Robot Programming Languages: A Study and a Design

Terri A. Noyes

COINS Technical Report 83-22

October 1983

Laboratory for Perceptual Robotics
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

(Communicated by Dr. Michael Arbib)

This research was supported by the National Science Foundation under Grant ECS-8108818.

ACKNOWLEDGEMENTS

I am very grateful to Dr. Michael Arbib, Thea Iberall, Gerry Pocock and Dr. Kenneth Overton for reading drafts of this report and providing invaluable suggestions in their areas of expertise. Dr. Overton also supplied Figure 8.

I would also like to thank Judy Franklin, Daniel Thompson, and Damian Lyons for their help and support.

CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER I

INTRODUCTION

I.1 Description and Intent

This report is intended to familiarize members of the Laboratory for Perceptual Robotics (LPR) at the University of Massachusetts at Amherst with different robot programming languages. It is meant to provide insight into desirable characteristics of robot programming languages (RPL's) in order to facilitate future development of our own software environment. This report examines three of the more advanced languages, each designed for use with a particular system, and extracts the unusual features, good qualities, and shortcomings of each.

Chapters II through IV are dedicated to AL, VAL and AML, respectively. A discussion of robot programming languages without regard to the robots they control would be futile, since anatomically distinct robots have different programming requirements. Therefore, an overview of the corresponding robot systems is given. The intended or most suitable applications of each are discussed. Syntactic characteristics are discussed, including the data structures provided, the conditional and looping constructs provided, and the ability to modularize a program into functional units. A section on manipulation encompasses position feedback, the "teaching" of positions in conjunction with manual control, actual motion control, and motion with sensing.

Chapter V encapsulates the similarities and differences of the three languages. Since FORTH is the low-level language we are presently using, we will examine its features. However, because it is not a robotics language per se, it has not been included in the comparison of the other three languages. Chapter VI is an overview of FORTH's qualities as a general language. Chapter VII discusses the current state of the Laboratory for Perceptual Robotics, the research being done in dynamic sensing and control, the use of FORTH as a base language for our proposed Perceptual Robotics Language, and the design philosophy of this language. Finally, this chapter defines possible elements of our language.

The intention here is not to reproduce the robotics languages in full, and the reader is assumed to have some familiarity with other high-level programming languages.

I.2 Terminology

We distinguish programming "implicitly" from programming "explicitly." Programming *implicitly* entails expressing robot actions with respect to a known and modelled environment, where the model is adjusted to any changes in the environment. A specific position and orientation may be referenced by name. Programming *explicitly* means specifying movement in terms of explicit locations in space [Nevins and Whitney, 1979].

The world coordinate frame (for a non-mobile robot) is a fixed reference cartesian coordinate frame. World coordinates indicate absolute position within a workspace. Joint coordinates specify the position of the manipulator in terms of the joint angles between successive links of an arm relative to the base of the robot. It is difficult to use joint coordinates, which depend on the kinematics of the arm and end effector.

A transformation is a mathematical quantity specifying a translation and a rotation. A transformation can be interpreted as an alternative coordinate frame in which to represent objects, including the manipulator. The position and orientation of the end effector comprise a tool coordinate frame.

Paul [1981] presents a thorough discussion of the mathematical manipulation of transformations and frames.

CHAPTER II

AL

II.1 AL System Overview

II.1.1 The Scheinman Arm

AL, or Automation Language, was developed by a group at the Stanford Artificial Intelligence Laboratory (SAIL) for use with two model Scheinman arms. The Stanford Scheinman arm is a six degree of freedom manipulator, and has 5 revolute joints and one linear joint (Figure 1). The end-effector is a two-fingered, parallel jaw gripper. To control the arm and run the AL system the power of a minicomputer is needed. A PDP-KL10 is used for compilation and loading, and a PDP-11/45 is used for execution.

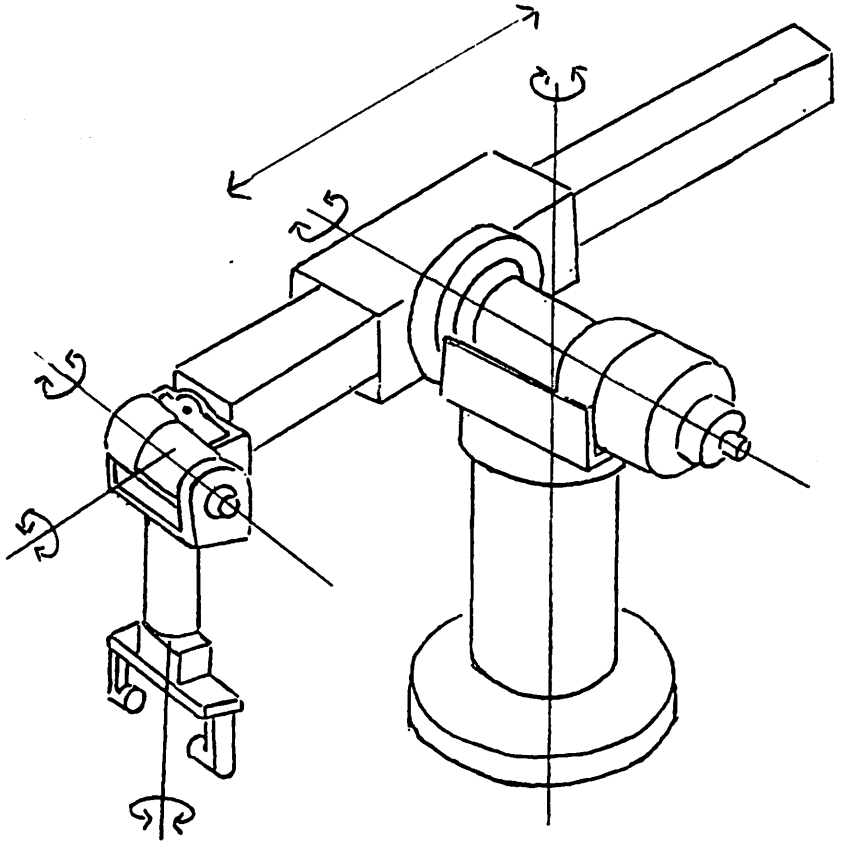

Figure 1. Kinematic Structure of the Stanford Scheinman Arm

II.1.2 General Language Characteristics

The most developed version of AL is a compiled language. During compilation AL uses planned values as a database on which trajectory calculations are made. Trajectories are modified later when the actual execution deviates from the planned execution. This design was chosen because the computational load involved in calculating trajectories was too much for the AL system to handle at runtime. However, now it is realized that with current improvements in hardware and with arm-servo software developments trajectories can be computed in real time. Very recently an interpreter has been built for the AL system.

AL is what is often called an implicit robot language (see Introduction). AL movements are in terms of grasped objects specified by coordinate transformations, and a world model is updated with each object location change.

All movements in AL are joint-interpolated. The joints are moved simultaneously, the speed of each joint being controlled so that all reach their end configuration at the same time. The time it takes for a movement is the maximum time required by any joint, unless a greater duration is specified. Joint-interpolated motion has the disadvantage that the path of the tool tip between points cannot be explicitly controlled, and caution must be used.

The force and torque (or moment) of the arm can be monitored and controlled in the AL system. One can also specify the duration and speed of motions.

AL permits the control of parallel processes by allowing the flow of control of the program to be distributed, 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.

II.1.3 Dynamic Model of the Robot Environment

II.1.3.1 Construction Using POINTY

The POINTY system may be used to aid in the construction of a world model description for AL programs. Use of the POINTY language involves manual control of the manipulator. A deformable pointer attached to the end effector is used to coordinatize features of interest on objects in the environment. (Calibration

techniques are used to acquire the displacement of the end of the tip from the gripper. The pointer is sufficiently rigid so as not to deform under unwarranted circumstances.) The positions and orientations are recorded as frames, and affixment properties (described in the next section) are specified. For further discussion of the POINTY system, see [Goldman and Mujtaba, 1979].

II.1.3.2 Updating the World Model

An unusual feature of AL is its affixment capability. *Affixment* is a means to define relationships between various features of an object and between separate objects that are in contact. Due to the high emphasis the AL system places on the moving and orientation of objects (since it is an implicit language), the *AFFIX* and *UNFIX* statements comprise a very valuable programming construct. They provide a means by which the world model can be updated independently of specific knowledge of each object transformation involved.

One can affix either *RIGIDLY* or *NONRIGIDLY*. With rigid affixment, when either frame changes, the other is changed so that the same relationship exists between them. In the nonrigid case, the relationship between them would be redefined. This is easily understood if one considers a simple case. Nonrigid affixment exists between a cup and a saucer. The cup moves with the saucer but not vice versa.

After an object has been affixed to the manipulator the user can concentrate on where objects are in relation to one another, and can disregard the control of the manipulator. The user can specify the desired position and orientation of the object, and AL will work out what the arm has to do to achieve this goal.

II.2 AL Data Structures

II.2.1 Data Types

AL provides a large number of data types, most of which could be built in other structured languages using their respective primitives, but whose preexistence is a considerable convenience.

The AL type *SCALAR* is a floating point number. However, one has the option to give the scalar a dimension of *time, distance, angle* or *force*. A dimensioned scalar must be explicitly associated with an appropriate dimensional unit (either *seconds, inches, degrees, centimeters, ounces, pounds, grams* or *radians*). This concreteness is due to the planning and location tracking design of AL, and dictates careful programming.

Dimensioned variables are used in the same manner as scalar variables. However, AL checks for conflicting units when performing addition, subtraction and assignment. Multiplication and division operations do not (if there is no assignment) require an exact match. In this way, intermediate results may be of a type not declared. One can use a *DIMENSION* statement to define new dimensions, and use macros to define new dimensional units (such as feet).

AL's *VECTORs*, essentially triples of scalars, may be manipulated using firmware cross product and dot product operations. The vectors *xhat,yhat,zhat*, and *nilvect* are predefined as (1,0,0), (0,1,0), (0,0,1) and (0,0,0) respectively. A *ROTATION* is a direction vector and an angle to indicate the amount of rotation about this vector, in the form of a 3x3 matrix.

A *FRAME* describes the position and orientation of an object. It consists 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.

Arrays of all algebraic data types (*SCALAR, VECTOR, TRANS, ROT, FRAME*) may be defined, with the restriction that any one array must be homogeneous.

There are several predefined constants and variables in AL. In addition to the vectors mentioned above, there are the following:

barm = location of the blue arm
yarm = location of the yellow arm
bhand = distance between fingers of blue arm
yhand = distance between fingers of yellow arm
bpark = rest position of blue arm
ypark = rest position of yellow arm

The first four are automatically updated variables, while *bpark* and *ypark* are constants.

II.2.2 Declaring and Modifying Variables

II.2.2.1 Formats

There is no explicit limit to the length of AL variables. The type of a variable in AL must be stated before it is used. The declaration statement used to define the dimension and type of each variable has the syntax: *<dimension> <data type> <variable-list>*. Array declarations are of the form: *<dimension> <data type> ARRAY <variable-list>*.

AL assignment is done by means of a left arrow, as in *somevar <- someexp.*

II.2.2.2 Examples Using Vectors, Rotations, Frames and Transformations

To construct a vector in the reference coordinate frame which has the same orientation as a vector in some frame, such as *xhat* in say *frame-1*, the *with respect to* operator *WRT* is used and one writes *vect <- xhat WRT frame-1.*

The following few examples illustrate the use of vectors, rotations, frames and transformations:

```
ROT rot-1, rot-2, rot-3;
VECTOR vect-1;
FRAME a-frame, b-frame;
TRANS trans-1, trans-2;

rot-1 <- ROT(zhat, 60 * deg);
rot-2 <- ROT(yhat, 90 * deg);
rot-3 <- rot2 * rot-1;
```

=> A rotation is made of 60 degrees around the z axis and then 90 degrees about the original y. A rotation of 90 degrees about y and then 60 degrees about the new z axis will produce the same result. Any number of rotations can be combined in this manner.

```
a-frame <- FRAME( ROT(zhat, 90 * deg), 3 * yhat * inches);
```

=> *a-frame* is displaced 3 inches from the reference frame of the robot in the Y direction. It's coordinate system is rotated 90 degrees around the Z axis of the reference frame, so that the new X axis points in the direction of the reference Y axis.

```
vect-1 <- a-frame * (zhat * inches);
```

=> this is equal to *VECTOR(0,3,1).*

```
trans-1 <- TRANS( ROT(xhat, 20 * deg), 2 * zhat * inches);
```

*vect-2 <- trans-1 * yhat * inches;*
=> trans-1 rotates yhat 20 degrees about the X axis, and then translates it by 2 inches along Z.

trans-2 <- a-frame -> b-frame
*=> a-frame * trans-2 = b-frame* This computes the transformation which brings one from *a-frame* to *b-frame*.

Any number of transformations may be multiplied. The order of interpretation is the same as for rotations – that is, ordering from right to left signifies performing all operations (rotations and translations) with respect to the reference coordinate frame, while ordering from left to right signifies performing all operations with respect to the newly derived frames.

II.3 AL Program Structure and Non-Motion Control

AL uses ALGOL-like control and block structures. AL programs are organized into *BEGIN-END* blocks (*COBEGIN-COEND* for parallel processing).

II.3.1 Modules: Procedures and Macros

One feature of AL, which is made possible by its compilation phase, is that macros with any number of parameters may be substituted for text. Macros can be constructed to serve the same purpose as a function or a procedure, and save time at execution. They provide the programmer with a good facility for developing coherent English-like task specifications.

An AL macro is defined by *DEFINE <macro-id> <parameters> = <body-of-macro>*. The body of the macro is the text to be substituted whenever *<macro-id>* is used in the program.

AL programs may be modularized by procedures, which have the format:

```
type PROCEDURE name (opt-param-list);
    BEGIN
        body-of-procedure
    END
```

Only if the procedure is to serve as a function and return a result as its value

should a type precede the declaration, and in this case *RETURN value* would be included in the body of the procedure.

Procedure calls may appear anywhere an expression might, and may also be recursive. The implications of this are discussed in Chapter V.

II.3.2 Conditional and Looping Constructs

A critical programming technique is the implementation of conditional expressions to make computational and executional decisions. AL uses *IF..THEN* statements and also supports the *IF..THEN.ELSE* extension.

AL has the conventional *WHILE..DO* and *DO..UNTIL* (*REPEAT..UNTIL*) expressions of other high level languages. A *FOR* loop in AL has the form *FOR svar <- s-exp1 STEP s-exp2 UNTIL s-exp3 DO statement*, where *svar* is some *SCALAR* variable and the s-exp's are scalar expressions of the same dimension. The value of the variable is initially *s-exp1*. It is incremented after each execution by the step *s-exp2*, and looping ends when it exceeds *s-exp3*.

Another convenient structure present in AL is the *CASE* statement. The AL *CASE* statement has two forms :

```
CASE s-exp of
    BEGIN
            statement 0;
            statement 1;
                .
                .
            statement n;
    END
                    and
CASE s-exp OF
    BEGIN
            c0 statement;
            c1 statement;
                .
                .
            cn statement;
            ELSE statement
    END
```

In both versions the scalar expression is evaluated and depending on its integer part

(truncation) one of the n statements is executed. In the first version the value is the number of the statement executed, and if this value is not between 0 and n an error results. Any statement may be null, and may be a series of statements enclosed by a *BEGIN-END* pair. In the second version, the integer value is matched with one of the scalar constant labels (there may be more than one scalar constant labelling each statement) or with the *ELSE* statement if there is no match. When no *ELSE* is used and the index value is not present an error results. While the AL *CASE* statement is convenient (and VAL and AML have no such provision), it is limited in that the indexing and labelling variables may be of type integer only.

II.3.3 Terminating Clauses

AL's *PAUSE* statement, used in the form *PAUSE time*, results in a suspension of execution for the time specified. Two other statements may be used to terminate an executing motion. *STOP dev* stops the device specified, and *ABORT list* stops the motion of all devices and prints out the elements of the (optional) list. This list may contain any number of variables, expressions or character strings.

These statements are useful when doing condition monitoring, which will be discussed in the next section.

II.4 Manipulation

II.4.1 Basic Motion Commands

The basic motion statement in AL has the following form : *MOVE frame TO endpos <modifying clauses>*. *frame* can be one of the system's two manipulators, or a frame affixed to an arm. In the second case the relation between the frame and the arm given by the affixment chain connecting them will be used so the motion results in a move to *endpos*. Various clauses may be used to influence a motion. Differential or relative motions may be specified by using a grinch sign (*x*), which represents the current position of the manipulator, in a statement such as *MOVE arm TO x - 2 * zhat * inches;*. The execution of this statement causes the specified arm to be displaced two inches from its current position along the negative z axis.

The addition of a duration clause to the basic *MOVE* statement, *WITH DURATION = 4 * seconds* for example, will cause the motion to be performed over a time interval of 4 seconds. When performing a multiple segment move the intermediate frames may be specified by means of a *VIA* clause, such as *VIA frame-1, frame-2 .. frame-n*. Velocity at an intermediate point and the duration from the previous point to this point may also be specified – i.e., *VIA frame WHERE VELOCITY = v, DURATION = n*. In this instance only one frame may be specified, and one of the modifying clauses may be absent and they may be in any order. *v* must be a velocity vector and *n* a time scalar.

One can also specify *deproach* and *approach* points in AL. These points are associated with the trajectory for the departure of the arm from the current position or its approach to a destination location. Unlike *VIA*, these points are in relation to the initial or the destination coordinate frames. The clauses are *WITH DEPARTURE = exp* and *WITH APPROACH = exp*. *exp* may be of type frame, vector or scalar. If of type frame the deproach point is specified by *fr*exp*, where *fr* is the destination frame if using an *APPROACH* clause and the current position if using a *DEPARTURE* clause. Similarly, if *exp* is a vector the deproach point is specified by *fr + exp WRT fr*, and if scalar it is given by *fr + (exp*zhat) WRT fr*, where *zhat* is the component along the z axis. When a clause is replaced by the predeclared macro *DIRECTLY*, the use of a deproach point is suppressed.

II.4.2 Condition Monitoring and Compliance

In AL it is possible to control torques and forces. To avoid incompatible requests the force components must be perpendicular. A force frame must be specified, and the directions of the applied forces and moments must be aligned with one of the axes of this current force coordinate system. One must specify whether the force frame is defined relative to the tool or world coordinate system. Consider the following clauses:

WITH FORCE = sval ALONG axis-vector OF frame IN coord-sys
WITH TORQUE = sval ABOUT axis-vector OF frame IN coord-sys
and
WITH FORCE-FRAME = frame IN coord-sys
WITH FORCE = sval ALONG axis-vector
WITH TORQUE = sval ABOUT axis-vector

and
WITH FORCE-FRAME = frame IN coord-sys
WITH FORCE(axis-vector) = sval
WITH TORQUE(axis-vector) = sval

where: *axis-vector* = *xhat*, *yhat* or *zhat* as defined previously
coord-sys = *HAND* or *WORLD* (*WORLD* is default)
sval = the magnitude of the force
frame = the orientation of the axes of the force frame

In the first group the force frame in all of the clauses must be the same. Only one force frame may be specified per move.

Force sensing, events, duration and various boolean expressions of variables can be explicitly monitored by AL. The general statement for this is *ON condition DO action*, where *action* is any statement or block. When a motion starts monitoring begins, and ceases when the motion does. If triggered the monitor is disabled, and remains disabled unless *ENABLE monitor* is added to the condition monitor clause. Similarly, *DISABLE monitor* may be used. When a *DEFER* directly precedes a condition monitor, the monitor is initially disabled. A sample of code in which a condition monitor is initially disabled, and then after three seconds is enabled is:

```
MOVE barm TO end-loc
  test: DEFER ON FORCE(zhat) > 10 * oz DO STOP
     ON DURATION > 3 * sec DO ENABLE test
```
[Goldman and Mujtaba, p. 58]

II.4.3 Gripper Control

The gripper can be controlled in three ways. *OPEN hand TO sval* and *CLOSE hand TO sval* moves the gripper to the opening *sval*. *CENTER* arm causes the gripper sides to move in slowly until one of the touch sensors triggers, indicating contact with the object has been made. At this point the arm will move till both touch sensors are activated while maintaining the position of the side that is in contact. Then the new position can be read to determine the location of the object.

II.4.4 Simultaneous Motion in AL

The primary purpose of the *COBEGIN-COEND* block in AL is to enable coordinated motion of more than one manipulator (two are used at Stanford). Also, computations may be performed when moving an arm, to save time. Statements enclosed by a *COBEGIN-COEND* share program control. The programmer must make certain that the processes being executed in parallel do not conflict, the most obvious case of this being when they access the same arm at the same time.

Parallel processes may be scheduled and coordinated using the *SIGNAL* and *WAIT* statements as follows. A count is kept of how many times each process has been signalled. *SIGNAL event* increments the count for the event, and if the new count is less than or equal to 0 a process waiting for the event to occur is freed for execution. *WAIT event* decrements the count associated with the event, and if the resulting count is negative the process issuing the wait is held until another process signals the event. If the count is 0 or positive there is no wait. Consider the following simple example:

```
BEGIN
 EVENT ready-receive, dropped;
 FRAME object, drop-position, mug, catch-position, dest;
 COBEGIN
   BEGIN
      MOVE yarm TO object;
      CENTER yarm;                    (yarm grasps object)
      AFFIX object TO yarm;
      MOVE object TO drop-position;   (takes object to release position)
      WAIT ready-receive;             (waits until barm in catch position)
      OPEN yhand TO 3.0*inches;       (releases object)
      SIGNAL dropped;                 (signals object has been dropped)
    END
   BEGIN
      MOVE barm TO mug;
      CENTER barm;                    (barm grasps mug)
      AFFIX mug TO barm;
      MOVE mug TO catch-position;     (moves mug to catch position)
      SIGNAL ready-receive;           (signals ready)
      WAIT dropped;                   (waits until object dropped)
      MOVE barm TO dest;
   END;
 COEND;
END;
```

II.5 Summary

The AL language has compiler-type versions and a fairly recent interpreter version. AL is a robust language. We can look towards AL for its parallel processing features, English language-like form, its abundance of data types and non-motion control constructs, and its monitoring capabilities. AL provides a large number of data types, most of which could be built in other structured languages using their respective primitives.

AL movements are in terms of grasped objects specified by coordinate transformations, and a world model is updated with each object location change. An unusual feature of AL is its affixment capability. Affixment is a means to define relationships between various features of an object and between separate objects that are in contact. Due to the high emphasis the AL system places on the moving and orientation of objects, the *AFFIX* and *UNFIX* statements comprise a very valuable programming construct. They provide a means by which the world model can be updated independent of specific knowledge of each object transformation involved.

AL is structured to enable coordinated motion of more than one manipulator (two are used at Stanford). Computations may be performed in parallel with manipulation also to save time. Statements enclosed by a *COBEGIN-COEND* share program control. The programmer must make certain that the processes being executed in parallel do not conflict, the most obvious case of this being when they access the same arm at the same time. Parallel processes may be scheduled and coordinated using general semaphore-type *SIGNAL* and *WAIT* commands.

AL is one of the few robot languages in which one can specify compliant moves. The force and torque (or moment) of the arm can be monitored and controlled in the AL system. One can also specify the duration and speed of motions.

Bolles et al. [1974] present specifications for the initial design of the AL language, together with the reasoning involved when making design choices. The AL user's manual should be consulted for a more complete syntactic description [Goldman and Mujtaba, 1979].

CHAPTER III

VAL

III.1 VAL System Overview

III.1.1 Unimation Puma Robot Series

The VAL language was designed for a revolute arm, the Unimation PUMA robot series. The robots have 6 rotational axes. A prototype is shown in Figure 2. VAL may be run on a microcomputer or small minicomputer.

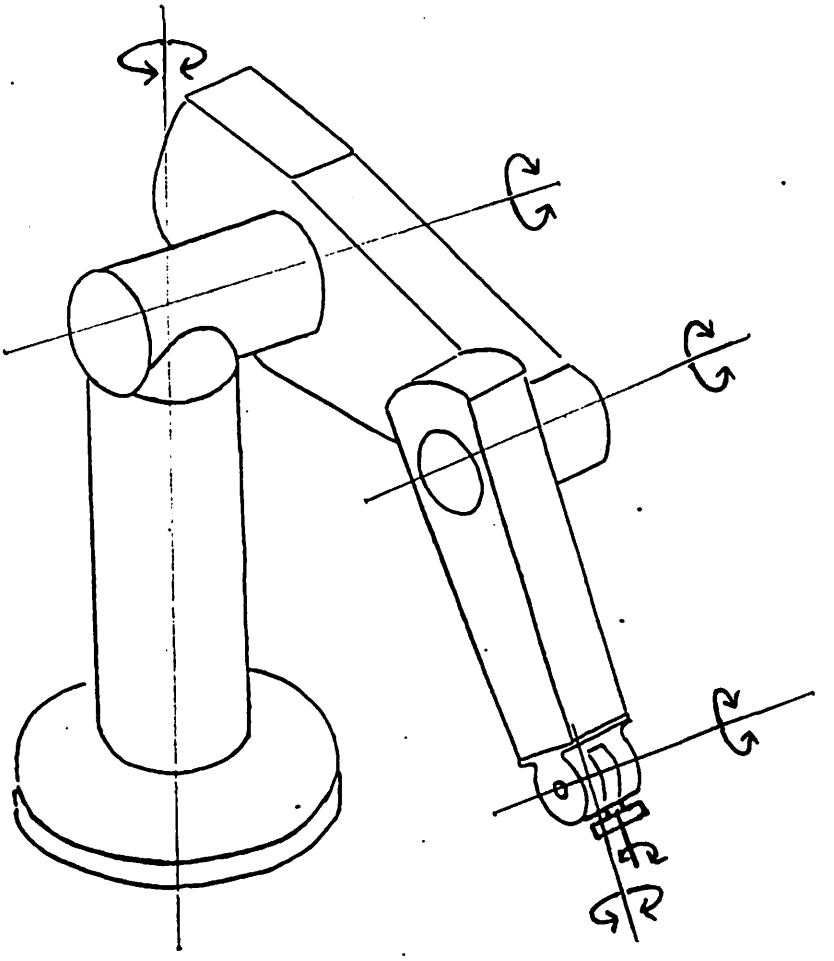

Figure 2. Kinematic Structure of the Unimation Puma 600 Robot

III.1.2 General Language Characteristics

VAL is an interpretive language and allows for much user interaction during program execution. One can write to or read from the disk, edit user programs, and display, define and modify variables while an application is running.

The VAL system has two schemes for controlling a path between two points. In addition to the joint-interpolated motion provided by the AL system, VAL also allows for straight line paths. VAL enables multiple segment trajectories in the same manner as the AL system. Unlike the AL system, VAL does not allow a user to specify the parameters of a trajectory such as speed or time, except in the case of a single joint move. VAL offers no means of performing compliant motion.

III.2 Data Structures

III.2.1 Data Types

In addition to an integer variable, VAL supports three types of location variables -- transformations, compound transformations and precision points. A precision point is a composite of all of the robot's joint variables. A transformation is a composite of the location and orientation in space of the robot with respect to a zero location, in the form *X, Y, Z, O, A, T*. *X, Y* and *Z* specify the position of a point centrally located between the fingers of the gripper in world coordinates, and *O, A* and *T* specify the orientation of the hand (in degrees).

Compound transformations allow one to specify the location and orientation of the end effector relative to the location and orientation specified by other transformations. For instance, if *CUP* is the name of a transformation specifying the location of a cup relative to the reference frame of the robot, and *OBJECT* is the relative transformation for the location of the object relative to the cup, then *CUP:OBJECT* defines the location of the object relative to the reference frame of the robot. It is possible to string together several relative transformations in this manner. Hence, in the above instance, if the object were to be grasped at a point *GRASP* (defined relative to the location of *OBJECT*), the statement *MOVE CUP:OBJECT:GRASP* moves the gripper to the grasp position.

III.2.2 Declaring and Modifying Variables

VAL location variables and integer variables are declared upon their first assignment within the program body, and are limited to 6 characters. An integer assignment is performed by the instruction *SETI intvar-1 = intvar-2 op intvar-3*, where the last two terms are optional.

There are several ways to define locations in VAL. *SET trans-1 = trans-2 [trans-3]..[trans-n]* and *SET prec-point-1 = prec-point-2* are two basic methods. The command *POINT locvar1 = locvar2* sets location variable 1 to the value of location variable 2, where, as discussed above, a location variable can be a precision point, a transformation, or a compound transformation. If the second argument is not included the value is not changed unless it has not been defined, in which case it gets set to a default value. *DPOINT locvar-1, locvar-2 .. locvar-n* deletes some number of location variables, though not compound transformations. *HERE locvar* defines the value of a location variable to be equal to the current robot location.

The VAL instruction *SHIFT trans BY dx,dy,dz* modifies the x, y and z components (all need not be specified) of the indicated transformation. *TOOL transformation* sets the value of the tool transformation to *transformation*. *INVERSE trans-1 = trans-2[:trans-3 ... :trans-n]* sets the value of *trans-1* to the matrix inverse of the right hand side. *FRAME trans-1 = trans-2, trans-3, trans-4* assigns the value to *trans-1* which describes the relationship of the second frame to that of the robot. This instruction is most often used to define a 'base' transformation for relative locations.

III.3 VAL Program Structure and Non-Motion Control

III.3.1 Modules

In VAL, a program is merely a sequence of instructions stored in a file. There is no program header and the name of the program is the file name. A VAL "subroutine" is such a program. VAL allows for up to ten nested program calls but *no passing of parameters*. The *RETURN* statement returns control to the calling program.

A VAL subroutine program is called using the statement *GOSUB program*. Within the subroutine a *RETURN skip-count* instruction must be executed to return control to the calling program. The *skip-count* signifies that execution is to be continued at (*skip-count* + 1) program steps following the expression which invoked the subroutine.

III.3.2 Conditional and Looping Constructs

VAL supports the *IF..THEN* conditional. VAL programs branch solely by way of the (primitive) *GOTO* statement, whose existence enhances the probability of unstructured, hard to follow code, but when used properly is sufficient.

III.3.3 Terminating Clauses

The execution of a VAL program is terminated by any one of three instructions. *PAUSE message* terminates a program, displays the message (optional) and allows execution to be continued if the user enters *PROCEED*.

The VAL *HALT message* unconditionally terminates program execution, and displays any specified message, while *STOP message* will terminate execution unless the program is set up for more than one run (as determined by the *EXECUTE* command) in which case the program starts again at the beginning.

III.4 Manipulation

III.4.1 Position Feedback

The *WHERE* command in VAL displays the current robot location in Cartesian base or world coordinates, in joint variables, and also displays the current hand location. *HERE locvar* defines the value of the location variable to be equal to the current robot location, in joint variables if the location variable is a precision point, and in world coordinates if it is a transformation.

III.4.2 Guiding

A series of positions may be recorded with a manual control unit by use of the *TEACH* command. Then each time the *RECORD* button on the unit is pressed, the value of a location variable is set to the position of the robot at that instant. The location variable names created have consecutive subscripts. For instance, the command *TEACH POS1* would cause the first recorded position to be stored in *POS1*, the second in *POS2*, and so on until the *RETURN* button on the manual control unit was pressed, terminating the session.

III.4.3 Motion Control: VAL Motion Commands

MOVE location moves the robot to the location and orientation specified by *location*. If the location name is followed by an exclamation point its value is set (while the instruction is being entered) to the robot position and orientation at the instant the *MOVE* instruction is completed by a *<CR>*. This calls for a joint-interpolated motion, as does *MOVET location, hand-opening*. With the *MOVET* instruction the hand opening is changed during the motion to *hand-opening* millimeters. The two instructions for straight-line motion are *MOVES location* and *MOVEST location, hand-opening*. In each instance the end effector is moved along a straight path and smoothly rotated to its final orientation. The *MOVET* instruction, like the *MOVE* command, has a set option indicated by an exclamation point.

DRAW dx,dy,dz moves the end effector along a straight line the specified offsets from the current position.

APPRO location, distance moves the end effector to a displacement *distance* along the tool z axis from the specified point. This instruction uses joint-interpolated motion and has the location set option described for the *MOVE* command.

DEPAR distance moves the end effector the distance specifed along the current z axis by joint-interpolation. *DEPARTS distance* performs the same function using straight-line motion.

The VAL command *DRIVE jt, change, speed* operates a single joint, changing the joint variable by *change* units.

VAL uses other motion commands to control its hand position. *OPEN hand-opening* and *CLOSE hand-opening* open or close the hand during the next motion sequence. The commands *OPENI hand-opening* and *CLOSEI hand-opening* open or close the hand immediately. *GRASP hand-opening, label* checks to see if the final opening is less than the specified amount. If so, the program branches to *label*.

III.5 Summary

VAL is typical of the earlier robotics languages, which tend to concentrate on manipulation and ignore the problems of data processing. VAL is very unstructured and hence it is difficult to write programs of any size in it.

VAL is an interpretive language, and thus statements are executed line by line. There are several advantages to this. A monitor command may be used to execute a single instruction. Thus, single moves can be made without having to execute a one-step program. While a VAL program is running the user can display, define and modify variables.

VAL has only one non-motion program control statement other than its *GOTO* and various terminating clauses. This in itself severely undermines VAL if we consider the several constructs offered by AL, AML and other structured programming languages.

A thorough syntactic description of the VAL language is given in the VAL user's guide [Unimation, 1980].

CHAPTER IV

AML

IV.1 AML System Overview

IV.1.1 The IBM Robot System

A Manipulator Language (AML) is used to operate the IBM Robot System 7565 manipulators. The robots have three linear joints, three revolute joints, and a gripper. The general configuration is sketched in Figure 3. The names of the joints on the IBM robots are *JX* (x motion), *JY* (y motion), *JZ* (z motion), *JR* (roll joint), *JP* (pitch joint), *JW* (yaw joint) and *JG* (gripper). The aggregate *ARM* is a conveniently predefined composite of the six joints.

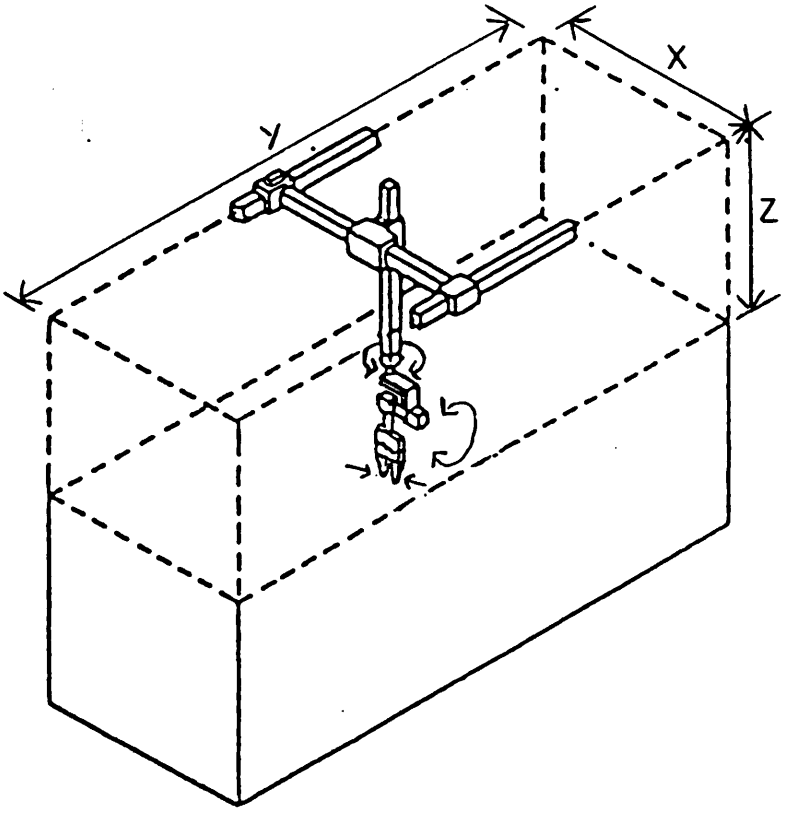

Figure 3. Kinematic Structure of the IBM 7565 Manipulator

IV.1.2 General Language Characteristics

AML is an interpretive language, and thus statements are executed line by line. There are several advantages to this. A monitor command may be used to execute a single instruction. Thus, single moves can be made without having to execute a one-step program. An executing AML program must be interrupted to enter commands interactively.

AML movements are specified in terms of explicit locations in space, and as yet very little has been attempted with world modelling.

To move the arm one specifies the joints to be moved and the displacements in inches or degrees, respectively. When a multiple joint move is specified, joint-interpolated motion occurs. One can control the rate of acceleration, speed and deceleration if desired. The IBM system is typically run on a minicomputer or microcomputer.

IV.1.3 External Device Interface

AML and the IBM system have extensive sensory capabilities. An interface is provided so that sensory equipment for a particular application may be connected in addition to the 6 built in force sensors or strain gauges in the gripper. When developing an application, if the hardware is not yet available it can be simulated by buttons on the control box. User defined sensors are treated in the same manner as built in system sensors.

DEFIO defines logical sensors and drivers as subfields of 16-bit input and output hardware registers in the system controller and returns integers that may be used to refer to the entities defined. One bit input might be used for determining whether a part is in position, or whether a tool is on or off. One bit output could be used to turn the tool on or off. For more sophisticated tasks, multiple bit input or output (up to 16 bits) can be used, for instance to count widgets or access digital encoders.

DEFIO is used as:

sensornum = DEFIO(iogroup,iotype,format,bitno,nbits,scale, offset)

GIO#	AML NAME	DESCRIPTION	S/1 ADDRESS
1		X feedback	AI point 0
2		Y feedback	AI point 1
3		Z feedback	AI point 2
4		Roll feedback	AI point 3
5		Pitch feedback	AI point 4
6		Yaw feedback	AI point 5
7		Gripper feedback	AI point 6
8		Right tip s.g.	AI point 7
9		Right pinch s.g.	AI point 8
10		Right side s.g.	AI point 9
11		Left tip s.g.	AI point 10
12		Left pinch s.g.	AI point 11
13		Left side s.g.	AI point 12
14		System reserved	AI point 13
15		system reserved	AI point 14
16		System reserved	AI point 15
17		Gripper LED	67
18		Pendant LEDs	4B
19	PB1	Pendant buttons	48
20	PB2	Pendant buttons	49
21		User DI	6C
22		User DI	6D
23		User DI	4C
24		User DI	4D
25		User DO	6E
26		User DO	6F
27		User DO	4E
28		User DO	4F

Figure 4. Group I/O

LIO#	AML NAME	DESCRIPTION	GIO POINTER
1000	PLEDS	Pendant LEDs	18
1001	LED	Gripper LED	17
1002	SRT	Right tip s.g.	8
1003	SRP	Right pinch s.g.	9
1004	SRS	Right Side s.g.	10
1005	SLT	Left tip s.g.	11
1006	SLP	Left pinch s.g.	12
1007	SLS	Left side s.g.	13
1008		System reserved	0
1009		System reserved	0
1010		System reserved	0
1011		System reserved	0
1012		System reserved	0
1013		System reserved	0
1014		System reserved	0
1015		System reserved	0
1016		User defined	User defined
1017		User defined	User defined
.		.	.
.		.	.
.		.	.
1079		User defined	User defined

Figure 5. Logical I/O

The parameters are:

iogroup = a group i/o number (Figure 4)
iotype = input (0) or ouput (1)
format = option to treat fields as two's complement or unsigned (1 or 0)
bitno = bit number of field
nbits = field is n bits long
scale = scale number of degrees or radians
offset = offset from scale

DELIO(sensornum) voids a *DEFIO* reference. The user can optionally define scale and offset factors that may be used to convert between hardware values (which are all integers) and floating point numbers corresponding to engineering units. These defined bits may be accessed by *SENSIO*. *SENSIO* is used to perform the input and output of defined values, and is used as *valueset* = *SENSIO(sensorset, type)*. The parameter *sensorset* is the name of a specific *DEFIO*, and *type* specifies a datatype or value to be used. For instance, suppose you want to simulate a feeder. First select a button on the pendant box. Then use a *DEFIO* to define that button as the feeder

input like *DEFIO(19,0,1,0,1)*. The 19 signifies pendant button group 19 (Figure 4), 0 specifies input, 1 specifies a two's complement representation, the next 0 specifies we are reading in bit 0, and 1 specifies the field is 1 bit long. Suppose the *DEFIO* returns 1016 for a label (Figure 5). Then *SENSIO(1016,0)* will read the appropriate pendant button, and a 0 returned means the button was not pressed. To simulate the feeder you would press button 1 on the pendant box and watch the application *fill* the feeder. Sensor output is also easily accomplished. Further discussion of the external device interface is beyond the scope of this report.

Sensors can be monitored so as to trigger the execution of a subroutine, or to terminate a robot motion. This will be discussed in more detail in section IV.4.3.2 (Sensor Monitoring).

IV.1.4 Gripper Sensing Capabilities

The sensory devices on the gripper provide for force feedback and object location. Forces may be detected by the use of the strain gauges in the tips, insides and outsides of the fingers.

If the fingertips push against an object the tip forces (*SLT,SRT*) increase. If a weight is lifted the forces decrease, and the weight of the held object roughly equals the absolute value of their sum. Weighing objects as they are moved could be quite useful for a sorting process in an application.

When the gripper is closed on an object the readings for the pinch strain gauges (*SLP,SRP*) increase, and if the gripper is opened the readings become smaller. If the gripper comes into contact with an object "behind" it the side readings (*SLS,SRS*) increase, and if the gripper is moved into an object in front of it the side readings decrease. There does exist slight cross-coupling between strain gauges, and in certain instances it is desirable to measure the effects and compensate.

An LED sensor across the fingers of the gripper may be monitored to detect the presence or absence of an object. When the beam is interrupted the sensor gives a reading of one, otherwise it returns 0. The LED is often used in calibration for precise movements such as required in assembly, and IBM has developed a package of calibration routines.

IV.2 Data Structures

IV.2.1 Data Types

AML supports the data type *aggregate*. An aggregate, like an array, is a data set in which each individual component is referenced by an integer subscript. Unlike the conventional array, however, an aggregate may have elements of different data types, like a Pascal record.

In AML there are three basic variable types, *INTEGER*, *REAL* and *STRING*. A *STRING* represents character strings enclosed in single quotes.

IV.2.2 Declaring and Modifying Variables

When declaring a variable in AML one has the option of simultaneously assigning it an initial value, thus saving one assignment statement per declaration. This value may be in the form of a constant expression, or an expression comprised of previously defined variables. The data type of the expressed value determines the data type of the variable. All variables must be declared before they can be referenced in an expression.

AML variables can be declared with one of the two keywords *NEW* or *STATIC*. In the main program, the treatment of variables based on either designation is the same. If declared within a subroutine, however, the difference is important. In this instance a *NEW* variable is defined upon each entry and the storage for it is cleared upon each exit. Large variables or aggregates with several elements should be declared as *NEW* whenever possible. A *STATIC* variable is declared upon the first call to the subroutine and its value is not destroyed when the routine is exited, although the name of the variable is detached from its allocated storage. The last value saved will be available from one subroutine call to the next. This alleviates having to pass the variable as a parameter, and keeps it local to the subroutine in which it is used. For the examples presented in this section *STATIC* and *NEW* should be considered interchangeable.

A string variable is declared as follows. To assign it an initial value, a declaration such as *NAME: NEW STRING 'SOMETHING'* is used, where in this case *NAME* would be defined as a string of length 9. To similarly declare this string without giving it the initial value we would use *NAME: NEW STRING(9).*

A variable declared with *VAR: NEW 5.0* would be of type *REAL*, and one declared with *VAR: NEW 5* would be of type *INTEGER*. To define but not initialize the variables one would use *VAR: NEW REAL* or *VAR: NEW INT* respectively.

The equal sign serves as the assignment symbol in AML in the conventional manner.

IV.3 AML Program Structure and Non-Motion Control

IV.3.1 Modules

AML allows for subroutine modules. The names of AML system subroutines are reserved words and may not be redefined. The format for an AML subroutine declaration is :

```
name: SUBR(opt-param-list);
  body-of-subroutine
END;
```

No distinction is made between a call to a user subroutine and a system subroutine (i.e., command) in AML. Therefore, with any extensions to AML (by way of added subroutine modules) the regular command syntax is preserved. One can access a subroutine from any point in the program or another subroutine as long as its declaration precedes the call. A standard call is made by specifying the subroutine name and any parameters (optional), in a form *name(param-1, param-2, .. param-n).* Subroutines can serve as functions by including a *RETURN* statement within, which returns a value to the caller. For instance, the call could be in a variable declaration such as *AVAR : NEW ASUB;*, or in an assignment such as *AVAR = ASUB;*, where the body of *ASUB* would contain a *RETURN(val).* Variables are accessible in the subroutine in which they are declared and in all subroutines called by that subroutine.

Normally when a subroutine is passed a parameter in AML the parameter is a copy of the variable in the calling routine, so that the value is not changed in the outside routine. In order to extend the effects of a parameter change in the called routine to the calling routine, the parameter must be passed by reference, which entails concatenating an exclamation point to the passed parameter. In this manner a pointer to the storage location of the variable is passed rather than a copy of its value, and because pointers use less storage this is more efficient.

The subroutines of AML may be recursive.

IV.3.2 Conditional and Looping Constructs

AML supports the *IF..THEN* statement and the *IF..THEN..ELSE* extension provided by most structured high level programming languages. AML has the conventional *WHILE..DO* and *REPEAT..UNTIL* expressions. In AML one can alter variables within a *WHILE..DO* condition. A simple *WHILE (I=I+1) LE NUM DO* (*LE* for 'less than or equal to') serves as a *FOR* loop. *WHILE (I=I-1) GE NUM DO* (*GE* for 'greater than or equal to') is a reverse *FOR* loop in the sense that one can count from an initial value *down to* a lesser value. Any positive or negative step value may be constructed.

IV.3.3 Terminating Clauses

The AML *BREAK* system subroutine, in the format *BREAK(data-items)*, causes program execution to be suspended and the data items to be displayed. During the suspension one can enter any AML subroutine from the keyboard and can set any values. To continue the program, *RETURN;* is entered.

IV.4 Manipulation

IV.4.1 Position Feedback

AML provides two ways to determine the position of the manipulator. *QGOAL* is a function which returns the last joint destination issued, and *QPOSITION* returns the actual position of the manipulator. *QPOSITION* should not be used in order to

repeat an action because each move produces some error, especially if the moving of objects is involved, and it would be desirable to move back to the commanded position so that the error would not build up over several iterations.

IV.4.2 Guiding

AML's *GUIDE* system routine allows you to move the joints of the robot manually with the control or pendant box. *GUIDE* returns the last commanded position when finished. One can easily record a series of locations in an array variable by calling *GUIDE* within a *WHILE* loop.

IV.4.3 AML Motion Control

IV.4.3.1 Basic Motion Commands

The basic motion statements of AML are *MOVE(joint, location)*, *MOVE(<joint-1,joint-2,...,joint-n>, <loc-1,loc-2,...,loc-n>)* or alternatively *MOVE(<joint-1, joint-2,...,joint-n>, common-loc)* . The move statement should be used in every instance where the location being moved to is known, or if that position can be calculated from a base point -- i.e., *MOVE(joint, QGOAL(joint) + displacement)*. As discussed earlier a multiple-joint move will result in a joint-interpolated motion, in which the case the joint with the furthest distance to travel from its current location becomes the "controlling" joint.

The non-trivial *MOVE* statement has the form *MOVE(joints,goals, monitors,<speed,acceleration,deceleration,settle>)*. The user can control each phase of a trajectory through parameters of the motion statement, or optionally by using the global commands *SPEED, ACCEL* and *DECEL*. If settle is turned off (specified as 0) the *MOVE* will not wait for the manipulator to settle so that it will just be in the general vicinity of the goal. I will refrain from discussing motion with sensory monitoring until the next section, where this will be discussed in detail.

The *DMOVE* command moves the joints an offset from their current positions, and has the same general form as the *MOVE* instruction. This is useful if it is desired to move a few inches away from a position for clearance. The *AMOVE* instruction can be used to speed up a motion sequence. *AMOVE* and *MOVE* use the

same type of trajectory, but instead of waiting for a move to complete, *AMOVE* returns so that the program can do other things while the motion finishes. One example is to use *AMOVE* when points are stored in a file. The file can be read, an *AMOVE* can be executed, and the next data point read. While the file is being accessed the previous motion is being executed. If another *AMOVE* or *MOVE* is issued before the first is finished, the system waits.

Two other useful motion statements exist. *STOPMOVE* is used to stop a motion already in progress. Consider the following:

```
SMOVE: SUBR;
   PARTEX : NEW STOPMOVE;
   BREAK( 'SUSPENSION OF MOVE',EOL
        'ARM AT ',QPOSITION(ARM),EOL);
   APPLY($AMOVE,PARTEX);
END;
```

A monitor key could be tied to the above subroutine, so that pressing this key would result in suspension of any move executing until the monitor RETURN key was pressed.

STOPMOVE is like *AMOVE* in that it does not wait for the stopping process to complete. If this is desired, it should be followed by a *WAITMOVE*.

IV.4.3.2 Sensor Monitoring

Motions can be influenced by sensory feedback using the *MONITOR* system subroutine. *MONITOR* is used to initiate the sensory feedback and link motions or subroutines to a monitor. *MONITOR* initiates reading of a specific sensor or set of sensors at regular time intervals and returns a small integer or set of integers labelling the monitors. A test condition triggers the monitor when a value is outside user-specified limits. When the sensor condition specified occurs during a motion linked to the *MONITOR*, the motion terminates and is considered complete.

Three test conditions are available when monitoring sensors. '1' will trigger a monitor if the sensor value is not within a given range, and '2' does so if the value is within the range specified. Test condition '3' determines whether or not the sensor value is within the limits specified relative to the initial value at the onset of monitoring. If a specified condition is met, any subroutine that is linked to a

monitor will be executed, or any motion linked to a monitor will be terminated. Consider the following simple case:

ZTOUCH: NEW MONITOR (<SLT,SRT>,1,0,10,'ADJUST');

MOVE(<JX,JY>,<X(1),Y(1)>,ZTOUCH);

The *MOVE* will be terminated and subroutine *ADJUST* executed when the monitor *ZTOUCH* is triggered, or when the tip forces are not within the limits specified (roughly 0 to 10 grams).

The *ENDMONITOR* subroutine both stops the monitoring of a specific set of monitors and clears the monitor indicator. *QMONITOR* returns -1 if a monitor has been triggered and 0 if it is undefined or has not been activated. To reset a monitor, or reenable it after it has been activated, *REMONITOR* is used. Up to 16 monitors may be accessed at one time by the four routines.

IV.5 Summary

AML's greatest advantage is its ability to interact with external devices and sensors. It is a well-structured, interpretive language and provides many of the control constructs offered by high-level programming languages.

To move the arm one specifies the joints to be moved and the displacements in inches or degrees, depending on the type of joint. When a multiple joint move is specified, joint-interpolated motion occurs. One can control the rate of acceleration, speed and deceleration if desired.

AML and the IBM system have extensive sensory capabilities. An interface is provided so that sensory equipment for a particular application may be connected in addition to the 6 built in force sensors or strain gauges in the gripper. In developing an application, hardware which is not yet available may be simulated using switches on the control box. User-defined sensors are treated in the same manner as built-in system sensors. Sensors can be monitored so as to trigger the execution of a subroutine, or to terminate a robot motion.

Meyer, Summers and Taylor [1982] describe important features of AML and discuss the considerations involved in its design. Example code for various applications is illustrated. The AML user's guide offers an explicit description of the language and several programming examples [IBM, 1981].

CHAPTER V

COMPARISONS

V.1 Interpreter vs. Compiler

VAL and AML are interpretive languages. An interpreter has just recently been built for the AL system, although all previous versions of AL were compiler languages.

Compiled programs execute faster than interpreted programs, but the difference may be negligible even when aiming for real time response, as in most robotics applications. The response time is more significantly dependent on the capacity of the manipulator and the complexity of the manipulation.

One asset of compiled languages is that compilers provide for more thorough syntactic error-checking, and errors may be detected in the compilation stage before a program is run. However, interpreted languages provide on-line error detection and usually on-line correction. For instance, VAL is very much an interactive language. One can write to or read from the disk, edit user programs, and display, define and modify variables while a program is executing. An executing AML program must be interrupted and suspended to do these same things, but IBM has provided a very good facility for debugging and "tracing" user programs.

V.2 Implicit vs. Explicit

As was discussed in the introduction, an *implicit* language has world-modelling conventions, whereas in an *explicit* language any world-modelling is programmed by the user. AL is a highly implicit robot language, and different facets of VAL may be considered implicit or explicit. VAL does not provide a way to generate a world model directly (like the POINTY extension of AL), and provides no means to automatically update the model (affixment of frames). VAL does have coordinate frame structures as a means to store positions and orientations of objects. The latest release of the AML language provides for frame representations and homogeneous transforms.

Because of their complexity, implicit languages require much more computing power than explicit languages. One major consideration in choosing or developing a language is how much complexity is necessary or desired. This is dependent on the application. For simple pick and place or playback operations an explicit language is all that is needed. The modelling problem is examined more closely in the next section.

V.2.1 Models of the Environment

Proper use of the frames and transforms of AL is advantageous but may be more than an unfamiliar user wishes to deal with, especially when programming a simple task. However, for any complex operation some kind of implicit model is fundamental. For instance, the visual and manipulative task of recognizing an obstacle and avoiding it by taking into consideration the size, shape and orientation of an object being moved or of the manipulator itself requires an internal model.

This kind of shape and space representation calls for a geometric model of some sort. Several of the newer robot languages have basic frame representations of objects, but do not have sophisticated obstacle avoidance mechanisms which use the internal information. One of the problems with using a model is that there are significant tradeoffs between storage, speed, and usefulness. One well-known path-planning scheme uses an approximate model (partitions the workspace, finds a sequence of partitions (free of obstacles) and traces the shortest path through them) [Lozano-Perez, 1980 and Brooks and Lozano-Perez, 1982].

One alternative which is being strived for in research is to develop more sophisticated sensing and reasoning systems which alleviate the need for more than a small information base, as opposed to the complex geometric data base required to represent a 3-dimensional environment without such continual sensing.

V.2.2 Hierarchical Control Schemes

Implicit programming has other merits. For one, it goes hand in hand with the development of a hierarchical control scheme. AL's world modelling and object-oriented nature are useful for defining levels of task specifications.

The ease of dividing a complex task into simpler subtasks is an important consideration in robotics language design. Control should be built up from primitives, and the transition between one state of a task and another should be facilitated by the language. (This is not to say that parallelism is not important -- both parallelism and serialism are appropriate in different instances. For further discussion of distributed control, see section VII.2.1.)

AL is the best of the three languages for task specifications in part because of its object oriented design. Its affixment capability is a very powerful mechanism. In addition, AL statements read very much like English sentences, and macros allow AL programmers to compound this factor.

Any "potentially intelligent" robot system, in which decisions are made dynamically in tune with a changing environment, must clearly be implicit in some sense.

V.3 General Language Capabilities

V.3.1 Recursion

The subroutines of AML and AL may be recursive. This cannot be considered a detriment in any way, although the use of recursive routines is controversial in specific instances. For instance, recursive subroutines require more run time and more memory than their non-recursive equivalents. A complete set of storage locations is allocated each time a procedure invokes itself, and entering and leaving a subroutine uses up time. Thus, in an application geared towards real time response, recursion should be used reasonably.

There are certain processes which are recursive by nature. One instance is that of path finding and obstacle avoidance algorithms. In such a problem an inner map might be constructed, and the key to this process is finding a path from any point to a desired location. At the uppermost level, the starting point in question would be the original position of the manipulator, and the self-nesting would be terminated when the starting point was the final destination. Recursion simplifies this process considerably.

V.3.2 Conditionals

A critical programming technique is the implementation of conditional expressions to make computational and executional decisions. All three of AML, AL and VAL use *IF..THEN* statements. However, only AML and AL support the *IF..THEN..ELSE* extension, a construct of Pascal, FORTRAN V, FORTH and others.

Both AL and AML have the conventional *WHILE..DO* and *REPEAT..UNTIL* (*DO..UNTIL* in AL) expressions of Pascal. AL also provides a *FOR* loop. A comparable feature of AML is the ability to alter variables within a *WHILE..DO* condition. A simple *WHILE (I=I+1) LE NUM DO* serves as a *FOR* loop of AL or Pascal. *WHILE (I=I-1) GE NUM* is equivalent to the *DOWNTO* control statement of Pascal, and a *FOR* loop in AL with step size of -1. This allows for any positive or negative step value to be constructed, as in AL.

It should be emphasized that VAL has only one control statement *(IF..THEN)* other than its terminating statements, *GOTO's*, and subroutine calls. This in itself severely undermines it if we consider the several constructs offered by AL, AML and other structured programming languages.

V.3.3 Input/Output Capabilities

One shortcoming of both AL and VAL is that they have no character variables or strings, which are clearly an asset to any user-system interaction, and are a construct of AML and of most high-level languages. For instance, one would want to be able to respond verbally to a prompt ('YES' is the most blatant example) and make program execution choices based on this verbal choice.

The designers of many robot languages have in the past ignored general i/o considerations, with little emphasis on a flexible programming environment. Any robot language should provide for both user options during execution, and file reading and writing (for instance to store and retrieve data points). AML is one of the few languages that does so.

V.3.4 Computational Provisions

VAL does not support floating point variables and does not have the arithmetic capabilities of a standard programming language. VAL does support coordinate transform data types and various operations on them, such as matrix inversion, but, for instance, would not enable you to program a different algorithm for matrix inversion. It is, in essence, a packaged manipulation language, and ignores traditional data processing and computation.

As was shown earlier, AL is a computationally rich language, providing extensive coordinate transform data types and operations as well as the standard arithmetic operations. AML has limited geometric operations, but because it is very much a high-level programming language, it is computationally rich in the standard sense and is easily extensible.

V.4 Motion

All three of AL, VAL and AML support both absolute and relative moves. All three languages also offer some convention for guiding. Guiding, or teaching positions in conjunction with manual control, should be included in any robot system, and in latter years was the primary method of programming a task. It is of course very limiting as the sole method of doing things.

All movements in AML are joint-interpolated - however, because the IBM Robot is cartesian the effect is visible only in the roll, pitch and yaw joints. AL and VAL allow for cartesian motion. VAL movements may also be joint-interpolated.

Trajectory control parameters can be specified in AL and AML. In AML, one can specify the initial acceleration, the maximum speed and the deceleration of a movement. In AL, one can specify the duration and speed of a movement, as well as

forces and torques maintained for compliant moves (see V.5). In VAL, one can specify the speed component of a trajectory for merely a single joint move.

AL is the only one of the three languages which is structured for the interaction of more than one robot. Perhaps AML does not have this capability because it was designed for a cartesian robot, and the rectangular workspace limits such interaction. However, synchronization primitives for interaction not only with other robots but feeders and special-purpose machines would be very worthwhile. AML perhaps makes up for this with its monitoring capabilities. Setting a monitor on a feeder, for instance, to determine when a bin is full is very easy to accomplish because of AML's external device interface.

One aspect of manipulator programming that has been ignored until just recently is the position error caused by robot inaccuracies or changing dynamics (speed of the robot, load). AML takes steps in minimizing this kind of error by providing the routines QPOSITION and QGOAL. QPOSITION returns the current position of the arm, and QGOAL returns the last commanded position. QGOAL should be used when iterating in order to return to the last position because the position error may become significantly large over several iterations.

V.5 Monitoring Capabilities and Compliance

The ability to control interaction conditions (output) through condition monitoring is a desirable component of a robot language. Also, an invaluable way of directing the execution of a program based on interaction factors (input) is to use condition monitoring. This is dependent upon the features of the system and what types of feedback it supports, whether vision, force or touch. Monitoring is enabled to different degrees in both AL and AML.

The ability to move compliantly or specify forces is a direct result of being able to detect environmental parameters and react accordingly (control interaction conditions). Compliant motion is built into the AL language. In AML, it is possible (in fact, very easy) to program a compliant move, but the results are by no means real-time.

CHAPTER VI

FORTH

This is not a complete presentation of FORTH. Only material which is of direct concern here is discussed - i.e., FORTH's execution speed, its qualities as a structured language, and the basic units of FORTH and how to manipulate them.

There are many different versions of FORTH. The one in use at LPR is the UMASS/LLE FORTH-79. Our implementation of FORTH enables optimization of code using PDP-11 FORTH Assembler. For brevity the Assembler version is not discussed here. However, documentation for both our FORTH and PDP-11 FORTH Assembler is forthcoming [Pocock]. Some of the words cited below are non-standard and particular to our version of FORTH.

VI.1 Language Overview

FORTH is what is known as a threaded language (both interpreted and compiled). It combines the convenience of an interpretive language with a greater execution speed. FORTH words can be typed and executed immediately as in an interpretive language. Alternatively, FORTH code may be saved on disk for later compilation and execution. The storage on disk is in blocks of 1024 characters. Two blocks reside in memory at any one time, and are swapped out using a least-recently-used policy.

The FORTH operating system has multitasking capabilites. Multitasking allows a user to have interdependent programs in various states of execution at one time. This feature is valuable in a robotics environment, and shall be discussed further in Chapter VII.

The procedures of FORTH are called "words". One defines a new word using existing words. Newly defined words are added to (compiled into) FORTH's "dictionary". The dictionary is a linked list of entries, each of which defines a FORTH word. A dictionary entry for a word consists of the length of the word name, the word name, the address of the previous word, the address of the machine code to be executed when the word is invoked, and a list of addresses of other

words to execute.

In addition to the primary vocabulary of FORTH, there may be several other vocabularies. In essence, a user's vocabulary is the conglomerate of all FORTH words that have been loaded into the dictionary, and of necessity contains the base FORTH vocabulary.

In FORTH, most calculations are performed by taking input from a user-controlled stack. While all languages use a stack, it is usually not under explicit programmer control.

The stack is an efficient alternative to variables as storage for temporary information. Speed is an important consideration. Accessing the contents of the address in memory that a variable refers to causes significant overhead. Stack manipulation allows for direct access. In some instances, use of a variable is more convenient, and FORTH provides for this.

FORTH operates in postfix (reverse Polish) notation (operands precede their operator).

VI.2 Data Representation and Manipulation

VI.2.1 Numeric Representation

In our version of FORTH, integers are stored as single or double precision. Floating point is also available. FORTH normally uses base 10 (decimal) arithmetic. The base is easily changed by executing any of the following four words:

```
BINARY   => base  2
HEX      => base  16
OCTAL    => base  8
DECIMAL  => base 10
```

While it does not have an explicit character type, FORTH provides for character manipulation. FORTH uses ASCII code to store characters. To put a letter on the stack, you can either enter its numeric value or use a special input statement called *KEY*. The *EMIT* command displays the ASCII representation of the top number on the stack.

VI.2.2 Data Structures

In FORTH, one can define variables, constants, and arrays.

VI.2.2.1 Variable and Constant Declarations

The format for a variable declaration is *init-val VARIABLE name*. Typing the name of a variable places its address on the top of the stack.

Constants are defined in the same way as variables using the reserved word *CONSTANT*. Typing the name of a constant leaves the value of the constant on the stack, not merely its address.

VI.2.2.2 Variable Data Modification and Retrieval

To modify a variable, the word *!* (pronounced "store") is used. A value is put on the stack, the variable name is typed, and then *!* is typed. The FORTH word @ retrieves the value at the address preceding it and leaves it on the stack. In other words, *COUNT* @ will place the value of the variable *COUNT* on the stack.

Other words which serve to store data in or retrieve data from a memory address are as follows:

C! (n addr,) : store the least significant 8-bits of *n* at addr.

C@ (addr, byte) : leave on the stack the contents of the byte at *addr* (with the high order byte set to 0 in a 16 bit field).

@ (addr, n) : leave on the stack the number contained at *addr*.

@@ (addr, n) : leave on the stack the number contained at the address contained in *addr*. That is, do an indirect fetch.

@! (n addr,) : store the value *n* at the address contained in *addr*. That is, do an indirect store.

! (n addr,) : store *n* at address *addr*.

0SET (addr,) : set the contents of address *addr* to zero.

ISET (addr,) : set the contents of address *addr* to one.

VI.2.2.3 Arrays

Arrays may be declared as *n ARRAY name*, to allot n words of storage. When *name* is later executed, it will place the storage address on the stack. The contents of *name* are not initialized. To define and initialize an array simultaneously, the word *IARRAY* would be used as *IARRAY name val1 , val2 , val3 ,...*

The size of the array is determined by the number of values. Arrays may be defined in other ways, depending on the nature of the data to be stored. Consider these:

()BYTE (n,) : a defining word used in the form: *n ()BYTE name* to define an array of *n* bytes. If *n* is odd, the array is aligned to end on a word boundary, i.e. an extra byte is added. Execution of *name* adds the value on the top of the stack to the base address of the array. Thus *0 name* returns the address of the first byte, *1 name* returns the address of the second byte, etc. (note: no bounds checking is performed)

()DIM (n,) : a defining word used in the form: *n ()DIM name* to create an array of *n* words. Execution of *name* adds the value on the top of the stack to the base address of the array. Thus *0 name* returns the address of the first word (note: no bounds checking is performed).

VI.2.3 Data Manipulation and Usage

VI.2.3.1 Arithmetic Words

ABS (n1, n2) : leave the absolute value of a number *(n1).*

MAX (n1 n2, n3) : leave the greater of two numbers.

MIN (n1 n2, n3) : leave the lesser of two numbers.

MINUS (n, -n) : leave the two's complement of a number. This is identical to *NEGATE.*

MOD (n1 n2, n3) : divide *n1* by *n2* leaving the remainder *n3. n3* has the same sign as *n1.*

NEGATE (n, -n) : leave the two's compliment of a number.

* *(n1 n2, n3)* : leave the arithmetic product of *n1 times n2. n3* is a 16 bit result, however, 32 bits are calculated internally.

*/ *(n1 n2 n3, n4)* : multiply *n1* by *n2* then divide the 32 bit result by *n3*, leaving the quotient *n4*.

+ *(n1 n2, n3)* : leave the arithmetic sum of *n1 plus n2*.

+! *(n addr,)* : add *n* to the 16 bit value stored at *addr*.

- *(n1 n2, n3)* : subtract *n2* from *n1* leaving the difference *n3*.

– *(n1 n2, n3)* : subtract *n1* from *n2* leaving the difference *n3*. This is identical to *SWAP* -.

/ *(n1 n2, n3)* : divide *n1* by *n2* leaving the quotient *n3* on the stack.

/MOD *(n1 n2, n3 n4)* : divide *n1* by *n2* and leave the remainder and quotient *n4* on the stack.

1+ *(n, (n+1))* : increment the top of stack *(n)* by one.

1- *(n, (n-1))* : decrement the top of stack *(n)* by one.

1+! *(addr,)* : increment the contents of address *addr* by one.

1-! *(addr,)* : decrement the contents of address *addr* by one. (same for twos of last 4, like 2+...)

IV.2.3.2 Logical Words

A variable that contains a boolean value is called a "flag". Basically, FORTH treats the numeric value 0 as *FALSE* and all non-zero values as *TRUE*. Several comparison tests are possible, all of which result in a true (1) or false (0) value being placed on the stack. Among those words which output a boolean value are: =, <, <=, <>, > and >=, which may be thought of in the conventional sense. Comparators are also provided (0<, 0<=, 0=, 0<>, 0>, and 0>=) which take as input one value, use 0 as their second operand, and output a boolean result.

Logical words, which are useful particulary in comparing the bitwise structure of a number with a mask, are contained in the primary FORTH vocabulary. The word *AND*, for instance, leaves the bitwise logical *AND* of the top two numbers on the stack. The words *OR* and *XOR* operate in the same manner, leaving the bitwise logical *OR* and *XOR* on the stack, respectively.

VI.2.3.3 Bit Manipulation Words

In addition to the logical words which perform bitwise comparisons of numbers, FORTH contains several other words for bit manipulation. A representative sample of those is listed below.

BIC (n mask, n) : clear the bits in *n* that are set to one in *mask*. That is, *AND n* with the one's complement of *mask*.

BIS (n mask, n) : set bits in *n* the are set to one in *mask*. That is, logically *OR mask* with *n*.

BITCLEAR (mask addr,) : clear bits at *addr* that are set in *mask*, ignore those that correspond to zero bits in *mask*.

BITSET (mask addr,) : set bits at *addr* that are set in *mask*, ignore those that correspond to zero bits in *mask*.

COM (n, n) : leave the one's complement of *n* on the stack.

SHL (n1, n2) : shift the element on top of the stack arithmetically one bit left.

SHR (n1, n2) : shift the element on top of the stack arithmetically one bit right.

->A (n1 m, n2) : arithmetically shift *n1 m* places to the right leaving the result *n2* on the stack. (note: the word *<-A* is not defined since it is identical to *<-L*)

->L (n1 m, n2) : logically shift *n1 m* places to the right leaving the result *n2* on the stack.

<-L (n1 m, n2) : logically shift *n1 m* bits to the left leaving the result *n2* on the stack.

VI.2.4 Stack Manipulation

VI.2.4.1 Stack Manipulation Words

Various words exist which operate on the stack directly without regard to the values of the entities on the stack. *DEPTH* for instance, determines the number of 16 bit values contained in the stack. This is useful in determining whether there are enough input values to perform a specific operation. *DROP* drops the top number on the stack (does not print it). There are duplicating words such as *DUP*, which duplicates the number on top of the stack, *OVER*, which leaves a copy of the second

number on the stack, and *PICK*, which picks the nth element from the stack. There are also words for merely changing the order of values on the stack. *ROT* rotates the top three values, bringing the deepest to the top. *SWAP* exchanges the top two stack elements. *SWAP-BYTES* swaps the bytes in the top of stack element.

VI.2.4.2 Conditional Stack Manipulation Words

There are also some conditional stack manipulation words. *?DUP* duplicates the top of the stack only if it has a non-zero value. *0>DUP* duplicates the top of stack if it is greater than zero (positive). These are just some of the many stack manipulation words, which provide much versatility in programming.

VI.2.5 Input/Output

FORTH is also a good language for input/output, although its low-level characteristics defy this assumption. The following words exhibit this:

QUERY : accepts input of up to 118 characters from the input terminal.

SPACE : transmits an ASCII blank to the current output device.

SPACES (n,) : transmits *n* spaces to the current output device. No action is taken if *n* is less than one.

TYPE (addr n,) : transmits *n* characters beginning at *addr* to the current output device. No action is taken if *n* is less than one.

Y/N? : accepts from the user on the current input device, a "YES" or "NO" response. If "YES" is entered, the true flag is returned. Any other response returns the false flag.

." : used in the form: ." *ccccccccc*". Accepts the text following from the input stream terminated by double quote ("). This text is transmitted to the current output device.

."CR : same as ." except that the output string if followed by a carriage return.

? (addr) : print on the current output device the value contained at address *addr*.

VI.3 FORTH Program Structure and Control

VI.3.1 Modules

The basic modules of FORTH (in fact, the only modules of FORTH) are its words. A FORTH word is immediately executable, as discussed above. The words ':' and ';' are special system words. In order to create a new word, all you do is put a colon and the name of the word in front of a sequence of already existent words, and terminate the string with a semicolon. This is what is termed a "colon definition". The FORTH dictionary was described above. To delete a word from the dictionary (and simultaneously all subsequent words) the *FORGET* command is used.

FORTH words and declarations are stored in blocks of 1024 characters. The *LOAD* command begins interpretation of block n by making it the input stream. If the block is terminated with *;S* then only that block will be loaded. Each subsequent --> at the end of a block causes the next block to be loaded.

VI.3.2 Conditional and Looping Constructs

FORTH is a structured language, and as such uses no *GOTO's* or labels for statements. It provides several conditional and looping constructs.

The "if-then" and "if-then-else" constructs of FORTH are implemented as follows:

flag IF.....THEN => executes the words specified if and only if the flag is true.

flag IF.....ELSE words THEN => the words between *IF* and *ELSE* will execute if the flag is true, and the words between *ELSE* and *THEN* will execute if the flag is false.

Any number of *IF-THEN's* and *IF-ELSE-THEN's* may be nested, as long as the keywords are in the proper sequence.

The *SEL* statement in FORTH is essentially a case statement. It has the form:

```
n    SEL
  <<  n1  ===>     ( action when n=n1)  >>
  <<  n2  ===>     ( action when n=n2)  >>
```

.
.
.

<< nm ===> (action when *n=nm*) >>
OTHERWISE (action when none of the above are true)
ENDSEL

The *OTHERWISE* clause is optional.

FORTH provides the looping constructs listed below:

n1 n2 DO.....LOOP => the loop index begins at *n2* and terminates at *(n1 - 1)*.

n1 n2 DO.....value +LOOP => at *+LOOP*, the index is incremented by the value on the top of the stack.

BEGIN.....flagEND
BEGIN.....AGAIN
BEGIN.....flagUNTIL
BEGIN.....REPEAT => *BEGIN* marks the start of a word sequence for repetitive execution. A *BEGIN* loop will be repeated until *flag* is true. The words after *UNTIL* or *REPEAT* will be executed when either loop is finished. *Flag* is always dropped after being tested.

VI.4 Summary

FORTH is both a language and an operating system. The FORTH operating sytem provides for multitasking, a feature which enhances a distributed control structure.

FORTH is what is known as a threaded language (both cross-interpretive and compiled). FORTH code executes faster than interpreted code in general. FORTH is well-structured, with no *GOTO's* or labels for statements. Its versatility surpasses both that of a low-level assembly language and that of a high-level structured language. Low-level bit manipulation may be performed in a variety of ways. This is extremely useful for bit-oriented access to hardware, perhaps to debug a new interface or to toggle switches. On the other side of the spectrum, FORTH has all of the niceties of a user-friendly language (disregarding the awkwardness of using postfix notation). FORTH is also quite fast, and a combination of FORTH and PDP-11 FORTH Assembler works well in a real-time environment.

CHAPTER VII

STEPS TOWARD THE DEVELOPMENT OF PRL: THE PERCEPTUAL ROBOTICS LANGUAGE

VII.1 The Laboratory for Perceptual Robotics

At present, the LPR possesses two manipulators. One is a large industrial arm, a prototype Cartesian Robot (CART) donated by the Flexible Automation Systems Program of the General Electric Company of Schenectady, New York. The CART consists of three prismatic (sliding) joint axes (*x,y,z*) and a revolute joint axis (*theta*). The CART's end effector is at present a two-fingered gripper developed and donated by the Digital Equipment Corportation (Figure 6 shows the CART and the gripper).

Digital Equipment PDP-11's will serve as single axis controllers for the CART. Currently an interrupt driver written in FORTH and running on a PDP-11/23 is simulating this architecture.

The LPR has recently acquired a three-fingered, nine degree of freedom Salisbury Hand (Figure 7). We will attach two or more tactile sensing arrays to opposing fingers of the hand. For the low-level control of the Salisbury Hand, we are designing a hierarchical control structure based on a network of DEC T-11 microprocessors (Figure 8). There will be one processor for each agonist tendon and one for the antagonist on each finger. Above this layer, there will be one processor per finger to synchronize fingers and integrate tactile information. There will then be at least one processor at the top of this hierarchy for explicitly controlling grasp.

Our second, "assistant" manipulator is a small revolute joint arm, the Rhino XR-1 of Rhino Robots, Inc. The Rhino is under simple bang-bang servo-control.

In order to research the use of vision, both static and dynamic, in a robotics domain, the lab has a General Electric TN-2200 C.I.D. camera. The camera has a 128-squared resolution. For tactile image interpretation, we have in our possession an 8x16 analog tactile array sensor developed by Overton [Overton, 1983]. A PDP-11/03

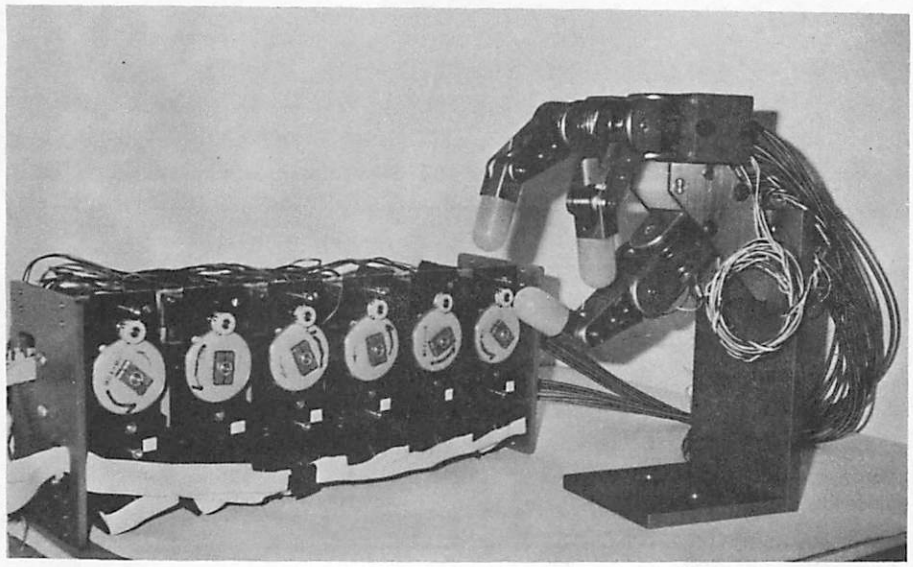

Figure 7. The Salisbury Hand and Tendon Motors

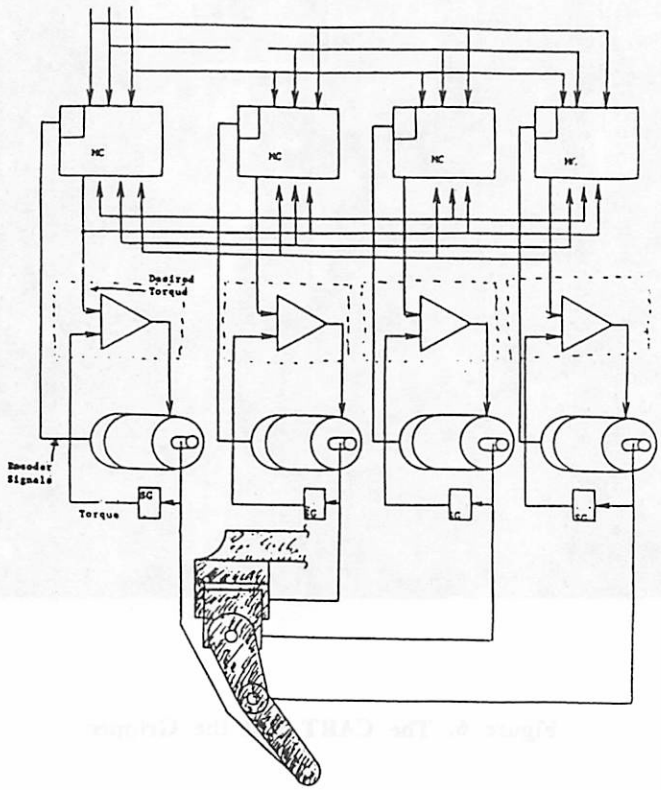

Figure 8. A Salisbury Finger and the Hierarchy of DEC T-11's

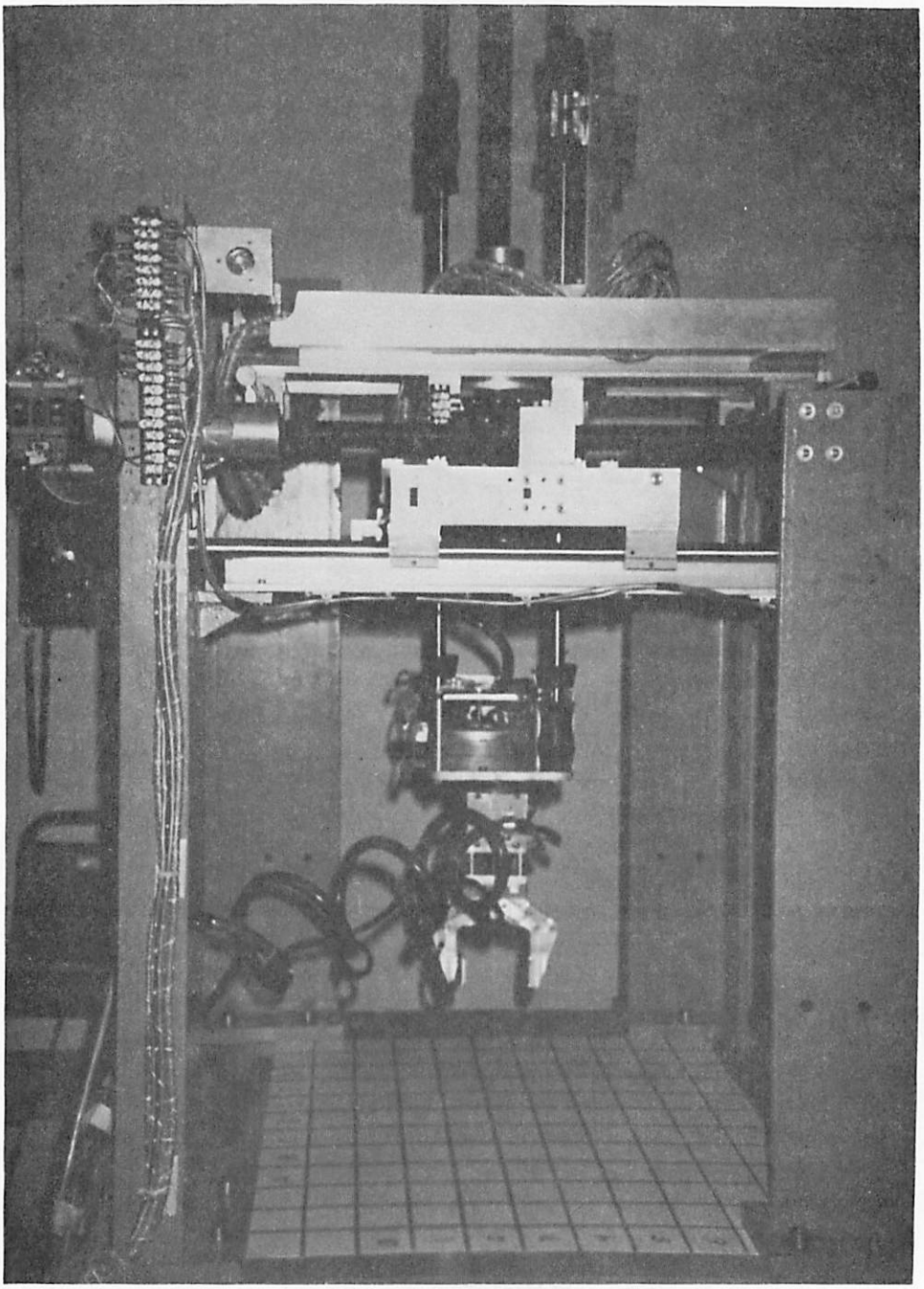

Figure 6. The CART and the Gripper

system is used for preprocessing of data from the camera and tactile sensor systems before it is passed to a VAX-11/750 system.

Sonar will be experimented with as a possible sense.

VII.2 Dynamic Sensing and Control

The sections below outline our current research efforts and goals and relate these to our language requirements.

VII.2.1 Schemas

In order to design our perceptual language for robot control, we must first address the questions of how actions will be coordinated and how sensory information will be integrated into our control structures.

Planning for a task takes into account all known, pertinent environmental parameters to build a cluster of actions which will effect the strategic objective. As a task progresses, the overall control mechanism must be able to adapt to unforeseen phenomena, modifying the world model and updating the current strategic plan (and its associated tactical sub-plans) accordingly. Raw sensory input must be interpreted in terms of objects or other "domains of interaction" before it can be used by the control mechanism.

We use the term "schema" to denote a representation of a collection of actions and sensory processing units. It couples the functioning of motor units with sensory processing. Motor schemas define a subclass of schemas in that their primary function is one of controlling the action of actuators, biological or mechanical. Perceptual, or sensory, schemas acquire and process sensory information.

A motor schema is an answer to the fundamental need to be able to effect structured changes in goal-directed actions as unforeseen conditions arise. A motor schema is a control system, continually monitoring feedback from the system it controls to determine the appropriate pattern of action to achieve the motor schema's goals. A perceptual schema is the process whereby the system determines whether a given domain of interaction characterizes the environment [Arbib, 1981 and Overton,

1983].

The overall design of a schema involves three basic components, an activation mechanism, an event section and a tracing component.

The activation mechanism monitors both the goal state(s) of the system and sensory input, and may activate or deactivate the schema based on preset conditions regarding the goal state or sensory input. For example, in cases where the activation mechanism perceives a supra-threshold change in some sub-section of the monitored environment, it can invoke the schema with the appropriate data to allow the event section to proceed to alter the robot's configuration to adapt to the perceived change. The components of the event (behavioral) section can function in either a parallel or serial fashion, depending on the interdependencies of the requisite action(s). There may also be a "tracing" component (not necessarily distinct from the others) to allow adaptation, or tuning, of the schema over time and to provide a history of the particular instantiation of the schema [Overton, 1983].

An example of the function of a schema in a robot system would be the control of a strategic task such as the retrieval of an object. In such a case, a problem arises when the robot encounters an object not in its world model while en route to the desired object. The activation mechanism of a schema will realize an "activation interrupt" is called for and spur the behavioral (event) sequence. The event sequence might consist of a visual analysis of the object, followed (serially) by the extension of a complex arm (which itself is a parallel action, since several joints may be concurrently activated) containing tactile sensors with which the control structures of the schema attempt to "fill in the blanks" of the visual interpretation. More specifically, suppose the results of image processing hypothesize the existence of an edge on the object, but with some inconsistency or breakpoint towards its middle. Then the perceptual schema must decide how to validate or disprove the hypothesis. Should the camera be oriented and positioned differently so as to eliminate any possible shadowing... or should the tactile sensor be used to probe the object? Decisions such as these may be resolved using probabilistic measures based on light intensities and proximity of the camera or tactile sensor to the object.

Thus, strategic plans are constantly modified through the low-level interpretation of the environment and maneuvering effected by the control structures that comprise the perceptual and motor schemas.

VII.2.2 Vision

VII.2.2.1 Static Vision

The approach of the VISIONS group for control of processing in object recognition can be extended to and simplified in a localized robotics environment. For the purposes of the LPR, because time is such an important consideration, we will combine rough processing and strict environmental constraints with domain-dependent control strategies to greatly simplify the recognition process. A more reliable system is created by integrating the processing from multiple sensory modes (vision and tactile), analyzing different aspects of the visual data (such as color, texture, and motion), and from local contextual cues.

The VISIONS approach makes use of "object schemas", each of which is associated with a particular object class and contains a sequence of actions (routines) to recognize that class [Weymouth, 1983].

The Robotics group will implement this schematic approach using VISIONS processing techniques as follows. We will make use of a "grouping schema", which groups objects based on any number of physical or abstract properties and activates routines to recognize its members. A characteristic data base for each known object will consist of average values for features on the object surface (texture, reflectance, intensity values or color) with an accepted deviation from each, and geometric characteristics. This combination of information will uniquely describe the object. Present in the characterization will be information on the position and orientation of the object, which can be easily updated as it is moved by the manipulator. Each object characterization will be given a name.

The name of each component object will be embedded in the corresponding grouping schemas so that positional information may be retrieved quickly via the characterizations. A schema may also group other schemas, in which case recognition routines would be activated by the component schemas. Any one object may be

grouped within a number of schemas.

This is sensible in light of what has already been done by VISIONS. Modular processes have been implemented for analyzing and matching 2D shapes, hypothesizing objects based upon the match of color and texture attributes of regions and stored object attributes, specific object matching routines based upon relative locations, analysis of perspective cues to place objects in space, matching of the size of stored objects with the hypothesized size of surfaces and volumes, shadow merging strategies, among others.

The VISIONS group and the ROBOTICS group are examining the range of control strategies for applying this information in the process of interpretation of visual and tactile data. These include issues of accessing relevant schemas based on prominent features, focus of attention mechanisms for selecting worthwhile portions of the sensory field for analysis, and the ways of decomposing knowledge hierarchically so that partial matching can be effective.

A preliminary set of primitives for vision that will be developed by the LPR is functionally defined in Chapter VII below.

VII.2.2.2 Dynamic Vision

Dynamic vision may be used to determine the absolute depth of objects and environmental surfaces. Depth maps can be defined for static scenes (objects do not move) by assigning depth as a function of displacement in a translating image sequence (via foci of expansion) or directly in a stereoptic system. Under a set of known camera and object motion constraints it has been shown that a system can yield absolute depth from simple calculations.

As a supplement to static object recognition methods (outlined above in static vision section), dynamic visual analysis can be used to refine segmentations and identify occluding contours. Segmentations can be refined by merging regions which move harmoniously and splitting regions which distort in ways which indicate they may consist of more than one distinct object. If direction of motion is ever discontinuous at more than one point (along a line, for example) then an object boundary is present. Given a depth map, foci of expansion can be derived by perspective analysis of the motion of objects at different depths in a dynamic scene.

Occluding edges exist at borders where there exists a depth disparity. Motion analysis can also provide a basis for high level semantic input in that object recognition can be constrained by motion requirements (i.e., the lip on the side of a conveyor does not move, in contrast to the conveyor itself and the objects upon it).

Information from our G.E. camera is now being used by Daryl Lawton and the Motion group in their study of Optic Flow. In the future, such Optic Flow information will be used by the arm's control routines for obstactle avoidance.

VII.2.3 Touch

Tactile sensing is performed over an area in order to provide a patterned response. This should be distinguished from force and torque sensing, both of which use information from one point. A binary tactile sensor signals the presence or absence of a contact force (and thus an object). An analog tactile sensor produces an output proportional to a contact force.

Binary or analog touch sensors fitted to an end effector can be used to identify an object or construct a representation of an object. Both are useful for determining the topology of a (small) surface. An object's edges can be traced to determine its shape, position and orientation. Analog touch sensors can be used to provide information about grasping forces, and also may be used to detect slippage during the acquisition of an object. Tactile sensors with a fairly high resolution may be used to obtain texture information.

Our initial work on interpretation from touch will use the Overton tactile array sensor. The sensor outputs analog signals which are converted into a digital image array. This image can then be processed in much the same manner as a picture from the G.E. camera is. With the help of the UMASS VISIONS group, we will develop routines to use this information in a real time environment.

There can be no question that dynamic sensing will improve the adaptability and capabilities of a robot. The coordination of touch and vision offers tremendous possibilities in the future for a robot that can interact intelligently with its environment.

VII.2.4 Coordinated Control

If we look to humans as an example, it is obvious that the coordinated use of two arms is more effective than the use of two arms independently. For instance, the handling of large objects sometimes requires more than one arm. Also, it is easy to envision robots with stationary bases passing things to each other.

The control of multiple arms and of the multi-fingered Salisbury hand will require extensive interaction between processors. The type of (software) synchronization structure necessary is not available with any of the languages studied. Semaphore-like interaction or event signalling will not do. Although coupled control of the fingers is unnecessary in some cases (for instance, when position-servoing trajectories through space), in most instances the individual controllers must cooperate [Salisbury and Craig, 1982]. Knowledge of the relative position of all fingers, the forces they are exerting or that are being exerted on them, and of the tactile information they gather is essential.

VII.3 FORTH and the Perceptual Robotics Language

VII.3.1 Additional Design Considerations

Our highly sensory robotic system, for which the acquisition of new equipment (robots, sensory devices and processors) is continual, requires a base language with which both new hardware devices can be easily tested and sensory inputs can be treated as bitwise logical units in the same manner as the AML system. The language should, however, be enough removed from assembly language so that low-level bit manipulation can be avoided by those who wish to program more sophisticated tasks. The three languages studied have this quality, although many of the earlier robotics languages do not. In addition, the language must enable modification of high level routines without forcing the shutdown of the CART and the loss of current "World State" information.

The threaded language qualities of FORTH and its multitasking capability enable and enhance both serial and parallel execution of processes in an interactive, real-time environment.

These concerns led us to FORTH. We have used it to aid in debugging newly built hardware, in the acquisition of sensory information, and in the control of the CART. With the development of the "Schema" language, FORTH becomes a good candidate to build upon.

The use of dynamic touch and visual feedback to plan movements in a general robot environment is being examined by the LPR. Our emphasis is on a high degree of sensory interaction. Our new language will not rely on absolute positional information to move the arm but rather rough directional information combined with continual sensory information (as do humans). Our initial work with FORTH has included such absolute motion routines (listed below), and they will be needed until our vision and tactile routines are more developed.

Continual monitoring of the environment renders coordinate transform capabilities unnecessary if not awkward to implement. This is distinct from other robotics languages such as AL and VAL, which do not have the sensory capabilities we do and therefore rely on absolute positional information. Our approach can be easily extended to the control of a mobile robot, whereas absolute information has no meaning in real space.

Our base-language motion routines will otherwise contain the desirable features of the three languages that have been studied. The specification of the acceleration and deceleration of the arm, for instance, is important in performing fine motions. With such a capability, the language must also detect ambiguities that often arise, and should be able to adjust itself. There must also be differential defaults which depend on the nature and distance of the move.

The PRL will realistically need a high degree of error checking and recovery built into each low-level routine.

The routines for vision, tactile sensing, and those which maneuver the Salisbury hand will be embedded in a distributed schema architecture.

VII.3.2 Functional Definition of Primitives

A preliminary list of primitives for vision, tactile sensing, grasping and motion have been functionally defined below, and will be actualized as appropriate. Single-facet motion and low-level control routines have been and will be written in FORTH. On a higher level, we have begun the definition of PRL to implement primitives for touch and vision.

(x,y,z) are the linear axes of the CART in a right-handed coordinate system, x and y being horizontal and z being the vertical axis. Arguments to the low-level routines are parenthesized.

VII.3.2.1 FORTH Routines

The routines defined below have been written in either FORTH or FORTH Assembler, and have been implemented on the CART with the two-fingered gripper. They are not to be considered a part of our new language -- however, they represent our progress thus far on the project. A subset of these will used as a base language from which we can support our Perceptual Robotics Language.

* Read.Counter: Read the position of one axis.

* Abs.Move.Single: Absolute single axis move.

* Max.Vel.Single: Set velocity for an axis.

* Abs.Move: Absolute move on all four axes.

* Max.Vel: Set all four velocities.

* Rel.Move.Single: Move to a postion delta units from the current position along the specified axis.

* Rel.Move: Move to a position (dx,dy,dz) units from the current position.

* Synch.Vels: Uses a scaling algorithm to set the maximum velocity for each axis based on the distances to travel.

* Absolute-Move: Takes as arguments the desired positions, synchronizes the velocities for each axis, and moves to the positions via the routine Abs.Move.

* Specmove: Move to a position (dx,dy,dz) units from the current position,

accelerating as (acc) until the desired velocity (vel) is reached, then approaching with deceleration (dec).

* Hover: Move within a specified distance (disp) of an object, the displacement being along the z axis. This enables us to quickly get near an object and slowly move in to do a delicate operation by means of Descend.

* Descend: Move gradually from the last Hover position to the last specified object along the vertical axis (z).

* Ascend: Move gradually along the positive z axis the displacement specified by the last Hover.

* Position: Enables the user to record manipulator positions in a specified file. Pressing a terminal key (RECORD) will record subsequent positions and pressing a key (END) will end the recording session. It is possible to record when the robot is moving or the system is otherwise preoccupied (in the editor, for instance). In this manner, the guiding (moving the arm and teaching positions) can be done manually or during program execution.

* Step.In: Takes a time delay and number of steps as arguments, and steps the gripper closed that number of steps from its current position. The time delay will control the apparent velocity.

* Step.Out: Takes the same parameters as Step.In and steps the gripper open.

* Grip.Pos: Takes a ramp delay and an absolute position and servos the gripper to that position.

* Vgrip: Allows the programmer to specify just a final position to the gripper. The routine then superimposes a five step velocity profile on the gripper motion.

* CLOSE: This routine takes no parameters and just closes the gripper until the position counter no longer changes. Note: the object is squeezed a little.

* Object-Velocity: Derive velocity vector of a moving object.

VT100 Display Routines for Overton Tactile Sensor Image

* Init-Display: Initialize the display for the touch sensor.

* Display: Displays the 16x8 tactile sensor image.

* Getval: Reads one input.

* Get-Image: Reads an entire image into an array.

* Display-Image: Coordinates the above four routines to display the image.

VII.3.2.2 PRL

The PRL is tentatively defined below. While initially intended for use with the CART, the system will eventually be compatible with other arms. These advanced perceptual motion routines are in fact Schemas, and the Schemas involving taction at present are intended for use with our Salisbury Hand. Low-level primitives for the control of the hand have not been included.

Members of the robotics group, in parallel with this work, have been postulating coordinated control programs for movements of the hand. The emphasis in this development is on neurophysiological studies and human and animal behavior. The study is documented in [Arbib et al., 1983]. However, at this point in time a "hand language" is not well-defined.

Advanced Perceptual Motion Schemas

* Touch: Approach a specified object (name) with the touch sensors enabled, and if a sensor is activated (a certain threshold is reached) perform a specified action.

* Roll: Invokes the Schema that moves the fingers in such a way that a rounded object tends to roll between them. Simultaneously, the Schema reviews and processes the tactile sensory data associated with such motions.

* Squeeze: Examines the extent of surface or object deformability.

* Brush: Examines the texture of a surface from the dynamic tactile patterns created by sliding or brushing a sensor over it.

* Follow-Edge: The tactile sensor follows an edge to map the object's geometry.

* Estimate-Distance: Estimates the distance from the arm to an object using visual information and a sonar range-finder.

* Predict-Contact: With the camera mounted on the arm, determines from the optic flow field whether the hand will make contact with an object.

* Intercept: Predict motion of an object in order to manipulate the hand so as to intercept the object.

* Characterize: Builds the characteristic data base for an object.

* Locate: Performs rough visual processing and extracts feature values in order to map an object characterization to the region(s) within the workspace which match (within a certain epsilon).

* Grasp-object: Cooperates with "Locate" to locate a specified object, manipulates the robot into the vicinity of the object, preshapes the hand, and then grasps the object.

* Release: Free a grasped object, or remove all positional constraints on the object imposed by the hand.

* Relax: Move the hand to a position of rest.

* Identify: Schemas such as Roll(ing), Squeeze(ing), Brush(ing), etc., in addition to static image processing Schemas are invoked to explore an object and determine its identity or spatial characteristics.

BIBLIOGRAPHY

Arbib, M. A., Iberall, T. and Lyons, D. "Coordinated Control Programs for Movements of the Hand," Technical Report Number 83-25, Computer and Information Science Department, University of Massachusetts at Amherst, 1983.

Arbib, M. A. "Perceptual structures and distributed motor control," *Handbook of Physiology -- The Nervous System, II. Motor Control* (V.B. Brooks, Ed.), American Physiological Society, Bethesda, MD, 1981.
also: Technical Report Number 79-11, Computer and Information Science Department, University of Massachusetts at Amherst, 1979.

Brooks, R. and Lozano-Perez, T. "A Subdivision Algorithm in Configuration Space for Findpath with Rotation," Artificial Intelligence Laboratory, AI Memo 684, Massachusetts Institute of Technology, December 1982.

Finkel, R., Taylor, R., Bolles, R., Paul, R. and Feldman, J. "AL, A Programming System for Automation", Stanford Artificial Intelligence Laboratory, AIM-177, November 1974.

Franklin, J. A. "Computer Interfaces and Operating Instructions for A Prototype Cartesian Robot," Technical Report Number 83-10, Computer and Information Science Department, University of Massachusetts at Amherst, July 1983.

International Business Machines, Inc. "IBM Robot System/1, AML Concepts and User's Guide," International Business Machines, Inc., 1981.

Lozano-Perez, T. "Spatial Planning: A Configuration Space Approach," Artificial Intelligence Laboratory, AI Memo 605, Massachusetts Institute of Technology, December 1980.

Lyons, D. "Gripper Manual," Internal Memorandum No. 3, Laboratory for Perceptual Robotics, Computer and Information Science Department, University of Massachusetts at Amherst, 1983.

Mujtaba, S., and Goldman, R. "AL User's Manual," Stanford Artificial Intelligence Laboratory, AIM-323, January 1979.

Nevins, J.L. and Whitney, D.E. "Robot Assembly Research," *Computer Vision and Sensor-Based Robots* (Dodd, G. and Rossol, R., eds.), General Motors Research Laboratories, 1979, pp. 275-321.

Overton, K. J. "The Acquisition, Processing, and Use of Tactile Sensor Data in Robot Control," Doctoral Dissertation, Computer and Information Science Department, University of Massachusetts at Amherst, 1983.

Paul,R. *Robot Manipulators, Mathematics, Programming and Control*, MIT Press, 1981.

Pocock, G. "UMASS/LLE FORTH Implementation Guide," Technical Report, Computer and Information Science Department, University of Massachusetts at Amherst [in preparation].

Salisbury, J.K. and Craig, J.J. "Articulated Hands: Force Control and Kinematic Issues," *International Journal of Robotics Research*, Vol. 1, No. 1, Spring 1982.

Taylor, R., Summers, P. and Meyer, J. "AML: A Manufacturing Language," *International Journal of Robotics Research*, Vol. 1, No. 3, Fall 1982.

Unimation Inc. "Puma Robot Technical Manual," Unimation Robotics, Unimation Inc., 1980.

Unimation Inc. "User's Guide to VAL, A Robot Programming and Control System," Unimation Robotics, Unimation Inc., June 1980.

Weymouth, T. "Schema-Guided Interpretation," Doctoral Dissertation [to appear], Computer and Information Science Department, University of Massachusetts at Amherst, 1983.