponential and logarithmic functions take dimensionless arguments and return dimensionless values. The exponentiation operator presents a problem for the parser, since during parsing, the value of the power to which the base is raised is unknown. The problem is recognized by giving an error message if either the base or index is not dimensionless.

Here is a short example using dimensioned scalars and functions.

```
SCALAR s1, s2;
DISTANCE SCALAR ds1;
TIME SCALAR tm1, tm2;
FORCE SCALAR fs1;
ANGLE SCALAR theta, phi;

ds1 ← 1.0 * inch;
tm1 ← 3 * sec;
fs1 ← 2.2 * ounces;
tm2 ← tm1 + 4.5;          {The constant 4.5 will be converted to
                           seconds after the relevant error message.}
theta ← 90 * deg;
phi ← theta * 4 * deg;    {This is a mistake: the right hand side has
                           dimension ANGLE * ANGLE.}
s1 ← SIN(30 * deg);
theta ← ACOS(.7);
ds1 ← SQRT(ds1 * 3 * inches );
phi ← ATAN2(s1, s2);      { same as arctangent(s1/s2) }
s1 ← LOG(33.0);
s2 ← s1 ↑ 3 ;
```

There are several predeclared scalars in AL:

```
SCALAR PI;                {3.14159...}
```

π is also recognized as the constant 3.14159....

```
DISTANCE SCALAR bhand, yhand;
                                  {These variables refer to the blue hand and
                                  yellow hand openings}
```

VELOCITY, *ANGULAR_VELOCITY*, and *TORQUE* are defined in terms of the primary dimensions in the generally accepted way.

It is also possible to define new dimensions, such as *acceleration*, by means of the dimension statement. New dimensional units, such as *feet*, can be defined with macros (c.f. section 4.5.8). For instance:

```
DEFINE feet = ⊂(12 * inches)⊃;
DIMENSION acceleration = VELOCITY / TIME;

acceleration SCALAR as1;
as1 ← 6.7 * feet /( sec * sec );  {= 6.7 * 12 * inches/sec/sec}
```

3.1.1.2 *VECTORS*

The world in which AL programs operate has three dimensions, and so we need more than just scalars. We will now introduce another data type: the *VECTOR*. It and the other algebraic data types which follow are similar to scalars in how they comprise arithmetic expressions and assignments.

We describe the world as a Euclidean space with three cardinal orthogonal axes, which meet at an origin. The actual alignment of these station axes is implementation dependent, though at SAIL and for the rest of this manual it will be assumed that the positive Z axis points upwards.

Vectors may represent entities having both direction and magnitude, e.g. displacement, velocity, acceleration. Like scalars, they may be dimensioned. Vectors can be constructed from three scalar expressions by means of the function *VECTOR*. The scalar expressions must all be of the same dimension, which the resulting vector will also have.

The available operations between vectors include addition, subtraction, dot product, and cross product. A vector may be multiplied or divided by a scalar. The direction unit vector (dimensionless) may be extracted by the function *UNIT*. Addition and subtraction are defined only on vectors of the same dimension. The dot product, cross product and multiplication by a scalar give results having the dimensions which are the product of the dimensions of the two arguments. The scalar magnitude of a vector is obtained by enclosing it within vertical bars. The operators are defined in the normal manner; for example, if we have a scalar *s* and two vectors:


```
v1 = VECTOR(x1, y1, z1) and v2 = VECTOR(x2, y2, z2)
```

then we have:

```
s * v1 = v1 * s = VECTOR(s * x1, s * y1, s * z1)
v1 + v2 = VECTOR(x1+x2, y1+y2, z1+z2)
v1 - v2 = VECTOR(x1-x2, y1-y2, z1-z2)
v1 . v2 = x1*x2 + y1*y2 + z1*z2
```

There are several predeclared vectors in AL:

```
VECTOR xhat, yhat, zhat, nilvect;      {These have values as follows}
xhat ← VECTOR(1,0,0);
yhat ← VECTOR(0,1,0);
zhat ← VECTOR(0,0,1);
nilvect ← VECTOR(0,0,0);
```

Here is one more example of the use of vectors:

```
VECTOR v;
DISTANCE VECTOR dv1, dv2, dv3;
SCALAR s;
DISTANCE SCALAR ds1, ds2;

ds1 ← 2 * inches;
dv1 ← VECTOR(4, 2, 6) * inches;
ds2 ← dv1 . yhat;                {So ds2 = 2 * inches}
v ← VECTOR(2, 1, 3);
v ← v - zhat;                    {So v = VECTOR(2, 1, 2) }
dv2 ← VECTOR(3, 0, 4) * inches;
ds1 ← |dv2|;                     {This assigns ds1 the magnitude of
                                 the vector dv1, which is a scalar of
                                 the appropriate dimension. So ds1 = 5
                                 * inches.}
dv3 ← VECTOR(4*inches, 2*inches, 6*inches); {dv3 is the same as dv1}
v ← UNIT(v);                     {So v = VECTOR(2/3, 1/3, 2/3) }
```

3.1.1.3 *ROTATIONS*

The next data type we will discuss is the rotation, or *ROT*, which represents either an orientation or a rotation about an axis. Rotations can operate on vectors and rotate them around the origin (without changing their length). They can also operate on other rotations (by matrix multiplication). To rotate a vector (about the station origin), multiply the vector (on the right) by the rot (on the left).

To compose rots, multiply them together; the one on the right will be applied first. The axis of rotation can be extracted by the function *AXIS* and the angle of rotation by enclosing the rotation expression within vertical bars. Rotations are dimensionless, and the user may not specify dimensions for this data type; however the amount of rotation about the axis has units of *ANGLE*.

A rotation can be constructed with the function *ROT*, which takes two arguments: a simple vector, which is the axis of rotation, and an angle, which is the amount to rotate. The direction of rotation follows the right hand rule, so a rotation of 90 degrees about the X axis moves the Y axis into the Z axis. This representation is far easier to write and understand than raw matrices. Here is an example showing the use of rotations:

```
ROT r1, r2, r3, r4;
ANGLE SCALAR alpha, beta, gamma;
VECTOR v;
r1 ← ROT(xhat, 90 * deg);
v ← r1 * zhat;      {v gets Z rotated 90 degrees about X, so v =
                    VECTOR(0,-1,0) }
r2 ← ROT(yhat, 45 * deg);
r3 ← r2 * r1;
      {Thus, r3 means first rotate 90 degrees about the X axis, then
      45 degrees about the original Y axis. An alternative
      interpretation is to first rotate by 45 degrees about Y, and then
      to rotate by 90 degrees about the new X axis.}
v ← AXIS(r2);       {This assigns v the axis of rotation of r2 = yhat.}
alpha ← |r2|;       {This assigns alpha the angle of rotation of r2 = 45
                    degrees.}
r1 ← ROT(xhat, alpha);
r2 ← ROT(yhat, beta);
r3 ← ROT(zhat, gamma);
r4 ← r3 * r2 * r1;
      {r4 is then a rotation with the following two meanings: Rotate
      by alpha degrees about the X axis of the station, then by beta
      degrees about the station's Y axis, and finally by gamma
      degrees about the station's Z axis. Or alternatively, rotate by
      gamma about the station's Z axis, then by beta about the new Y
      axis, and finally by alpha about the doubly new X axis. Both of
      these interpretations yield the same result; use whichever one
      you find most comfortable.}
```

There is one predeclared rot, called *nilrot*, defined as *ROT(zhat, 0 * deg)*.

3.1.1.4 *FRAMES*

In working with objects in the real world we need to specify both their position and orientation. To do this we introduce a new data type, the *FRAME*, which represents a coordinate system. It has two components: the position of the origin (a distance vector) and the orientation of the axes (a rot). Features on an object can be specified with respect to the object's coordinate system. The term location, where used, will refer to the position and orientation of a frame.

There are several predeclared frames in AL. *Station* represents the reference frame of the work station. Associated with each manipulator is a frame whose value (updated at the end of each motion) is the position of that manipulator. Currently, there are four such frames: *barm*, *yarm*, *garm* and *rarm*, associated with the blue, yellow, green and red arms respectively. Also associated with each arm is a rest, or park position; these are *bpark*, *ypark*, *gpark* and *rpark* respectively.

A frame may be constructed by calling the function *FRAME*, which has two arguments: a rot (for the orientation) and a distance vector (for the position). The orientation or position of a frame can be extracted by the functions *ORIENT* and *POS*. To transform a point specified by a distance vector in the coordinate system of some frame into station coordinates, multiply the frame (on the left) by the vector (on the right). To translate a frame by some amount, simply add/subtract a distance vector to/from it. Finally, to construct a vector in station coordinates which has the same orientation as a vector in some frame, such as *xhat* in say *f1*, the "with respect to" operator *WRT* is used and one writes *xhat WRT f1*. For any vector *v* and frame *f* the following are equivalent (the dimensions of the result are the same as those of *v*):

v WRT f ≡ (f * v) - POS(f) ≡ ORIENT(f) * v

Here are a few examples using frames.

```
FRAME f1, f2;
f1 ← FRAME( ROT(zhat, 90 * deg), 2 * xhat * inches);
      {The frame f1 sits 2 inches from the station in the X direction.
      Its coordinate system has X where the station's Y axis points.}
v1 ← xhat WRT f1;          {This evaluates to VECTOR(0,1,0).}
f2 ← f1 + v1 * inches;     {Just like f1, but with origin at (2,1,0).}
v2 ← f1 * (zhat * inch);   {This evaluates to VECTOR(2,0,1).}
```

3.1.1.5 *TRANSFORMS*

The last of the algebraic data types is the transformation or *TRANS*. Transes are used to transform frames and vectors from one coordinate system to another. Like frames, they consist of two components: a rotation and a vector. The application of a trans first rotates its operand about the station origin, and then translates the result. Transes can be composed in the same manner as rotations, the one on the right being applied first.

A trans consists of a rotation part having units of angle and a translational (vector) part having some other physical unit - usually distance. When "multiplying" by a trans, one is really multiplying by the rotational part and then adding the vector component. The matrix operation of multiplying transes together produces a trans. The vector parts of two transes multiplied together must have the same dimensions, and the vector part of the product will have the same result. For convenience, we will refer to the dimension of a trans as being that of the vector part. When a trans is applied to a vector, both must have the same dimension, the one for the trans being defined above. The resulting vector is of the same dimension. When a trans operates on a frame, it must be a distance trans. When transes are composed, they must agree in dimension, and the result will have the same dimension. Unless declared otherwise, transes will be assumed to have dimensions of distance.

One can construct a transform by use of the function *TRANS*, which takes two arguments: a rot (the rotational part) and a vector (the translational part). Another convenient way to specify a trans is by forming it from two frames. The arithmetic operator "→" applied to two frames produces a trans which takes the origin of the first frame across to the origin of the second, performing a rotation first to get the axis aligned. When a frame is used in a context demanding a transformation, it will be understood as a shorthand for the distance trans leading to it from the station.

Here are a few examples using transes.

```
TRANS t1, t2, t3, t4;
t1 ← TRANS( ROT(xhat, 30 * deg), 2 * zhat * inches );
v1 ← t1 * yhat * inches;
                    {t1 rotates yhat 30 degrees about the X-axis, and then
                     translates it by 2 inches along Z = (0,.866,2.5).}
t2 ← f1 → f2;       {Thus f1 * t2 = f2.}
v2 ← t2 * (xhat * inches);
                    {v2 is f2's x-axis as seen from f1}
t3 ← t2 * t1;       {t3 means to first perform the transformation given by t1,
                     and then that specified by t2.}
f3 ← f1 * f2;       {This expresses the position of f2 in f1's coordinate
                     system. Equivalent to (station → f1)*f2.}
t5 ← INV(t1);       {This expresses the inverse transformation of t1.}
```


The null transformation, equivalent to *TRANS(nilrot,nilvect)*, is called *niltrans*.

The initial distinction between frames and transes has lessened as work with AL has progressed. The current distinction is that frames may be affixed to each other. In general a trans can appear anywhere a frame can, and vice versa. For example to get at either of a trans's two components the extraction operators, *ORIENT* and *POS*, would be used. Whether or not the two data types will be merged remains to be seen. An evolving view considers frames to be labels associated with physical objects or locations in space and transes the relationship between these physical objects. In such a case, frames would not have dimensions associated with them, but there will be some relationship between them and other frames.

3.1.2 Block structure - i.e. "what's a program"

An AL program consists of a sequence of statements which will result in the manipulator successfully performing a desired task. While the simplest AL program consists of a single simple statement, any reasonable program will be made of many statements *S1, S2, S3, ...* separated by semicolons, and surrounded by the reserved words *BEGIN* and *END*. This composite arrangement of

```
BEGIN
   S1;
   S2;
   .
   .
   Sn
END
```

is known as a block statement. The statements *(S1, S2, ...)* within the block may themselves be other block statements. Indentation has no effect on the program and serves only to make the program more readable.

In order to keep track of blocks within other blocks, they may be named with strings within double quotes immediately following the *BEGIN* and the corresponding *END*. The strings after a corresponding *BEGIN* and *END* pair should be the same, or there should be no string after the *END*; otherwise there will be an error message. The following is an example of block naming:

```
BEGIN "MAIN"
    S1;
    S2;
    BEGIN 'INN □
       S3a;
       S3b;
    END "INNER";
    S4
END
```

Like SAIL or ALGOL, AL requires that an identifier be declared before it is used. The effect of an identifier is only within the block it is declared. Outside the block, any reference to those identifiers will give an error message. An error message will result if the same identifier is declared more than once in a given block, unless subsequent declarations are within blocks internal to the given block. Consider the following example:

```
BEGIN "BLK_1"
   SCALAR i,k,m;
   i←1;
   BEGIN "BLK_2"
      SCALAR i;   {denotes a new variable "i" distinct from
                        the "i" declared in BLK_1 above}
      i←2;
      m←i;        {So m=2; i refers to the second declaration of i}
   END "BLK_2";
   k←i;           {So k=1 since after exiting "BLK_1" i = 1 again}
END "BLK_1";
```

In the inner block *"BLK_2"* the variable *i* is a new variable distinct from the ε defined in *"BLK_1"*. Had the *SCALAR* statement been absent in block *"BLK_2"*, the value of *k* and *i* at the end of execution would have been **2**.

3.1.3 A simple program

As mentioned before, an assignment statement consists of a variable name, a left arrow ("←"), and an expression of the correct type. When an assignment statement is executed, the value of the expression on the right hand side is computed, and the result becomes the new value of the variable on the left hand side. Care must be taken to ensure that the data type of the expression is the same as that of the variable. During compilation, AL will check for type and dimensional consistency across opposite sides of the left arrow, and complain if it finds any incompatibility.

The print statement prints out the values of the variables and the strings during execution time. It consists of the reserved word *PRINT* followed by an open parenthesis, a list of arguments separated by commas and a close parenthesis. The arguments may be variable names or the names of predefined

constants, or they may be string constants which consist of characters enclosed by double quotes.

Here is a simple AL program that will compute and print out the current arm locations and the distance between them:

```
BEGIN
   DISTANCE SCALAR s1;
   DISTANCE VECTOR v1;
   PRINT ("THE BLUE ARM IS AT ", barm);
   PRINT ("THE YELLOW ARM IS AT ", yarm);
   v1 ← POS(barm) - POS(yarm);
                       {v1 is the vector ● between the centers of
                        the hands}
   s1 ← | v1 |;        {s1 is the absolute distance between the hands}
   PRINT ("THE DISTANCE BETWEEN THE BLUE AND YELLOW FINGERS IS ",
      s1, " INCHES");
END
```

Other statements possible within a block will be discussed in the following sections.

3.2 Simple move statement

The simplest motion program is one which will move an arm to a known location. When the two Stanford arms *barm* and *yarm* are not in use they are placed in statically balanced positions with the fingers pointing downwards so that a power failure does not result in the arms collapsing. The two PUMA arms *garm* and *rarm*, when not in use, are placed with the arms and fingers pointing straight up so that they are out of the way of the other arms and the user. (Since power must be applied to release the brakes, there is no danger of the PUMAs collapsing during a power failure.) The resting positions of the arms with the described pointing directions (orientations) of the fingers are known as *bpark*, *ypark*, *gpark* and *rpark* respectively.

For purposes of this document when we refer to an arm we shall mean the blue arm unless otherwise obvious from the context.

Let us assume that the arm is in any arbitrary position, and we want to move it to the park position under computer control. The statement to do this would be

```
MOVE barm TO bpark;
```

During execution, AL works out a trajectory (the position of each of the joints from the initial value to the final value as a function of time) from the current location to the park location so that the motion is accomplished gracefully subject to the constraints of maximum acceleration and torque imposed by the

motors.

It is also possible to specify differential motions. The grinch sign, "⊕", is used to represent the current position of the arm. The following statement would cause the arm to move down 2 inches.

```
MOVE barm TO ⊕ - 2 * zhat * inches;
```

3.2.1 More about *barm* and *bpark*

Let us now consider *bpark* and *barm* for a moment. *Bpark* specifies completely the way the arm is to be parked. It specifies the center of the hand by giving the cartesian coordinates, and in addition it indicates that the hand is pointing downwards. Since there are six joints, specifying only the cartesian coordinates of the hand is insufficient since it is possible to have an infinite number of different hand orientations with the center of the finger tips in the same position.

Barm is the name of a coordinate system whose origin lies centrally between the fingers of the hand, and whose z-axis points in the same direction as the fingers, the y-axis passes through the centers of the fingers, and the x-axis is determined from these two axes by use of the right hand rule (fig. 3.1). The value of *barm* depends on the position and orientation of the hand, and consists of a vector which defines the position of the center of the hand in the world coordinate system, and a rot which defines how the arm coordinate system is rotated in terms of the coordinate system of the station. *Station* is the frame which is the reference coordinate system, and the vector part is set at (0,0,0). Our station coordinate system has the z-axis pointing upwards, the y-axis horizontal and parallel to the short side of the table and pointing towards the window (i.e in a direction pointing from the pedestal of the yellow arm to the pedestal of the blue arm). The x-axis is horizontal and parallel to the long side of the work table, and points towards the far wall (fig. 3.2).

In the park position the hand points downward with the center of the hand at coordinates (43.53, 56.86, 9.96) * inches. The coordinate system is rotated 180 degrees about the y-axis. Thus the value of *bpark* is as follows:

```
FRAME( ROT(YHAT, 180*degrees), VECTOR(43.53, 56.86, 9.96)*inches )
```

The instruction *MOVE barm TO bpark* has the effect of moving the coordinate system whose name is *barm* to the new position and orientation described by *bpark*.

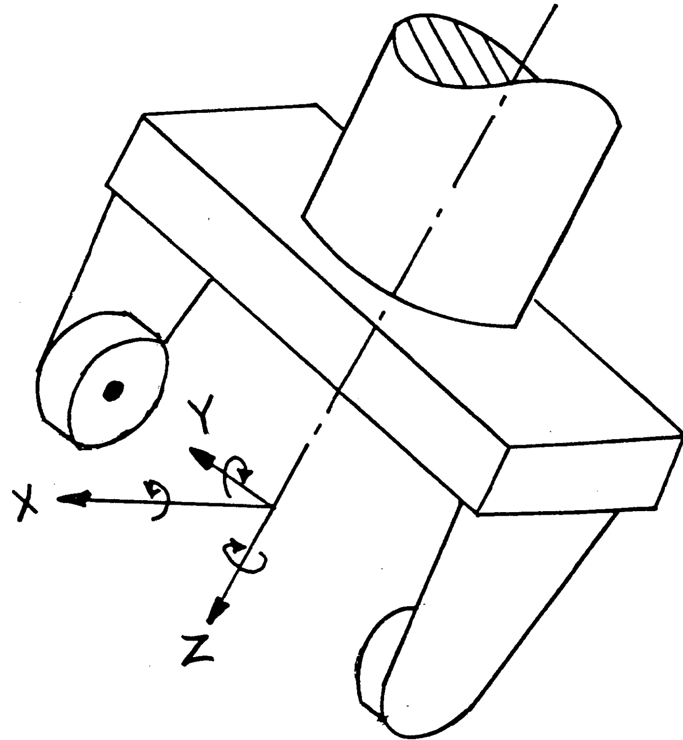


Fig. 3.1 Diagram of hand showing coordinate axes

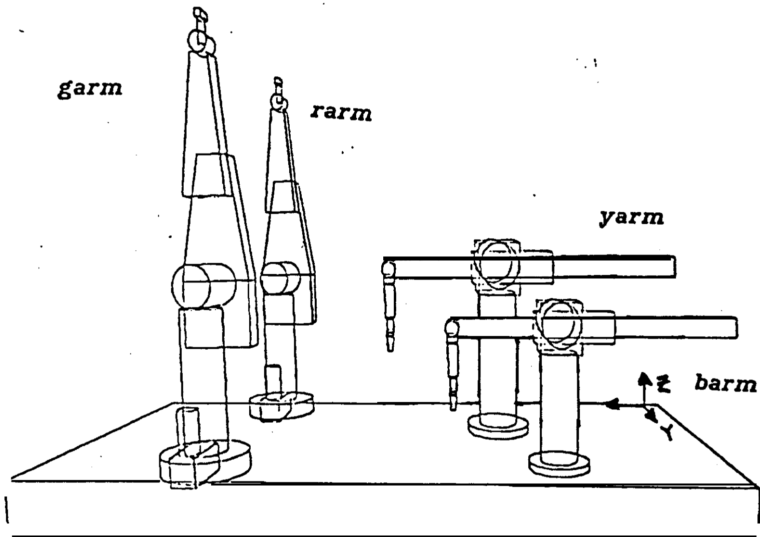


Fig. 3.2 Diagram of four arms on table

3.3 Using the fingers: *OPEN*, *CLOSE* & *CENTER*

Our manipulator end effector (hand) consists of two fingers which can move together or apart when instructed to do so by the *OPEN* or *CLOSE* command, which specifies the width to which the hand opening must go. An example of this particular instruction is

```
OPEN BHAND TO 2.5*inches
```

The general form of the instruction is

```
OPEN  <hand> TO <scalar_exp>
CLOSE <hand> TO <scalar_exp>
```

where <hand> is either of the reserved words *bhand* or *yhand*, and <scalar_exp> consists of a scalar expression of dimension distance, i.e. its units should ultimately be reducible to inches or cm or some such unit of measure of distance. For the PUMA arms it is not possible to specify a specific opening width since they are currently binary devices: either fully open or closed. Therefore, for *ghand* and *rhand* the *TO <scalar_exp>* is not needed, e.g.

```
OPEN ghand
CLOSE rhand
```

The *OPEN* or *CLOSE* instruction moves both fingers simultaneously at the same speed. The *OPEN* command will open the hand to the desired size. The *CLOSE* instruction will keep on moving the finger until the touch sensors trigger, and signal an error if the hand opening is smaller than the desired opening. (The *CLOSE* instruction will be implemented in the near future.) If there is a heavy object between the fingers, the fingers or motors might get damaged, while a light object may get moved by the fingers. The *CENTER* command prevents these undesirable results by causing the fingers to move toward each other slowly until one of the touch sensors triggers to let the system know that contact has been made with the object. At this point the whole arm will shift to maintain the position of the finger which is in contact with the object, and the cycle of moving fingers and arm will continue until both touch sensors trigger. When this happens, the new position of the arm can be read to determine the position of the object. Note that the *CENTER* command does not "center" the object between the fingers, but rather ensures that the hand grasps the object without moving the object. The *OPEN* and *CLOSE* commands are used when the position of the object to be grasped is known precisely or when the object is to be moved to a precise spot. The *CENTER* command takes an arm as its argument as follows.

```
CENTER <arm>
```

Note that *CENTER* does not currently work for the PUMA arms.

The use of these statements will be illustrated in the following example used to grab a 2-inch cube, move it over 10 inches in the X direction, and then release it.

```
BEGIN
   FRAME cube, new_place;
   cube ← FRAME(ROT(XHAT,180*deg), VECTOR(20,30,1)*inch);
                                  { defines position of cube center }
   new_place ← cube + 10*xhat*inches;
   OPEN bhand TO 3*inches;     {open the hand }
   MOVE barm TO cube;          { get arm to cube }
   CENTER barm;                { grasp cube without moving it }
   MOVE barm TO new_place;     { put the cube where we want it }
   OPEN bhand TO 3.0 inches;   { open the hand, releasing the block
   MOVE barm TO bpark;         { all done, park the arm }
END
```

3.4 Intermediate points - *VIA*, *APPROACH* and *DEPARTURE*

Many objects have shapes which necessitate care as the arm approaches or departs from them. The motion clause *WITH APPROACH = appr* will cause the arm to approach its destination after having passed through the point determined by vector *appr* in the coordinate system of the destination. In station coordinates this point would be *dest + appr WRT dest*. The motion clause *WITH DEPARTURE = depr* similarly specifies a departure point. Section 4.4.2 indicates the effect of *appr* or *depr* taking on non-vector values.

If no approach point is given, a default approach of 3 inches along the Z axis of the station will be used. If no departure point is specified, the approach point from the last motion, if any, will be used. Approach points relate to the destination of the current move command, while departure points relate to the starting position of the arm for the current command. To move the arm directly from the frame position at the beginning of the motion, the clause *WITH DEPARTURE = NILDEPROACH* should be used. To move the arm directly towards the desired frame position indicated in the current statement, the clause *WITH APPROACH = NILDEPROACH* should be used.

If the destination is a frame constant or expression then *NILDEPROACH* will be the default approach point.

The predeclared macro *DIRECTLY* will accomplish the same purpose as the two clauses

WITH APPROACH = NILDEPROACH
WITH DEPARTURE = NILDEPROACH

The *APPROACH* and *DEPARTURE* clauses allow the user to specify at most a three segment motion - from the current position to the departure point, from the departure point to the approach point, and from the approach point to the destination. Usually these intermediate points are in terms of the coordinate system of either the current position or the destination.

Sometimes it is necessary to move an object through additional locations in space, or to have more than the three segment motions described above. Examples are cases where objects in the way of the moving manipulator have to be avoided, or the arm has to pass through an opening. In such situations the *VIA* clause may be used to specify the frames through which the arm must pass.

In this example, the arm picks up a brick on the ground and places it on the floor of the oven, which is at the same level as the ground, but the arm has to pass through the open door which is above ground level.

```
BEGIN "Put brick into oven"
   FRAME brick, oven, oven_door;
   brick←FRAME(ROT(yhat,90*degrees) ,VECTOR(10,30,3)*inches);
                              { define initial position of brick }
   oven←FRAME(ROT(yhat,90*degrees),VECTOR(10,40,3)*inches);
                              { define final position of brick )
   oven_door←FRAME(ROT(yhat,90*degrees),VECTOR(15,40,4)*inches);
                              { define position of oven door }

   OPEN bhand TO 3*inches;
                  {make sure hand opening is large enough}
   MOVE barm TO brick
      WITH APPROACH = 3*zhat*inches;
                  { so for brick with hand in horizontal position,
                    note that brick z-axis is parallel to station x-axis}
   CLOSE bhand TO 1.7*inches;
                  ( grasp the brick }
   MOVE barm TO oven VIA oven_door
      WITH DEPARTURE = -3*xhat*inches;
                  { move brick into oven through oven door after lifting vertically }
   OPEN bhand TO 3.0*inches;
                  { release the brick }
   MOVE barm TO bpark VIA oven_door;
                  { go park the arm }
END
```

3.5 Modelling objects - affixment & indirect moves

Since assembly often involves attaching one object to another, AL has an automatic mechanism to keep track of the location of a subsidiary piece of the assembly as the main assembly is moved; the mechanism is called affixment. For example, there might be a frame called *pump* and another called *pump_base*. At some stage in the assembly, the *pump* is bolted to *pump_base*. At this time it is appropriate to execute the statement

AFFIX pump TO pump_base

This statement informs AL that motions of the *pump_base* are to affect the location of *pump*. Note that the *AFFIX* statement does not call any routines to generate the code to actually perform the bolting operation. The statement merely informs AL that at this stage in the program execution, *pump* is to be considered affixed to *pump_base*.

The particular case in which object frames are attached to the arm frame is of special importance. Once *pump* is affixed to *barm*, for instance, the user can forget about the arm, and just concentrate on where and how *pump* has to move; AL will take care of how to move the arm to achieve the desired result. This is an indirect move where the user need not specify arm motion.

When affixing frames to one another, the user must specify the relative transformation between the frames, and whether the affixment is rigid or nonrigid. The relative transformation can be specified within the affixment statement, or if the positions of the two frames are already defined, just stating that they are to be affixed will automatically compute the necessary trans.

The form of the affixment statement is as follows:

```
part ← <frame exp>;
fixture ← <frame exp>;
AFFIX part TO fixture NONRIGIDLY;
```

or alternately,

```
AFFIX pump TO pump_base AT <trans exp> RIGIDLY;
```

RIGIDLY implies that the affixment is symmetric, so that changes in value of one frame imply changes in the other. A *RIGID* affixment is normally used when the objects are physically joined together rigidly, e.g. the *pump* being bolted to the *pump_base* or an arm grasping an object. In the above example, movement of *pump* will affect *pump_base*, and movement of *pump_base* will affect *pump*.

A *NONRIGID* affixment is used when one object is resting on another: e.g. *part* resting in *fixture*; *part* moves with the *fixture*, but if only *part* is moved, *fixture* stays put.

A frame could be affixed to more than one frame, and affixments may be chained together. The affixment relationship can be broken by means of the UNFIX statement as follows:

```
UNFIX pump FROM barm;
```


All the frames rooted in *pump* (e.g. *pump_base*) will remain rooted in *pump*, and will no longer be affected by *barm* or its motion.

The following examples illustrate the stacking of one block on top of another with and without the use of affixment to illustrate its usage and convenience during programming.

```
BEGIN "block stacking without affixment"

  FRAME blk1, blk1_grasp, blk1_top, blk2, blk2_grasp, finplace;
  DISTANCE SCALAR graspheight, blk1 length, blk2length, blk1 width,
                  blk2width, blk1height;
  ROT stand;

  stand ← ROT(XHAT,180.*degrees);

  blk1width ← 1.5*inches;        blk2width ← 1.5*inches;
  blk1length ← 2.4*inches;       blk2length ← 2.4*inches;
  blk1height ← 2*inches;         graspheight ← 0.75*inches;
                             { define dimensions of the blocks }

  blk1 ← FRAME(nilrot,VECTOR(10,30,0)*inches);
  blk2 ← FRAME(nilrot,VECTOR(6,30,0)*inches);
                             ( define bottom corner of blocks )
  finplace ← FRAME(nilrot,VECTOR(8,40,0)*inches);
                             { define final position of bottom of block 1 }

  blk1_grasp ← FRAME(stand,VECTOR(blk1 length/2,blk1 width/2,graspheight));
                             ( define grasping position of block 1 )
  blk1_top   ← FRAME(nilrot,VECTOR(0,0,blk1 height));
                             { define position of top of block 1 )
  blk2_grasp ← FRAME(stand,VECTOR(blk2length/2,blk2width/2,graspheight));
                             { define grasping position of block 2 )

  OPEN bhand TO 3.6*inches;

  MOVE barm TO blk1*blk1_grasp WITH APPROACH = 3*zhat*inches;
                                   ( arm moves to grasping position of blk1 )
  CENTER barm;                     { hand grasps blk1 }
  MOVE barm TO finplace*blk1_grasp WITH APPROACH = 3*zhat*inches;
                                   { arm moves so that blk1 is in final place )
  OPEN bhand TO 3.6*inches;        (hand opens to release blk1}

  MOVE barm TO blk2*blk2_grasp WITH APPROACH = 3*zhat*inches;
                                   ( arm moves to grasping position of blk2 )
  CENTER barm;                     { hand grasps blk2 }
  MOVE barm TO finplace*blk1_top*blk2_grasp WITH APPROACH = 3*zhat*inches;
                                   { arm moves to put blk2 on top of blk1 }
  OPEN bhand TO 3.6*inches;        (hand opens to release blk2 )

  MOVE barm TO bpark; PRINT ("all done");

END "block stacking without affixment";
```

Note that for each motion the destination is an expression consisting of a

local coordinate system and a point in that system (e.g. *blk1*blk1_grasp*). Another way to write the same program is as follows, where AL automatically takes care of the bookkeeping of which coordinate system to use. The same number of declarations are still needed, but now the motion statements are clearer. Note that because the destination of each motion is no longer an expression AL will automatically use the standard approach.

```
BEGIN "block stacking using affixment"

  FRAME blk1, blk1_grasp, blk1_top, blk2, blk2_grasp, finplace;
  DISTANCE SCALAR graspheight, blk1length, blk2length, blk1width,
            blk2width, blk1height;
  ROT stand;

  stand ← ROT(XHAT,180.*degrees);

  blk1width ← 1.5*inches;      blk2width ← 1.5*inches;
  blk1length ← 2.4*inches;     blk2length ← 2.4*inches;
  blk1height ← 2*inches;       graspheight ← 0.75*inches;

  blk1 ← FRAME(nilrot,VECTOR(10,30,0)*inches);
  blk2 ← FRAME(nilrot,VECTOR(6,30,0)*inches);
  finplace ← FRAME(nilrot,VECTOR(8,40,0)*inches);

  AFFIX blk1_grasp TO blk1 at
       TRANS(stand,VECTOR(blk1length/2,blk1width/2,graspheight)) RIGIDLY;
  AFFIX blk1_top TO blk1 at
       TRANS(nilrot,VECTOR(0,0,blk1height)) RIGIDLY;
     { top and grasping position of block1 are defined with respect to bottom }
  AFFIX blk2_grasp TO blk2 at
       TRANS(stand,VECTOR(blk2length/2,blk2width/2,graspheight)) RIGIDLY;
     { grasping position of block2 defined with respect to bottom }

  OPEN bhand TO 3.6*inches;

  MOVE barm TO blk1_grasp;     { arm moves over the grasping position of blk1 }
  CENTER barm;                 { hand closes over blk1 }
  AFFIX blk1 to barm RIGIDLY;  { blk1 and all its parts are attached to arm }
  MOVE blk1 TO finplace;       { note that blk1 is moved, not barm }
  OPEN bhand TO 3.6*inches;    { this physically releases the block }
  UNFIX blk1 from barm;        { blk1 is released from the arm in the world model }

  MOVE barm TO blk2_grasp;
  CENTER barm;
  AFFIX blk2 to barm RIGIDLY;
  MOVE blk2 TO blk1_top;       { move bottom of blk2 to the top of blk1 }
  OPEN bhand TO 3.6*inches;
  UNFIX blk2 from barm;

  MOVE barm TO bpark;   PRINT ("all done");

END "block stacking using affixment";
```


3.6 Sensing forces - simple condition monitors

When we want to use threshold values of sensory information to perform certain actions, we make use of condition monitor clauses. The syntax is as follows:

ON <condition> DO <action>

A simple example would be to rotate the wrist of the arm (assumed vertical) and stop when a torque of 50 ounce-inches is encountered - perhaps that indicates that we have tightened something to the required torque. An example of such a statement would then be

```
MOVE barm TO barm*FRAME(ROT(zhat, 90*degrees),nilvect*inches)
     ON TORQUE(zhat) ≥ 50 * ounces*inches  DO STOP barm;
```

The effect of this statement is obvious; the *STOP* command stops the motion of the arm immediately after the force is encountered. Note the specification of the direction of the detected torque, *zhat*, and the threshold amount (50 ounce-inches).

Assume we want to find the height of an object and that the object is expected to be in a given location, and that its height is expected to be between 2 and 12 inches.

```
BEGIN
  FRAME object;
  DISTANCE SCALAR height;

  CLOSE bhand TO 0*inches;   { bring fingers together }

  MOVE barm TO object + 14*zhat*inches;  { arm is vertically above the object }

  MOVE barm TO ⊗ - 13*zhat*inches
                         { symbol ⊗ here means current position of barm }
    WITH DURATION = 10*seconds
    ON FORCE(ZHAT) ≥ 10*ounces DO STOP;
        { try to move arm down 13 inches slowly and stop when a force is
          encountered; i.e. contact is made }

  height ← POS(barm).zhat - 0.3*inches;
        { take the z-component of the arm's current location and subtract the
        distance between the center and edge of the fingers to give the actual
        height of the object }

  PRINT("HEIGHT OF OBJECT IS ", height, " INCHES");
END;
```

At present only the blue Stanford arm can sense forces. In the near future force sensing wrists will be mounted on the PUMA arms which will enable them to

sense forces also.

3.7 Force and stiffness control

In addition to sensing forces and torques with the arm AL permits the user to apply controlled forces and torques to objects held in the hand. The user may also control the apparent stiffness of objects held in the hand. These two capabilities permit the user to perform assemblies that normally would be impossible with strict position control alone. Typical applications include using the arm to follow the contour of a surface or to place a pin in a hole.

The basic philosophy behind using stiffness control in assemblies is that the user still specifies a nominal position trajectory for the grasped object to follow. Based upon the constraints of the task the user may then command the arm to behave with a particular stiffness as it follows the trajectory. The arm follows the nominal trajectory if no constraint is met. However, if the grasped object makes contact with a constraining surface the contact force applied will depend upon the positioning error and the stiffness in the direction of the error. The user may dictate low stiffness in the direction of expected constraint to prevent excessive contact forces and binding. For example to place a peg in a hole the user would specify low stiffnesses in the directions perpendicular to the hole axis and a high stiffness in the direction of insertion. The final stage of a peg insertion may then be expressed as follows:

```
BEGIN "FINAL_INSERT"
    {.. assume peg is held in the hand and AFFIXed to barm .. }
    MOVE peg TO hole_entrance;
    MOVE peg TO hole_bottom
       DIRECTLY
       WITH STIFFNESS = (VECTOR(10,10,90)*oz/inch,
                    VECTOR(200,200,200)*oz*inch/radian)
       WITH DURATION = 2*SECONDS;
END "FINAL_INSERT";
```

The stiffness specification above causes the arm to be compliant for translations in the X and Y directions and relatively stiff in the Z direction. The arm will also be relatively stiff for rotations in all three directions.

In addition to controlling the stiffness in the six principal axes of a grasped object the user may specify the location of the controlled frame relative to the hand so that the center of stiffness may be located where desired. This permits the arm to behave like a programable remote center compliance device or have other stiffness properties suitable for particular tasks. Pure forces may also be applied by making the stiffness zero in the desired direction and adding a bias force by means of the FORCE or TORQUE which looks similar to the the FORCE or TORQUE condition monitor except that an equality sign is used for the magnitude of

the force or torque. In the following statement, the arm will apply a constant force in the x-direction.

```
MOVE barm TO barm+12*zhat*inches
   DIRECTLY
   WITH STIFFNESS = (VECTOR(0,90,90)*oz/inch,
               VECTOR(200,200,200)*oz*inch/radian)
   WITH FORCE(XHAT) = 40*ounces
   WITH DURATION = 3*seconds;
```

A detailed example of force and stiffness control will be found in Appendix VI.

3.8 Control structures: *IF*, *FOR* & *WHILE* statements

AL has many of the traditional ALGOL control structures, including conditionals and loops. There are no jumps in AL. In this section we shall describe the *IF*, *FOR* and *WHILE* statements.

The *IF* statement has the form:

```
IF <condition>
   THEN <statement>
   ELSE <statement>
```

The *ELSE* part is optional. The condition is some boolean expression involving one of the operators $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$. Boolean expressions can be built up out of relational operators, the logical connectives $\wedge$ (*AND*), $\vee$ (*OR*), $\neg$ (NOT), $\oplus$ (*XOR*, exclusive or), $\equiv$ (*EQV*, the logical equivalence) or the logical constants *TRUE* or *FALSE*. The condition may also be some arithmetic scalar expression. If the condition is true (non-zero) the statement following the *THEN* is executed. Otherwise the statement following the *ELSE*, if present, will be executed.

The *FOR* loop has the form:

```
FOR <s var> ← <s expr> STEP <s expr> UNTIL <s expr> DO <statement>
```

where *<s var>* stands for "scalar variable" and *<s expr>* stands for "scalar expression of same dimension". The initial value of the variable is the value of the first expression; every time the statement is executed, its value is incremented by the value of the second expression, and the process repeats until the value exceeds that of the third expression. If the step size is negative, the right things happen. A test is made before the first iteration, so it is possible that the loop will not get executed at all.

The *WHILE* loop is as follows:

WHILE <condition> DO <statement>

where *<condition>* is the same as above. The condition is checked and if it is true the statement is executed. The process is repeated until the condition becomes false.

The following example illustrates the use of the *IF*, *FOR* and *WHILE* statements in a program where the arm picks up castings from one place, puts the good ones on a pallet in 6 rows of 4 and discards the defective ones. The castings come in batches of 50, but it is not known ahead of time how many batches there will be.

```
BEGIN "sort castings"
   FRAME pickup, garbage_bin, pallet;
   SCALAR pallet_row,pallet_column, good, bad;
   DISTANCE SCALAR packing_distance;
   SCALAR ok, more_batches, casting_number;
   packing_distance←4*inches;

   pallet_row←1; pallet_column←0; good←0; bad←0;
   casting←pickup;

   OPEN bhand TO 3*inches;

   MOVE barm TO pickup DIRECTLY;
   CENTER barm;
   IF (bhand < 1.5*inches) THEN more_batches←FALSE ELSE more_batches←TRUE;
      {an alternate way to state this is
                more_batches ← NOT (bhand < 1.5 * inches); }
   WHILE more_batches DO
      BEGIN "sort 50 castings"

      FOR casting_number← 1 STEP 1 UNTIL 50 DO
         BEGIN "sort casting in hand"
         ok ← FALSE;
         AFFIX casting TO barm RIGIDLY;
         MOVE casting TO pickup + 3*zhat*inches
            ON FORCE(zhat)≥ 20*ounces DO ok←TRUE; {see if it weighs enough }

         IF ok THEN
            BEGIN "good casting"
            good←good+1;
            IF pallet_column=4
               THEN BEGIN pallet_column←0; pallet_row←pallet_row + 1; END
               ELSE pallet_column←pallet_column+1;
            MOVE casting TO pallet +
                  VECTOR( pallet_column*packing_distance,
                        pallet_row*packing_distance,0*inches)
                  WITH APPROACH = 3*zhat*inches;
            UNFIX casting FROM barm;
            OPEN bhand TO 3*inches;
            IF (pallet_column=4) AND (pallet_row=6)
               THEN BEGIN "pallet full"
               pallet_column←0; pallet_row←1;
```



```
                  { code to remove this pallet and get new pallet }
                  END "pallet full";
               MOVE barm TO pickup;
               END "good casting"
            ELSE
               BEGIN "defective casting"
               bad←bad+1;
               MOVE casting TO garbage_bin DIRECTLY;
               OPEN bhand TO 3*inches;
               UNFIX casting FROM barm;
               MOVE barm TO pickup;
               END "defective casting";

            casting←pickup;
            CENTER barm;
            END "sort casting in hand";

         IF (bhand < 1.5*inches) THEN more_batches ← FALSE;
         END "sort 50 castings";

      MOVE barm TO bpark;

      PRINT("THERE WERE ", good," GOOD CASTINGS AND ", bad," DEFECTIVE CASTINGS");

   END "sort castings";
```

3.9 Control structures (cont): *CASE* & *UNTIL* statements

Two of the other traditional ALGOL control structures in AL are the *CASE* and *UNTIL* statements.

The *CASE* statement comes in several forms. The regular *CASE* statement has the form:

```
CASE <index> OF
   BEGIN
   <statement 0>;
   <statement 1>;
   <statement 2>;
       :
   <statement n>
   END
```

The scalar index expression is evaluated and depending on the integer part of its value one of the following statements is executed. If the index is zero then statement 0 is chosen, if the index is one then statement 1 is chosen, and so on up till n. If the index is negative, or greater than the number of statements, an error is reported. Any of the statements may be null, e.g. "<statement 1>;;<statement 3>", in which case if the index were two nothing would be done.

There is also a numbered version of the *CASE* statement:

```
CASE <index> OF
    BEGIN
[C0] <statement>;
[C1] <statement>;
[C2] <statement>;

[Cn] <statement>;
ELSE <statement>
    END
```

where each statement has one or more non-negative scalar constants labelling it. Again, the index expression is evaluated and if its integer part is the same as one of the Ci's then the statement with that label is executed. Otherwise, if an *ELSE* is present, the statement it labels is executed. If no *ELSE* is present, an error occurs if the integer part of the index is negative or greater than the largest Ci, otherwise nothing is done. Note that the *ELSE* statement may appear anywhere in the list of statements; it need not be at the end.

Here is an example using the numbered *CASE* statement to select the appropriate action to perform when given one of several possible parts.

```
BEGIN
   SCALAR part_number;
   FRAME pick_up,base,base_grasp,cover,cover_grasp,side,side_grasp,...;

  {Initialization code including the following macro definitions:
   DEFINE base_num = ...;
   DEFINE cover_num = ...;
   DEFINE side_num = ...;
   which will be used for clarity in a numbered case statement.}

   {Now go get the part at pick_up and do whatever is appropr[illegible] with it.}

   PRINT("Enter the part's number:");
   part_number ← INSCALAR;      {INSCALAR reads in a scalar from the console keyboard }
       {Have the user type in the part's number. In the future this might
        be done automatically using vision.}
   CASE part_number OF
       BEGIN
       [base_num] BEGIN {Code to handle base.}
                   base ← pick_up;
                   MOVE barm TO base_grasp;
                   CENTER barm;    {Grab it}
                   AFFIX base TO barm;
                   {Rest of code for base.}
                 END;

       [cover_num] BEGIN
                   {Code to handle cover.}
                 END;
```



```
       {Repeat for other known parts: side,etc.}

       ELSE        BEGIN
                     PRINT("Unknown part number",crlf);
                     {Code to recover from error}
                   END
       END;

       {Rest of program.}
END;
```

The *UNTIL* statement is as follows:

```
DO <statement> UNTIL <condition>
```

where the statement is repeatedly executed until the condition becomes true. This is similar to the *WHILE* statement described in the previous section, with the exception that the *WHILE* loops while the condition is true, whereas the *UNTIL* loops until the condition becomes true. Note that the body of an *UNTIL* loop is always executed at least once.

As an example of the use of the *UNTIL* statement, here is a program excerpt that gets a good casting, discarding any bad ones it finds in the process. It is similar to the example in the previous section.

```
BEGIN
   SCALAR success;
   {Initialization code}

   success ← false;
   casting←pickup;
   MOVE barm TO casting_grasp;

   DO BEGIN {Try to get a good casting}

      CENTER barm;
      AFFIX casting TO barm RIGIDLY;
      MOVE casting TO pickup + 3*zhat*inches      {See if it weighs enough}
         ON FORCE ≥ 20*ounces ALONG zhat OF station DO success ← true;

      IF ¬success THEN {Get rid of a bad casting}
        BEGIN
        MOVE casting TO garbage_bin DIRECTLY;
        OPEN bhand TO 3*inches;
        UNFIX casting FROM barm;
        casting←pickup;
        MOVE barm TO casting_grasp
        END

     END UNTIL success;

   {Code for rest of program}
END;
```

3.10 Simultaneous motion: *COBEGIN-COEND, SIGNAL-WAIT*

So far we have considered single arm moves. To perform simultaneous movements of arms, two new concepts have to be introduced. The *COBEGIN-COEND* block has the same effect as the *BEGIN-END* block, except that statements within the block are executed simultaneously.

Thus the following will park all four arms at the same time.

```
COBEGIN
    MOVE barm TO bpark;
    MOVE yarm TO ypark;
    MOVE garm TO gpark;
    MOVE rarm TO rpark;
COEND;
```

Simple synchronization is possible within the context of simultaneous execution. This is achieved by means of explicit events and the *SIGNAL* and *WAIT* statements. Every different event that the user wishes to use should be declared in a declaration statement as follows:

```
EVENT e1,e2,e3
```

The *EVENT* is distinct from algebraic data types (e.g. scalars) and cannot be assigned a particular value by the user in his program by means of the regular assignment statement. With each event is associated a count of how many times it has been signalled. Initially, the count is zero, that is, no signals have appeared, and no processes are waiting. The statement

```
SIGNAL e1
```

increments the count associated with event *e1*, and if the resulting count is zero or negative, one of those processes waiting for *e1* is released from its wait and readied for execution. The statement

```
WAIT e1
```

decrements the count associated with event *e1*, and if the resulting count is negative, the process issuing the *WAIT* is blocked from continuing until a signal comes along. If the count is zero or positive, there is no waiting.

The following example shows the use of the *SIGNAL* and *WAIT* commands, although it may be done without these constructs. The blue arm picks up an object and moves to a passing location, where it makes sure that the yellow arm has grasped it before releasing it.


```
BEGIN
    EVENT passed, caught, ready_pass;
    FRAME steel_beam, pass, catch;

    COBEGIN
        BEGIN "blue"
        MOVE barm TO steel_beam;
        CENTER barm;
        AFFIX steel_beam TO barm;      { barm gets steel beam }
        MOVE steel_beam TO pass;       { takes it to passing position }
        SIGNAL ready_pass;             { barm says it is ready }
        WAIT caught;                   { waits for yellow arm to catch}
        OPEN bhand TO 3.0*inches;      { when yellow arm ready releases beam}
        UNFIX steel_beam FROM barm;
        SIGNAL passed;                 { barm announces it has released beam }
        END "blue";

        BEGIN "yellow"
        OPEN yhand TO 3.0*inches;      {meanwhile yellow hand is opened }
        MOVE yarm TO catch;            { yellow arm goes to catching position }
        WAIT ready_pass;               {yarm waits till ther • is something to grab}
        CENTER yarm;                   { grasps it}
        SIGNAL caught;                 { yarm announces it caught it }
        WAIT passed;                   {waits for blue arm to releas • it }
        MOVE yarm TO pallet;
        END "yellow";
    COEND;
END;
```

A second example illustrates the use of *SIGNAL* and *WAIT* in resource sharing. The example in the last section where castings are sorted will be used but assume that the two arms are doing similar jobs, and that a single overhead crane is used to take away the full pallets and bring in empty pallets. Blue and yellow pallets are used to correspond to the appropriate arms. The code for the program will be similar to the previous section, except that the section which states { *code to remove this pallet and get new pallet* }, in the block labeled "*pallet full*", will use *SIGNAL* and *WAIT* to ensure that the crane is not asked to go to two locations at the same time, and that it is asked to go to a location only when it is needed.

```
BEGIN
    EVENT blue_pallet_full, blue_pallet_empty;
    EVENT yellow_pallet_full, yellow_pallet_empty;
    EVENT crane_free;
    SCALAR more_blue_pallets, more_yellow_pallets;

    more_blue_pallets←TRUE;      more_yellow_pallets←TRUE;
    SIGNAL crane_free;

    COBEGIN
        BEGIN "load blue pallets"
            BEGIN "sort castings"  {code from section 3.8}

            IF (pallet_column=4) AND (pallet_row=6)
```

```
        THEN BEGIN "pallet full"
             pallet_column←0; pallet_row←1;
             SIGNAL blue_pallet_full;
             WAIT blue_pallet_empty;
             END "pallet full";


     END "sort castings";
     SIGNAL blue_pallet_full;       {to get last pallet out of the way}
     WAIT blue_pallet_empty;
     more_blue_pallets←FALSE;  {to stop crane waiting for blue pallet,
                               otherwise crane program will get stuck in
                               "change blue pallet" block.}
END;

BEGIN "load yellow pallets"
     BEGIN "sort castings" {similar to blue pallets except use yellow
                           arm and yellow pallet}


        IF (pallet_column=4) AND (pallet_row=6)
        THEN BEGIN "pallet full"
             pallet_column←0; pallet_row←1;
             SIGNAL yellow_pallet_full;
             WAIT yellow_pallet_empty;
             END "pallet full";


     END "sort castings";
     SIGNAL yellow_pallet_full;
     WAIT yellow_pallet_empty;
     more_yellow_pallets←FALSE;
END;

WHILE more_blue_pallets
DO BEGIN "change blue pallet"
        WAIT blue_pallet_full;
        WAIT crane_free;           {wait for crane to be freed }
             {code to use crane to change blue pallet}
        SIGNAL blue_pallet_empty;
        SIGNAL crane_free;
   END;

WHILE more_yellow_pallets
DO BEGIN "change yellow pallet"
        WAIT yellow_pallet_full;
        WAIT  crane_free;          { wait for crane to be freed }
             {code to use crane to change yellow pallet}
        SIGNAL yellow_pallet_empty;
        SIGNAL crane_free;
   END;

  COEND;
END;
```


3.11 Arrays

Sometimes we would like a variable to refer to more than one value. As an example consider a base plate with three screw holes in it. During the assembly, code to insert a screw into each hole will be written. Rather than repeatedly writing the same code for each screw hole, it would be preferable to write it once and somehow use a *FOR* loop to repeat it for all the holes. An array will allow us to do this.

An array is a variable that can have multiple values. In the above example we had three frames: first_hole, second_hole and third_hole. We can define a frame array: hole[1:3] which allows us to reference the three screw holes as: hole[1], hole[2] and hole[3]. More formally an array definition is of the form:

<type> ARRAY <name1>[bounds], <name2>[bounds]

where type specifies the array's data type, and bounds indicates the size of the array and how the elements of it are referenced. Our example above used a one dimensional array. An example of a two dimensional array is:

SCALAR ARRAY foo[1:3,1:4]

which would look like:

```
foo[1,1] foo[1,2] foo[1,3] foo[1,4]
foo[2,1] foo[2,2] foo[2,3] foo[2,4]
foo[3,1] foo[3,2] foo[3,3] foo[3,4]
```

There is no upper limit on the number of dimensions an array may have. The array bound pairs may be either scalar constants, variables or expressions. The bounds may have positive or negative values, as long as the lower bound is smaller than the upper bound For example:

VECTOR ARRAY u[-3:3], v[n:n+5], w[0:3,1:m]

where n and m are scalar variables. Space is allocated for arrays upon entry of the block in which they are defined, so the sizes of v and w will depend on the values of n and m when the definition occurs.

Arrays are used in programs just like regular variables. For example:

FOR i ← 1 STEP 1 UNTIL 4 DO foo[1,i] ← foo[2,i] * foo[3,i]

At runtime a check is made that each subscript falls within the lower and upper bounds given for the dimension it specifies. Subscripts outside the bounds

cause an error message to be printed. Only the integer part of the subscript is used.

Here is an example to d o the screw insertion task mentioned at the beginning of this section.

```
BEGIN
   FRAME ARRAY hole[1:3];
   FRAME base_plate;
   SCALAR i;

  {Initialization and start of the program including definition of the locations of
   the base_plate and the screw holes:
   base_plate ← FRAME(....);
   AFFIX hole[1] TO base_plate RIGIDLY AT TRANS(....);
   AFFIX hole[2] TO base_plate RIGIDLY AT TRANS(....);
   AFFIX hole[3] TO base_plate RIGIDLY AT TRANS(....);
   Screws will be defined with the z-axis pointing downward.
   Code to get the screw driver into the hand is also included.)

  {Now insert the three screws}

   FOR i ← 1 STEP 1 UNTIL 3 DO
    BEGIN
      screw ← screw_dispenser;        {Define location of new screw}
      MOVE driver_tip TO screw;       {Get a screw - not really this easy}
      AFFIX screw TO driver_tip;
      MOVE screw_tip TO hole[i];      {Screw is just above the screw hole}

      COBEGIN
        MOVE screw TO ⊗ - 0.75 * zhat * inches          {Push down with arm}
            WITH FORCE = 20 * ounces ALONG zhat OF screw
            WITH DURATION = 2.5 seconds;
        OPERATE driver                      {Drive in the screw}
            WITH ANGULAR_VELOCITY = 200 * rpm
            WITH DURATION = 3 * seconds;
      COEND;

      UNFIX screw FROM driver_tip;   {Release the screw}
    END;
END
```

3.1 2 Procedures

There are times when we wish to do the same operation at several places in the program. Rather than place the entire sequence at each of these points it is often desirable to code it up once as the body of a procedure or subroutine, and at each point in the program where the operation is required have a call o n the procedure. As an example during an assembly there may be a number of screws that need to be inserted. A procedure to do this insertion will be shown after the syntax for procedures has been explained.

Procedures are defined as follows:


```
<type> PROCEDURE <name> (parameter list);
        <statement>
```

where the statement is executed each time the procedure is called. A simple procedure to park the arm and open the fingers could be written as:

```
PROCEDURE park;
    BEGIN
    MOVE barm TO bpark;
    OPEN bhand TO 3 * inches;
    END;
```

Any time in the program the user wants to move the arm to the park position and open the hand all she need type is the statement:

```
park
```

which will call the procedure. Sometimes a procedure will be used to return a result needed for computation (i.e., the procedure will be used as a function). This is done by using the *RETURN* statement:

```
RETURN (value)
```

which returns value as the result of the procedure. For example a procedure to determine the height of the blue arm might be written:

```
DISTANCE SCALAR PROCEDURE height_barm;
        RETURN(POS(barm) . zhat );
```

Any time the height of the blue arm is needed one would call the procedure. Note the declaration of the data type that the procedure returns. We can generalize this procedure s o that for a ų frame it returns the height of the frame. To d o this we introduce the use of parameters to pass a value to the procedure. The generalized procedure and a sample of it in use is as follows:

```
DISTANCE SCALAR PROCEDURE height (FRAME f);
        RETURN( POS(f) . zhat );

PRINT("The height of the pallet is:", height(pallet_top));
```

when the procedure is called the parameter f is bound to the value of pallet_top, and every reference to f in the body of the procedure will refer to pallet_top. Parameters can be passed by reference, which is the default for variables and arrays, or by value, the only way expressions are passed. If a variable is passed by reference then its value can be modified by the procedure. For example a procedure to refine the location of a frame by grasping it with the arm and then reading the position of the arm might be written:

```
PROCEDURE refine (REFERENCE FRAME obj);
      BEGIN
      OPEN bhand TO 3*inches;
      MOVE barm TO obj;
      CENTER barm; {This will sense the object's position}
      obj ← barm
      END;
```

When the procedure returns, the frame passed as its argument will have a new value.

A traditional example of a procedure used in most programming tutorials is the factorial function: fact(1) = 1, fact(2) = 2*1, fact(3) = 3*2*1, etc. Here are two ways of writing factorial in AL; the first is iterative, while the second is recursive (i.e. it calls itself).

```
SCALAR PROCEDURE ifact (SCALAR n);
      BEGIN
      SCALAR i, prod;
      prod ← 1;
      FOR i ← 2 STEP 1 UNTIL n DO prod ← prod * i;
      RETURN( prod );
      END;

SCALAR PROCEDURE rfact (SCALAR n);
      IF n > 1 THEN RETURN( n * rfact(n-1) )
                ELSE RETURN( 1 );
```

A procedure to do the screw insertion operation is as follows:

```
PROCEDURE insert_screw (FRAME hole_location);
 BEGIN
  screw←screw_dispenser;
  MOVE driver_tip TO screw;        {Get a screw - not really this easy}
  AFFIX screw TO driver;
  MOVE screw_tip TO hole_location;       {Screw is just above screw hole}

  COBEGIN
    MOVE screw TO ® - 0.75 * zhat * inches        {Push down with arm}
       WITH FORCE = 20 * ounces ALONG zhat OF screw
       WITH DURATION = 2.5 seconds;
    OPERATE driver                          {Drive in the screw}
       WITH ANGULAR_VELOCITY = 200 * rpm
       WITH DURATION = 3 * seconds;
  COEND;

  UNFIX screw FROM driver          {Release the screw}
 END;
```

Now the loop to insert three screws in the example in the previous section would be:

```
FOR i ← 1 STEP 1 UNTIL 3 DO insert_screw( hole[i] );
```


3.13 Hints to the Programmer

3.13.1 Upward pointing grasping positions

The AL user will quickly realize that under normal usage, the frame *barm* usually has its Z axis pointing downwards in station coordinates. Since we are used to thinking in terms of an upward positive Z direction, it is sometimes convenient to define another frame affixed rigidly to *barm* but with the Z-axis pointing upwards, and the Y axis either parallel or anti-parallel to the station Y axis. With such a frame, the user can define grasping frames with the station orientation if the hand points downwards. The following statements will set up a frame called *bgrasp* to accomplish what we want.

```
FRAME bgrasp;
AFFIX bgrasp TO barm AT TRANS(ROT(xhat,180*deg), nilvect*inches) RIGIDLY;
```

3.13.2 Initialization and program end

Initialization of the arm and hand to known positions before starting is a good idea to ensure that the first movement from an unknown position does not result in the arm accidentally hitting objects in the way. The user should get the arm to a known location and the hand to an appropriate opening so that the first motion does not hit an object in the workplace.

It is good policy to park the arm at the end of the program by using:

```
MOVE barm TO bpark;
```

This leaves the arm in a statically balanced location.

3.13.3 Slowing down movements

When trying out a program for the first time when it is not known how the arm will behave, the use of a *speed_factor* greater than two will slow down all motions in the program (c.f. section 4.4.6 for details). The user should assign a value to *speed_factor* at the beginning of the program as follows:

```
speed_factor ← 4.0
```

For convenience, three predeclared macros *SLOW*, *CAUTIOUS*, and *QUICK*, assigning values of 4.0, 6.0 and 1.0 respectively to *speed_factor*, may be used instead of the assignment statement described above. The "normal" default speed factor is 2.0. Setting the speed factor to 1.0 speeds up the motion; however, it should be be noted that trying to make the motion too fast requires high motor torques which result in AL shutting down the motors during motion as an error.

4. THE AL LANGUAGE

AL is an ALGOL-like source language extended to handle the problems of manipulator control. This chapter describes the features of the AL language. It is presumed that the reader has read the previous chapter which introduces the AL language in a tutorial fashion.

4.1 Basic constructs

4.1.1 Programs

AL programs are organized in the traditional block structure of ALGOL. A program in AL consists of either a single statement or a block statement, which is a sequence of statements, separated by semicolons, and surrounded by the reserved words *BEGIN* and *END* (or *COBEGIN* and *COEND*). Blocks may be named by placing a string constant immediately after the *BEGIN* (or *COBEGIN*). This name will be checked against the string (if any) that follows the matching *END* (or *COEND*), and if the two strings do not match, an error will be reported.

BEGIN "block name" S; S; S; S END "block name"

4.1.2 Variables

A variable name is a string of alphanumeric characters and underscore, "_", starting with a letter. Variables must be declared before being used. AL follows normal variable scoping rules: variables may only be referenced within the block they are declared in, or in blocks nested within that block. The same variable name may be declared in several blocks, in which case any references to it refer to the innermost declaration enclosing the reference.

4.1.3 Comments

Comments are text inserted into the program to make it more readable. Comments can be written in two forms. The compiler will ignore all text between the reserved word *COMMENT* and the next semicolon encountered. Comments may also be enclosed by curly brackets "{}".

4.2 Data types and expressions

4.2.1 Algebraic data types: *SCALAR, VECTOR, ROT, FRAME, TRANS*

The basic data types in AL were chosen to facilitate working in the three dimensions of the real world. Scalars are floating point numbers like reals in other computer languages. Vectors are 3-tuples specifying (X, Y, Z) values, which represent quantities like translations, velocities, and locations with respect to some coordinate system. Rotations are 3x3 matrices representing either an orientation or a rotation about an axis. A rotation, or rot, is constructed from a vector, specifying the axis of rotation, and a scalar, giving the angle of rotation. Frames are used to represent local coordinate systems. They consist of a vector specifying the location of the origin, and a rotation specifying the orientation of the axes. Transes are used to transform frames and vectors from one coordinate system to another. Like frames they consist of a vector and a rotation.

4.2.2 Labels, Events and Strings

Labels, events, and strings are data types that are declared in the same manner as the algebraic data types. There are two kinds of labels: statement labels and condition monitor labels. Condition monitors are labelled for reference by the *ENABLE* and *DISABLE* statements (c.f. section 4.4.4.2). Statements are labelled for use in debugging. A label consists of an identifier followed by a colon. Labels must be declared before being used.

Events are used in conjunction with the *SIGNAL* and *WAIT* statements (c.f. section 4.5.4) used to synchronize parallel processes.

Strings consist of characters enclosed within a pair of double quotes. String variables are provided primarily to pass strings to procedures. The only operation currently defined on strings is assignment.

4.2.3 Arrays

Multi-dimensional arrays are available in AL. They may be of any algebraic data type or of type event. Array bounds may be scalar constants, variables, or expressions; they may be positive or negative integers. The only constraint is that the lower bound be smaller than the upper bound. At runtime a check is made that each subscript falls within the lower and upper bounds given for the dimension it specifies. Subscripts outside the bounds cause an error message to be printed.

Arrays are allocated upon entry of the block in which they are defined, and deallocated upon block exit.

4.2.4 Dimensions

AL allows physical dimensions to be associated with variables. The known dimensions are: *TIME, DISTANCE, ANGLE, FORCE, TORQUE, VELOCITY, ANGULAR_VELOCITY & DIMENSIONLESS*. New dimensions may be defined if desired by means of the *DIMENSION* statement:

DIMENSION <new dimension> = <dimension expression>

where the operators defined in <dimension expression> are (,),*,/ and *INV*, which takes the inverse of its argument, e.g. INV(TIME) = 1/TIME.

Dimensioned quantities are just like regular ones, except that they are multiplied by the appropriate reserved word: *SEC, CM, DEG, GM, INCHES, RPM, OZ, LB*, and *LBS* (also *SECOND, SECONDS, INCH, OUNCES, DEGREES, RADIAN* and *RADIANS*). For example:

```
VELOCITY VECTOR v;
v ← xhat * inches / sec
```

Other units may be defined using macros (c.f. section 4.5.8), e.g.:

```
DEFINE feet = ⊂(12 * inches)⊃
```

AL checks for consistent usage of dimensioned quantities: addition and subtraction, along with frame, trans and rot operations require exact dimension match, while scalar and vector multiplication and division produce a quantity of new dimension.

4.2.5 Declarations

The declaration statement is used to define the data type and dimension of each variable used in a program. It has the form:

<dimension> <data type> <list of variables>

where *<dimension>* is one of the predefined dimensions in AL (*TIME, DISTANCE, ANGLE, FORCE, TORQUE, VELOCITY & ANGULAR_VELOCITY*), or a user defined dimension. *<Data type>* is one of the following: *SCALAR, VECTOR, ROT, FRAME, TRANS, STRING, EVENT & LABEL*. Only the algebraic data types: *SCALAR, VECTOR* and *TRANS* may have a dimension associated with them. Unless otherwise specified, scalars and vectors are considered dimensionless, while transes are considered to be of dimension distance (c.f. section 3.1.1.5).

Array declarations are of the form:

<dimension> <data type> ARRAY <list of array segments>

where each array segment in the list consists of one or more variable names followed by a list of lower-upper bounds pairs enclosed in square brackets "[]", e.g. "name1,name2,...[L1:U1, L2:U2,...]".

4.2.6 Arithmetic expressions

Here is a summary of the arithmetic operators available. They are grouped by the data type of their resulting value. These abbreviations are used: 's' = scalar, 'v' = vector, 'r' = rotation, 'f' = frame, 't' = trans.

Scalar operators

```
s + s       scalar addition
s - s       scalar subtraction
s * s       scalar multiplication
s / s       scalar division
s ↑ s       scalar raised to a scalar power
s MAX s     maximum
s MIN s     minimum
s DIV s     integer quotient after applying INT to each argument
s MOD s     integer remainder after applying INT to each argument
v . v       dot product of two vectors
|s|         absolute value of a scalar
|v|         magnitude of vector (vector norm)
|r|         extracts angle of rotation
```

Scalar functions

```
INT(s)       integer part of s
SQRT(s)      square root
SIN(s)       sine (all trigonometric functions are in degrees)
COS(s)       cosine
TAN(s)       tangent
ASIN(s)      arc-sine
ACOS(s)      arc-cosine
ATAN2(s,s)   arc-tangent of s/s
LOG(s)       natural logarithm
EXP(s)       e raised to the s power
RUNTIME      current system time in seconds (i.e. time of starting AL system)
RUNTIME(s)   current system time in seconds minus s
INSCALAR     reads a scalar from the console
```

Boolean operators

s <rel> s	returns true if relation is satisfied, else returns false possible relations are: $<,\leq,=,\geq,>,\neq$
s ∧ s , s AND s	logical and
s ∨ s , s OR s	logical or
s ⊗ s , s XOR s	logical exclusive or
s ≡ s , s EQV s	logical equivalence
¬ s , NOT s	logical not
QUERY	reads a boolean from the console (c.f. section 4.5.7)

Vector operators

VECTOR(s,s,s)	construct vector given (x,y,z) components
(s,s,s)	same effect as VECTOR(s,s,s)
s * v	dilation of a vector
v / s	contraction of a vector
v + v	vector addition
v - v	vector subtraction
v * v	vector cross product
r * v	rotation of a vector
t * v	transformation of a vector
f * v	transformation of a vector - shorthand for (station → f) * v
v WRT f	a vector in station coordinates pointing the same way as v points in f's coordinate system. v WRT f ≡ ORIENT(f)*v ≡ (f*v) - POS(f)
UNIT(v)	vector of unit length pointing in the same direction as v
POS(f)	vector position of frame or trans
AXIS(r)	axis of rotation

Rotation operators

ROT(v,s)	constructs rotation of s degrees about v
(v,s)	same effect as ROT(v,s)
ORIENT(f)	orientation of a frame or trans
r * r	composition of two rotations (the one on the right is applied first)

Frame operators

FRAME(r,v)	constructs frame of orientation r at position v
CONSTRUCT(v,v,v)	makes a frame: first vector gives the position, second a point on the x-axis, third is a point in the xy-plane
f + v	translation of a frame
f - v	translation of a frame
t * f	transformation of a frame
f * f	transformation of a frame - shorthand for (station → f) * f

Transform operators

TRANS(r,v)	constructs trans which will cause a rotation of r followed by a translation of v
(r,v)	same effect as TRANS(r,v)
f → f	transformation which maps from the first frame to the second
t * t	composition of two transes (the one on the right is applied first)
INV(t)	take the inverse of t

The operators in AL generally follow "normal" precedence rules, i.e., functions are evaluated first, followed by exponentiations before multiplications or divisions, which in turn are performed before additions and subtractions. The order of operation can be changed by including parentheses at appropriate points. In an expression where several operators of the same precedence occur at the same level, the operations are performed from left to right.

TABLE OF PRECEDENCE

functions, (), | |, NOT, unary operations
→ ↑
* / . MAX MIN DIV MOD
WRT
\+ -
= ≠ < > ≤ ≥
∧
∨ ⊗
≡

4.2.7 Predeclared constants and variables

PI = 3.14159... (can also be written as π)

STATION is a frame which has standard station coordinates

BARM, YARM, GARM, RARM
: are the locations of the blue, yellow, green and red arms respectively

BHAND, YHAND, GHAND, RHAND
: are the distances between the fingers of the blue, yellow, green and red arms respectively

BARM_ERROR, YARM_ERROR, GARM_ERROR, RARM_ERROR
: are errors associated with the blue, yellow, green and red arms for the last motion of the appropriate arm.

BHAND_ERROR, YHAND_ERROR, GHAND_ERROR, RHAND_ERROR
: are errors associated with the blue, yellow, green and red hands for the last motion of the appropriate hand

DRIVER_ERROR is the error associated with the last motion of the socketdriver

BPARK, YPARK, GPARK, RPARK
: are the rest positions for the blue, yellow, green and red arms

respectively and the values are as follows:

```
BPARK ← FRAME(ROT(yhat,180*degrees),VECTOR(43.53,56.86,9.96)*inches);
YPARK ← FRAME(ROT(yhat,180*degrees),VECTOR(40,14,9)*inches);
GPARK ← FRAME(ROT(zhat,180*degrees),VECTOR(83.2,46.13,67.7)*inches);
RPARK ← FRAME(ROT(zhat,180*degrees),VECTOR(84.8,12.87,67.7)*inches);
```

DRIVER_GRASP, DRIVER_TIP
are coordinate frames for the grasping location and the tip of the socket driver respectively

DRIVER_TURNS is a scalar which keeps count of the number of turns of the driver

TRUE and FALSE have the obvious meanings (TRUE = 1, FALSE = 0)

XHAT is VECTOR(1,0,0)

YHAT is VECTOR(0,1,0)

ZHAT is VECTOR(0,0,1)

NILVECT is VECTOR(0,0,0)

NILROT is ROT(zhat, 0 * DEG)

NILTRANS is TRANS(nilrot,nilvect*inches)

CRLF is a string constant that prints as a carriage return followed by a line feed

NULL is the null string of zero length

π when included within double quotes in a printing statement, sends a beep to the VT05 terminal.

4.2.8 Some examples

```
DISTANCE VECTOR v1,v2; {some declarations}
ANGLE SCALAR theta;
SCALAR ARRAY s1,s2[1:5], s3[-3:3,1:2];
FRAME f1,f2;
EVENT ready;

ROT(zhat,90*deg) * v1    {v1 rotated 90 degrees about
                              the station's Z axis}
v1 . yhat            {the Y component of v1}
f1 * xhat            {f1's X axis in station coordinates}
3 * s1[2]            {the second element of the
                          array s1 multiplied by 3}
```

4.3 Affixment: *AFFIX & UNFIX*

The relationships between the various features of an object, and between different objects, may be modelled by use of the *AFFIX* statement. The general form for the *AFFIX* statement is:

AFFIX f1 TO f2 BY t AT <expr> <affix type>

The effect of the above is to establish a trans that expresses the relationship

between *f1* and *f2*. If *<BY t>* is present the resulting trans will be associated with the variable *t* making the affixment relation modifiable by the user, otherwise an internal variable will be created. The initial value of the trans is specified by the *<AT expr>* part of the statement. If none is given then the current values of *f1* and *f2* are used to create a trans taking *f2* to *f1* (*f2* → *f1*). There are two flavors of affixment possible, and *<affix type>* specifies whether the affixment is to be done *RIGIDLY* or *NONRIGIDLY*. Rigid affixment is symmetric; when either frame is given a new value the other is updated to preserve the relationship between them. Non-rigid affixment is asymmetric; when *f2* is changed, the value of *f1* is updated, whereas when *f1* is modified, the trans describing the relationship between *f1* and *f2* is recomputed to express the new relationship between them. An example of non-rigid affixment would be a plate on a tray; the plate moves with the tray, but not vice versa. If *<affix type>* is not specified, rigid affixment will be assumed.

An affixment relation can be broken by use of the *UNFIX* statement:

UNFIX f1 FROM f2

4.4 Motions and Device operation

4.4.1 The basic *MOVE* statement

The basic *MOVE* statement is of the form:

MOVE <controllable frame> TO <dest> <modifying clauses>

which will cause the specified arm to be moved so it has the same position and orientation as the destination frame expression *<dest>*. A grinch sign, "⊛", can be used in *<dest>* to represent the current position of *<controllable frame>* when the motion is executed. *<Controllable frame>* may be either an actual manipulator (*barm* or *yarm*) or a frame which has been affixed to one of the arms. In the latter case, the physical relationship between the frame and the arm, described by the affixment chain connecting them, will be used so the motion results in the frame being moved to *<dest>*. The motion may be modified in many different ways through the use of the various *<modifying clauses>* described below.

4.4.2 Intermediate points: *VIA, DEPARTURE & APPROACH*

In the case where a motion must go through a series of intermediate points (to avoid obstacles, for instance), the intermediate frames may be specified by means of a *VIA* clause, such as:

VIA f1,f2,f3,f4,f5

where *f1,...f5* are frame expressions. The motion will pass through the points in the order they are specified. It is also possible to specify the arm's velocity at a via point, and the duration of the motion from the last given point to the via point. This full *VIA* clause looks as follows:

VIA f WHERE VELOCITY = <v>, DURATION <rel> <n> THEN <statement>

where *v* is a velocity vector, *n* is a time scalar and <rel> can be ≤, = or ≥. Note that unlike the first mentioned form, only one frame *f* may be given in this format. One or both modifying clauses of velocity and duration may be present, in either order. If the *THEN* part is included, the <statement> will begin execution when the motion reaches the VIA point f. The user is responsible for ensuring that <statement> can in fact be executed during motion. (An example of an inconsistency is for <statement> to ask the current arm to move to another location without first stopping the arm.) To execute a statement at the beginning or end of motion, make use of the *DEPARTING* or *ARRIVAL* condition monitor described in section 4.4.4.1.

It is also possible to specify deproach points, which are points associated with departure of the arm from its current location, or its approach to the destination location. Unlike via points, deproach points are expressed with respect to the initial or destination coordinate systems. The clauses are as follows:

WITH DEPARTURE = <exp> THEN <statement>

and

WITH APPROACH = <exp> THEN <statement>

where *<exp>* may be as follows. Depending on whether the *APPROACH* or *DEPARTURE* clause is used, *<fr>* represents either the destination frame or the current location.

type of *<exp>*:	deproach point in station coordinates:
frame	<fr> * <exp>
vector	<fr> + <exp> WRT <fr>
scalar	<fr> + (<exp> * zhat) WRT <fr>

It is also possible to indicate that no deproach point is to be used by specifying *<exp>* as *NILDEPROACH*.

The AL predeclared macro *DIRECTLY* expands into the two clauses:

WITH DEPARTURE = NILDEPROACH
WITH APPROACH = NILDEPROACH

4.4.3 Force and Stiffness control

The blue arm has a force sensing wrist with eight semiconductor strain gages mounted just above the fingers. (It is the silverly looking disk between the hand and the wrist.) This force sensing wrist makes it possible to have the blue arm sense and apply specified forces and moments. (Sensing forces is discussed in Section 4.4.4 below.) The following clauses currently are only valid for motions of the blue arm. In the near future, force wrists will be mounted on the PUMA arms.

At the moment, force application (section 4.4.3.2), sensing (section 4.4.1) and stiffness (4.4.3.1) can be done with respect to a coordinate frame that is moving with the hand. It is expected that additional code will be added in the future to allow specification in a fixed coordinate system.

For the sake of completeness of description, the syntax will be given for specifying the coordinate system of reference to be either HAND or WORLD. However, AL will currently ignore the coordinate frame of reference if one is given and act as if it were in the hand coordinate system.

4.4.3.1 Spring force application - stiffness

The stiffness clause specifies the apparent stiffness of the object in the hand while following the specified trajectory. The object follows the nominal trajectory specified in the *MOVE* statement if no constraint is encountered during the motion. If the object encounters a constraining surface, then the contact force applied will depend on the stiffness coefficient along the direction of contact. The modifying clause that permits this takes one of the following two forms:

WITH STIFFNESS = (v,v) ABOUT <frame> IN <coord sys>
WITH STIFFNESS = (s,s,s,s,s,s) ABOUT <frame> IN <coord sys>

The first *v* (or the first three *s*'s) has dimensions of *FORCE/DISTANCE* and the second *v* (or the next three *s*'s) has dimensions of *TORQUE/ANGLE*. The components of the first *v* represent the force spring constants in the X, Y, and Z directions, while the components of the second *v* represent the torque spring constants about the X, Y, and Z directions. (In the second form, the first three scalars represent the force spring constants in the X, Y, and Z directions, while the next three scalars represent the torque spring constants about the X, Y, and Z directions.) The ABOUT part defines the center of compliance about which the arm complies, and if it is left out, has a default of NILTRANS. <Coord sys> can be either *HAND* or *WORLD*. This permits the arm to behave as a programmable center of compliance. Note again that currently control is possible only in the hand coordinate frame. A stiffness component of magnitude zero means that the arm will be compliant, i.e. move away from any external force in that direction.

4.4.3.2 Constant force application

In addition to the apparent spring forces and moments that specification of the STIFFNESS clause permits, constant forces and moments can be applied using the FORCE and TORQUE clauses. To avoid incompatible requests the force components must always be orthogonal. To insure this, a force frame must be specified, and the directions of the applied forces and moments must be aligned with one of the cardinal axes of this current force coordinate system. Also specified is whether the orientation of the axes changes as the hand moves, i.e. is the force frame defined relative to the hand or the table (world) coordinate system. The clauses to do all this are as follows:

WITH FORCE = <sval> ALONG <axis-vector> OF <frame>
IN <coord sys>
WITH TORQUE = <sval> ABOUT <axis-vector> OF <frame>
IN <coord sys>

WITH FORCE_FRAME = <frame> IN <coord sys>
WITH FORCE = <sval> ALONG <axis-vector>
WITH TORQUE = <sval> ABOUT <axis-vector>

or

WITH FORCE_FRAME = <frame> IN <coord sys>
WITH FORCE(<axis-vector>) = <sval>
WITH TORQUE(<axis-vector>) = <sval>

where:
<axis-vector> = xhat, yhat or zhat.
<coord sys> = HAND or WORLD (WORLD is currently ignored)
<sval> = the magnitude of the force
<frame> = the orientation of the axes of the force frame

In the first form the specified force frame in all of the clauses must be the same. If *IN <coord sys>* is not specified, HAND is assumed, while if *OF <frame>* is omitted, *STATION* is assumed. Note that only one force frame may be specified per move.

A short form is also available for those motions which only need to apply or sense one force, but not both. It looks like either:

WITH FORCE = <sval> ALONG <vect> OF <frame> IN <coord sys>

or

WITH FORCE(<vect>) = <sval>

This generalizes in the obvious way for *TORQUE* and for force sensing. If no *<frame>* and *<coord sys>* are specified then a force frame in hand coordinates is automatically created with it's x-axis aligned along *<vect>*. Otherwise the specified

coordinate system is used and a force frame is created with it's x-axis along *<vect>* WRT *<frame>*.

Note again that the description of the world coordinate frame is given only for completeness, and that AL currently does force sensing in hand coordinates only.

4.4.3.3 Zeroing the force wrist

Since the force wrist readings are very sensitive, the readings will change with changing hand orientation even when there is no load, since the weight of the hand will be measured by the force wrist. Thus, it is necessary to take the readings with respect to the base readings. The base readings may be set or not set by means of the *FORCE_WRIST* clause as follows:

WITH FORCE_WRIST ZEROED
WITH FORCE_WRIST NOT ZEROED

In the first case, the wrist is read before the motion, and subsequent readings are with respect to this base reading. In the second case, the force_wrist is not read and the readings are given with respect to the last time base readings were obtained. The default case is *WITH FORCE_WRIST ZEROED*.

The wrist can also be zeroed by means of the *SETBASE* command as follows:

SETBASE;

4.4.3.4 Collecting force components

The force and torque vectors acting at the blue hand at any time (regardless of whether the arm is moving or at rest) can be obtained by means of the *WRIST* command which takes two arguments, v1 (a force vector variable), and v2 (a torque vector variable). The components of the resultant force and torque acting on the blue hand are returned in the variables v1 and v2. The syntax is as follows:

WRIST(v1,v2);

The *GATHER* clause permits specified components of force and torque acting at the blue hand to be collected during the motion for later graphic display on the PDP-10. The syntax is as follows:

WITH GATHER = (p1,p2,..,pn)

where each p represents one component to be collected, and is one of the following: FX, FY, FZ, MX, MY, MZ, T1, T2, T3, T4, T5, T6, TBL. The meaning of each of these is as follows:

FX	force component in X-direction
FY	force component in Y-direction
FZ	force component in Z-direction
MX	torque component about X-direction
MY	torque component about Y-direction
MZ	torque component about Z-direction
T1-T6	joint torques at Joints 1 through 6
TBL	data in table coordinates (rather than hand coordinates)

The force and/or torque plots can be seen if the GAL module is active when the program is executed (c.f. Section 5.6).

4.4.4 Condition monitors

4.4.4.1 Types: force, duration, event, boolean, arrival and departing

During the course of an arm motion it may be desired to monitor some condition, or set of conditions, and to execute an action if the condition has occurred. The condition monitor clause is used for this purpose. It has the following general form:

ON <condition> DO <action>

Currently the conditions that can be monitored include force sensing, duration, events, and various boolean expressions of variables. *<Action>* may be any valid AL statement or block. The only restriction is that if a motion statement is the only statement in *<action>* then it must be surrounded by *BEGIN* and *END* to prevent ambiguity.

The monitoring will begin with the start of the motion and continue until the motion terminates. If the monitor triggers, then after it finishes its action, it will become dormant and cease checking its condition. It is possible to modify this by use of the *ENABLE* and *DISABLE* statements described below (section 4.4.5.2).

When sensing forces and moments the following clauses are used:

```
ON FORCE <rel> <sval> ALONG <axis-vector> OF <frame>
                              IN <co-ord sys> DO <action>
ON TORQUE <rel> <sval> ABOUT <axis-vector> OF <frame>
                              IN <co-ord sys> DO <action>
```

or


```
WITH FORCE_FRAME = <frame> IN <co-ord sys>
ON FORCE <rel> <sval> ALONG <axis-vector> DO <action>
ON TORQUE <rel> <sval> ABOUT <axis-vector> DO <action>
```

or

```
WITH FORCE_FRAME = <frame> IN <co-ord sys>
ON FORCE(<axis-vector>) <rel> <sval> DO <action>
ON TORQUE(<axis-vector>) <rel> <sval> DO <action>
```

where: *<axis-vector>*, *<co-ord sys>*, *<sval>* and *<frame>* are the same as in section 4.4.4 above and *<rel>* is either ≥ or <, the condition monitor triggering when the force or moment exceeds or goes below the specified magnitude respectively. As in applying forces there is a short form when only one force is being sensed or applied:

```
ON FORCE <rel> <sval> ALONG <vect> OF <frame>
                              IN <co-ord sys> DO <action>
```

or

```
ON FORCE(<vect>) <rel> <sval> DO <action>
```

If only the absolute magnitude of a force is important, and not which direction along an axis, then the following may be used instead of the previous two clauses:

```
ON |FORCE| <rel> <sval> ALONG <vect> OF <frame>
                              IN <co-ord sys> DO <action>
```

or

```
ON |FORCE(<vect>)| <rel> <sval> DO <action>
```

This generalizes to TORQUE in the obvious way.

Note again that the description of force sensing in world coordinates is given for completeness. Currently AL does force sensing in hand coordinates only.

The condition monitor:

ON DURATION ≥ n * seconds DO <action>

will trigger its action *n* seconds after being enabled at the start of the motion.

ON <event> DO <action>

means do the action if *<event>* is signalled (by another condition monitor or some other parallel processes).

ON <boolean expression> DO <action>

has the effect of evaluating the boolean expression, made up of algebraic variables, and if it is true (non-zero) performing the desired action. If the expression is false the condition monitor goes to sleep for a short while (currently 100 milliseconds) before evaluating and checking the expression again.

Two other special condition monitors are used to synchronize statement execution with the beginning and end of a motion statement.

ON ARRIVAL DO <action>

has the effect of performing the desired action when the MOVE statement has been successfully performed.

ON DEPARTING DO <action>

causes <action> to commence when the MOVE starts up. Actually this has the same effect as ON *DURATION* = *<n>* to *<action>* where <n> is a snd) time duration.

4.4.4.2 *ENABLE* and *DISABLE* - labelled condition monitors

A condition monitor has two states: enabled and disabled. In the enabled state it will trigger its conclusion if the condition it is checking for occurs. In the disabled state the condition monitor is inactive. As mentioned above a condition monitor is enabled when the motion is started, and disabled upon the conclusion of the motion. Once a condition monitor triggers it will become disabled, unless it is explicitly reenabled. This reenabling is done by means of an *ENABLE* statement placed in the conclusion of the condition monitor.

With the *ENABLE* and *DISABLE* statements it is possible to change the state of an arbitrary condition monitor that has been named by putting a label immediately before the reserved word *ON*. The syntax of these statements is:

ENABLE <condition monitor>

and

DISABLE <condition monitor>

Prefacing a condition monitor with the reserved word *DEFER* will cause it to be initially disabled. It can then be explicitly enabled later. Here is an example where a condition monitor is initially disabled, and then after three seconds is enabled:

```
MOVE barm LO dest
    test: DEFER ON FORCE(zhat)≥ I 0 * oz DO STOP
        ON DURATION ≥ 3 * sec DO ENABLE test
```


4.4.5 User error handler - *ERROR* and *RETRY*

When there is an error during a motion, AL aborts the motion and, normally, awaits a user reponse. This brings the program to a temporary halt, and the user has to type at the terminal to resume execution. Some errors can be anticipated by the user and dealt with by the program by use of the *ERROR* clause. The syntax is like that of the condition monitor; however, the error code is checked after the motion, unlike other condition monitors where the condition is checked during the motion.

ON ERROR = <n> DO <statement>

The error code is indicated by the appropriate bits in the value of <n> being on. Currently, <n> can be the sum of any number of the following quantities: PANIC_BUTTON, EXCESSIVE_FORCE and TIME_OUT. These quantities actually represent error codes whose numerical values are predefined in AL. A PANIC_BUTTON error occurs when the user hits the panic button. An EXCESSIVE_FORCE error occurs if the arm tries to move too fast, or if it hits something, and the motor output torque required to continue moving becomes too high. The TIME_OUT error occurs when the motion time is too long (this sometimes happens when the NO_NULLING clause causes the arm to try to reach its destination precisely, but the error is not large enough to generate enough torque to bring it within limits).

One of the actions that can occur in <statement> (the body of the error handler) is the RETRY statement that will cause the aborted motion to be retried. If this instruction is present, it must not be inside a *FOR* loop.

Note also the error codes are recorded in the appropriate variable (i.e. *barm_error* for the blue arm, *yhand_error* for the yellow hand, etc.), each time the device is used, so that the error code may be referred to at a later point in the program if desired.

4.4.6 Other clauses: *DURATION, SPEED_FACTOR, NULLING & WOBBLE*

Here are some other clauses that can be used to modify motions.

WITH DURATION <rel> <sval>

causes the resulting motion to take the amount of time specified by *<sval>*, which should be of dimension *TIME*. <rel> can be ≤, = or ≥.

WITH SPEED_FACTOR = <sval>

describes the speed of motion. The nominal time for the motion computed by AL

will be multiplied by <sval> which should be ≥ 1, and this product will be used as the time for the motion. There are four predefined macros: *QUICKLY*, *NORMALLY*, *SLOWLY*, and *CAUTIOUSLY* which will set the speed factor for the motion to 1, 2, 4 and 6 respectively.

The default speed factor for motions is 2, so the arm moves at a reasonable speed. This can be changed by assigning the desired default multiplier to the predeclared variable *SPEED_FACTOR* with a regular assignment statement:

SPEED_FACTOR ← <new default speed factor>

There are also three predefined macros: *QUICK*, *SLOW* and *CAUTIOUS*, which set the default speed factor to 1, 4 and 6 respectively.

WITH NULLING

informs the runtime system to null out errors at the end of this motion. There is also a *WITH NO_NULLING* clause which is the current default. There are two macros PRECISELY and APPROXIMATELY which achieve the same results.

WITH WOBBLE = <sval>

adds a small sinusoidal motion to the outer three joints causing them to shake a bit. It is useful for breaking small friction forces and for seating parts. <Sval> is a small constant of dimension *ANGLE* that is usually about 2 or 3 degrees.

4.4.7 Controlling the fingers: *OPEN*, *CLOSE* & *CENTER*

The fingers can be controlled in several ways.

OPEN <hand> TO <sval>

and

CLOSE <hand> TO <sval>

causes the fingers to open or close so that they are a distance <sval> apart. <Sval> is any scalar expression of dimension *DISTANCE*. Currently there is no difference between the *OPEN* and the *CLOSE* statement. Eventually *CLOSE* will stop the motion of the fingers if both touch sensors are triggered. For binary hands (e.g. the current hands on the PUMA arms) the TO clause is omitted.

CENTER <arm>

closes the fingers of the specified arm until both touch sensors indicate contact has been made. Furthermore if one finger makes contact before the other, *CENTER* causes the arm itself to move so that the object being grasped is not

pushed by the finger. *OPEN* and *CLOSE* only move the fingers, and if the object being grasped is not centrally located between the fingers, the object will be moved or, if it is fixed in place, excessive force might be exerted by the fingers, thereby aborting the motion.

4.4.8 *STOP* & *ABORT*

There are two ways of terminating motions before they finish:

STOP <device>

and

ABORT(<print list>)

The *STOP* statement causes the indicated device to stop. <Device> may be a physical manipulator or a frame affixed to an arm. If <device> is not specified, and the stop statement appears in the scope of a move statement, then the arm used for the motion will be the one stopped. The *ABORT* statement is used for more drastic occasions. It will stop the motion of all devices, print out the elements of the <print list> (see the description of the *PRINT* statement, section 4.5.7, below), and transfer control to 11DDT. The user may continue the program execution by typing <alt>P to 11DDT. Usually these statements appear in the body of condition monitors, though they may be appear at any point in the program.

4.4.9 Other devices

4.4.9.1 The *OPERATE* statement

The *OPERATE* statement is provided to control devices interfaced to the AL system. Its syntax is similar to that of the *MOVE* statement:

OPERATE <device> <modifying clauses>

where <device> is the device being controlled, and the <modifying clauses> describe what action the device shall perform. Currently only the socketdriver is available, so only its syntax will be given.

```
OPERATE driver WITH ANGULAR_VELOCITY = n * rpm <direction> ;
OPERATE driver WITH TORQUE = n * oz * inches <direction> ;
OPERATE driver WITH DURATION = n*seconds;
```

The first two clauses are mutually exclusive since we can only consistently specify either the angular velocity or torque. <direction> is either CLOCKWISE or COUNTER_CLOCKWISE and specifies the direction of rotation. CLOCKWISE and COUNTER_CLOCKWISE can be abbreviated as CW or CCW respectively. The default direction is clockwise. The DURATION clause, similar to that in the MOVE

statement, specifies the duration of the motion.

Event, expression and duration condition monitors can be applied to the OPERATE statement as in the MOVE statement. One variable that is of interest in expression condition monitors for the OPERATE statement for the driver is DRIVER_TURNS, which keeps track of the number of rotations of the driver since the beginning of the current statement. Additional condition monitors that check angular velocity and torque acting on the driver will be added in the future.

4.4.9.2 The ADAC interface

There is an ADAC interface interface with 64 A/D channels and 4 D/A channels. The AL system can read the values of the voltages on the A/D convertor by means of the function ADC which takes as argument a channel number between 0 and 63. Output from the program is possible on channels 1 through 4 of the D/A convertor. The syntax of the statements is shown below:

```
x ← ADC(i)    {assigns to x the value in volts on channel i where
                 0 ≤ i ≤ 63}
DAC(j,v)      {this statement outputs v volts on channel j
              where -10≤v≤+10 and 1≤j≤4}
```

4.4.9.3 The Vision Module

A Machine Intelligence Corporation VS-100 Vision Module is interfaced to the AL system and can be accessed by means of a number of auxiliary AL procedures. These procedures are defined in the file VISION.AL[AL,HE] which must be included in the user program by the statement

```
REQUIRE SOURCE_FILE "VISION.AL[AL,HE]";
```

The text of this file is included in Appendix VIII and further documentation is available in the files VM.DOC and COMM.TXT on [DOC,HE], and in the manufacturers' literature. (The actual communication between AL and the Vision Module is achieved by means of the *VM* command, which is of interest only to AL wizards.)

4.4.9.4 The VAL controllers

The PUMA arms are directly controllable under AL. Sometimes it may be necessary to communicate with them directly through VAL which lives on the LSI-11 in the PUMA controller. A string can be sent to VAL by the following AL command:


```
VAL("string") ;
VAL("string",WAIT);
VAL("string",NOWAIT);
```

Once the string has been sent over to VAL, AL will either wait for VAL to finish the command (default), or immediately proceed to the next AL statement. Note that the hardware configuration to drive the PUMAs using AL directly and through VAL are different, and the user should ensure that the hardware is set up appropriately.

4.5 Non-motion statements

4.5.1 Assignment statement

The assignment statement:

<variable> ← <expression>

causes the value represented by *<expression>* to be assigned to the variable appearing to the left of the assignment symbol. The data type and physical dimension of the expression on the right hand side of the assignment symbol must be the same as the data type and dimension of the variable on the left hand side.

Assignments are valid for scalars, vectors, rots, transes, frames and strings.

4.5.2 Traditional control structures: *IF, FOR, WHILE, UNTIL, CASE*

AL has many of the traditional ALGOL control structures.

The *IF* statement has the form:

IF <boolean expression> THEN <statement> ELSE <statement>

The *ELSE* part is optional. If *<boolean expression>* is true (non-zero) the statement following the *THEN* is executed. Otherwise the statement following the *ELSE*, if present, will be executed.

The *FOR* loop has the form:

FOR <s var> ← <s expr> STEP <s expr> UNTIL <s expr> DO <statement>

where *<s var>* is a scalar variable and the *<s expr>*'s are scalar expressions of the same dimension. The initial value of the variable is the value of the first expression; every time the statement is executed, its value is incremented by the value of the second expression, and the process repeats until the value exceeds

that of the third expression. If the step size is negative, the right things happen. The test is made before the first iteration, so it is possible that the loop will not be executed at all.

The *WHILE* loop is as follows:

WHILE <boolean expression> DO <statement>

The boolean expression is checked and if it is true the statement is executed. This process is repeated until the condition becomes false.

The *UNTIL* statement is as follows:

DO <statement> UNTIL <boolean expression>

where the statement is repeatedly executed until the condition becomes true. This is similar to the *WHILE* statement described above, with the exception that the *WHILE* loops while the condition is true, whereas the *UNTIL* loops until the condition is true.

There are two forms that the *CASE* statement may take. The regular *CASE* statement has the form:

CASE index OF BEGIN S0; S1; S2; ... Sn END;

The index is evaluated and depending on the integer part of its value one of the statements will be executed. If the index is zero then S0 is chosen, if the index is one then S1 is chosen, and so on up till n. If the index is negative, or greater than the number of statements, an error is reported. Any of the statements may be null, e.g. "S1;; S3", in which case if the index were two no statement would be executed.

There is also a numbered version of the *CASE* statement:

CASE index OF BEGIN [C0] S; [C1] [C2] S; ... [Cn] S; ELSE S END

where each statement has one or more non-negative scalar constants labelling it. The index expression is again evaluated and if it is the same as one of the Ci's then the statement with that label is executed. If no constant matches the index then nothing is done, unless an *ELSE* is present in which case the statement it labels is executed. If the index is negative or greater than the largest Ci an error occurs, unless there is an *ELSE* present. Note that the *ELSE* statement may appear anywhere in the list of statements, not necessarily at the end.

4.5.3 Procedures

Procedures are defined as follows:

<type> PROCEDURE <name> (parameters);
<statement>;

where the statement is executed each time the procedure is called. Only those procedures that return a result need their type specified. The data types of the parameters may be modified by the reserved words: *VALUE* and *REFERENCE*. Reference is the default. It is necessary when defining a procedure to specify the dimensions of any arrays that are to be used as a parameter so that the number of dimensions associated with the array in the procedure body can be checked. For example:

PROCEDURE foo(FRAME ARRAY pnts[1:4,1:3]);

Procedures can return a result by means of the *RETURN* statement which has the form:

RETURN (value)

which returns value as the result of the procedure. The *RETURN* statement may not appear inside condition monitors or *COBEGIN-COEND* blocks.

Procedure calls take the normal form of the procedure name followed by the list of arguments: name(arglist). They may appear anywhere an expression might, or alone by themselves as a procedure statement. If a typed procedure appears in a procedure statement then the result it returns will be discarded.

4.5.4 Parallel control: *COBEGIN-COEND, SIGNAL & WAIT*

In addition to the normal sequential execution of statements within a *BEGIN-END* pair, AL allows blocks of code to be executed in parallel by placing them in a *COBEGIN-COEND* block. Upon entering the *COBEGIN* block control is divided among the various processes to be executed simultaneously. Upon the termination of all of these processes control will be passed to the part of the program following the *COEND*. It is the user's responsibility to ensure that the code being executed in parallel is sufficiently independent (e.g. two processes don't try to use the same arm at the same time), and that no deadlock situations occur.

The purpose of the *COBEGIN* construct is to allow simultaneous independent manipulator control. It is not particularly useful to execute purely computational code in parallel, though doing computation while an arm is moving can

save time. The scheduling algorithm used is to start up one process and execute it until it is blocked, and at that point another process will be run. A process can be blocked by waiting for an event, by pausing, doing I/O, or by initiating a motion.

Parallel processes may be synchronized by means of explicit events and *SIGNAL* and *WAIT* statements. With each event is associated a count of how many times it has been signalled. Initially, the count is zero, that is, no signals have appeared, and no processes are waiting. The statement:

SIGNAL e1

increments the count associated with event *e1*, and if the resulting count is zero or negative, one of those processes waiting for *e1* is released from its wait and readied for execution. The statement:

WAIT e1

decrements the count associated with event *e1*, and if the resulting count is negative, the process issuing the WAIT is blocked from continuing until another process signals *e1*. If the count is zero or positive, there is no waiting.

4.5.5 Statement condition monitors

Condition monitors, besides modifying motions, may also appear as statements. The description in section 4.4.5 also applies to statement condition monitors. When its defining statement is executed the statement condition monitor will become enabled. It will become disabled when it triggers, is explicitly disabled (it must be labelled for this to occur), or its local block is exited. The reserved word *DEFER* still causes a condition monitor to be defined in an initially disabled state.

Scope rules come into play regarding when condition monitors may be enabled or disabled. An enable or disable statement may only refer to a condition monitor that is defined in the same block as itself or in a block containing it.

4.5.6 *PAUSE* statement

The statement:

PAUSE <sval>

will result in the program going to sleep for the time specified by *<sval>*, which should be of dimension *TIME*.

4.5.7 I/O

At runtime strings and variable values may be typed out using the *PRINT* statement:

PRINT(<arg1>,<arg2>,...,<argn>)

where the *<arg>*'s are either algebraic expressions or variables, or string constants. Strings are delimited by double quotes. *CRLF* is a predefined string which prints as a carriage return followed by a line feed.

The statement:

PROMPT(<print list>)

is syntactically like the *PRINT* and *ABORT* statements. Upon encountering a PROMPT statement the AL runtime system prints out all the items in the print list and then prints the message:

"Type P to proceed"

and waits for a P to be typed. Unlike the ABORT statement control does not pass to 11DDT and hence any parallel processes (e.g. COBEGIN) will continue to be executed. As an example:

PROMPT("Move barm to work station origin"); org ← barm;

There are two arithmetic operators to read in a value from the VT05 console. *INSCALAR* reads in a scalar, prompting the user with: "SCALAR, please: ". *QUERY* reads in a boolean. It is like *PROMPT* in that it can have a print list. After typing the print list the user is asked to "Type Y or N: ". For example:

PRINT("How tall is casting?"); height ← INSCALAR;
WHILE QUERY("More to do?") DO ...

4.5.8 Macros

AL possesses a general purpose text macro facility. The syntax for a macro definition is:

DEFINE <macro id> <parameters> = ⊂<macro body>⊃

where *<macro id>* is the name of the macro, *<macro body>* is the text to be substituted whenever *<macro id>* is encountered in the program, *<parameters>* if present is a list of arguments for the macro, separated by commas and enclosed by

parenthesis. Only undeclared identifiers may be used as macro parameters. When the macro is expanded the actual arguments will be substituted into the macro body wherever the parameters appear. If this value is anything other than a simple token it must be surrounded by the delimiters ⊂⊃. The *<macro body>* is also delimited by ⊂⊃.

Here are two examples of the use of macros:

```
DEFINE feet = ⊂12 * inches⊃;
DEFINE grasp(frob) = ⊂MOVE barm TO frob;
                      CENTER barm;
                      AFFIX frob TO barm RIGIDLY⊃;

size ← 10.4 * feet; {Expands to 10.4 * 12 * inches}
grasp(handle);      {Expands to:
                     MOVE barm TO handle;
                     CENTER barm;
                     AFFIX handle TO barm RIGIDLY;}
```

4.5.9 *REQUIRE* statement

REQUIRE statements allow the user or his program to communicate with the AL compiler. No code is generated as a result of a *REQUIRE* statement, and the effect of the *REQUIRE* statement is global and persists after exiting the block in which it was invoked. Another *REQUIRE* statement or some other termination condition is necessary to undo or stop the effect.

REQUIRE SOURCE_FILE "<file_name>"

The file named will be the source of future input until an end of file is encountered, at which time the code following the require will be read. The source file will be assumed to be a disk file, unless specified as a teletype file by "TTY:" in front of its name.

A teletype file does not need a name, but if it has one, the teletype input will be saved on a disk file with the given name and default extension TTY. Parsing action on teletype inputs will begin each time a carriage return is hit. The file is closed by typing a <control><meta><linefeed>. The current operating system allows only one teletype file to be open at a time.

The file name can be one of:

"NAME"
"NAME.EXT"

"NAME[P,PN]"
"NAME.EXT[P,PN]"

where P and PN represent the project and programmer names respectively.

REQUIRE MESSAGE "<message>"

Anything appearing within the double quotes will be printed out at the user's terminal.

REQUIRE ERROR_MODES "<mode flags>"

While the AL parser may ask for user responses to errors during program compilation, it is possible to predefine the standard treatment of errors by setting certain flags with the *REQUIRE ERROR_MODES* statement. The flags are set by including the relevant letter within the quotes, and reset by including a minus sign in front of the code letter. The following flags are available:

- L - errors, if any, will be logged in a file with extension LOG
- A - compilation will continue automatically after each error message is printed.
- M - the system will prompt the user only for modifiable errors
- F - strict dimension checking will not be carried out across assignment statements, condition monitors, etc. Undimensioned variables will be coerced according to the context in which they appear. Error messages will be generated only for inconsistent usage.

REQUIRE COMPILER_SWITCHES "<compile switches>"

All the switches that are used in the command line (see Chap 5) can be specified here. This is an alternative to specifying the switches in the command line. Only letters (without the slash) should be within quotes.

5. USING AL

This chapter describes the steps involved in compiling and executing an AL program and what to do in the event of errors. In the following description where commands are typed by both the user and by the system, the system response will be shown in italics.

5.1 Compilation of user programs

To compile and prepare the binary load module for the PDP-11 do the following:

1. Create a file called "FOO.AL" with your program in it, where "FOO" may be any name you wish.

2. Get your job to monitor level and type "COMPILE FOO".

2a. The system program SNAIL which handles requests like COMPILE will give the message

Swapping to SYS: AL. DMP

and then start AL at the parser. The parser will then say

AL: FOO

When the parser hits a page boundary in your file, it will type "*1*" or whatever the number of the page that it is starting to read.

2b. When the parsing is complete, the parser swaps to the AL compiler, which types "*ALC*".

2c. When the compiler completes and code emission, it deletes the S-expression file and swaps to the cross-assembler PALX for the PDP-11. "*PALX n*", where *n* is the version number of the PALX compiler, is typed out at the user terminal.

2d. The PALX compiler swaps to ALSOAP, which cleans up the user area by deleting the intermediate file with extension .ALP, that is created during the compilation of the AL program.

2e. The job gets back to monitor level.

If you misspell the name of your file then SNAIL will complain

File not found: FOO

where "FOO" is your misspelling.

At any time during steps 2a through 2e above, you could get an error message from the parser, compiler or PALX. See section 5.4 about these.

5.1.1 Compilation with switches

Compilation may be done with switches if desired by including the desired switches within parentheses as "COMPILE FOO.AL(KS)". Switches are relevant only if you are modifying or debugging the AL system. Effects of the different switches are shown below:

K	Keep the intermediate .ALP file
S	Generate a symbol file (.ALS)
L	Generate a PALX assembly listing
I	Keep the .SEX file
X	extra switch for new trial compilations (to generate a binary file compatible with ALX.SAV)
N	swap to new AL compiler ALCNEW instead of ALC
B	run BAIL immediately after scanning the command line;

5.2 Loading and executing the AL program

When your program "FOO.AL" has got through to ALSOAP without grief, you are ready to execute the program on the PDP-11.

1. Type "DO AL[AL,HE]" followed by carriage return. This initiates a series of instructions which are described later (5.3). When you see

f=

type "FOO", the name of your program, followed by carriage return. You will then see a number of lines printed out. The last line will be

DDT STARTED AT 130000

2. Locate the brake control box(es) for the blue and/or yellow arm(s), and the position(s) of the panic button(s), and check that all the brake switches are pointing away from the RELEASE position. Keep your finger poised over the panic button of the appropriate arm at all times while the AL program is being executed (procedure for starting it is in step 3), and be prepared to press it immediately if it should appear that something unpredictable or disastrous is about to happen. Pulling the yellow cord that runs around the table will turn off power to the arms,

and can also be used in the event of an emergency. Take care not to lean on the cord accidentally. If you are using the green and/or red PUMA arm(s), note that they do not have control boxes if you are using them under AL. In that case, pull on the yellow emergency cord to cut off power to the arms.

Locate the red button on the underside of the short end of the hand-eye table closest to the Stanford arms. Press it to turn on power for the arms.

If you are using either or both PUMA arms, locate the appropriate controller box(es) (under the hand-eye table), and turn the controller on by flipping the toggle switch if it is not already on. Make sure the black rotary switch is in the RUN position.

3. Now go to the VT05 which is a white colored terminal with a dark screen in the area of the hand-eye table, and on it, you should see an asterisk "*" and a flashing cursor. Make sure you have the panic button under your thumb and then type

* START <alt><alt> G

to begin execution of your program. Note that just <alt>G will also work. AL will print out at the VT05:

AL RUNTIME SYSTEM

4. If you are using either of the PUMA arms, AL will ask you to turn on the PUMA arm power as follows:

ARM POWER FOR PUMA(S) ENABLED. TURN ARM POWER ON NOW.
(TYPE <CR> TO CONTINUE)...

This is done by pushing on the black button below the light and the the "ARM POWER ON" label on the front of the PUMA controller. The red light above the "ARM POWER ON" label should then come on. Then type <CR> at the VT05. AL may then type the following:

GREEN ARM CRUDELY INITIALIZED...

This indicates that the green arm is not accurately calibrated. AL then asks:

DO YOU WANT TO CALIBRATE THE GREEN PUMA?

Type Y or N followed by <CR> depending on whether or not you want the arm to be recalibrated. Generally it is a good idea to calibrate the arm. If you type Y, each of the joints will move a short distance, and after that, AL will type the

following:

GREEN PUMA IS CALIBRATED.

The GREEN above would read RED if the RED arm is used in the program. If both PUMA arms are used in the program, AL will ask questions for both of them.

5. The VT05 will beep just before the start of each motion by the arm. Messages or values will be printed where appropriate. When program execution is complete, the following message will appear:

ALL DONE NOW. SEE YOU AROUND!
ELAPSED TIME = 24.928 SECONDS

NO ACTIVE PROCESSES LEFT. YOU'RE IN DDT.

BE; SRFADL+50>>BPT

Type "<alt>G" to re-execute the program from the beginning.

6. When you have finished using the AL system you should ALWAYS turn off power to the arms by pulling on the yellow cord that runs around the table. Then it does not matter if somebody hits <alt>G, and it also makes it impossible for someone not at the hand-eye table to accidentally run a program that tries to move the arm. So make a habit of pulling the yellow cord on your way out. Also, type "X" or <CALL> to 11TTY to terminate execution of that program. Finally type "D ARM" to deassign the arms.

5.3 Complete runtime execution sequence

The following is the complete sequence of operations required to load and execute an AL program once a binary file has been prepared. It is given in case some error occurs when the user types a DO AL[AL,HE].

1. Type "A ARM" to have the arms assigned to your job.

2. Type "R 11TTY" to execute the program that loads your program into the PDP-11. 11TTY will respond with

CORE SIZE = 28K
VERSION USING <device>
TYPE ? FOR HELP
*

where device is either VT05 or TERMINAL. The asterisk is 11TTY's way of

prompting for user input.

The way to change (toggle) between the two devices is to type "V" immediately after the asterisk, and 11TTY will fill in the rest of the line and ask for the next prompt as follows:

VERSION USING <other device>
*

An alternate way to get the device of your choice is to use an extended command by typing "A" followed by "VT05" or "TERM" to select the desired device.

AN EXTENDED COMMAND VT05
*

It is desirable to use the device VT05 so that once execution starts, you can be independent of the PDP-10.

3. Type "Z" to zero out the core, followed by the memory size, currently 500000, then a carriage return to confirm the instruction. 11TTY will respond as follows:

ZERO CORE [CONFIRM] 500000<cr>
*

4. The AL interpreter and the runtime system is then loaded by typing "G" for getting the core image binary file, followed by the name of the file AL[AL,HE] and a carriage return.

GET SAV FILE - AL[AL,HE]<cr>
*

5. The user's AL program binary file is then loaded by means of typing "O" for overlay, followed by the name of the file and a carriage return.

OVERLAY BIN FILE - FOO<cr>
*

6. The next step is to get the program started by typing "S" then "D" followed by a carriage return.

START AT (1000) (D FOR DDT) - D<cr>
DDT STARTED AT 130000
*

7. Now go to the VT05 and after making sure you are ready to push the panic button type

*START <alt><alt>G

AL will print out at the VT05:

AL RUNTIME SYSTEM

Any other input or output will be typed at the VT05.

When program execution is done, the following message will be printed out at the VT05.

ALL DONE NOW. SEE YOU AROUND!
ELAPSED TIME = 24.928 SECONDS

NO ACTIVE PROCESSES LEFT. YOU'RE IN DDT.

0£, SRFADL+50>>BPT

Typing <alt>G will re-execute the program from the beginning.

5.4 Error Corrections and Recovery

Errors can occur at various stages during program compilation and execution, and it is important to be able to continue from the error point as gracefully as possible. Some errors may be patched up according to the wishes of the user, while others may be fixed up by the AL system, with the user having no say other than whether to continue with the execution or to abort it. This section attempts to describe the kinds of errors encountered during program compilation and execution, and what action the user can take when such errors do occur.

5.4.1 Parsing errors

Errors detected in the parsing phase are easiest to correct and patch. For minor errors it is possible to proceed after correction without going back to the source file.

The parser outputs error messages, and gives the user the option of
- (a) editing the source file
- (b) aborting the compilation
- (c) taking the standard fixup
- (d) backing up to and changing the source code from the

beginning of the innermost statement.

The last feature is particularly advantageous when the compilation is a long one, and the error is a minor one which can be easily corrected - e.g. errors which are due to misspellings, missing operators, and even some simple cases of syntactically incorrect statements.

Error messages are generated whenever the parser comes across something it does not like. Some messages are warning messages which tell the user what he should not do in the future. An example of this is the case where identifiers are declared in a block but never referenced, resulting in carrying more variables than necessary.

The most common errors are dimension and type incompatibility. Dimension checking is done across assignment statements and force, torque, and duration expressions and conditions. Whenever there is inconsistency, an error message is generated. While dimensional inconsistency may not cause any grief during execution of the program, checks for it enable certain errors (e.g. wrong variables being used) to be pinpointed early during the compilation phase. A more serious error occurs when the data types are incompatible (e.g. assigning a vector expression to a scalar variable), and needs to be corrected, as otherwise the error will cause trouble in the compilation and execution phases.

Dimension checking can be made less stringent by including the F switch in the REQUIRE ERROR_MODES statement. In this case, dimensionless variables will be coerced to the type that makes them compatible with the other terms in the expression or statement

TYPICAL ERROR MESSAGES:

TYPE MISMATCH

This message is printed when an identifier, factor or term is ○ a different type than that expected in the context of the expression.

DISTANCE DIMENSIONS DON'T MATCH ON ASSIGNMENT STATEMENT

The meaning is obvious, but the error is not serious and AL will allow the code to continue compiling, since dimension checking is not done during execution of the program.

BLOCK NAME AT END DOES NOT AGREE WITH THAT AT THE BEGINNING

This error occurs when there is a misspelling in the names within strings at the corresponding places, or if the *BEGINs* and *ENDs* are mismatched.

TRYING TO ASSIGN VALUE TO ARM OR DEVICE

The user is trying to assign a value to an arm or a hand. This is disallowed in a program because the values reflect the state of the real world during execution, and cannot be changed by the user.

<variable> NOT DEFINED, WILL DEFINE IT.

The user has put an undeclared variable on the left hand side of an assignment statement. This message could be due to a misspelling.

An error message, followed by *CONTINUE WILL FLUSH STATEMENT*

This means that the parser will be unable to do any form of fixup, and that it will just flush the statement by ignoring any further text until the next semi-colon is read.

ERROR CORRECTION

Whenever the parser detects an error, it prompts the user with a "$". The user should respond with a single character as follows:

C or <cr>	continue with standard fixup
<lf>	continue automatically for non modifiable errors.
A	continue automatically with standard fixup for future errors.
E	edit source file at the place the error occurs
R	restart the program - type in the command line
T	terse information giving only the different options available
V	verbose information giving characters and their effects
X	exit from program
L	log errors in a logging file
M	modify source code - user will be presented the offending line
B	invoke BAIL for debugging the AL parser (useful only if debugging AL and BAIL is loaded in the system)

Any other character will cause either a list of the above information to be printed out, or just a list of the possible options.

The most useful response for the user is "M", "C" or "E". The first is particularly useful when a minor error (e.g. spelling error) occurs towards the end of a long compilation, and the user does not want to have to start from the

beginning again. "C" is useful when the error can be corrected by a standard fixup, while "E" is used to correct more serious problems by going back to the source file.

Note that the "M" option is not always available. There are situations where interactive error recovery is impossible - e.g. when in the middle of a macro expansion, and so the user is not allowed to make any changes.

If any errors have been corrected interactively, at the end of the parsing phase AL will ask the user if an updated copy of his source file is to be saved.

5.4.2 Compiler errors

You should not get any error messages from the compiler. If you do get any messages, it is probably due to a bug in the compiler and should be reported.

5.4.3 PALX errors

The principal PALX error occurs when the program is very long. PALX then gives the message that *there might not be enough space*, in which case, the program should be broken down into smaller subprograms. If any other error message is given by PALX, it is an AL bug, and the user is requested to report it.

5.4.4 Loading errors

11TTY is the program that loads the PDP-11 with the core image of the AL interpreter and runtime system. The instruction "DO AL[AL,HE]" has the effect of a number of instructions which includes assigning the PDP-11 to your job, zeroing the memory of the PDP-11, loading the AL interpreter and runtime system, overlaying it with the user program load module and starting 11DDT on the PDP-11. Further details are given in section 5.3.

There are several things that could go wrong during this sequence of events. The message will be printed out at the terminal of the user.

Already assigned to job 33
?
Arm is busy. Will you wait?

This message is printed when some other job has the arms assigned to it.

When this happens, you may type Y and wait until the ARM becomes available, or you can find out who is using it by getting back to the operating system by hitting <CALL> and then typing

PJ ARM

to which the system will reply that the ARM is not is use, or indicate which job has the ARM assigned to it. If the job is not using the ARM, you should request that the ARM be deassigned, and then try the "DO" instruction again. If the job is using the ARM, you should try again later.

PDP-11 STOPPED, RESTART

Restarting the entire sequence from Zeroing the core should take care of this problem. If it does not, it should be repeated.

NO RESPONSE WHEN YOU TYPE ANYTHING ON THE VT05.

If there is no response on the VT05 when you expect some output e.g. when you do not get the asterisk and the flashing cursor, 11TTY may be in "TERMINAL" rather than "VT05" mode. Type ∀ several times on the terminal and let the mode toggle from one to the other until "VT05" mode is obtained.

5.4.5 Runtime errors

During the execution of the user program, several things can cause the program to stop. The following are the common error messages that are printed on the VT05 by the runtime interpreter.

5.4.5.1 Motion associated errors

Here are errors associated with the arms and other devices that occur during motions. The device name at the beginning tells which device ran into trouble. For arms the number at the end tells which joint ran into problems. Joints are numbered outwards from the shoulder. Joint 7 is the hand.

PANIC BUTTON PUSHED

This error occurs when the panic button is pushed, or someone has leaned on the edge of the table, thereby pulling on the yellow cord, and shutting off the power supply. RETRY<alt>G will try the current motion again if the panic button was pushed, but it will give the next message if the yellow cord was pulled.

ARM INTERFACE POWER SUPPLY TURNED OFF
(CHECK JOINT BRAKE SWITCHES)

When this error message appears, check all the brake switches on the panic button box, and make sure that all the brakes are applied. If any of the brakes are in the released position, toggle them to the set position, and then try again by

typing RETRY<alt>G. If you get the same error again, press the large red button on the underside of the short side of the table nearest the wall to turn on the arm power, and try again. If you get the error again, it may be that the arm interface power really is off at the source, in which case you should get help from one of the personnel in the lab.

RARM - EXCESSIVE FORCE ENCOUNTERED BY JOINT n
to retry the move, RETRY$G
to move arm directly to destination, FINISH$G

This error occurs when the movement to be made requires too high a force. It could occur when:

(1) the arm encounters an object during the course of the motion (get it out of the way)
or (2) the time specified for the motion is too short (make it longer next time)

BARM - TIME OUT FOR JOINT n

This occurs usually at the end of a motion when the arm is prevented from going to its final destination but the error is insufficient to cause a high enough motor torque requirement (0 give a joint force error.

YARM - STOP LIMIT EXCEEDED FOR JOINT n

There is a software joint operating range which is lower than the hardware joint operating range for safety purposes, and when the limits are exceeded, this error message is generated. Usually this message occurs if continuation of compilation had been allowed in the compilation phase when a "destination location not accessible" message was generated. Again the offending joint number is indicated.

OTHER ERRORS

The following are internal or hardware errors over which the user has little control. They are given for the sake of completeness. Such error messages should be reported.

BACKGROUND JOB TOOK TOO LONG TO EXECUTE
SERVO DEAD
A/D ERROR
NO ARM SOLUTION WHILE SERVOING
SERVO ERROR = n
where n represents the following:

1	COULD NOT ATTACH TO REQUESTED JOINT(S)
2	INCORRECT NUMBER OF JOINTS REQUESTED TO BE DRIVEN
3	WIPERS COULD NOT BE READ WITHIN THEIR OPERATING RANGE
4	ARM SOLUTION DOES NOT EXIST
5	UNKNOWN TOUCH SENSOR REQUESTED
6	NO MORE FREE SLOTS IN TOUCH SENSOR EVENT LIST
11	ZERO VELOCITY TACHOMETER READING OUT OF RANGE
12	ATTEMPTED TO SWITCH ARMS WHILE FORCE SERVOING
13	NO MORE FREE SLOTS IN FORCE SENSOR EVENT LIST
14	NEED ALL 6 ARM JOINTS IN ORDER TO DO FORCE SENSING/COMPLIANCE
15	CAN'T FORCE SERVO MOTION WITHOUT POLYNOMIAL
20	JOINT STARTED OUTSIDE OF PERMITTED OPERATING RANGE
400	JOINT IS DOWN, INOPERABLE
1000	CATASTROPHIC A/D ERROR HAS OCCURRED
40000	NO ARM SOLUTION WHILE DOING FORCE COMPLIANCE

5.4.5.2 Non-Motion errors

INCOMPATIBLE PCODE VERSION. PROCEED AT YOUR OWN RISK

This means that the binary file assembled for the user program is incompatible with the current runtime system. The solution is to recompile the user AL program.

FREE STORAGE EXHAUSTED

Only very large programs will cause this error. It has been largely eliminated with the addition of more memory.

NO VALUE FOR VARIABLE - USING DEFAULT.

This error is caused by attempting to access a variable before it has been assigned a value. Proceeding will use a value of zero, nilvect, nilrot, or niltrans, depending on the data type of the variable.

USER PDL OU

This is a fatal error caused by a bug in either the hardware or the runtime system. Sometimes restarting the program will cause this error to go away.

CAN'T INITIALIZE ARM. REFERENCE POWER SUPPLY OUT OF RANGE.

The arm initialization routine ran into trouble due to the arm reference

power supply drifting. The program may be continued by typing <alt>P, but this should be done with extreme caution, and the user should be extremely alert with a finger over the panic button to cause an immediate stop if the arm does something unexpected. The arm will be offset from the place it would go if the reference power supply were not out of range. Please report this error.

5.4.5.3 Continuation from runtime errors

To continue from a runtime error there are several possibilities that are indicated on the VT05.

<alt>G	will cause execution to begin from the start of the program.
<alt>P	will cause execution to continue from the next statement.
RETRY<alt>G	will attempt to retry the aborted move.
FINISH<alt>G	will move the arm directly to its destination in the aborted move very slowly.
PARK<alt>G	will move the blue and yellow arms to their respective park positions.

Note that *RETRY* and *FINISH* are only applicable after getting a motion associated error.

5.5 Hints on debugging

There are several instructions that are available for the user to determine which part of his program is giving him problems.

5.5.1 Parse time debugging aids

REQUIRE MESSAGE COMMAND (c.f. section 4.5.9)

The message can be used to inform the user where he is in the program, but since the user is normally familiar with his program, it would be used where there are long compilations of several source files, and the user wants some description of the contents of some source file. Another use is to output a message to set parameters during compilation, and follow it directly with a REQUIRE SOURCE_FILE "TTY:FOO.AL". The user can then make the required assignments from the teletype.

REQUIRE ERROR_MODES "LA" (c.f. section 4.5.9)

This message is particularly useful if the compilation is to be done non-interactively. Errors (if any) will automatically be collected in a file with extension LOG, and the parser will try to continue from errors the best it can.

5.5.2 Runtime debugging aids

One way to help debug the program during execution is to output values and messages during the execution by means of the *PRINT* command (c.f. section 4.5.7). It is useful for printing out actual values of variables at execution.

11DDT is an assembly-language symbolic debugger for the PDP-11, and its use is outside the scope of this document. It is primarily used by AL wizards to debug the runtime system. The user is exposed to 11DDT to the extent that he uses it to start or continue execution of his program using the <alt>G, RETRY<alt>G, and <alt>P commands.

5.6 The GAL program module for graphing force data

The GAL module has been developed for graphing data collected by means of the *GATHER* clause (c.f. section 4.4.3.4). GAL is a program run on the PDP-10 that communicates with the AL runtime system. To invoke GAL, instead of using the command DO AL[AL,HE], use the command

R GAL

GAL will respond with

AL Force Data Gathering Module

and prompt with an asterisk to which you need to type your responses. Here is a summary of the valid responses and their effects.

FX, FY, FZ	display force data along specified axis
MX, MY, MZ	display torque data about specified axis
T1, T2, T3, T4, T5, T6	display torque data about specified joint
G	graph mode select - toggles between continuous or discrete
S	scaling for force axis of graphs - toggles between fixed & automatic
P	produces plot file for XGP (via PLOT) - asks for file name
C	continue with next gathering move
W	wait after each gathering move - toggled (cleared by Auto-continue)
A	Auto-continue, if on AL won't stop between gathering moves -- toggled
D	start 11DDT
Z	zero memory of PDP-11
R	reload AL runtime system
X	reload experimental AL system
L	load new AL program - asks for file name
1	load and run 11TTY
E	exit

Q quit (same as exit)
H help (to print out all this information)
? same as help

You should type R to reload the AL runtime system, which will ask for the name of your AL program, and D to start 11DDT, and then type <alt>G at the VT05 terminal.

To produce a hardcopy plot of the currently displayed force graph one uses the P command which will ask for a file to store the plot in, along with a title for it. Later the PLOT program can be run to read in the file, re-display the force graph, and write out a copy to the XGP (Xerox Graphics Printer). To invoke PLOT use the command:

RU PLOT[AL,HE]

and answer its prompts.

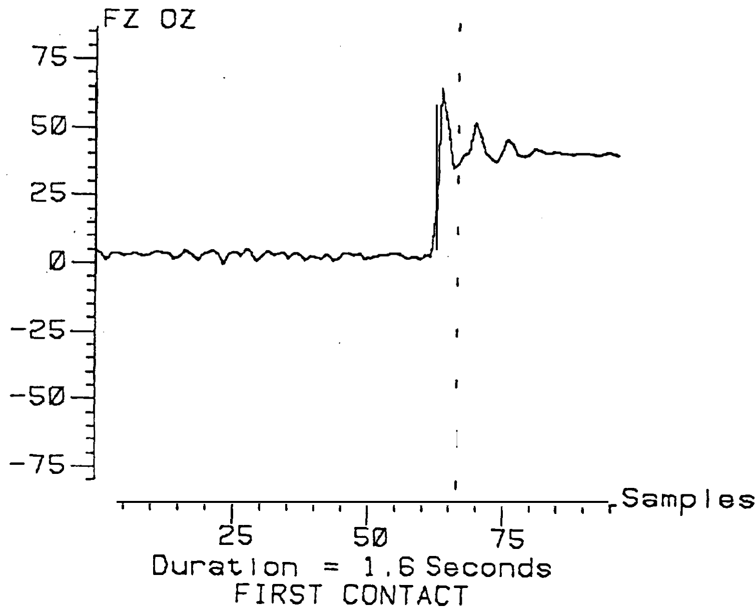


Fig. 5.1 Sample Output of force graphics

6 • POINTY

6.1 Description of POINTY

6.1.1 Introduction

The concept of *FRAMES* as a data structure in AL and their affixment to form an object model should be clear to the reader by now. The generation of such affixments of frames is a non-trivial task, especially if the frames to be affixed to each other have different orientations. If the object is physically available, the user would need to measure distances, angles, and positions, and by doing some rotation of frames would be able to determine the relationships between the frames. Such a procedure is tedious and error-prone in all but the simplest cases.

Given the object, a means of generating the affixment structure is needed. The ideal case would be to present the physical object or its design drawing to the computer by utilizing vision, etc., and let the system build the affixment structure. However, the features of interest on the object are dependent on the nature of the assembly procedure, and may not bear any relationship to the shape of the parts. One way of generating an affixment structure is to use human assistance. The human operator will point out the features of interest on the object, and the system will take care of the book-keeping involved in keeping straight the relationships between the various features.

The interactive construction of world model descriptions for AL programs has been achieved using POINTY, a system developed and implemented at SAIL. It makes use of the ability to read arm positions to define points of interest on the object.

By moving the manipulator around manually and reading the location, the user is able to record various positions on the object. He then tells the system how the various locations are related to each other so that an object model can be generated such that all the required features on the object are known once the position and orientation of one point is known.

POINTY provides the ability to execute AL statements. This allows the user to try out various move statements before putting them into an AL program, permits the arm to be reoriented, and allows differential moves with the same orientation.

6.1.2 Pointing with a manipulator

6.1.2.1 Implicit specification of frames

The Scheinman arm has 6 degrees of freedom, which allows it to be positioned at an arbitrary position and in an arbitrary orientation. Frames also have 6 degrees of freedom, corresponding to 3 components of translation and 3 angles of rotation. It follows that if a single pointing of the manipulator is to imply a unique frame explicitly, there are no spare degrees of freedom. The absence of spare degrees of freedom makes it quite difficult to position the manipulator accurately, since all motions fine or gross require the movement of the same members, and also limits obstacle avoidance.

It is not difficult to guide the arm manually to a good grasping position to pick a part out of a fixture or pallet. It can be quite difficult to guide it manually to a good orientation such that when the manipulator attempts to remove the part, there is no binding. The need for orientation accuracy becomes more crucial when it is being used to define a world model, since any angular error may be multiplied by some long moment arm in the AL program.

To avoid this difficulty, it is sometimes convenient to use multiple pointings to define each frame implicitly. The first pointing may define the origin of the frame, the second may define one axis of the frame, and the third may define one plane of the frame. In this manner, each pointing determines position only, and there is no need to have orientation precision.

A simplification is possible when the orientation is parallel to that of some other known frame, e.g. *station* or some other predefined frame, in which case, the orientation frame can be specified from the known frame, and the location determined by means of a single pointing. Another possibility is to move the arm to the desired location manually, and then use sensing to obtain a more precise estimate.

6.1.2.2 Pointer

The manipulator extremity must be provided with some sort of sharp pointer so that it can be used as a precise measuring tool. The pointer must have a shape suitable for reaching into awkward places such as the inside of a screw hole, the interior of a box, and so forth. In order to make the pointer shape compatible with all kinds of unforeseen obstructions, it is desirable to design a pointer which may be bent by the user into an arbitrary shape. Such a special device will be referred to as a bendy pointer.

Whenever the user wishes, he may deform the bendy pointer into any new configuration which appears to be convenient for the next operation. Having

deformed the pointer, the user must calibrate its new end position by using the pointer to point to a standard fiducial mark at a known location in the laboratory. From the frame of the fiducial and the frame of the gripper, the system can infer the translation which takes the gripper frame into the bendy pointer. An alternative to the bendy pointer would be a tool set consisting of an assortment of rigid pointers of commonly useful shapes which could be quickly attached or detached. Whatever type of pointer is used, it must be reasonably rigid under gravity to prevent it deforming accidentally while being positioned.

6.1.3 System hierarchy

The POINTY system resides on two computers during execution - the PDP-10 where the parsing and of source code is performed, and the PDP-11 which is responsible for executing pcode and moving the manipulators.

AL statements and expressions are accepted by POINTY. The command line scanner (parser) prompts the user for input of a new statement by an asterisk "*". If it is waiting for the continuation of a statement or expression, it prompts with "****>>>". Parsing of the current input line begins when the user hits a carriage return. The first token of the input line is compared with entries in the symbol table. If there is a match, a fixed sequence of parsing will be followed, depending on the token. If no match is found, the parser checks to see if it is a variable that is on the left hand side of an assignment statement by checking to see if the next symbol is a back arrow "←".

The user interface communicates with the user by giving out error messages when the parser does not recognize something, or if the user wants to edit values of variables (e.g. orientation of frames) etc. without using an assignment statement.

Display routines update the screen of the user's terminal to reflect the current state of the affixments, values of variables, procedures and macro definitions, or shut off the display altogether if necessary.

The file input/output facility contains routines for saving and restoring variables and values in and from a text file of AL declarations. These AL declarations and assignments may be used directly by an AL program, or they may be read in by POINTY if the user needs them for re-initialization to a known world state. In addition, the terminal session may be saved.

The PDP-11 part of the POINTY system consists of an extended AL runtime system with code to handle the interaction with the PDP-10 and to display the joint angles and arm positions.

6.2 Executing POINTY

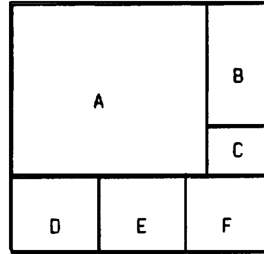
To execute POINTY type the instruction from the monitor.

R POINTY

followed by a carriage return. This instruction first loads the PDP-11 with the POINTY runtime system and starts it up so that it is continuously reading the arm joints and printing it out on the VT05 screen. POINTY may then generate a display on the screen which is continuously updated as more instructions are executed. (In some versions of POINTY, the display is left out to allow faster initialization.) The following shows the state of the display after several instructions.

```
STATION (NILROT,NILVECT)                                    | BHAND   1.20
 -BASE (NILROT,(15.0,12.0,.500))                            | YHAND   .000
 -HANDLE (NILROT,(35.0,32.0,.500))                          | OFFSET 3.00
   *HANDLE_TOP ((Y,180.)*(Z,90.0),(2.10,.340,5.05))         |
   *HANDLE_REF (NILROT,(1.10,2.30,.100))                    |
 +YARM (NILROT,NILVECT)                                     |
 +BARM ((Y,180.)*(Z,.002),(43.5,56.8,10.9))                 |
   *BGRASP ((Y,180.)*(Z,-180.),NILVECT)                     |
                                                            |-------------
                                                            | MOVE
                                                            | BARM
--------------------------------------------------------------------------
*O DECLAR.AL         | NILROT  (Z,.000)          | NILVECT(.000,.000,.000)
                     | RT_AP(Y,180.)*(Z,-90.0)   | APPR (3.00,.000,.000)
                     |                           |
 SAVED.TTY           |                           |
```


The boxes will be referred to later by the following letters:



A: affixment tree,
   frames and transes
B: scalars
C: default moves
D: output files
E: rotations
F: vectors

POINTY is now ready to accept instructions, prompting with an asterisk, as it does each time it awaits a new command. Single instructions may terminate with a carriage return or with a semi-colon and a carriage return, and POINTY will then try to execute the instruction. Multiple instructions on the same line must be separated by semi-colons, and the last instruction followed by a carriage return. On seeing a carriage return, POINTY tries to execute the instruction if it is meaningful, otherwise it will await more input and the next carriage return by prompting with ****>>>.

e.g. a1 ← 3 <cr> {POINTY will assign value 3 to variable a1}

a1 ← 3 + <cr> {POINTY will wait for more input}

In the initial state of the display, Box A indicates the frames known to POINTY: *station*, *barm*, *yarm*, *garm*, and *rarm*. Box B has values of *bhand*, *yhand*, *ghand*, and *rhand* corresponding to the hand opening of the blue, yellow, green and red arms respectively. Boxes E and F initially contain the definitions of the predefined rotation *nilrot* and the predefined vector *nilvect*.

6.3 POINTY instructions

POINTY will now accept AL instructions, and execute them. In addition, there are additional instructions that facilitate interaction with the user. In this section we will look at the instructions that POINTY is capable of handling and classify them according to basic constructs, data types and expressions, affix and unfix, motions and device operation, non-motion statements, and POINTY specific statements.

6.3.1 Basic constructs

When POINTY sees a complete statement, it executes it immediately. The definition of statement is recursive, as in AL, and it may be a single statement or a block statement. A whole program may thus be treated as a statement.

Variable names are exactly the same as in AL, consisting of a string of alphanumeric characters and the underscore following a letter.

Both forms of the AL comment statement may be used, and the COMMENT statement is allowed in POINTY to keep AL programs compatible with POINTY. In addition, the user can type comments and opinions interactively, and they will be saved in a logging file.

6.3.2 Data types, expressions, declarations and dimensions

6.3.2.1 Explicit and implicit declarations

Explicit declarations of *SCALAR*, *VECTOR*, *ROT*, *FRAME*, *TRANS*, *STRING*, *EVENT*, and *LABEL* data types may be made as in AL. Array declarations for the first seven data types can also be made as in AL.

In the POINTY assignment statement, as in AL, the expression on the right hand side is evaluated and assigned to the variable on the left hand side. If the variable on the left hand side has not been declared, the assignment statement implicitly declares the variable as having the type of the evaluated expression. An error message will be generated if the variable has been previously declared, and the right hand side expression type is different from the left hand side.

Examples:

```
s4 ← 2*3;            { declares s4 as a scalar, will appear in box B }
v4 ← zhat + yhat;    { declares v4 as a vector, will appear in box F }
r5 ← nilrot;         { declares r5 as a rot, will appear in box E }
f5 ← bpark;          { declares f5 as a frame, will appear in box A }
```

6.3.2.2 Predeclared variables and constants

The following AL predeclared variables and constants are recognized by POINTY - scalars: *bhand, yhand, ghand, rhand,* vectors: *nilvect, xhat, yhat, zhat,* rotation: *nilrot,* frames: *station, barm, yarm, garm, rarm, bpark, ypark, gpark, rpark,* trans: *niltrans,* strings: *null, crlf,* and dimensional constants: *inch, inches, deg, degree, degrees.* Where *barm, yarm, garm, rarm,* or a frame attached to an arm is used in an expression, the current value computed from the present arm position will be used.

6.3.2.3 Implicit data types

POINTY allows implicit specification of data types. In particular, since the number and data types of arguments are different for the various data types (except between *FRAME* and *TRANS*), they may be declared without the qualifier. If POINTY is unsure whether a declaration is for a frame or a trans, it will assume it is a trans, and will change it to a frame type when the variable associated with it is used in an affixment statement. Note that in the display the reserved words *VECTOR*, *ROT*, *TRANS*, *FRAME* are left out to save space, and that *xhat*, *yhat*, and *zhat* are abbreviated *x*, *y*, *z*, where it is obvious.

vectors:	VECTOR(<scalar> , <scalar> , <scalar>) or (<scalar> , <scalar> , <scalar>)
rotations:	ROT(<vector>, <scalar>) or (<vector>, <scalar>)
frames:	FRAME(<rot>, <vector>) or (<rot>,<vector>)
transes:	TRANS(<rot>, <vector>) or (<rot>,<vector>)
strings:	"<any list of characters except double quote>"

6.3.2.4 Expressions

POINTY accepts all the algebraic expressions that the AL parser is capable of handling. The following summarizes the valid operations. Where they have the same meaning as in AL, they are not described in detail.

SCALAR	s+s, s-s, s*s, s/s, s↑s, v.v, |s|, |v|, |r|, s MAX s, s MIN s, s DIV s, s MOD s, INT(s), INSCALAR, SIN(s), COS(s), TAN(s), SQRT(s), ASIN(s), ACOS(s), ATAN2(s,s), LOG(s), EXP(s), RUNTIME, RUNTIME(s), XCOORD(v), YCOORD(v), ZCOORD(v), XCOORD(t), YCOORD(t), ZCOORD(t) {XCOORD, YCOORD, ZCOORD return the X,Y,Z components of vector v or of vector part of t}	
BOOLEAN	s <rel> s, s∧s, s∨s, s⊕s, s≡s, ¬s, QUERY	
	ARMREACH(arm,f)	{Returns TRUE if arm can reach destination location f}
	ISAFFIXED(f,f)	{returns TRUE if the two frames are affixed to each other}
VECTOR	s*v, v*s, v/s, v+v, v-v, v*v, r*v, POS(f), f*v, v WRT f, UNIT(v),	

	AXIS(r), t*v,	
	v REL f ≡ f*v	{ *v* is a vector expressed in the coordinate frame *f*. The expression represents the coordinates of the vector in station coordinates.}
ROTATION	ORIENT(f), r*r	
FRAME	(+v, f-v, t*f, f*t	
	f1 REL f2 ≡ f2*f1	{*f1* is a frame expressed in the coordinate frame *f2*. The expression determines the frame expressed in station coordinates.}
	CONSTRUCT(v,v,v),CONSTRUCT(f,f,f)	
		{constructs a frame using the location part of the three frames, or the three vectors: the first position defines the origin, the second the x-axis and the third the x - y plane of the desired frame. This avoids having to guide the manipulator to a desired orientation precisely.}
	↑f, ↓f, $f, αf	{returns a frame having the location part as that of f but with different orientations. ↑ gives the vertical component of orientation, i.e. if ORIENT(f) = rot(zhat,a) * rot(yhat,b) * rot(zhat,c) then ↑f = FRAME(rot(zhat,c), POS(f)); ↓ gives the orientation of bpark position, i.e. rot(yhat,180); $ gives the station orientation, i.e. nilrot, and α gives the orientation of bgrasp when the arm is in the park position, i.e. rot(zhat,1 80)}
TRANS	f→f, t*t, INV(t)	

6.3.3 Affixment tree operations - *AFFIX & UNFIX*

The *AFFIX* instruction is similar to the AL *AFFIX* instruction, but allows *RIGIDLY* and *NONRIGIDLY* to be abbreviated as "*" and "+" respectively:

AFFIX f1 TO f2;
AFFIX f1 TO f2 RIGIDLY;
AFFIX f1 TO f2 NONRIGIDLY;
AFFIX f1 TO f2 *;
AFFIX f1 TO f2 +;

Frame *f1* is affixed to the *f2*. Unless specified otherwise, the affixment is *RIGID*.

Every newly defined frame is shown with respect to the *station* frame (indicated with a "-" on the display). The affixment trees appear on box A of the display as they are constructed. Frames may also be affixed with the relative transform between them being specified as follows:

AFFIX <identifier> TO f3 AT (<rot>,<vector>) ;
AFFIX <identifier> TO f3 AT (<rot>,<vector>) RIGIDLY ;
AFFIX <identifier> TO f3 AT TRANS (<rot>,<vector>) NONRIGIDLY ;

If <identifier> is not a frame, a new frame is defined before it is affixed. This instruction is used mainly for reading in AL instructions generated during a previous POINTY session.

The *UNFIX* instruction is written as in AL. *Frame_1* is unfixed from *frame_2* and affixed independently to *station*.

UNFIX frame_1 FROM frame_2;

6.3.4 Motion and Device operations

6.3.4.1 MOVE statement

The current arm position is used whenever the arm is referred to directly, and to compute the values of any affixed frames. The user may not assign values to the frames *barm*, *yarm*, *rarm* and *garm*.

The MOVE command in AL is applicable to POINTY with all the clauses and condition monitors that AL is capable of handling. Thus POINTY will recognize *VIA*, *DEPARTURE*, *APPROACH*, *DURATION*, *SPEED_FACTOR*, *NULLING*, *WOBBLE*, *FORCE_WRIST*, *GATHER*, *STIFFNESS*, and *ERROR* clauses, *FORCE*, *TORQUE*, *DURATION*, event and boolean condition monitors, and can *ENABLE* and *DISABLE* them. POINTY will try to execute a motion when a complete motion command is specified. In addition to the absolute positions that can be specified in AL, POINTY allows the specification of differential motions (possible in AL by defining a macro which expands to ⊂TO ⊗ +⊃ by using a *BY* instead of *TO* and a vector instead of a frame expression, as follows:

MOVE f1 BY <vector>
MOVE f1 BY <vector> WRT f2

These instructions are equivalent to:

MOVE f1 TO ⊗ + <vector>
MOVE f1 TO ⊗ + <vector> WRT f2

Differential moves parallel to the x, y or z axes of the station may be specified by the following instructions.

MOVEX f1 BY <scalar>;
MOVEY f1 BY <scalar>;
MOVEZ f1 BY <scalar>;

These instructions are equivalent to the AL instruction

MOVE f1 TO ⊗ + <scalar>*<axis>

To reduce repetitive typing, a move instruction similar to the last executed move instruction (shown in box C of the display) may be given by merely typing the last part of the instruction. Hence it is possible to state

TO f1;
TO f1 + <vector>;
TO f1 + <vector> WRT f2;
BY <vector>;
BY <vector> WRT f2;
BY <scalar>;

The last form may be used only after a differential movement instruction along a principal axis *xhat*, *yhat*, or *zhat*.

6.3.4.2 Hand motion - *CENTER, OPEN, CLOSE*

The syntax and use of *CENTER* is similar to that in AL.

CENTER <arm>; { <arm> may be left out }

closes the fingers slowly, moving the <arm> to accomodate to the location of any object positioned between the fingers. If <arm> is left out, the last arm moved will be used.

The syntax for hand motions are similar to those in AL except that differential movements may also be specified.

OPEN <hand> TO <scalar>;
CLOSE <hand> TO <scalar>; { absolute opening or closing }
OPEN <hand> BY <scalar>;
CLOSE <hand> BY <scalar>; { differential opening or closing }

If the next motion statement is to open or close the same hand, the instruction may be abbreviated as follows:

TO <scalar>;
or BY <scalar>;

6.3.4.3 *DRIVE* command

POINTY permits the movement of individual joints (which is not permitted by AL). The syntax is as follows:

DRIVE BJT(<joint number>) TO <scalar>;
DRIVE YJT(<joint number>) TO <scalar>;
DRIVE BJT(<joint number>) BY <scalar>;
DRIVE YJT(<joint number>) BY <scalar>;

The indicated joint of *barm* or *yarm* is moved to <scalar> or by <scalar>. <Joint number> is an integer which represents the joint and can take a value of 1 through 6. The dimension of <scalar> must be angles for joints 1,2,4,5,6 and distance for the prismatic joint 3.

Short forms exist as for the other motion instructions.

TO <scalar>;
or BY <scalar>;

Currently the PUMA arm joints cannot be moved individually the same way. (They may be moved under simulated joystick control as described in section 6.3.4.4.) When the PUMA arm joints are movable under the *DRIVE* command, they will be referred to as *GJT* and *RJT* for the green and red arms respectively, and the joint angle or change in joint angle will be in degrees for joints 1 through 6.

6.3.4.4 PUMA simulated joystick commands

POINTY permits the PUMA arms to be controlled directly by the keyboard of the VT05 where the keys on the VT05 keyboard act as buttons on a joystick controller or a control box. (Actually the keyboard is used here as a simulated joystick.) The instructions take as argument the name of the arm, put the arm in the appropriate control mode, and await future input from the VT05. In the following description, <arm> is either GARM or RARM.

CALIB(<arm>)

enables the arm power to be turned on for the appropriate PUMA arm and to do a preliminary calibration of the joint potentiometer if desired. The power to the PUMA arm is turned on by pressing the black button on the right hand part on the front of the PUMA controller. The "ARM POWER ON" light should go on when

you press the button, and remain on when you release it, otherwise it means that the main power is is not on, in which case you have to turn that on by pressing the large red button on the underside of the short end of the hand-eye table closest to the Scheinman arms.

PFREE(<arm>

allows the user to select the joint to be freed up by typing an integer from 1 through 6 on the VT05 console. The joint can be be moved manually. To stop the freedom of the joint, type any character on the VT05 console.

PGRAV(<arm>

computes the gravity model of the arm and keeps the arm in static balance by putting out the compensating torque on the motors, permitting the user to move the arm manually to any position and orientation. Typing any character on the VT05 console gets you out of this mode.

PJOINT(<arm>)

enables single joint motion under keyboard control. Typing the joint number from 1 through 6 selects the joint. With the hands in the home position on the VT05 keyboard (i.e. left hand on keys ASDF and right hand on keys JKL;) depressing the first finger of the right hand gives rise to motion in the positive direction, while the first finger of the left hand gives rise to motion in the negative direction; the little finger on the right hand speeds up the motion, and the little finger on the left hand slows down motion. In terms of actual characters, the keys are as follows:

A	slow down motion
;	speed up motion
F	move in negative direction
J	move in positive direction

The keyboard is scanned continuously, so that the direction, speed and joint number can be changed during motion. A character not understood will cause the arm to stop while Q for quit will end the mode and return control to the PDP-10.

PTABLE(<arm>)
PTOOL(<arm>)

enable the mode for Cartesian motion in table (station) and tool (hand) coordinates. Again, the keys on the VT05 console control the direction and speed of the motion. In the home position, the first three fingers indicate X, Y, Z axes respectively, the right and left hands indicate positive and negative direction, and

the little fingers indicate fast or slow. The character codes are as follows:

A	slow down motion
;	speed up motion
F	negative X direction
D	negative Y direction
S	negative Z direction
J	positive X direction
K	positive Y direction
L	positive Z direction

6.3.5 Non-motion AL statements

6.3.5.1 Assignment statements

The assignment statement in AL is recognized by POINTY, and may be used for implicit declarations. In addition, POINTY permits the changing of various components of variables rather than the whole variable by using *POS*, *ORIENT*, *XCOORD*, *YCOORD* and *ZCOORD*.

```
ORIENT(f)← <rot expression>;    { changes the orientation part of
                                  f, where f is a frame or trans variable}
POS(f) ← <vector expression>;   { changes the vector part of f,
                                  where f is a frame or trans variable}
XCOORD(v) ← <scalar expression>;
                                 { if v is a vector, changes the x-component;
                                  if v is a frame or trans, changes the
                                  x-component of POS(v)}
```

YCOORD and ZCOORD have the same effect as XCOORD except that the Y and Z components are changed instead of the X component.

6.3.5.2 Control structures

The control structures *IF*, *FOR*, *WHILE*, *UNTIL*, *CASE*, *COGEBIN-COEND*, and the synchronization commands *SIGNAL* and *WAIT* are all applicable in POINTY. The statement will not be executed unless POINTY recognizes that it is complete. Any errors occurring during parsing will return control to the top level.

6.3.5.3 Procedures and Macros

Procedures and macros can be defined as in AL. The macro definition is completed when POINTY sees a matched ⊂⊃ pair, and procedure definitions are complete after POINTY sees a complete statement after the procedure declaration.

POINTY permits the use of default arguments in procedures and macros. The value of the default argument is given during declaration by including the value in parentheses immediately after the argument. The following is an example.

```
PROCEDURE MOVE2(FRAME f1,f2(bpark -3*zhat*inches),f3(bpark));
        MOVE f1 TO f3 VIA f2;
```

The procedure call *MOVE2(yarm,ypark-3*zhat*inches,ypark)* will move the yellow arm to ypark via ypark-3*zhat*inches; the procedure call *MOVE2(yarm,ypark-3*zhat*inches)* will cause it to try to go to bpark via ypark-3*zhat*inches, while the procedure call *MOVE2(yarm)* will cause it to try to go to *bpark* via *bpark-3*zhat*inches*. Notice that if we gave a default value to *f1* as well by making the declaration

```
PROCEDURE MOVE2(FRAME f1(barm),f2(bpark-3*zhat*inches),f3(bpark));
        MOVE f1 TO f3 VIA f2;
```

we could make a procedure call MOVE2 without arguments that moved the blue arm to *bpark* via *bpark-3*zhat*inches*.

Text substitution by macros is also possible in POINTY. The syntax is similar to that in AL, and default arguments to macros are defined the same way as for procedures, i.e. the value of the default argument follows the parameter in parentheses.

```
DEFINE ARM = ⊂barm⊃;
DEFINE V1(A,B,C)=⊂VECTOR (A,B,C)⊃;
```

Note the delimiters used around the body of the macro definition. In the macro definition, the parameter names must be hitherto undeclared variable names. Using those names for some other purpose in the future will not affect the macro definition.

The macro name can be used just about anywhere where the body gives a valid statement or statements. Thus the following are valid:

```
MOVE ARM TO BPARK;
VECT1← V1(0,0,1);
VECT2← V1(⊂2*3⊃,1,4);
```

Note also the use of the delimiters when the parameter substituted is not a single token but an expression or a series of tokens.

A macro definition may be changed by means of the *REDEFINE* command. Thus if we wish to change the definition of V1 which has been defined above, we

could do so as follows:

```
REDEFINE V1(D,E,F)=⊂VECTOR(D,E,F)*4⊃;
```

Note that the *EDIT* command (section 6.3.11) when used with macros actually utilizes the *REDEFINE* command.

6.3.5.4 Condition monitors

Condition monitor statements are executed as in AL. When POINTY comes across a condition monitor statement, the appropriate condition statement is set up and enabled unless it is a deferred condition monitor. Condition monitors can be *ENABLE*d and *DISABLE*d as in AL.

6.3.5.5 I/O

The *PRINT*, *PROMPT* and *QUERY* statements of AL are also applicable in POINTY and have the same format.

6.3.5.6 *REQUIRE* statement

POINTY accepts the REQUIRE statement of AL, and only ERROR_MODE "F" and SOURCE_FILE are applicable. The other possibilities of the REQUIRE statement are ignored.

6.3.6 Deletion statement

Variables may be deleted by means of the delete statement. If the deleted variable is a frame identifier any subtrees rooted in it are also deleted. Examples of the delete statement are

```
DELETE s1,s2,s3,v1,v2,f1,f2;
QDELETE s1,s2,s3,v1,v2,f1,f2;
DELETE ALL;
QDELETE ALL;
```

The use of *ALL* deletes all the user declared variables. If no argument is given, it is assumed to be *ALL*, but in the case of *DELETE*, the user is asked to confirm that he does in fact want to delete all the variables. The variables will disappear from the relevant boxes in the display. If a variable name is given that POINTY does not understand, POINTY will assume a spelling error, and let the user correct the name. If the user does not want POINTY to inform him that the variable does not exist, he should use *QDELETE* instead of *DELETE*. The *QDELETE* command is useful when macros or identifiers are to be read in from a file whose names may be the same as those already defined in the symbol table.

6.3.7 Display routines

6.3.7.1 Data Disk screen

The standard display has been described in section 6.2.1, and it shows as much useful information as possible by omitting the use of reserved words like *VECTOR*, *TRANS*, etc, and by abbreviating *XHAT*, *YHAT*, *ZHAT* to *X*, *Y*, *Z*, and not displaying the values of POINTY defined constants.

There are now four display modes available. In the table display mode, scalars, vectors, transes, frames, rots, the default move statement, and the files used in the current session are shown. Owing to lack of space on the display, macros and function expression definitions are not displayed in this mode. The REDISPLAY command gets to this mode from other modes.

The type display mode allows the user to see all the current definitions of the specified data type.

DISPLAY <data type> { where <data type> is SCALAR, VECTOR, ROT,TRANS,FRAME,MACRO,STRING,PROCEDURE,EVENT}

This display mode permits the display of more variables of a data type than is possible in the standard display, and the display of macros and procedures. When the user is more interested in seeing what he has typed so far, the display mode most useful is the no display mode, invoked by the command

NODISPLAY

This eliminates the display altogether, and just prints out the series of commands typed by the user. Finally, the variable display mode, invoked by the SHOW command, allows the user to display selected variables. The syntax is as follows:

SHOW <variable list>

where <variable list> is a list of variable names separated by commas.

The command NOUPDATE causes the values of variables on display not to be updated, while the command UPDATE undoes effect of NOUPDATE.

6.3.7.2 VT05 screen

The VT05 screen shows values of the blue arm joints and A/D readings of each of the potentiometers, and is updated continuously. Here are the commands for various modes of control of the VT05 display.

COMMAND	explanation
VT05_ON	Continuously displays arm joints and A/D readings
VT05_OFF	Shuts of display of arm joints and A/D readings
VT05_YELLOW	Displays yellow arm data
VT05_BLUE	Displays blue arm data
VT05_RED	Displays red arm data (not currently implemented)
VT05_GREEN	Displays green arm data (not currently implemented)

6.3.7.3 Force Graphics

The *GRAPH* command in POINTY displays the force components that were gathered for the last motion command that had the *GATHER* components specified. This command plots the force information graphically, and permits the plot files of graphs to be saved on the disk. (Programs running directly under the AL runtime system make use of the GAL program module for displaying force data as described in section 5.6.)

6.3.8 File input/output

File input and output is necessary to generate the affixment trees for AL instructions, as well as to save the results of a POINTY session and to make use of the results of previous sessions.

6.3.8.1 Saving current state - *WRITE*

The *WRITE* instruction is used to write on the indicated file the AL instructions required (declarations, assignments, and affixments) to define variables and preserve the current state of the world. The syntax is as follows:

```
WRITE;                     { into file last written }
WRITE <id>;                { into file last written }
WRITE INTO <file>;         { write everything into <file> }
WRITE <id> INTO <file>;
```

If the *<id>* part is omitted, all the variables (except *station*, *fiducial*, *pointer*, *yarm* and *barm* and other predeclared variables) are output, otherwise only the indicated frame and the subtrees rooted in it, or the identifier is output. Since frames are affixed to other frames in terms of their relative transes, any frame to be saved should be affixed independently to *station* in order to obtain its absolute location.

POINTY permits output to different files. If the file named does not exist, it is created, and the current time and date written out before the required information specified by the user. If it exists and was used during the current session, the output is appended. If it exists but no input has been done to it during the current session, the current time and date are put on a fresh page, followed by the desired output. If no output has been done so far, and no file name is specified, output will be directed to a file DECLAR.AL on the area of the current user.

The fact that POINTY appends to existing files rather than overwrites them may be inconvenient, since it may result in multiple declarations of the same variable. However, this inconvenience is small compared to the disastrous results that may occur if overwriting were allowed and the user accidentally asked that an important data file be overwritten. Asking for confirmation before overwriting is one solution, but not a suitable one in the cases where only certain variable values need to be updated, while those of others need to be preserved.

6.3.8.2 Getting a given world state - *READ & QREAD*

The *READ* and *QREAD* commands will read the specified file of AL instructions to bring the state of POINTY's world to a known state, or to a state that was saved at the end of the previous terminal session, so that in addition to being input files to AL, POINTY generated files may be used to store instructions to build the necessary frame tree structure and assign values to variables.

```
READ;          { reads from DECLAR.AL }
READ <file>;
QREAD;
QREAD <file>;
```

Since movement commands may also be given in the input file, the user should be careful that the commands do not cause disastrous motions to occur. The *READ* command will print out the input file as it is being read. The *QREAD* command will execute faster since it does not print out the input file.

Printing can also be controlled from within the file by means of the *ECHOON* and *ECHOOFF* commands which are used as follows:

```
ECHOON;
ECHOOFF;
```

The file contents will be printed on the terminal after an *ECHOON*, and will continue until an *ECHOOFF* is encountered. These commands may be used for selective printing of the input file.

6.3.8.3 Saving a terminal session - *PHOTO*

The POINTY system saves what the user types and error messages in the disk file POINTY.PHT[PNT,HE] with the time, date and name of the user. This permits ᵃ ⁿ examination of the circumstances under which errors occurred.

The user may wish to direct a copy of the terminal input and output to a file of his own choosing. The command to do this is as follows:

```
PHOTO <filename>
```

This will cause a copy of terminal input and output to be saved in the file called <filename>. Calling *PHOTO* again with a different filename will change the destination. Currently there is no provision for shutting off the recording on the user file except by ending the session.

6.3.9 HELP module

POINTY has a help module which provides interactive help by displaying information on the screen. Entry to the help module or helper can be invoked by the command *HELP*. Once in this mode, reserved words lose their meaning and are treated as keywords, and there is access to an on-line data base, which is organized as a directed graph of nodes. Each node has a keyword and corresponding message, and terminal nodes have null keywords associated with them. At any time, the user selects a keyword; if the keyword is valid, the keywords and messages of all the successor nodes are displayed. Since there is no matching of the null string keyword, the user cannot go past a terminal node. The user thus steps through successive nodes in the graph structure or is able to back up along the path he has t aken to get to the current node. At any point, the user can trace through the path taken to get to the current node, and access any of its ancestors, immediate descendants and sibling nodes, or get back to the starting node. Since data for the helper is stored on a random access disk file, being read only when needed, the messages can be as detailed and informative as possible without occupying too much memory during program execution.

If the message is too long to fit on the screen, the <formfeed> and <vert tab> keys may be used to scroll the information up and down the screen.

The ? node corresponds to the node that prints out help messages, and directions on the use of the helper. The *menu* node corresponds to a list of classes of keywords corresponding to instructions and description of the hardware, software, and the AL system.

```
******************************** H E L P   M O D E ********************************
RES     gives a list of reserved words
RESH    gives one line descriptions of the reserved words
DISP    describes display commands
POW     describes turning on and off the power to the arms
PUMA    describes use of PUMA arms with POINTY
PDP-11  gives information on the PDP-11 interface
VT05    describes VT05
OPER    shows available operators and functions
EXP     valid expressions
FILE    Gives the file management commands
ESC_I   ESCAPE I command
STATE   classes of statements
ERROR   talks about how to handle errors
SYNTAX  syntax of statements
DEBUG   details of the high level debugger
SYSTEM  building up a new POINTY system
BAIL    describes the SAIL debugger and how to get at it


ANCESTORS:    MENU   ?
BRETHREN:     HELP BUGS MENU ??
************************************************************************************
```

Fig. 6.1 Sample of display when using help command at menu node

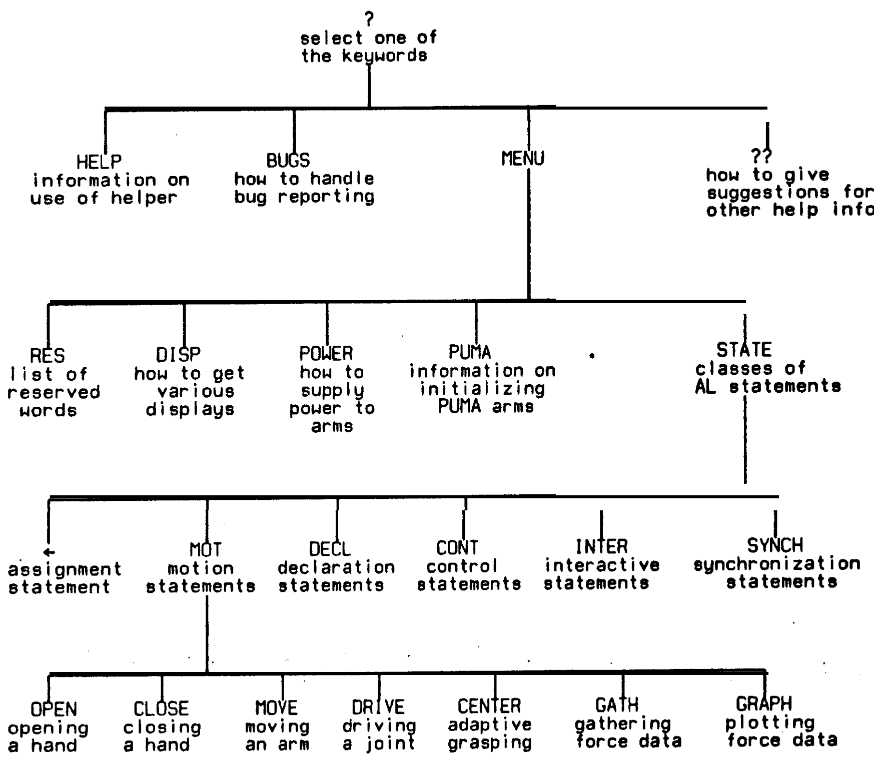


Fig. 6.2 Typical path through the help module

The above diagram shows part of the HELP module data structure and how a user would step through it. On entering the help module he is told to select one of the keywords "HELP", "BUGS", "MENU", "??". Suppose he chooses "MENU". Then he is told the various categories of information available to him (e.g "RES" will get him information on reserved words, "DISP" on display, "POWER" on turning on the power, "STATE" on classes of statements, and so on). Suppose he selects "STATE". Then he is told about assignment, motion, declaration, control, interactive and synchronization statements. If he selects "MOT", he is told that information on the various motion statements may be obtained by typing "OPEN", "CLOSE", "MOVE", "DRIVE", "CENTER", "GATHER" and "GRAPH".

At any time within the HELP module, the user gets to the ancestor node by typing "UP", to the top by typing "?", out of the module by typing "DONE", or he may choose to type a keyword at the same level, one along the path he chose to get to the current node, or a keyword corresponding to one of the descendant

nodes. For example, at "MOT", the legitimate keywords are "STATE", "MENU", "OPEN", "CLOSE", "MOVE", "DRIVE", "CENTER", "GATH", "GRAPH", "?", "UP", "DONE".

6.3.10 Interactive debugger

An interactive high-level debugger has been implemented in POINTY. Its syntax is similar to that of BAIL[Reiser 75], the source level debugger for the programming language SAIL[Reiser 76]. In order to activate the debugger, the debug mode must be set by means of the DEBUGON command. To leave the debug mode and resume normal mode, the instruction DEBUGOFF must be given.

When in debug mode, simple statements are executed immediately, while there is a pause at the beginning of compound statements and control is returned to the user. In the debug mode, the user is prompted by the symbol ":*:". Input can be edited in the standard line editor to allow correction of mistyping. The activation character is the semicolon or the carriage return, whichever comes first.

While in debug mode, AL expressions typed by the user are evaluated in the context of the program where execution was suspended. The evaluation is performed just as if the user had inserted an extra statement into the original program at the point where execution was suspended. The AL expression must be legitimate at this point.

AL statements are numbered starting from 1, and each time there is a new statement, the numbering count is incremented by 1. Thus if a BEGIN-END block is being executed, the first "BEGIN" takes the number 1, the first statement in the block takes the number 2, and so on. In addition, if procedures are declared in the debug mode, the statements in the body are numbered starting from 1. Reference to an instruction can be made through its statement number, or "coordinate", as described in BAIL[Reiser 75]. Reference to a statement in a procedure requires both the name of the procedure and the statement number, which begins with "1" for the first statement in the block.

6.3.10.1 Setting breakpoints

As in BAIL, break points can be set by means of the BREAK command as follows:

```
BREAK(STMNT#);
BREAK("PROCEDURE_NAME", STMNT#);
```

Here STMNT# refers to the statement number of the statement, and PROCEDURE_NAME refers to the name of the procedure if the break point is to be inside the procedure. The break point is inserted just before the statement, and when execution reaches this point in the program, control returns to the user in

DEBUG mode.

6.3.10.2 Permanent breakpoints

The statement HALT puts a permanent breakpoint in the program and returns control to the user in DEBUG mode when program execution reaches this point. This instruction is inserted in the text of the program being debugged, unlike the BREAK instruction that is used interactively in the process of debugging ● Proceeding past the breakpoint can be done as described in the section "Resuming Execution".

6.3.10.3 Abort execution

The QUIT command abandons execution of the current program and gets control back to top level. This command is currently valid only outside the body of a procedure.

6.3.10.4 Restarting a program

The RESTART instruction begins re-executing the current user statement or program. It can be called only from outside the body of a procedure.

6.3.10.5 Displaying source level code

The source code at specified statements can be displayed by means of the TEXT command. Here is a set of legitimate calls to the TEXT command.

```
TEXT;
TEXT(MIN#);
TEXT(MIN#,MAX#);
TEXT("PROCEDURE_NAME",MIN#);
TEXT("PROCEDURE_NAME",MIN#,MAX#);
```

MIN# and MAX# are integers. The source text is shown from MIN# to MAX# when MAX# is greater than MIN#. If MAX# is less than MIN#, then the text shown is from MIN# to MIN#+MAX#. If MAX# is omitted, then only the text at MIN# is shown. If both numbers are omitted, the next instruction to be executed is displayed.

6.3.10.6 Listing the existing break points

The command TRAPS shows the existing break points in the program, and is used to keep track of where the current break points are.

6.3.10.7 Removing a break point

A break point set by the BREAK command can be removed by the UNBREAK command. The syntax of the command is as follows:

```
UNBREAK(STMNT#)
UNBREAK("PROCEDURE_NAME", STMNT#)
```

The debugger complains if there is no break point at the corresponding statement.

6.3.10.8 Resuming execution

The command !!GO or <CONTROL>G resumes execution of the program from the point of suspension until another break point or the end of the program is encountered.

The command !!STEP or <CONTROL>S resumes executing the current statement until a new statement is encountered, at which point execution of the statement is suspended. This command allows single stepping into procedures or compound statements.

The command !!GSTEP or <CONTROL>X continues execution until the end of the current statement. It considers a compound statement or a procedure call to be a single statement, and stops after execution of that statement.

As an example, consider the following routine called ACQUIRE and the calling program.

```
     PROCEDURE ACQUIRE(FRAME object);
A1.  BEGIN
A2.       OPEN bhand TO 3.8*inches;
A3.       MOVE barm TO object WITH APPROACH=-3*zhat*inches;
A4.       CENTER barm;
A5.  END;

1.   BEGIN
2.        FRAME f1;
3.        f1←FRAME(ROT(YHAT,180*degrees),VECTOR(30,40,5)*inches);
4.        ACQUIRE(f1);
5.        IF bhand<0.2*inches THEN PRINT("Could not get it");
6.        HALT;
7.   END;
```

Now suppose that there is a break point at statement 4 in the mai

program. !!GO or <control>G will cause execution and return control at statement 6 which is HALT !!GSTEP or <control>X will execute the procedure ACQUIRE and return control to the user at the beginning of statement 5. !!STEP or <control>S will step through the procedure ACQUIRE and return control to the user at statement A1 which is the first statement in the procedure ACQUIRE.

6.3.11 Miscellaneous commands

EDIT <variable>;

loads the line editor with the value of the variable and allows the user to edit it. This is particularly useful when the user wants to change the rotation part of a transformation without changing the vector part. <variable> can be a scalar, vector, rot, trans, frame or macro. When it is a macro name, the macro definition is displayed line by line so that the appropriate modifications can be made.

RENAME <variable>;

allows the user to change the name of the variable.

DUMP_VARIABLES(<filename>);

and LOAD_VARIABLES(<filename>);

dump the currently declared variables and their values in the disk file <filename> and restores variables from the POINTY written file <filename> respectively. Note that this works with simple variables only at the moment, and that the data is stored in a compact POINTY readable form which cannot be read by text editors. This facility is provided for fast saving and restoration of a state of the world.

READMESSAGE

tells POINTY to look for future input from its job mailbox. This command permits another job on the PDP-10 to send POINTY and AL instructions format through the interjob mailbox system. Any messages received are printed out at the terminal. The instruction *STOPMESSAGE* tells POINTY to accept future input from the terminal.

EXIT;

exits from POINTY.

Some error recovery procedures are available. Whenever an undeclared identifier is used where POINTY expects a known variable or its value, POINTY will keep asking for a corrected name until it is given something it can work with, or

the user hits <control>C to get out of the query loop.

<ESCAPE>I

typed on the terminal will cause termination of program execution at the end of the current input line or statement, whichever comes first. All typeahead will be flushed, and if POINTY is reading from an input file, it will stop. The next input accepted will be from the keyboard. This command is used to get out of runaway executions when instructions are being executed from a file through the *READ* command, or out of infinite loops.

6.4 Hints on using POINTY

6.4.1 Suggested sequence for using pointer

The following is a suggested sequence of steps for using POINTY after initializing if you are using the pointing tool.

1) Use the arm to grasp the fiducial point and type the instruction

FIDUCIAL ← BARM;

2) Put the pointer in the hand of the arm and grasp it tightly. Point the tip of the pointer to the fiducial point and type the instruction

POINTER ← FIDUCIAL;

3) Now affix the pointer to the arm frame

AFFIX POINTER TO BARM;

4) For any object, it is desirable to find a reference point for the reference frame. In order to be able to locate the object quickly for future use, it is desirable to have the orientation parallel to the station orientation. Thus the pointer should be used to point at the reference frame, and the following instruction typed

origin ← $ POINTER;

5) The frames for other features of interest are found by using *barm,pointer, CONSTRUCT*. Let us call these new frames *f1, f2, f3*;

6) These frames should be affixed to the origin by the instructions

AFFIX f1 TO origin RIGIDLY;

AFFIX f2 TO origin RIGIDLY;
AFFIX f3 TO origin RIGIDLY;

7) Before exiting from POINTY, do not forget to save the frame tree you are interested in.

6.4.2 Hints

1) It is possible to record the values of variables during a session by asking to edit those variables. The values will be saved within the file collecting the terminal output.

2) If the POINTER is physically removed from the arm, the user need not bother to *UNFIX* it. So far as POINTY is concerned, there is an imaginary pointer in the hand. If the user can put back the pointer in the same position later, the values will still be valid. For access to difficult places, the bendy pointer (6.1.2.2) can be used; however, it must be redefined each time it is bent.

3) Certain positions may be read more easily by moving the arm there and grasping, rather than using the pointer. In that case the value of barm should be used.

4) It is a good idea to save the current value of a frame within another variable before moving it, so that if you later decide to backup, the value will be available.

5) While objects may be in any arbitrary orientation, it is generally easier to use POINTY if the principal axes of frames are parallel or orthogonal to the station axes.

6) Remember that if you want to recall what you did during a POINTY session, a record is kept in the file POINTY.PHT[PNT,HE].

6.4.3 Accurate Part Relocation

Once the part or part fixture locations are taught accurately, future use of the data involves relocating the parts at the same spot with a high degree of accuracy. One way would be to outline the part or fixture on the tabletop with a pencil. A second way would be to tape it to the tabletop. Neither way is desirable, since the first way relies on the outlines not being deleted accidentally, and the second precludes setting up any other objects or parts. Judicious use of the tabletop flexible fixturing system allows the part or part locations to be located quickly and accurately at a subsequent time. The flexible fixturing system consists of the aluminum table top plate with holes, clamps, bolts, locating pins and wingnuts. Parts or part holders can be located by pushing them against the

locating pins, and held securely by the bolts, clamps and wingnuts. The holes are identified by means of grid coordinates, so that a record can be kept of the location of the fixtures on the tabletop.

Appendix I. AL RESERVED WORDS, PREDECLARED CONSTANTS AND MACROS

Reserved Words

ABORT
ABOUT
ACOS
ADC
AFFIX
ALONG
AND
ANGLE
ANGULAR_VELOCITY
APPROACH
ARRAY
ARRIVAL
ASIN
AT
ATAN2
AXIS
BEGIN
BY
CASE
CENTER
CLOCKWISE
CLOSE
COBEGIN
COEND
COMMENT
COMPILER_SWITCHES
CONSTRUCT
COUNTER_CLOCKWISE
EOS
DAC
DEFER
DEFINE
DEPARTURE
DEPARTING
DIMENSION
DIMENSIONLESS
DISABLE
DISTANCE
DIV
DO
DURATION
ELSE
ENABLE
END
EQV
ERROR
ERROR_MODES
EVENT
EXP
FOR
FORCE
FORCE_FRAME
FORCE_WRIST
FRAME
FROM
GATHER
HAND
IF
IN
INT
INV
INSCALAR
LABEL
LOG
MAX
MESSAGE
MIN
MOD
MOVE
NO_NULLING
NONRIGIDLY
NOT
NOWAIT
NULLING
OF
ON
OPEN
OPERATE
OR
ORIENT
PAUSE
POS
PRINT
PROCEDURE
PROMPT
QUERY
REFERENCE
REQUIRE
RETURN
RETRY
RIGIDLY
ROT
RUNTIME
SCALAR
SETBASE
SIGNAL
SIN
SOURCE_FILE
SPEED_FACTOR
SQRT
STEP
STIFFNESS
STRING
STOP
TAN
THEN
TIME
TO
TORQUE
TRANS
UNFIX
UNIT
UNTIL
VAL
VALUE
VECTOR
VELOCITY
VIA
VM
WAIT
WHERE
WHILE
WITH
WOBBLE
WORLD
WRIST
WRT
XOR
ZEROED

Predefined constants

π
BPARK
CM
CRLF
DEG
DEGREE
DEGREES
EXCESSIVE_FORCE
FALSE
GM
GPARK
INCH
INCHES
LB
LBS
NULL
OUNCE
OUNCES
OZ
NILDEPROACH
NILROT
NILTRANS
NILVECT
PANIC_BUTTON
PI
RADIAN
RADIANS
RPARK
RPM
SEC
SECOND
SECONDS
STATION
TIME_OUT

TRUE
XHAT
YHAT
YPARK
ZHAT

Predefined identifiers

BARM
BARM_ERROR
BHAND
BHAND_ERROR
DRIVER
DRIVER_ERROR
DRIVER_GRASP
DRIVER_TIP
DRIVER_TURNS
FIXED_JAW
GARM
GARM_ERROR
GHAND
GHAND_ERROR
MOVING_JAW
RARM
RARM_ERROR
RHAND
RHAND_ERROR
VISE
VISE_ERROR
YARM
YARM_ERROR
YHAND
YHAND_ERROR

Predefined macros

APPROXIMATELY
CAUTIOUS
CAUTIOUSLY
CW
CCW
DIRECTLY
NORMALLY
PRECISELY
QUICK
QUICKLY
SLOW
SLOWLY

Words used with gather

FX
FY
FZ
MX
MY
MZ
T1
T2
T3
T4
T5
T6
TBL

Appendix II. POINTY RESERVED WORDS & PREDEFINED CONSTANTS

In addition to the words recognized from AL, POINTY recognizes the following:

Reserved words

ALL
ALPRIN
ARMREACH
BAIL
BJT
BREAK
BY
CALIB
DDT
DEBUGOFF
DEBUGON
DELETE
DISPLAY
DRIVE
DUMP_VARIABLES
ECHOOFF
ECHOON
EDIT
EXIT
GRAPH
HALT
HELP
INTO
ISAFFIXED
LINE
LOAD_VARIABLES
MOVEX
MOVEY
MOVEZ
NODISPLAY
NOELF
NOFOLD
NOUPDATE
PFREE
PGRAV
PHOTO
PJOINT
PPCODE
PRTIME
PTABLE
PTOOL
PWCODE
QBAIL
QDELETE
QREAD
QUIT
READ
READMESSAGE
READWRIST
REDEFINE
REDISPLAY
REL
RENAME
RESETSTATUS
RESTART
RESUME_MESSAGE
SAVECOREIMAGE
SETSTATUS
SHOW
STOPMESSAGE
TEXT
TIME_OUT
TRAPS
UNBREAK
UPDATE
VT05
VT05_BLUE
VT05_GREEN
VT05_OFF
VT05_ON
VT05_RED
VT05_YELLOW
XCOORD
YCOORD
YJT
ZCOORD

Predefined Identifiers

BGRASP
POINTER

Appendix III. AL COMMAND SUMMARY

```
BLOCKS          BEGIN S; S; S; ... S END
                COBEGIN S; S; S; ... S COEND

DECLARATIONS:   TIME SCALAR ts1, ts2;            LABEL l1,l2;
                DISTANCE VECTOR dv1,dv2;              STRING st1,st2;
                ROT r1,r2;                        FRAME f1,f2;
                TRANS t1,t2;                      EVENT e1,e2;
                FRAME ARRAY f1[s1:s2],f2[s3:s4,s5:s6,...];

PROCEDURES:     PROCEDURE p1; S;
                SCALAR PROCEDURE sp1(VALUE SCALAR vs1,vs2;
                       REFERENCE ROT rr1;
                       SCALAR ARRAY as1[2:3]); S;

OPERATIONS:
scalar  s:   s+s,s-s,s*s,s/s,s↑s,|v|,|r|,|s|,v.v,s MAX s, s MIN s, s MOD s, s DIV s,
vector  v:   VECTOR(s,s,s),(s,s,s),s*v,v*s,v/s,v+v,v-v,v*v,r*v,t*v,f*v,v WRT f,
                     UNIT(v),POS(f),AXIS(r)
rot     r:   ROT(v,s),(v,s),r*r,ORIENT(f)
frame   f:   FRAME(r,v),f+v,f-v,f*t,CONSTRUCT(v,v,v)
trans   t:   TRANS(r,v),(r,v),f→f,t*t,INV(t)
boolean b:   ¬b, NOT b, b∧b, b AND b, b∨b, b OR b, b≡b, b EQV b, b⊕b, b XOR b,
                     s<s,s≤s,s=s,s≠s,s>s,s≥s
dimension d: d*d,d/d,INV(d)

FUNCTIONS:
scalar       INT(s),SQRT(s),SIN(s),COS(s),TAN(s),ASIN(s),ACOS(s),ATAN2(s,s),LOG(s),
             EXP(s),INSCALAR,RUNTIME,RUNTIME(s)
boolean      QUERY(print list)

AL           s:   π,PI,BHAND,YHAND,GHAND,RHAND,DRIVER_TURNS
CONSTANTS    v:   XHAT,YHAT,ZHAT,NILVECT
AND          r:   NILROT
VARIABLES    f:   STATION,BPARK,YPARK,GPARK,RPARK,⊗(valid only in MOVE)
                  BARM,YARM,GARM,RARM,DRIVER_GRASP,DRIVER_TIP
             t:   NILTRANS
             b:   TRUE, FALSE
    strings:      CRLF,NULL
    units:        CM,INCH,INCHES,OUNCES,OZ,GM,LBS,SEC,
                  SECONDS,DEG,DEGREES,RADIANS,RPM
    dimensions:   DISTANCE,TIME,FORCE,ANGLE,TORQUE,ANGULAR_VELOCITY,VELOCITY,
                  DIMENSIONLESS

STATEMENTS:
comment      COMMENT <any text without semicolon>;
             { <any text> }

control      FOR s ← <scalar> STEP <scalar> UNTIL <scalar> DO <statement>;
             IF <condition> THEN <statement> ELSE <statement>;
             IF <condition> THEN <statement>;
             WHILE <condition> DO <statement>;
             DO <statement> UNTIL <condition>;
             CASE <scalar> OF BEGIN S;S;... S END;
             CASE <scalar> OF BEGIN [i1] S; [i2] S; ...ELSE j←0; [i3][i4] S END;

affix        AFFIX f1 TO f2 AT t1 RIGIDLY;
             AFFIX f3 TO f4 BY t2 NONRIGIDLY;
             AFFIX f3 TO f4 BY t2 AT t1 NONRIGIDLY;
```



```
unfix        UNFIX f5 FROM f6;

condition    ON FORCE(<vector>) <rel> <force scalar> DO <statement>;
monitor      ON TORQUE <rel> <torque scalar> ABOUT <vector> DO <statement>;
statement    ON |FORCE| <rel> <force scalar> ALONG <axis vect> OF f1 DO ..;
and          ON TORQUE <rel> <torque scalar> ABOUT <axis vect> OF f1 IN HAND DO ..;
clauses      ON DURATION ≥ <time scalar> DO <statement>;
             ON ARRIVAL DO <statement>;
             ON DEPARTING DO <statement>;
             ON ERROR = <scalar> DO <statement>;
             <label>: DEFER ON <event> DO <statement>;
                  <rel> is ≥ or <

with         FORCE, TORQUE, DURATION similar to condition monitor
clauses      WITH FORCE_FRAME = <frame> IN <co-ord sys>
             WITH SPEED_FACTOR = <scalar>
             WITH APPROACH = <distance scalar> or <distance vector> or <frame>
             WITH DEPARTURE = ....
             WITH WOBBLE = <scalar>
             WITH NULLING or NO_NULLING
             WITH FORCE_WRIST ZEROED or NOT ZEROED
             WITH STIFFNESS = (v,v) ABOUT f
             WITH STIFFNESS = (s,s,s,s,s,s) ABOUT f IN WORLD
             WITH GATHER = (FX,FY,FZ,MX,MY,MZ,T1,T2,T3,T4,T5,T6,TBL)

enable       ENABLE <label>
disable      DISABLE <label>

motion       MOVE f1 TO <fval>;
             MOVE f1 TO <frame> VIA <frame>,<frame>,<frame>;
             MOVE f1 TO <fval>
                  VIA <frame> WHERE DURATION = <time scalar>,
                                    VELOCITY = <velocity vector>
                                    THEN <statement>
                  <more clauses>;
             MOVE f1 TO <fval> <more clauses>;
             OPEN  <hand> TO <distance scalar>;
             CLOSE <hand> TO <distance scalar>;
             CENTER <arm>;
             OPERATE <device> <clauses>
             STOP <device>;
             RETRY;

print        PRINT(<e>,<e>,...,<e>)      e is an expression, variable or string constant
abort        ABORT(<e>,<e>,...,<e>)      similar to print
prompt       PROMPT(<e>,<e>,...,<e>)     similar to print
pause        PAUSE <time scalar>;

wrist        WRIST(fv,tv);
             SETBASE;

signal       SIGNAL e1;
wait         WAIT e1;

assignment   <var> ← <expression>

return       RETURN
             RETURN(<expression>)
```

```
require     REQUIRE SOURCE_FILE "DSK:FILE.EXT";
            REQUIRE COMPILER_SWITCHES "LSK";
            REQUIRE ERROR_MODES "LAMF";
            REQUIRE MESSAGE "<message>";

macro       DEFINE <macro_name> ≡ ⊂ <macro_body> ⊃;
            DEFINE <macro_name>(m1,m2,...) ≡ ⊂ <macro_body> ⊃;

MACROS:     DIRECTLY          WITH APPROACH=NILDEPROACH
                              WITH DEPARTURE=NILDEPROACH
            CAUTIOUS          SPEED_FACTOR ← 6.0
            SLOW              SPEED_FACTOR ← 4.0
            QUICK             SPEED_FACTOR ← 1.0
            CAUTIOUSLY        WITH SPEED_FACTOR = 6.0
            SLOWLY            WITH SPEED_FACTOR = 4.0
            NORMALLY          WITH SPEED_FACTOR = 2.0
            QUICKLY           WITH SPEED_FACTOR = 1.0
            PRECISELY         WITH NULLING
            APPROXIMATEIV     WITH NO_NULLING
```


Appendix IV. POINTY COMMAND SUMMARY

```
PROCEDURES:      PROCEDURE p2(VALUE SCALAR vs1,vs2(23)); S;

OPERATIONS:
vector ∀ :   v REL F
frame f:     CONSTRUCT(f,f,f),f1 REL f2

FUNCTIONS:
boolean      ARMREACH(arm,f),ISAFFIXED(f,f)

STATEMENTS:

affix        AFFIX f3 TO f4 ;
             AFFIX f3 TO f4 AT II *;

motion       MOVE f1 BY <vector exp>;
             MOVEX or MOVEY or MOVEZ f1 BY <scalar>;
             OPEN <hand> BY <scalar>;
             CLOSE <hand> BY <scalar>;
             DRIVE BJT(<jt no>) TO <scalar>;
             DRIVE YJT(<jt no>) BY <scalar>;

assignment   POS(<var>) ← <vect expression>;
             ORIENT(<var>) ← <rot expression>;
             XCOORD(<var>) ← <scalar expression>;

input/output READ;        READ <file>;          QREAD <file>;
(file)       WRITE;       WRITE INTO <file>;    WRITE <id_list>;
             WRITE ALL INTO <file>;             WRITE <id_list> INTO <file>;
             PHOTO <file>;

edit         RENAME <var>;
rename       EDIT <var>;

display      DISPLAY SCALAR;
(terminal)   REDISPLAY; NODISPLAY;
             SHOW var1,var2,var3,...,varn;

(vt05)       VT05_ON;    VT05_BLUE;       VT05_GREEN;
             VT05_OFF;   VT05_YELLOW;     VT05_RED;

deletion     DELETE s1,s2,v1,v2,...;
             DELETE;
             DELETE ALL;

macro        DEFINE m2(mm1,mm2,...,mmx(mdx),mmy(mdy),mmz(mdz))≡ ⊂ <macro_body> ⊃;

help         HELP;        HELP <keyword>;
```

Appendix V. AL EXECUTION SUMMARY

```
      DEVICE      USER RESPONSE          AL RESPONSES

1.    Terminal    Create file FOO.AL

2.                COMPILE FOO.AL
2a.                                      Swapping to SYS: AL. DMP
                                         AL: FOO 1 2 3 ...

2b.                                      ALC

2c.                                      PALX 246

2d.                                      ALSOAP

3.                DO AL[AL,HE]
                                         f-
                  FOO
                                         CORE SIZE = 28K
                                         VERSION USING VT05
                                         GET SAV FILE - AL[AL,HE]
                                         OVERLAY BIN FILE - FOO
                                         AN EXTENDED COMMAND - VT05
                                         START AT (1000) (D FOR DDT) - D
                                         DDT STARTED AT 130000

4.    VT05                               *
                  START <alt><alt> G
            or    <alt>G
                                         AL RUNTIME SYSTEM
                                         <any output from your program>
                                         ALL DONE NOW. SEE YOU AROUND!
                                         ELAPSED TIME = 24.928 SECONDS

                                         NO ACTIVE PROCESSES LEFT. YOU'RE IN DDT.
                                         BE;SRFADL+50>>BPT
5.                <alt>G {for re-execution}

6.    ARM         pull on the yellow cord
      POWER       around table to kill arm power
```


Appendix VI. AL examples

Here are several sample AL programs. A brief description is given for each of the programs given.

ENGINE ASSEMBLY

This program causes the arm to pick up a crankshaft assembly and lower it into the body of the engine. It then picks up the top of the engine and places it over the crankshaft, thereby completing the assembly.

```
BEGIN "assemble engine"
{ program to place the crankshaft assembly and the engine top on
    the engine body}

FRAME engine_top,engine_top_final,crankshaft,crankshaft_final;
FRAME bgrasp;

engine_top ← FRAME (ROT(ZHAT,90.000*DEGREES),VECTOR(51.0,34.2,3.13)*INCHES);
engine_top_final ← FRAME (ROT(ZHAT,45.000*DEGREES),VECTOR(57.3,49.3,10.2)*INCHES);
crankshaft_final ← FRAME (ROT(ZHAT,45.000*DEGREES),VECTOR(57.3,49.2,8.48)*INCHES);
crankshaft ← FRAME (ROT(ZHAT,90.000*DEGREES),VECTOR(51.3,40.3,4.09)*INCHES);

AFFIX bgrasp TO barm AT TRANS(ROT(xhat,180*degrees),nilvect*inches) RIGIDLY;

PRINT("ASSEMBLING ENGINE");

OPEN bhand TO 3.0*inches;       { open hand }
MOVE bgrasp TO crankshaft;
CENTER barm;                    { grab the crankshaft }

crankshaft ← bgrasp;

AFFIX crankshaft TO bgrasp RIGIDLY;

MOVE crankshaft TO crankshaft_final + 3*zhat*inches
     WITH DURATION=2*seconds;   { take crankshaft above engine }

MOVE crankshaft TO crankshaft_final - 0.3*zhat*inches
     WITH WOBBLE = 0.1*DEGREES
     WITH DURATION = 5*seconds; { insert piston }

UNFIX crankshaft FROM barm;
OPEN bhand TO 3.7*inches;       { release crankshaft }

MOVE bgrasp TO engine_top slowly;

CENTER barm;
engine_top←bgrasp;

AFFIX engine_top TO barm RIGIDLY;
MOVE engine_top TO engine_top_final + 1.8*zhat*inches;
MOVE engine_top TO engine_top_final + 1.0*zhat*inches;
                              { by trial and error it was found
                              that doing this reduced oscillation
                              of crankshaft assembly }
MOVE engine_top TO engine_top_final - 0.3*zhat*inches
```

```
WITH FORCE_FRAME = STATION IN FIXED
WITH WOBBLE = 0.1 *DEGREES
WITH DURATION = 5*seconds
ON FORCE(zhat)≥80*ounces DO STOP engine_top;

UNFIX engine_top FROM barm;
OPEN bhand TO 3.8*inches;

MOVE bgrasp TO bgrasp + VECTOR(-4,-4,0)*INCHES;
                    ( can't move out straight because elbow joint
                     (joint 5) will be at limit, so we nwe the
                      band sideways )

MOVE barm TO bpark;          { all done now }

END "assemble engine"
```

SHIFTING CASTINGS

This program causes the arm to shift a row of three castings back and forth between two positions.

```
BEGIN "casting shifter"
  {program to shift a row of three castings back and forth between two positions}
 FRAME casting, casting_grasp, pick_up, set_down, line_1, line_2;
 DISTANCE VECTOR dpick, dset;
 SCALAR i,j,k;

 DEFINE TIL = ⊂STEP 1 UNTIL ⊃;

 line_1 ← FRAME(ROT(zhat,90 * degrees),VECTOR(28,30,0) * inches);
 line_2 ← FRAME(nilrot,VECTOR(32,24,0) * inches);

 AFFIX casting_grasp TO casting              {describe casting}
      AT TRANS(ROT(xhat,180 * degrees),VECTOR(1.2,1.5,1.87) * inches) RIGIDLY;

 OPEN bhand TO 3.5 * inches;

 FOR k ← 1 TIL 2 DO                            {do it all twice}
   BEGIN "outer loop"
     pick_up ← line_1 ;
     set_down ← line_2 - 0.8 * zhat * inches;        {[illegible] casting position}
     dpick ← -4 * yhat * inches;
     dset ← +4 * xhat * inches;

    FOR i ← 1 TIL 2 DO       {move the castings [illegible] and back again}
      BEGIN "inner loop"

        casting ← pick_up;
        MOVE barm TO casting_grasp;            {go get first one}

        FOR i ← 1 TIL 3 DO                     {[illegible] castings}
         BEGIN
         CENTER barm;
         casting_grasp ← barm;             {grab one}
         AFFIX casting TO barm RIGIDLY;
         MOVE casting TO set_down + 2 * zhat * inches    {shift it over}
                WITH APPROACH = 2 * inches;
```



```
         MOVE casting TO set_down DIRECTLY            {& place it on the table}
              ON FORCE(zhat) ≥ 100 * oz DO STOP;
         OPEN bhand TO 3.5 * inches;                  {release it}
         UNFIX casting FROM barm;
         pick_up ← pick_up + dpick;                   {update where next one is}
         set_down ← set_down + dset;                  {& where next goes}
         casting ← pick_up;
         IF i < 3 THEN MOVE barm TO casting_grasp      {go get next one}
                       WITH DEPARTURE = -3 * inches;
         END;

       pick_up ← line_2;          {get ready to move them back}
       set_down ← line_1 - 0.8 * zhat * inches;
       dpick ← +4 * xhat * inches;
       dset ← -4 * yhat * inches;
     END "inner loop";
   END "outer loop";

 MOVE barm TO bpark WITH DEPARTURE = -4 * inches;   {when done put the arm away}

END;
```

CASTING INSPECTOR

```
BEGIN "casting inspector"
  {The arm is moved to a pick up point where it grabs a casting.
       Depending on the weight of the casting it is either rejected or accepted.
       Rejected castings are dropped in a garbage bin, while accepted ones
       are lined up in a row. The program terminates after finding three
       good castings.}
 FRAME pick_up, set_down, garbage, casting, casting_grasp;
 DISTANCE VECTOR dset;
 SCALAR good_castings, heavy;

 set_down ← FRAME(nilrot,VECTOR(15,40,-0.8) * inches); {initial locations}
 pick_up ← FRAME(ROT(zhat,90 * degrees),VECTOR(4,44,0) * inches);
 garbage ← FRAME(ROT(zhat,90 * degrees),VECTOR(18,45,7) * inches);
 dset ← -4 * xhat * inches;

 AFFIX casting_grasp TO casting             {describe casting}
      AT TRANS(ROT(xhat,180 * degrees),VECTOR(1.2,1.5,1.87) * inches) RIGIDLY;

 OPEN bhand TO 3.5 * inches;

 good_castings ← 0;

 DO                    {loop to find 3 good castings}
  BEGIN

    casting ← pick_up;
    MOVE barm TO casting_grasp;              {go get a casting}
    CENTER barm;
    casting_grasp ← barm;
    AFFIX casting TO barm RIGIDLY;
    heavy ← false;                           {see if it weighs enough}
    MOVE casting TO ⊗ + 2 * zhat * inches DIRECTLY
        ON FORCE(-zhat) ≥ 85 * oz DO heavy ← true;

    IF heavy THEN
```

```
        BEGIN "good casting"
         MOVE casting TO set_down * 2 * zhat * inches DIRECTLY;
         MOVE casting TO set_down DIRECTLY
              ON FORCE(zhat) ≥ 90 * oz DO STOP;   {place it on table with others}
         OPEN bhand TO 3.5 * inches;
         UNFIX casting FROM barm;
         good_castings ← good_castings + 1;        {update • good ones}
         set_down ← set_down + dset;               {& where next goes}
         MOVE barm TO ⊗ + 3 * zhat * inches;
         END "good casting"
       ELSE
         BEGIN "bad casting"
         MOVE casting TO garbage DIRECTLY;
         OPEN bhand TO 3.5 * inches;               {trash bad one}
         PRINT("defective casting!",crlf);
         UNFIX casting FROM barm;
         END "bad casting";

    END
  UNTIL good_castings ≥ 3;          {repeat until we find 3 good castings}

  MOVE barm TO bpark DIRECTLY;              {put arm away when done}

END;
```

FORCE DEMONSTRATIONS

This program demonstrates the use of force sensing and compliance over a series of short examples.

```
BEGIN "force demo"
REQUIRE ERROR_MODES "F";    {coerce dimensionless quantities}

SCALAR thresh,dum,ybias,zbias,dur;
FRAME scr,hole,cup;
ROT stand;

scr ← FRAME(ROT(zhat,66.0*deg)*ROT(zhat,178.9*deg)*ROT(zhat,-113.4*deg),
              VECTOR(8.161,39.062,5.092)*inches);
stand ← ROT(xhat,180.0 * degrees);
hole ← FRAME(stand, VECTOR(17.7,27.5,1.625)*inches);
cup ← FRAME(stand, VECTOR(7.164,32.803,3.125)*inches);

MOVE barm TO scr WITH DURATION = 2*seconds;

WHILE QUERY("DO TRIGGER DEMO? ") do
       BEGIN   {this demo prints out when the threshold for • specified
                is exceeded in the principal directions}
       PRINT("trigger level? - "); thresh ← INSCALAR;
       PRINT("duration? - "); dur ← INSCALAR;
       MOVE barm TO scr DIRECTLY
              WITH GATHER=(fx,fy,fz,mx,my,mz) {gather all six components }
              WITH FORCE_FRAME = station
                     {π on the next three lines mean beep the terminal }
               ON |FORCE| ≥ thresh * ounces ALONG xhat DO print("xtriggerπ ")
              ON |FORCE| ≥ thresh * ounces ALONG yhat DO print("ytriggerπ ")
              ON |FORCE| ≥ thresh * ounces ALONG zhat DO print("ztriggerπ ")
```



```
              WITH DURATION = dur * seconds;
       END;

WHILE QUERY("DO COMPLIANCE DEMO? ") DO
       BEGIN        {makes arm compliant in various directions and planes}

       PRINT("FREE IN X DIRECTIONπ",crlf);
       MOVE barm TO scr DIRECTLY WITH DURATION = 1.5 * seconds;
       OPEN bhand TO 2 * inches; PAUSE(1 * second);
       CENTER BARM;
       MOVE barm TO scr DIRECTLY
              { for next 15 seconds arm will be compliant to external forces
              in x direction, i.e. pushing in x direction causes arm to move •
              along X direction, while the arm resists forces in other directions }
              WITH DURATION = 15 * seconds
              WITH FORCE_FRAME = station
              WITH STIFFNESS = (0,40,40,4000,400,400) ABOUT niltrans;

       PRINT("FREE IN X-Y PLANEπ",CRLF);
        MOVE barm TO scr DIRECTLY WITH DURATION = 1.5 * seconds;
       MOVE barm TO scr DIRECTLY
              { for next 15 seconds arm will be compliant to external forces
              in X-Y direction, i.e. it will comply to forces in the horizontal
              plane while resisting vertical forces }
              WITH DURATION = 15 * seconds
              WITH FORCE_FRAME = station
              WITH STIFFNESS = (0,0,40,2000,2000,400) ABOUT niltrans;

       PRINT("CENTER OF COMPLIANCE AT -4 INCHES",CRLF);
       MOVE barm TO scr DIRECTLY WITH DURATION = 1.5 * seconds;
       MOVE barm TO scr DIRECTLY
              { for next 10 seconds forces the arm will try to comply to
              about a point 4 inches above initial hand location (remember
              that compliance is specified in hand coordinates) }
              WITH DURATION = 10 * seconds
              WITH FORCE_FRAME = station
              WITH STIFFNESS = (20,20,20,30,30,30)
                     ABOUT FRAME(nilrot,-4 * zhat * inches);

       PRINT("CENTER OF COMPLIANCE AT 6 INCHES",CRLF);
        MOVE barm TO scr DIRECTLY WITH DURATION = 1.5 * seconds;
       MOVE barm TO scr DIRECTLY
              ( for next 10 seconds forces the arm will try to comply to
              about a point 6 inches along the fingers or the initial
              hand location (remember that compliance is specified in hand
              coordinates) }
              WITH DURATION = 10 * seconds
              WITH FORCE_FRAME = station
              WITH STIFFNESS = (15,15,15,20,20,20)
                     ABOUT FRAME(nilrot,6 * zhat * inches);
       END;

WHILE QUERY("DO PUSH DEMO? ") DO
       BEGIN   {pushes a rectangular block into a rectangular bd • ...
       FRAME here;
       PROMPT("PUT BLOCK IN HAND π");
        MOVE barm TO hole * 4 * zhat * inches DIRECTLY WITH DURATION=2 * seconds;
       MOVE barm TO hole - zhat * inches DIRECTLY
              WITH DURATION = 2 * seconds
              WITH FORCE_FRAME = station IN world
```

```
        ON |FORCE(zhat)| < 15 * ounces DO STOP barm;
    PRINT("OK - TOUCHED"); { block in place }

    here←barm;
    MOVE barm VIA here-2 * zhat * inches
        {now push down on the block}
        TO here DIRECTLY
        WITH DURATION = 4 * seconds
        WITH FORCE_WRIST NOT ZEROED
        WITH FORCE_FRAME = station
        WITH STIFFNESS = (30,30,30,30,30,30) ABOUT NILTRANS;
    END;

WHILE QUERY("DO EDGE FOLLOW DEMO ") DO
    BEGIN { this follows an edge }
    ybias←10; zbias←15; { suggesting possible values for ybias and zbias}
    PRINT("ybias force is: (",ybias,")"); ybias←INSCALAR;
    PRINT("zbias force is: (",zbias,")"); zbias←INSCALAR;
    PRINT("PUT BLOCK IN HAND π",CRLF);dum←QUERY("READY? ");
    FRAME here,side,hole;
    hole ← FRAME (STAND, VECTOR(17.7,27.5,1.625)*INCHES);
    side←hole+4 * yhat * inches;
    MOVE barm TO side + Z * zhat * inches WITH DURATION=2 * seconds;
    MOVE barm TO side - zhat * inches DIRECTLY
        WITH FORCE_WRIST ZEROED
        WITH DURATION = Z * inches
        WITH FORCE_FRAME = station
        ON |FORCE(zhat)| ≥ 15 * ounces DO STOP barm;
    STOP barm;
    PRINT("OK - TOUCHED");

    here←barm;
    MOVE barm TO here • 8 * xhat * inches
        VIA here • 4 * xhat WHERE VELOCITY = 4 * xhat * inches/second
        WITH DURATION = 4 * seconds
        DIRECTLY
        WITHFORCE_WRIST NOT ZEROED
        WITH FORCE_FRAME = station
        WITH STIFFNESS = (60,5,5,800,2000,30) ABOUT FRAME(NILROT,1*ZHAT)
        WITH FORCE(YHAT) = ybias * ounces
        WITH FORCE(ZHAT) = zbias * ounces
        with force(xhat) = .5*sqrt(zbias↑2+ybias↑2); {friction compensation}
    PAUSE(1*sec);
    MOVE barm TO barm + Z * zhat * inches DIRECTLY WITH DURATION • 1 * second;
    END;

WHILE QUERY("DO COFFEE DEMO? ") DO
    BEGIN    {picks up a coffee cup and waves it back and forth,
        releases the coffee cup when someone has grabbed it }
  SCALAR    got;
    cup ← FRAME (stand, VECTOR(7.164,32.803,3.125)*INCHES);
    OPEN bhand TO 3.5 * inches;
    MOVE barm TO cup WITH DURATION = 3 * seconds;
    CENTER barm;
    MOVE barm TO ⊗ + Z * zhat * inches DIRECTLY WITH DURATION = 1 * second;
                    {arm has the coffee cup}
    got←FALSE;
    WHILE ¬got DO BEGIN
        { waves arm back and forth until someone has grabbed the cup }
        MOVE barm VIA cup • VECTOR(-3,0,4) * inches
```



```
                VIA cup • VECTOR(-5,8,4) * inches
                VIA cup + VECTOR(-6,16,4) * inches
                TO cup • VECTOR(-5,8,4) * inches
                DIRECTLY
                WITH DURATION = 7 * seconds
                WITH FORCE_FRAME = station
                ON |FORCE(xhat)|≥20 * ounces DO
                    BEGIN
                    STOP barm;
                    got←TRUE;
                    END
                ON |FORCE(yhat)|≥20 * ounces DO
                    BEGIN
                    STOP barm;
                    got←TRUE;
                    END
                ON |FORCE(zhat)|≥20 * ounces DO
                    BEGIN
                    STOP barm;
                    got←TRUE;
                    END;

            IF got THEN OPEN    and TO 3.5 * inches;
            END:
        N

MOVE barm VIA cup • VECTOR(10,0,4) * inches TO bpark
    WITH DURATION = 4 * seconds;

END "force demo";
```

VISION MODULE

This program illustrates the use of the Vision module to do an inspection task.

```
BEGIN "inspct"
REQUIRE SOURCE_FILE "VISION.AL[AL,HE]";
REQUIRE ERROR_MODES "F";

FRAME view_point, set_down, set_down_bad, blob;
FRAME camera_park, camera_deploy, camera_show, cam1, cam2;

SCALAR x0,y0,xp,yp,do_it,i,j,parked,th,th0,nb,ng,prototype,distance,dd2,good;

InitVision;
parked ← FALSE;
x0 ← 0; y0 ← 0; nb ← 0; ng←0;
speed_factor ← 2.5;
view_point ← FRAME(ROT(zhat,-110*deg)*ROT(yhat,180*deg),
                VECTOR(48.810,25.510,7.00)*inches);
set_down ← FRAME(ROT(zhat,90*deg)*ROT(yhat,180*deg),
                VECTOR(32.0,34.0,5.00)*inches);
set_down_bad ← FRAME(ROT(zhat,90*deg)*ROT(yhat,180*deg),
                VECTOR(32.0,24.0,5.00)*inches);
camera_deploy ← FRAME(ROT(zhat,-23*deg)*ROT(yhat,-85*deg),
                VECTOR(45.884,26.826,32.0)*inches);
```

```
MOVE rarm TO camera_deploy DIRECTLY WITH SPEED_FACTOR = 3.0;

camera_show ← FRAME(ROT(zhat,-90*deg),vector(53.65,23.01,38.75)*inches);
cam1 ← FRAME(ROT(zhat,-180*deg),VECTOR(53.65,23.01,38.75)*inches);
cam2 ← FRAME(ROT(zhat,30*deg),VECTOR(53.65,23.01,38.75)*inches);

MOVE barm TO bpark DIRECTLY;
OPEN bhand TO 2.0 * inches;

IF QUERY("Show camera?") THEN
  BEGIN  {Pan crowd watching demo}
  MOVE rarm TO camera_show DIRECTLY WITH SPEED_FACTOR = 6;
  PAUSE 1*sec;     PROMPT("Ready to go on?");
  MOVE rarm TO camera_show DIRECTLY WITH SPEED_FACTOR = 9
       VIA cam1, camera_show, cam2;
  MOVE rarm TO camera_deploy DIRECTLY WITH SPEED_FACTOR = 4;
  END;

MOVE barm TO view_point VIA view_point + VECTOR(-7,5,4) DIRECTLY;
PROMPT("Put body between the fingers");
CENTER barm;
view_point ← barm;
OPEN bhand TO 3.2*inches;
MOVE barm TO ⊗ + VECTOR(-10,0,3)*inches WITH DEPARTURE = -3*inches;
                          {Get arm out of way}
Erase;
Picture(j);   {Get transformation from vision coordinates to arm coordinates}
GetFeature(1,xcent,x0);
GetFeature(1,ycent,y0);
GetFeature(1,THETA,th0); th0 ← th0 * (180.0 / pi);

PRINT("OK - here we go....",crlf);

DO BEGIN   {Go grab assemblies}
 do_it ← TRUE;
 good ← 0;

 WHILE do_it DO
  BEGIN
  i ← 0;
  DO BEGIN  {Get a.blob or time out after a minute}
   Erase;
   IF Picture(j) THEN PRINT("VM error: ",VM_Status,crlf);
   IF j=1 THEN RECOGNIZE(1,prototype,distance,dd2) ELSE distance ← 99;
   IF distance > 4*inches THEN
     BEGIN
     i ← i + 1;
     IF i<3 ∨ parked THEN PAUSE 2 * sec
      ELSE
       BEGIN  {Park arm if nothing going on}
       MOVE barm TO bpark WITH DURATION ≥ 2 * sec;
       parked ← TRUE
       END;
     END;
   END UNTIL distance < 4 ∨ i > 10;

   IF distance < 4 THEN          {Looks like we got one!}
     BEGIN
     GetFeature(1,xcent,xp);
     GetFeature(1,ycent,yp);
```



```
     GetFeature(1,THETA,TH); th ← th * (180.0 / pi);
     blob ← FRAME(ROT(zhat,th-th0),VECTOR(xp-x0,yp-y0,0));
     IF parked ∧ prototype ≤ 3 THEN MOVE barm TO bpark - 20*yhat*inches;
     IF good > 7 THEN do_it ← FALSE ELSE good ← good + 1;
     CASE prototype OF BEGIN
       [1] BEGIN   { gside - good assembly on its side }
           MOVE barm TO view_point * blob - 1.8*zhat
              WITH APPROACH = -1;
           CENTER barm;
           MOVE barm TO set_down;
           ng ← ng + 1;
           END;
       [2] BEGIN   { gup - good assembly with the stem up }
           MOVE barm TO view_point * blob
              WITH APPROACH = -1;
           CENTER barm;
           MOVE barm TO set_down;
           ng ← ng + 1;
           END;
       [3] BEGIN   { bad - not a good assembly }
           MOVE barm TO view_point * blob - 2.3*zhat
              WITH APPROACH = -1
              ON |FORCE(zhat)| ≥ 30 DO STOP;
           CENTER barm;
           MOVE barm TO set_down_bad;
           nb ← nb + 1;
           END;
       ELSE PRINT("unrecognized object",crlf)
       END;
     OPEN bhand TO 3.2;
     parked ← FALSE;
     END
    ELSE IF i > 10 THEN do_it ← FALSE;

  END;

 END UNTIL ¬QUERY("πKeep going?");

COBEGIN
  MOVE barm TO bpark;
  MOVE rarm TO camera_park DIRECTLY WITH SPEED_FACTOR = 3.0;
COEND;

END
```

Appendix VII. POINTY examples

LOCATE_ZUP

LOCATE_ZUP is used to determine the x and y coordinates of the axis of an upright cylinder. The macro tells the user to move the arm to the approximate location of the object, and then it does a center, reads the hand position, opens the hand, rotates it 90 degrees, closes it again and takes a second reading, and then produces a frame with station orientation. A similar macro which rotates the wrist 60 degrees is used for hexagonal cylinders.

```
DEFINE LOCATE_ZUP(ACTUAL_POS)=⊂
    PROMPT (" MOVE ARM TO APPROX POSITION OF ACTUAL_POS ");
            { lets user prompt when he is ready }
    CENTER BARM;    { use sensing to get position }
    OPEN BHAND TO BHAND_MAX;
            { BHAND_MAX has been defined elsewhere as 3.8 inches }
    MOVE BGRASP TO FRAME(ORIENT(↑ BARM)*ROT(ZHAT,90*DEGREES),POS(BGRASP));
            { so now we move the arm so that the wrist is rotated
            90 degrees but the hand points vertically downwards }
    CENTER BARM;
    ACTUAL_POS←FRAME(NILROT,POS(BGRASP));
            { and we determine the final position of the object
            but give it station orientation }
    OPEN BHAND TO BHAND_MAX;
    MOVEZ BGRASP BY 3*inches;   { open hand and get the arm out of the way }
    ⊃;
```

MOVE_AND_READ

This macro is used to ask the user to move the arm to a certain location. The position is then recorded.

```
DEFINE MOVE_AND_READ(POSITION) =
    ⊂
    { simple macro that asks user to move arm to a position and records it }
    PROMPT("MOVE ARM TO POSITION");
    POSITION ← BARM;
    ⊃;
```

MOVEMACRO3

This macro is used to define three positions and a new macro that will make the arm go through these positions. It illustrates the use of the MOVE_AND_READ macro, and may be used to teach motion through a series of three frames that will avoid obstacles in its path.


```
DEFINE MOVEMACRO3(MACNAME,P1,P2,P3) =
    ⊂
    MOVE_AND_READ(P1);
    MOVE_AND_READ(P2);
    MOVE_AND_READ(P3);
    DEFINE MACNAME =
        ⊂
        MOVE BARM VIA P1,P2 TO P3;
        ⊃;
    ⊃;
```

WRIST calibration routines

This is a set of macros and definitions used for the calibration of the wrist force sensor by POINTY. The source file is WRIST.LIB[1,JKS] or WRIST.LIB[TMP,MSM]. To do automatic calibration, do a READ WRIST.LIB or QREAD WRIST.LIB command.

```
{HAND LOCATIONS FOR CALIBRATION ROUTINE}
FRAME P32;
P32 ← FRAME (ROT(ZHAT,179.8*DEG)*ROT(YHAT,90.1*DEG)*ROT(ZHAT,90.4*DEG),
            VECTOR(4.050,44.876,20.234)*INCHES);
FRAME P31;
P31 ← FRAME (ROT(ZHAT,-180.00*DEG)*ROT(YHAT,90.067*DEG)*ROT(ZHAT,-90.000*DEG),
            VECTOR(4.11,44.8,20.2)*INCHES);
FRAME P22;
P22 ← FRAME (ROT(ZHAT,-180.00*DEG)*ROT(YHAT,90.067*DEG)*ROT(ZHAT,-179.96*DEG),
            VECTOR(4.11,44.8,20.2)*INCHES);
FRAME P21;
P21 ← FRAME (ROT(ZHAT,179.91*DEG)*ROT(YHAT,90.201*DEG)*ROT(ZHAT,-.64774*DEG),
            VECTOR(4.14,44.8,20.2)*INCHES);
FRAME P12;
P12 ← FRAME (ROT(ZHAT,56.937*DEG)*ROT(YHAT,.24644*DEG)*ROT(ZHAT,122.47*DEG),
            VECTOR(14.4,44.8,30.5)*INCHES);
FRAME P11;
P11 ← FRAME (ROT(ZHAT,-90.086*DEG)*ROT(YHAT,179.79*DEG)*ROT(ZHAT,89.659*DEG),
            VECTOR(14.4,44.7,9.94)*INCHES);

FRAME WEIGHTSTORE;
WEIGHTSTORE ← FRAME (ROT(ZHAT,-42.3*DEG)*ROT(YHAT,179.7*DEG)*ROT(ZHAT,-132.7*DEG),
            VECTOR(4.248,47.293,1.600)*INCHES);

FRAME TORQUESTORE;
TORQUESTORE ← FRAME (ROT(ZHAT,161.0*DEG)*ROT(YHAT,178.9*DEG)*ROT(ZHAT,159.5*DEG),
            VECTOR(12.007,43.654,1.335)*INCHES);

FRAME TORQUESTORE2;
TORQUESTORE2 ← FRAME (ROT(ZHAT,161.0*DEG)*ROT(YHAT,178.9*DEG)*ROT(ZHAT,-16.9*DEG),
            VECTOR(12.007,43.654,1.335)*INCHES);

{MACROS TO DRIVE ARM AND APPLY LOADS FOR WRIST SENSOR CALIBRATION}

{This group of macros makes use of the READWRIST( nnn ) command in POINTY to
process wrist sensor readings. nnn may be replaced by the following arguments:

 INIT - initialize
```

```
READ - read force sensors
BASE - base readings
RESOLVE - resolve forces and moments
CALIB - calibrate
SAVECALIB - save calibration data set on disk file
SAVECALIBPAL-save palx version of transposed calibration data
SAVERAWDATA - save wrist readings
COMPILEPALFILE - compile FORCAL.PAL on current ppn
COMPILEPALFILEONTOALHE - compile FORCAL.PAL onto AL,HE
DISPRAWDATA - display wrist raw data
RENAMEFILE - change force calibration data file}
```

{The following macro applies 6 linearly independent loads to the wrist sensor and derives the calibration matrix necessary to convert sensor readings into forces and moments. This is achieved by moving the hand to three different orientations and recording the strain gage readings resulting from the (known) hand weight. The hand is then directed to pick up a 500 gm weight and repeat the measurements in two different orientations. Finally the hand is directed to pick up a long bar which applies a known torque to the sensor. Further information on this calibration method may be found in "The Kinematic Design and Force Control of Computer Controlled Manipulators", Stanford A.I. Memo-313 by Bruce E. Shimano}

```
DEFINE WRISTCALIB =
    ⊂
    NODISPLAY;
    OPENBHAND TO 3.5;
    READNOLOAD(P11,P12,1);          {Read weight of hand with z-axis vertical}
    READNOLOAD(P21,P22,2);          {Read weight of hand with x-axis vertical}
    READNOLOAD(P31,P32,3);          {Read weight of hand with y-axis vertical}
    GETWEIGHT(WEIGHTSTORE);         {Pick up a 500 gm weight}
    READWITHLOAD(P21,P22,4);        {Read hand weight+500gm with x-axis vertical}
    READWITHLOAD(P31,P32,5);        {Read hand weight+500gm with y-axis vertical}
    REPLACE(WEIGHTSTORE);           {Place 500 gm weight on table}
    GETTORQUER(TORQUESTORE);        {Pick up 3.17 kg-cm torque device}
    READWITHTORQUE(P22,P21,6);      {Apply torque about z-axis and tea ● readings}
    REPLACE(TORQUESTORE2);          {Place torque device on table}
    MOVEBARM TO Pl1;                {Move arm out of the way}
    READWRIST(COMPUTE);             {Compute 6x8 calibration matrix}
{   READWRIST(SAVECALIBPAL);}       {Save assembly code version of calibration table}
{   READWRIST(SAVECALIB);}          {Save SAIL readable version of calibration table}
    REDISPLAY;
    ⊃;
```

{This macro takes strain gage readings with only the weight of the hand acting upon the sensor}

```
DEFINE READNOLOAD(F1,F2,N) =
    ⊂
    PRINT("MOVING TO F1",CRLF);
    MOVE BARM TO F1;
    READWRIST(READ);
    READWRIST(BASE);
    PRINT ("MOVING TO F2",CRLF);
    MOVEBARM TO F2;
    READWRIST(READ);
    READWRIST(CALIB,N);
    ⊃;
```

{This macro takes strain gage readings with the 500 gm weight in the hand}

```
DEFINE READWITHLOAD(F1,F2,N) =
    ⊂
```



```
    PRINT("MOVING TO F1",CRLF);
    MOVEBARM TO F1 ;
    READWRIST(READ);
    READWRIST(BASE);
    PRINT ("MOVING TO F2",CRLF);
    MOVEBARM TO F2;
    READWRIST(READ);
    READWRIST(CALIB,N);
    ⊃;
```

{This macro takes strain gage readings with the torque device in the hand}

```
DEFINE READWITHTORQUE(F1,F2,N) =
    ⊂
    MOVEBARM TO F1;
    MOVEBARM TO F1 ;
    PRINT ("APPLY NEGATIVE Z TORQUE ",CRLF);
    READWRIST(READ);
    READWRIST(BASE);
    MOVE BARM TO F2;
    REPLACE(TORQUESTORE2);
    GETTORQUER(TORQUESTORE);
    MOVEBARM TO F1 ;
    PRINT ("APPLY POSITIVE Z TORQUE",CRLF);
    MOVEBARM TO F1 ;
    READWRIST(READ);
    READWRIST(CALIB,N);
    MOVEBARM TO F2;
    ⊃;

DEFINE GETWEIGHT(F1 ) =
    ⊂
    MOVE BARM VIA F1+2*ZHAT TO F1;
CENTER BARM;
    OPEN BHAND BY 1.0*INCH;
    DRIVE BJT(6) BY -90*DEGREES;
    CENTER BARM;
    ⊃;

DEFINE GETTORQUER(F1 ) =
    ⊂
    MOVE BARM VIA F1+3*ZHAT*INCHES TO F1;
    CENTER BARM;
    MOVEBARM TO BARM+4*ZHAT*INCHES;
    ⊃;

DEFINE REPLACE(F1 ) =
    ⊂
    MOVEBARM VIA F1+3*ZHAT*INCHES TO F1;
    OPEN BHAND BY 1.0*INCH;
    MOVEBARM TO F1+3*ZHAT*INCHES;
    ⊃;

{macros for convenience}
DEFINE RR = ⊂ READWRIST ( READ ) ; READWRIST ( RESOLVE ) ; ⊃;
DEFINE RB = ⊂ READWRIST ( READ ) ; READWRIST ( BASE ) ; ⊃;
```

Appendix VIII. Vision Module Routines

The following is a listing of the source code file VISION.AL[AL,HE]

```
{Routines to call the Vision Module and have it do it's thing.
 For more details see VM.DOC & COMM.TXT ON [DOC,HE]}

SCALAR VM_STATUS;

{Definitions of status values: }

DEFINE    OK =⊂0⊃;                     {Successful}
DEFINE    NYI = ⊂1⊃;                   {"Not Yet Implemented"}
DEFINE    BadCmnd = ⊂3⊃;               {CmnNum out of range}
DEFINE    OutOfRange = ⊂5⊃;            {Argument too low or too high}
DEFINE    ReservedCommand = ⊂7⊃;       {Command number reserved for future use}
DEFINE    NotFound = ⊂9⊃;              {Blob is not on active blobs list}
DEFINE    BadBlob = ⊂11⊃;              {Blob specification is illegal}
DEFINE    NoBlob = ⊂13⊃;               {Active blobs list is empty}
DEFINE    BadFeatNum = ⊂15⊃;           {Illegal feature number)
DEFINE    BadName = ⊂17⊃;              {Name not recognized from table}
DEFINE    Duplicate = ⊂19⊃;            {New name same as old name (prototypes)}
DEFINE    BadRestartOption = ⊂21⊃;     {Option NEQ 0 or 8 for Restart}
DEFINE    RestartAbort = ⊂23⊃;         {Vision System restarted during command}

SCALAR PROCEDURE InitVision;
  {Initialize communications software. This routine must b called bfor ●
     anx other routines can be invoked.}
  BEGIN
  PROMPT("Is Vision Module running & is external-computer-control on? ");
  VM_STATUS ← VM(0,1,0,0,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Restart   (VALUE SCALAR how);
  {Restart the Vision System. Either hard or soft restart can be selected.}
  BEGIN      {how= 0 forhard restart, ≠ 0 for soft restart}
  IF how THEN how ← 8.0;
  VM_STATUS ← VM(1,1,0,how,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Picture (REFERENCE SCALAR numblobs);
  {Read a camera image into the image buffer and process it according to the
     currently set parameters.}
  BEGIN
  VM_STATUS←VM(2,1,0,0,1,0,numblobs);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE RePicture (REFERENCE SCALAR numblobs);
  {Reprocess the current image buffer using the currently set parameters.}
  BEGIN
  VM_STATUS←VM(3,1,0,0,1,0,numblobs);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE GetFeature (VALUE SCALAR blobnum, idnum; REFERENCE SCALAR valu);
  {Return the value of one or more blob features. se ● next pa ● for list of
```



```
     selectable features.}
  BEGIN
  VM_STATUS ← VM(4,2,0,blobnum,0,idnum,1,6,valu);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Blink   (VALUE SCALAR blobnum, color);
  {"Blink" the specified blob (Draw a line around the perimeter).
     The outline can be either bright or dark.}
  BEGIN
  IF color THEN color← -1;
  VM_STATUS ← VM(5,2,0,blobnum,0,color,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE DelBlob (VALUE SCALAR blobnum);
  {Delete a blob from the blob list. This removes the blob descriptor from
     free storage. If the blob was previously "blinked", it is "unblinked"
     by drawing a dark line over its outline.}
  BEGIN
  VM_STATUS ← VM(6,1,0,blobnum,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Calibrate (VALUE SCALAR blobnum);
  {Calibrate the various scale factors in the Vision System. The specified
     blob is used (o calculate the size of pixels and the origin of the
     Vision System coordinate system. The calibration blob must be round
     and its size must have been previously specified as the value of the
     variable "CALIBRATION SIZE".}
  BEGIN
  VM_STATUS ← VM(7,1,0,blobnum,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Remember (VALUE SCALAR proid; VALUE STRING name);
  {Create an empty prototype descriptor in the Vision System and optionally
     give it a name.}
  BEGIN
  VM_STATUS ← VM(8,2,0,proid,4,name,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Train(VALUE SCALAR blobnum, proid);
  {Add the features of a specified blob into the specified prototype. Both the
     mean and the variance about the mean are calculated for ibos ● features
     which can be used for recognition.}
  BEGIN
  VM_STATUS ← VM(9,2,0,blobnum,0,proid,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Recognize (VALUE SCALAR blobnum; REFERENCE SCALAR proid, d1, d2);
  {Identify which prototype is closest to the specified blob. All prototypes
     known to the system are compared to the specified blob and the closest
     one is identified and its "distance" from the blob is calculated.}
  BEGIN
  VM_STATUS ← VM(10,1,0,blobnum,3,0,proid,2,d1,2,d2);
  RETURN(VM_STATUS);
  END;
```

```
SCALAR PROCEDURE WhereAre (VALUE SCALAR proid, matches, newpic;
                          SCALAR ARRAY vals[1:5,1:5]);
  {Identify all blobs which are acceptably close to the specified prototype.
     A list of their identities and information about them is returned.}
  BEGIN
  SCALAR b1,b2,b3,b4,b5; {Temporary storage}
  SCALAR d1,d2,d3,d4,d5;
  SCALAR x1,x2,x3,x4,x5;
  SCALAR y1,y2,y3,y4,y5;
  SCALAR o1,o2,o3,o4,o5;

  IF newpic > 0 THEN newpic ← 1 ELSE IF newpic < 0 THEN newpic ← 255; {='377}

  CASE matches OF BEGIN
[0]    BEGIN
       PRINT(crlf,"WhereAre: max number of matches must be at least 1",crlf);
       VM_STATUS ← OutOfRange;
       RETURN(VM_STATUS);
       END;
[1]    VM_STATUS ← VM(11,3,0,proid,0,1,0,newpic,5,0,b1,2,d1,2,x1,2,y1,2,o1);
[2]    VM_STATUS ← VM(11,3,0,proid,0,2,0,newpic,10,0,b1,2,d1,2,x1,2,y1,2,o1,
                          0,b2,2,d2,2,x2,2,y2,2,o2);
[3]    VM_STATUS ← VM(11,3,0,proid,0,3,0,newpic,15,0,b1,2,d1,2,x1,2,y1,2,o1,
                 0,b2,2,d2,2,x2,2,y2,2,o2,0,b3,2,d3,2,x3,2,y3,2,o3);
[4]    VM_STATUS ← VM(11,3,0,proid,0,4,0,newpic,20,0,b1,2,d1,2,x1,2,y1,2,o1,
                 0,b2,2,d2,2,x2,2,y2,2,o2,0,b3,2,d3,2,x3,2,y3,2,o3,
                          0,b4,2,d4,2,x4,2,y4,2,o4);
ELSE  VM_STATUS ← VM(11,3,0,proid,0,5,0,newpic,25,0,b1,2,d1,2,x1,2,y1,2,o1,
                 0,b2,2,d2,2,x2,2,y2,2,o2,0,b3,2,d3,2,x3,2,y3,2,o3,
                 0,b4,2,d4,2,x4,2,y4,2,o4,0,b5,2,d5,2,x5,2,y5,2,o5)
  END;

  vals[1,1]←b1; vals[1,2]←d1; vals[1,3]←x1; vals[1,4]←y1; vals[1,5]←o1;
  IF matches > 1 THEN
      BEGIN
      vals[2,1]←b2; vals[2,2]←d2; vals[2,3]←x2; vals[2,4]←y2; vals[2,5]←o2;
      END;
  IF matches > 2 THEN
      BEGIN
      vals[3,1]←b3; vals[3,2]←d3; vals[3,3]←x3; vals[3,4]←y3; vals[3,5]←o3;
      END;
  IF matches > 3 THEN
      BEGIN
      vals[4,1]←b4; vals[4,2]←d4; vals[4,3]←x4; vals[4,4]←y4; vals[4,5]←o4;
      END;
  IF matches > 4 THEN
      BEGIN
      vals[5,1]←b5; vals[5,2]←d5; vals[5,3]←x5; vals[5,4]←y5; vals[5,5]←o5;
      END;
  IF matches > 5 THEN
      PRINT(crlf,"WhereAre: max number of matches can't be > 5",crlf);

  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE WhereIs (VALUE SCALAR proid, newpic;
                REFERENCE SCALAR blobnum, dis, x, y, ori);
  {Special case of WhereAre. Identifies one blob which is acceptably dos
     to the specified prototype. Information about it is returned.}
```



```
  BEGIN
  IF newpic > 0 THEN newpic ← 1 ELSE IF newpic < 0 THEN newpic ← 255; {='377 = -1}
  VM_STATUS ← VM(11,3,0,proid,0,1,0,newpic,5,0,blobnum,2,dis,2,x,2,y,2,ori);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Forget (VALUE SCALAR proid);
  {Delete a prototype descriptor from the Vision System.}
  BEGIN
  VM_STATUS ← VM(12,1,0,proid,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE SRead (STRING name; REFERENCE SCALAR valu);
  {Return the value of a switch. The value will be 0 if the switch is off,
     and -1 if the switch is on.}
  BEGIN
  VM_STATUS ← VM(13,1,4,name,1,0,valu);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE SWrite (STRING name; SCALAR valu);
  {Write the value of a switch. The value should be 0 to turn the switch off,
     and -1 to turn it on.}
  BEGIN
  IF valu THEN valu ← -1;
  VM_STATUS ← VM(14,2,0,valu,4,name,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE VRead (STRING name; REFERENCE SCALAR valu);
  {Read the value of a variable.}
  BEGIN
  VM_STATUS ← VM(15,1,4,name,1,2,valu);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE VWrite (STRING name; SCALAR valu);
  {Write the value of a variable.}
  BEGIN
  VM_STATUS ← VM(16,2,2,valu,4,name,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Erase;
  {Erase graphics overlay.}
  BEGIN
  VM_STATUS ← VM(24,1,0,0,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE ClearW (VALUE SCALAR x, y, dx, dy, color);
  {Clear a rectangular area in graphics overlay.}
  BEGIN
  IF color THEN color ← -1 ;
  VM_STATUS ← VM(25,5,0,x,0,y,0,dx,0,dy,0,color,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE Draw (VALUE SCALAR x, y, dx, dy, color);
```

```
  {Draw a vector on the display. The vector can be either bright or dark.}
  BEGIN
  IF color THEN color ← -1;
  VM_STATUS ← VM(26,6,0,x,0,y,0,dx,0,dy,0,color,0;0,0);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE DText (VALUE SCALAR x, y; VALUE STRING text;
                                        REFERENCE SCALAR nx, ny);
  {Display a text string on the display. The characters will always b bright.}
  BEGIN
  VM_STATUS ← VM(27,3,0,x,0,y,4,text,2,0,nx,0,ny);
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE ProRead (VALUE SCALAR proid; SCALAR ARRAY des[1:144]);
  {Read the contents of a prototype descriptor from the Vision System.
     This is currently a core image of the prototype. for now, it is
     intended for saving and restoring the prototypes only.}
  BEGIN
  PRINT(crlf,"Read Prototype not implemented",crlf);
  VM_STATUS ← NYI; {Not Yet Implemented}
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE ProWrite (SCALAR ARRAY des[1:144]);
  {Write a prototype descriptor to the Vision System.     The format should be
     identical to that previously read from the Vision System by ProRead.}
  BEGIN
  PRINT(crlf,"Write Prototype not implemented",crlf);
  VM_STATUS ← NYI; {Not Yet Implemented}
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE PicRead (VALUE SCALAR rasnum; SCALAR ARRAY data[1:16]);
  {Read the contents of one row of the currently selected image buffer.}
  BEGIN rasnum←1; {to avoid parser warning}
  PRINT(crlf,"Picture Read not implemented",crlf);
  VM_STATUS ← NYI; {Not Yet Implemented}
  RETURN(VM_STATUS);
  END;

SCALAR PROCEDURE PicWrite (VALUE SCALAR rasnum; SCALAR ARRAY data[1:16]);
  {This writes one row in the currently selected image buffer.
     The format is the same as that returned by PicRead.}
  BEGIN rasnum←1; {to avoid parser warning}
  PRINT(crlf,"Picture Write not implemented",crlf);
  VM_STATUS ← NYI; {Not Yet Implemented}
  RETURN(VM_STATUS);
  END;

{Definitions of feature numbers: }

DEFINE    NEXT = ⊂0⊃;            {Link of all active blobs}
DEFINE    PARENT = ⊂2⊃;          {Address of parent blob}
DEFINE    BCOLOR = ⊂4⊃;          {0 if black, -1 if white}
DEFINE    NCELLS = ⊂8⊃;          {Area in pixels}
DEFINE    PerimOn = ⊂10⊃;        {Perimeter list intensified on display}
DEFINE    TOTALCELLS = ⊂12⊃;     {Area in pixels, including holes}
DEFINE    EBOUNDARY = ⊂14⊃;      {Pointer (0 and of boundary list}
```



```
DEFINE    NHOLES = ⊂16⊃;         {Number of holes in the blob}
DEFINE    BOUNDARY = ⊂18⊃;       {Pointer to start of boundary list}
DEFINE    XMIN = ⊂20⊃;           {Minimum X (relative to XBASE)}
DEFINE    NBSEG = ⊂22⊃;          {Number of boundary segments}
DEFINE    XMAX = ⊂24⊃;           {Maximum X (relative to XBASE)}
DEFINE    NBCELLS = ⊂26⊃;        {Total perimeter length = XPERIM+YPERIM}
DEFINE    YMIN = ⊂28⊃;           {Minimum Y (relative to YBASE)}
DEFINE    MODEL = ⊂30⊃;          {Pointer to matching prototype}
DEFINE    YMAX = ⊂32⊃;           {Maximum Y (relative to YBASE)}
DEFINE    EXLINK = ⊂34⊃;         {Extra word for list links}
DEFINE    XPERIM = ⊂36⊃;         {Perimeter length in X-direction}
DEFINE    RMINI = ⊂38⊃;          {Index of minimum radius point}
DEFINE    YPERIM = ⊂40⊃;         {Perimeter length in Y-direction}
DEFINE    RMAXI = ⊂42⊃;          {Index of maximum radius point}

DEFINE    SIGX = ⊂44⊃;           {Sigma X for the blob (relative to XBASE)}
DEFINE    SIGY = ⊂48⊃;           {Ditto Y (relative to YBASE)}
DEFINE    SIGXX = ⊂52⊃;          {Sigma X squared, times 12}
DEFINE    SIGXY = ⊂56⊃;          {Sigma X times Y, times 2}
DEFINE    SIGYY = ⊂60⊃;          {Sigma Y squared}

DEFINE    AREA = ⊂64⊃;           {Area in calibrated units}
DEFINE    XCENT = ⊂68⊃;          {X centroid (relative to calibrated center)}
DEFINE    YCENT = ⊂72⊃;          {Ycentroid (relative to calibrated center)}
DEFINE    MAJOR = ⊂76⊃;          {Length of major axis of best ellipse}
DEFINE    MINOR = ⊂80⊃;          {Length of minor axis of best ellipse}
DEFINE    THETA = ⊂84⊃;          {Angle of major axis of best ellipse}
DEFINE    PERIMETER = ⊂88⊃;      {Perimeter length}
DEFINE    TOTALAREA = ⊂92⊃;      {Area + hole area}
DEFINE    RMIN = ⊂96⊃;           {Min (perim-cg)**2}
DEFINE    RMAX = ⊂100⊃;          {Max of same}
DEFINE    RMINANG = ⊂104⊃;       {Angle of minimum radius vector}
DEFINE    RMAXANG = ⊂108⊃;       {Angle of maximum radius vector}
DEFINE    AVRAD = ⊂112⊃;         {Avg (perim-cg)**2 for all perim points}
DEFINE    LENGTH = ⊂116⊃;        {Dimensions of a bounding box}
DEFINE    WIDTH = ⊂120⊃;         { aligned with theta}
DEFINE    ORIENTATION = ⊂124⊃    {Orientation of object (unambiguous)}

DEFINE    HOLEAREA = ⊂128⊃;      {Area of holes (and their holes, etc)}
DEFINE    HOLERATIO = ⊂132⊃;     {Ratio of hole area to total area}
DEFINE    AXRATIO = ⊂136⊃;       {MINOR / MAJOR}
DEFINE    PSQUARED = ⊂140⊃;      {PERIMETER squared}
DEFINE    PPEA = ⊂144⊃;          {PSQUARED / AREA}
DEFINE    RADRATIO = ⊂148⊃;      {RMIN / RMAX}
DEFINE    LENRATIO = ⊂152⊃;      {WIDTH / LENTH}
DEFINE    YDIFF1 = ⊂156⊃;
DEFINE    YDIFF = ⊂160⊃;         {Y-Height}
DEFINE    XDIFF1 = ⊂164⊃;
DEFINE    XDIFF = ⊂168⊃;         {X-Width}
DEFINE    BOXAREA = ⊂172⊃;       {XDIFF * YDIFF}
DEFINE    BXARATIO = ⊂176⊃;      {BOXAREA / TOTALAREA}
DEFINE    XCENT2 = ⊂180⊃;
DEFINE    YCENT2 = ⊂184⊃;
DEFINE    CGDIST = ⊂188⊃;        {XCENT squared + YCENT squared}
DEFINE    PEROU1 = ⊂192⊃;
DEFINE    PEROUND = ⊂196⊃;       {4 * PI * AREA / PSQUARED}
DEFINE    ANGMOD = ⊂200⊃;        {RMAXANG - RMINANG (+-PI range)}
```

Appendix IX. Generating a new system

Introduction

This appendix is given for reference of system hackers and those who want to put up a new AL or POINTY system. It assumes familiarity with the WAITS operating system of the Stanford Artificial Intelligence Laboratory, and the use of the programs associated with it.

AL system

AL parser and compiler

These are the files that make up the AL parser and compiler.

```
AL parser          PARSE.SAI[AL,HE]

AL compiler        ALC.SAI[AL,HE]
                   GOBBLE.SAI[AL,HE]
                   PASS3.SAI[AL,HE]
                   ALPRIN.SAI[AL,HE]
                   ARITH.SAI[AL,HE]
                   INTDEF.SAI[AL,HE]
                   ALREC.SAI[AL,HE]
```

To generate a new AL parser, do the following (comments after semicolons are by way of explanation; the period at the front of the line is prompt of the WAITS system.

```
.AL AL,HE                    ;alias to [AL,HE] area
.COMP PARSE.SAI[AL,HE](R)    ;compile the source file with R switch

.LOAD/SAV PARSE              ;load PARSE with the SAIL runtime system
                             ;save the core image in PARSE.DMP

.REN ALOLD.DMP←AL.DMP[AL,HE]
                             ;renames old AL parser in case we need
                             ;it later
.RENAME AL.DMP ← PARSE.DMP ;renames PARSE.DMP to AL.DMP so that
                             ;calls to the AL system will get the
                             ;new system
```

Here is how to create a new AL compiler.

```
.AL AL,HE                    ;alias to [AL,HE] area
```



```
.REN ALCOLD.DMP←ALC.DMP      ;save old compiler
.R LOADER                    ;run loader which will prompt with a *
*ALC<alt>                    ;note the use of the <alt> key
```

When the loader is done, do the following:

```
.SETUWP
.SSAVE ALC
```

If you name the new compiler ALCNEW instead of ALC, then you can get that compiler instead of ALC by using the /N switch in the COMPILE FOO.AL statement.

The following instruction will take care of assembling all the files and making up the AL compiler in batch mode. The date and time indicate when the compilation is to begin.

```
.BATCH /DATE=5-may-78/TIME=1200/DO @BAT[AL,HE]
```

AL runtime system

These are the source files that make up the AL runtime system.

```
AL pcode interpreter     AL.PAL[AL,HE]
                         INTERP.PAL[AL,HE]
                         ALIO.PAL[AL,HE]
                         LARGEB.PAL[AL,HE]
                         SMALLB.PAL[AL,HE]
                         FLOAT.PAL[AL,HE]
                         GRAPHS.PAL[AL,HE]
                         VISIO.PAL[AL,HE]
                         VALIO.PAL[AL,HE]

AL arm code              ALARM.PAL[AL,HE]
                         ARM.PAL[AL,HE]
                         ARMSOL.PAL[AL,HE]
                         BEJCZY.PAL[AL,HE]
                         ARMSOL.PAL[AL,HE]
                         ARITH.PAL[AL,HE]

kernel                   K1.PAL[11,SYS]
```

The file AL.SAV is made up of the above files as shown:

```
AL.SAV[AL,HE] = AL.BIN[AL,HE] + ALARM.BIN[AL,HE] + K1.BIN[11,SYS]
```

The way to generate a new runtime system is:

```
.AL AL,HE                    ;alias to [AL,HE]
.REN ALOLD.SAV←AL.SAV        ;in case we need to save old
                             ;runtime system
.R 11TTY
*Zero $00000
*Load - AL.BIN
*Overlay - ALARM.BIN
*Overlay - K1.BIN[1 1,SYS]
*An extended command - TOP $00000
*Version using VT05
*Dump sav file - AL          ; this creates a new AL.SAV[AL,HE]
*
```

POINTY system

These source files make up the POINTY runtime system.

AL source code interpreter	POINTY.SAI[PNT,HE]
	MAINPR.SAI[PNT,HE]
	FILES.SAI[PNT,HE]
	DISPLY.SAI[PNT,HE]
	FORMAT.SAI[PNT,HE]
	OUTPUT.SAI[PNT,HE]
	TALK11.SAI[PNT,HE]
	HELP.SAI[PNT,HE]
	EXPR.SAI[PNT,HE]
	EXEC.SAI[PNT,HE]
	SCANNR.SAI[PNT,HE]
	PARSE.SAI[PNT,HE]
	INIT.SAI[PNT,HE]
	WRIST.SAI[PNT,HE]
	GRAPH.SAI[PNT,HE]
	PPCODE.SAI[PNT,HE]
	UTIL.SAI[PNT,HE]
	PCODE.SAI[PNT,HE]
	PPROC2.SAI[PNT,HE]
	PPROC.SAI[PNT,HE]
	PCALL.SAI[PNT,HE]
	SYMBOL.SAI[PNT,HE]
	PNEW.SAI[PNT,HE]
	DEBUG.SAI[PNT,HE]
	OUTDPW.FAI[PNT,HE]

POINTY pcode interpreter	POINTY.PAL[PNT,HE]
	AL.PAL[AL,HE]
	PINTRP.PAL[AL,HE]
	INTERP.PAL[AL,HE]
	ALIO2.PAL[PNT,HE]
	LARGEB.PAL[AL,HE]
	SMALLB.PAL[AL,HE]
	FLOAT.PAL[AL,HE]
	GRAPHS.PAL[AL,HE]
	VISIO.PAL[AL,HE]
	VALIO.PAL[AL,HE]
	TALK10.PAL[PNT,HE]
	PINTRP.PAL[PNT,HE]
POINTY arm code	PNTARM.PAL[PNT,HE]
	ALARM.PAL[AL,HE]
	ARM.PAL[AL,HE]
	ARMSOL.PAL[AL,HE]
	BEJCZY.PAL[AL,HE]
	ARMSOL.PAL[AL,HE]
	ARITH.PAL[AL,HE]
	PARM.PAL[AL,HE]
	EULER.PAL[AL,HE]
kernel	K1.PAL[11,SYS]
AL file for POINTY data structure	PNTY.AL[PNT,HE]
PPN and name of users	USERS.DAT[PNT,HE]
Initialization message	PNTMSG.INI[PNT,HE]
Initialization of PONTYO	POINTY.INI[PNT,HE]

The intermediate files are made up as follows:

POINTY.SAV[PNT,HE] = POINTY.BIN[PNT,HE] + PNTARM.BIN[PNT,HE] + K1.BIN[1 1,SYS] + PNTY.BIN[PNT,HE]

PONTYO.DMP[PNT,HE] = POINTY.REL[PNT,HE] + SAIL runtime system (POINTY.REL will take care of loading the other .REL files)

POINTY.DMP[PNT,HE] = PONTYO.DMP[PNT,HE] + POINTY.SAV[PNT,HE] + other initialization files

Creating the POINTY runtime system is a bit more involved than creating the AL system. However, since the instructions to compile all the files necessary for the PDP-10 part of POINTY is kept in a DO file, all you need to do is type:

```
.DO PCOMP[PNT,HE]
```

This results in the appropriate files getting compiled, and POINTY.REL[PNT,HE] is loaded with them and the SAIL runtime system. The core image is then saved in PONTY0.DMP[PNT,HE]. We next build up the runtime system.

```
.AL PNT,HE                    ;alias to [PNT,HE]
.R 11TTY
*Zero 500000
*Load - POINTY.BIN
*Overlay - PNTARM.BIN
*Overlay - K1.BIN[11,SYS]
*Overlay - PNTY.BIN
*An extended command - top 500000
*Version using vt05
*Write loc = 160000           ; this is to let the runtime system
Set to - 1                    ; know it is POINTY
                              ; otherwise it thinks it is AL
*Dump sav file - POINTY       ; this creates POINTY.SAV[PNT,HE]
*Start at (D for DDT) - D
DDT started at 130000

        <type <alt>G on the VT05 and wait for printing of values>

*X
Exit
```

The PDP-11 part is now running. We now need to get the interpreter started as follows:

```
.R PONTY0
```

This loads and executes the file PONTY0. The runtime system on the PDP-11 is restarted. (If for some reason you do not want PONTY0 to restart the program on the PDP-11, write the value 1 into location 160036 with 11TTY before running PONTY0.) The system is then initialized and POINTY instructions can be typed in.

We can use this configuration of loading the PDP-11 and then the PDP-10

each time we want to use POINTY. However, since initialization takes a long time especially if the system is busy, saving the core images of the PDP-10 and PDP-11 saves the state of the world, so that in future calls to POINTY, the initialization can be bypassed.

Before saving the core image, we may want to have put in a message that is printed out when POINTY in reinvoked at a future time. We can do this by means of the *RESUME_MESSAGE* command which takes a string constant as argument as follows:

```
*RESUME_MESSAGE(<a string message>)
```

Finally, the instruction to actually save the core image is as follows:

```
*SAVECOREIMAGE <filename>
```

<filname> can take any name except POINTY, and the extension is ignored; the extension .DMP will be used. POINTY is disallowed as <filename> to prevent inadvertently writing over an existing POINTY file. This instruction suspends execution of the program on the PDP-11, and causes POINTY (interactive source code interpreter) to read the core image of the PDP-11 and save the data within its own core image in the PDP-10. The whole core image is then written out onto a disk file. Typically, the names chosen are P0, P1,... Let us assume that it is P0. The interpreter then restarts execution of the runtime system on the PDP-11, and then resumes execution.

The next thing to do is to get out of the interpreter by means of the EXIT command, and to test that P0 in fact does the right thing. To do so, type:

```
.RU P0
```

and the PDP-11 will be reloaded and the interpreter restarted on the PDP-10. If you are happy and satisfied that the new system works, rename it to POINTY.DMP as follows:

```
.REN POINTY.DMP←P0.DMP
```

Debugging POINTY

For debugging the PDP-10 part, compile POINTY.SAI and the source file which needs to be debugged with BAIL, the SAIL debugger. Then to get access to BAIL during runtime, type the instruction:

```
*BAIL
```

If you want to include a string that could be subsequently typed into BAIL, it may be included in parentheses after the word BAIL as shown below:

```
*BAIL(BREAK("GTOKEN");!!GO;);
```

The above will get into BAIL, place a break point at GTOKEN and resume execution. An alternate form permits you to type in your instructions into a text file (without the table of contents page - unformatted file in the editor E) called QUERY.TXT. The *QBAIL* instruction will get you into BAIL and act as if you had typed the contents of QUERY.TXT to BAIL.

```
*QBAIL
```

To get access to 11DDT during execution, type the instruction DDT as follows to POINTY:

```
*DDT
```

This causes the runtime system to give control to 11DDT so that it is possible to step through the code on the PDP-11.

Other debugging aids can be utilized by means of the *SETSTATUS* and *RESETSTATUS* commands as shown:

```
*SETSTATUS(<parameter>,<value>);
*RESETSTATUS(<parameter>,<value>);
```

where <value> is an integer constant. If <value> is left out, the default is 1 for *SETSTATUS* and 0 for *RESETSTATUS*. <parameter> is one of the following reserved words, and the given effects when the values are non-zero.

NOELF	no output to the PDP-11 (instructions are only parsed and not executed)
NOFOLD	no constant folding and evaluation of constant scalars and reals (usually POINTY will try to evaluate constant expressions during parsing)
LINE	prints out the expanded version of the last statement on the terminal (useful for checking macro expansion)
PPCODE	the pcode form of the statement being parsed is printed out at the terminal
PWCODE	the pcode of the statement being parsed is appended to the file PPCODE.FOO
ALPRIN	subsequent printout of variables is in AL format rather than POINTY format
PRTIME	prints the execution time on the PDP-10 for the current

instruction

Force wrist calibration

If it is found that the calibration of the force wrist has drifted, automatic calibration of the force wrist can be done by using the *READWRIST* instructions described in Appendix VII.

Appendix X. REFERENCES

Binford,T.O., Grossman,D.D., Liu,C.R., Bolles,R.C., Finkel,R.A., Mujtaba,M.S., Roderick,M.D., Shimano,B.E., Taylor,R.H., Goldman,R.H., Jarvis,J.P., Scheinman,V.D., Gafford,T.A.; *EXPLORATORY STUDY OF COMPUTER INTEGRATED ASSEMBLY SYSTEMS*; Progress report 3 covering period from December 1, 1975 to July 31, 1976. Stanford Artificial Intelligence Laboratory MEMO AIM-285, Computer Science Department Report STAN-CS-76-568 Stanford University. August 1976.

Binford,T.O., Liu,C.R., Gini,G., Gini,M., Glaser,I., Ishida,T., Mujtaba,M.S., Nakano,E., Nabavi,H., Panofsky,E., Shimano,B.E., Goldman,R., Scheinman,V.D., Schmelling,D., Gafford,T.; *EXPLORATORY STUDY OF COMPUTER INTEGRATED ASSEMBLY SYSTEMS*; Progress report 4 covering period from August 1, 1976 to March 31, 1977. Stanford Artificial Intelligence Laboratory MEMO AIM-285.4, Computer Science Department Report STAN-CS-76-568 Stanford University. June 1977.

Finkel,R., Taylor,R., Bolles,R., Paul,R., Feldman,J.; *AL, A PROGRAMMING SYSTEM FOR AUTOMATION*; Stanford Artificial Intelligence Laboratory MEMO AIM-243 Computer Science Department Report STAN-CS-74-456 Stanford University. November 1974.

Finkel,R.A.; *CONSTRUCTING AND DEBUGGING MANIPULATOR PROGRAMS*; Stanford Artificial Intelligence Laboratory MEMO AIM-284, Stanford Computer Science Department Report STAN-CS-76-567 Stanford University. August 1976.

Goldman,R.; *RECENT WORK WITH THE AL SYSTEM*; Proceedings of the 5th International Joint Conference on Artificial Intelligence, Massachusetts Institute of Technology, Boston, August 1977.

Grossman,D.D., Taylor,R.H.; *INTERACTIVE GENERATION OF OBJECT MODELS WITH A MANIPULATOR*; Stanford Artificial Intelligence Laboratory Memo AIM-274, Stanford Computer Science Report STAN-CS-75-536 Stanford University. December 1975.

Mujtaba,M.S., Goldman,R.; *AL USERS' MANUAL*; Stanford Artificial Intelligence Laboratory, Stanford University. First Edition November 1977. Second Edition January 1979 (Stanford Artificial Intelligence Laboratory Memo AIM-323, Computer Science Department Report STAN-CS-79-718).

Mujtaba,M.S.; *POINTY, AN INTERACTIVE SYSTEM FOR ASSEMBLY*; 16 mm color with sound, 10 Minutes, Stanford Artificial Intelligence Laboratory, Stanford University, December 1977.

Mujtaba,M.S., Salisbury,J.K.; *FLASHLIGHT FACTORY*; Ten minute film showing flashlight assembly under AL. Stanford Artificial Intelligence Laboratory, Computer

Science Department, Stanford University. September 1979.

Mujtaba,M.S.; *Current Status of the AL Manipulator programming system*; Proceedings of the Tenth International Symposium on Industrial Robotics, Milan, Italy March 5-7, 1980.

Paul,L.; *MODELLING, TRAJECTORY CALCULATION AND SERVOING OF A COMPUTER CONTROLLED ARM*; PhD. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-177, Computer Science Department Report STAN-CS-72-311 Stanford University. March 1973.

Shimano,B.E.; *THE KINEMATIC DESIGN AND FORCE CONTROL OF COMPUTER CONTROLLED MANIPULATORS*; Stanford Artificial Intelligence Laboratory MEMO AIM-313, Stanford Computer Science Department Report STAN-CS-78-660 Stanford University. March 1978.

Taylor,R.H.; *A SYNTHESIS OF MANIPULATOR CONTROL PROGRAMS FROM TASK-LEVEL SPECIFICATIONS*; Stanford Artificial Intelligence Laboratory MEMO AIM-282, Computer Science Department Report STAN-CS-76-560 Stanford University. July 1976.

Appendix XI. ACKNOWLEDGEMENTS

We wish to acknowledge the following contributors who have brought the AL system to its current state of development: Tom Binford, Bob Bolles, John Craig, Ray Finkel, Jerry Feldman, Ron Goldman, Tom Gafford, Maria Gini, Pina Gini, Dave Grossman, Norman Haas, Tatsuzo Ishida, Pitts Jarvis, Oussama Khatib, Jeff Kerr, Bill Laaser, Dick Liu, Jim Maples, Hamid Nabavi, Lou Paul, Enrico Pagello, Ted Panofsky, Mike Roderick, Gene Salamin, Ken Salisbury, Vic Scheinman, Bruce Shimano, Barry Soroka, Russ Taylor, Rick Vistnes, Lee Winnick.

This work was supported by the National Science Foundation through the following grants: NSF-APR-74-01390-A04, NSF-DAR-78-15914 and NSF-MEA-80-19628.

9. INDEX

!!GO 113
!!GSTEP 113
!!STEP 113

!!DDT 8, 12, 66, 72, 76-77, 83,
  88-89, 151
!!AII 11-12, 78-79, 83-84, 88

ABORT 66
ADAC interface 8, 67
ADC 67
affixment 6, 11, 30-33, 55-56,
  66, 90, 92, 97
AL
  command summary 120
  execution summary 124
  software 11
  system hardware 8
ALAID 104
ALPRIN 151
ALSOAP 11, 75
ANGLE 15, 51
ANGULAR_VELOCITY 17, 51, 66
APPROACH 29, 57
APPROXIMATELY 65
arithmetic operators
  AL 52-54
  POINTY 96-97
ARMREACH 96
arrays 5, 44, 46, 50, 52, 70, 95
ARRIVAL 57, 63
assignment statement 7, 14-15,
  23, 68, 74, 92, 95, 102
AXIS 18-19

BAIL 76, 82, 111, 150
BEGIN 14, 22, 41, 49, 69-70
bendy pointer 91
bgrasp 97
BJT 100
block 22-23, 41
  naming 22, 49
  statement 22, 49, 95
  structure 6, 14, 49
bpark 25, 55
BREAK 112-113
8V ss, 98, 100

calibration
  pointer 91
  PUMA 77, 100
  vision system 140
  wrist 136-138
CASE 38, 69
CAUTIOUS 48, 65
CAUTIOUSLY 64
CCW 66
CENTER 28, 65, 99
CLOCKWISE 66
CLOSE 28, 65, 99
COBEGIN 41, 49, 70
ƐOƐNO 41, 49, 70
command summary
  AL 120
  POINTY 123
comments 15, 49, 95
compilation 74-76
condition monitors 7, 34, 50,
  57, 61-63, 67, 70-71, 98,
  104
CONSTRUCT 53, 97
COUNTER_CLOCKWISE 66
crlf 55
CW 66

DAC 67
data types 14
  EVENT 50
  FRAME 20, 50, 90
  LABEL 50
  ROT 18, 50
  SCALAR 15, 50

STRING 50
TRANS 21, 50
VECTOR I 7, 50
DEBUG 111-112
debugger 111
debugging aids 87
DEBUGOFF III
DEBUGON 111
declaration 23, 51, 95
DEFER 63, 71
DEFINE 51, 73, 103
DELETE 104
DEPARTING 57, 63
DEPARTURE 29, 57
deproach points 29, 57
design philosophy 3
dimension checking 15
dimensions 15, 51
DIRECTLY 29, 57
DISABLE 50, 62, 63
display I I, 60, 88-89, 92-94, 97, 105-106
DISTANCE 15, 51
AL[AL,HE] 76
DRIVE 100
DUMP_VARIABLES 114
DURATION 57, 62, 64, 66

ECHOOFF 107
ECHOON 107
EDIT 104, 114
ENABLE 50, 61, 63
END 14, 22, 41, 49, 69-70
ERROR_MODES 74, 81, 87
ERROR clause 64
errors 80-88
correction 80
loading 83
messages 80-81
PALX 83
parsing 80
recovery 7, 80
response 81

runtime 84
events 41, 50-51, 70
EXCESSIVE_FORCE 64
execution 76
EXIT 114
expressions 52, 96

FALSE 36, 55
fiducial 91
files I I
.ALP II, 75
.ALS II
.BIN II
.LOG II, 87
.NEW II
.AIV II
extension 11
input/output 106
logging 11
name 11
S-expression 11
FINISH 8 7
flexible fixturing I 16
FOR 36, 64, 68
force 35
FORCE 15, 51, 59, 61-62
FORCE_FRAME 59, 61-62
FORCE_WRIST 60
force application 35, 58-59
force sensing 34, 58-59, 61-62, 129
FRAME 6, 20, SI, SI, 90

GAL II, 61, 88
GATHER 60, 88, 106
GJT 100
gpark SS
GRAPH 106
grinch ZS, 86

HALT 113
HAND 58-59
HELP module 108

I/O 72
identifier 14
IF 36, 68
initialization 48
INSCALAR 52, 72
ISAFFIXED 96

KL10 8

LABEL 50-51
LINE 151
LOAD_VARIABLES 114

macros 6, I 7, 12, 102-103
MESSAGE 74, 87
MOVE 24-25, 28-30, 56, 98
MOVEX 99
MOVEY 99
MOVEZ 99

NILDEPROACH 29, 57
nilrot I 9
niltrans ZZ
nilvect 18
NO_NULLING 64-65
NODISPLAY 105
NOELF 151
NOFOLD 151
NONRIGIDLY 31, SS, 9 7
NORMALLY 64
NOUPDATE 105
NOWAIT 68
NULL SS
NULLING 65

ON 61-63
OPEN 28, 65, 99
OPERATE 66
ORIENT 20, 97, 102

PALX 11-12, 15, 81
panic button 64, 76-77, 80, 84, 86
parallel processes 6, 41, 50, 62, 70
PARK 87
PAUSE 71
pcode II
PDP-10 8, 88
PDP-II 8, 83
cross assembler 11
load module 75
PFREE 101
PGRAV 101
PHOTO 108
PJOINT 101
plantime system 4
PLOT 89
pointer 91
POINTY command summary 123
POINTY sequence 115
POS 20, 102
PPCODE 151
precedence relations 54
PRECISELY 65
predeclared variables 54, 95
PRINT 23, 72
procedures 6, 45, 70, 102
program 22, 49
PROMPT 72
PRTIME 151
PTABLE 101
PTOOL 101
PUMA 8
calibration 100
talking to VAL 67
PWCODE 151

QBAIL 151
QDELETE 104
QREAD 105
QUERY 53, 72, 96
QUICK 48, 65
QUICKLY 64
QUIT 112

READ 107
READMESSAGE 114
READWRIST 136-137, 152
REDEFINE 103-104
REDISPLAY 105
REFERENCE **JO**
REL 97
RENAME 114
REQUIRE 67, 73-74, **81**, 87
RESETSTATUS 151
RESTART 112
RESUME_MESSAGE 149
RETRY 64, 87
RETURN **JO**
RIGIDLY **8I**, 55, 97
RJT **100**
ROT 18, 51, 53
rpark 55
RUNTIME 52, **96**
runtime system 4, 146

SAVECOREIMAGE 150
SCALAR 15, 51
Scheinman arm 8, 91
SETBASE **60**
SETSTATUS 151
SHOW 105
SIGNAL 41, 50, 70
SLOW 48, 65
SLOWLY 64
software organization 11
SOURCE_FILE 67, **TI**
SPEED_FACTOR 48, 64
station **ZO**, 25, 54
stiffness 35, 58
STOP 34, 66
STOPMESSAGE 114
strings 22, 50-51, 72
synchronization 41

IEXI 112
TIME 15, 51
TIME_OUT 64
TORQUE 17, 51, 59, **61-62**, **66**
TRANS 21, 51, 54
TRAPS 112
TRUE 36, 55
type incompatibility 81

UNBREAK **IE8**
UNFIX **8I**, 55, 98
UNIT 17-18
UNTIL 40, **69**
upeaIE 105

VAL 67
VALUE **TO**
variable 14, 49
VECTOR 17, 51, **SE**
VELOCITY 17, 51, 57
VIA 29-30, 56-57
Vision Module 8, 67
VM 67
VT05 72, 77, 79, 84, 93, 100-101
VT05_BLUE **106**
VT05_GREEN **106**
VT05_OFF **106**
VT05_ON **106**
VT05_RED **106**
VT05_YELLOW 106

WAIT 41, **SO**, **68**, **JO**
WHILE 36, **68**
WOBBLE 65
WORLD 58-59
WRIST **60**
WRITE **106**
WRT 20

XCOORD **96**, **102**
XGP 8 9
xhat 18

YCOORD **96**, 102
yhat 18

YJT 100
ypark 25, 55

ZCOORD **96**, 102
ZEROED **60**
zhat 18