# An Environment for Programming a PUMA 260 Work Cell

Eric McConney

March 14, 1986

Computer Vision and Robotics Laboratory
Department of Electrical Engineering
McGill University

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Engineering.



Postal Address: 3480 University Street, Montréal, Québec, Canada H3A 2A7

# An Environment for Programming a PUMA 260 Work Cell

Eric McConney

## Abstract

An improved Robotics Applications Programming environment (RAP) has been developed for the PUMA 260 robot for use in the repair of hybrid integrated circuit boards. The system features full control of the robot, vision system, X-Y linear stage, motorized microscope, and digital input/output interface module. In addition to the normal editing and filing capabilities, this unique programming environment provides the ability to compose and execute programs "concurrently" as opposed to the more traditional Edit - Compile - Link - Run sequence. Debugging capabilities are featured, including the ability to set break points, and single step a program both in the forward and backward directions. Further facilities allow the user to pause, continue or abort a running program. RAP is programed in the C language and runs under the UNIX 4.2 B S D operating system. Full spelling and syntactic checking of command lines is achieved by the use of keyword matching techniques. The RAP system greatly simplifies the programming task for such a complex environment, allowing the user to concentrate on the activities being performed rather than on the programming details. The program is being successfully applied to the repair of hybrid integrated circuits in the McGill computer vision and robotics laboratory.

# An Environment for Programming a PUMA 260 Work Cell

Eric McConney

## Résumé

Cette thèse présente un environnement de programmation robotique qui a été développé pour le PUMA 260 afin de procéder à la réparation des circuits hybrides. Le système permet le contrôle du robot, du système de vision par ordinateur, de la table X-Y ainsi qu'une multitude d'autres périphériques. Grâce à ce système, il est possible de composer, de modifier, de mémoriser et d'exécuter des programmes sur le champ, ce qui est un avantage face à la séquence traditionelle d'édition, de compilation et d'exécution Il y a des facilités pour la vérification des programmes, et d'autres qui permettent à l'utilisateur de suspendre temporairement le programme à des points spécifiés ainsi que de les exécuter en mode pas-à-pas, soit dans le sens du programme soit en sens contraire. De plus, il est possible de suspendre, de continuer ou d'arrêter un programme pendant l' exécution de ce dernier. Cet environnement a été programmé en language C et fonctionne présentement sur un système d'exploitation UNIX 4.2 BSD. Les erreurs de syntaxe étant détectées immédiatement, la tâche du programmeur se trouve grandement facilitée, ce qui lui permet de se pencher davantage sur ce qui doit être fait plutôt que sur la manière dont les programmes seront exécutés. Cet environnement est présentement utilisé avec succès pour la réparation des circuits hybrides dans le laboratoire de vision par ordinateur et de robotique de l'Université McGill.

March 14, 1986

# Acknowledgements

I would like to extend my thanks to my advisor, Dr. A. S. Malowany, for his guidance and encouragement in every aspect of this project. It has been both educational and a pleasure to work with him.

Thanks are also due to my colleagues, for their help and advice in many areas of this project. In particular I would like to thank M. Parker for his help with the debugging of some of the more convoluted constructs of my code, and A. Mansouri for help with the vision section of this project and the translation to French of the abstract of this thesis.

The tools used in this project would not have been possible without the help, advice and mechanical expertise of J. Földvari to whom I would like to extend my thanks.

I am grateful to my family, in particular my mother and father for their support, without which this project would have never been completed.

March 14, 1986

# Table of Contents

# List of Figures

# Chapter 1 Robotic Systems

## 1.1 Introduction

The development of robotic systems is a new field which is of great interest to industry. Historically, all manufacturing assembly has been performed manually, but with increasing process complexity and as large corporations strive for greater efficiency in their production operations, the use of robots is likely to become commonplace.

In 1975, a company called Vicarm Inc., which was involved in making manipulators for research laboratories, developed a test program, to incorporate a robot manipulator interfaced with a microcomputer, for the purpose of demonstrating the manipulator [Shimano 1]. Vicarm was sold to Unimation Inc. in 1977 and, based on the computer robot interface system, Unimation developed an industrial grade robot, under contract to the General Motors corporation. The robot, called PUMA (Programmable Universal Machine for Assembly) was delivered in 1978.

Since then, studies have been done to investigate communications between slave robots, sensors and supervisory control systems. The objective of current research work is to develop general applications, using sensory feedback, which are reliable in complex assembly operations [Lieberman and Wesley 2]. This presents several interesting opportunities for the researcher.

The repair of hybrid integrated circuit boards has been selected by the author as a vehicle for the development of a robotic programming environment called the RAP (Robotic Applications Programming) system. The environment which has been created, offers a friendly interface to robotic programming, suitable both for the novice and experienced

programmer. It enables the programmer to select syntactically correct commands from a menu, provides help prompts, contains editing and debugging features and generally permits the programmer to create robotic applications with great ease

### 1.1.1 The Need for Robots

The most basic form of manufacturing assembly, being manual in nature, is suitable in low volume operations, where manufacture by hand is a positive marketing factor. That is, where the consumer desires a personally customized product, for a specific purpose or need, and is prepared to pay a premium for it

At the high end of the assembly volume spectrum, where the "nuts and bolts" of industry are of standard design, the production process is characterized by high capital cost equipment, which automatically transfer materials and parts between machines. This hard automation may be extensive, depending on the nature of the items being manufactured.

Robot control systems are of interest to industries in the intermediate region of the assembly volume spectrum where certain aspects of assembly are critical to the production process. This volume range is characterized by an assembly sequence in which already manufactured parts, tools and fixtures have to come into close contact with each other, in a harmonious manner, to achieve the desired assembly goal.

The most widespread use of industrial robots today, is to carry out repetitive functions such as those involved in gripping parts, picking them up and performing routine assembly operations on them. However such basic tasks, involving the most simple teach and repeat commands, have applications which are limited to jobs such as spray painting, material handling, machine loading and spot welding.

As manufacturing systems become more sensitive to part tolerances, particularly in miniaturized processes, the need for precision becomes apparent. Such precision can be achieved only in an environment in which the robots interact with supervisory control and sensory devices, thereby enabling motions to be described with mathematical accuracy, in addition to providing the capability for sensory feedback and decision making.

Problems in the development of robotic systems to carry out manufacturing assembly operations are characterized by the need to create a friendly environment, in which a

programmer may specialize in creating programs, without being encumbered by exhaustive technical programming detail.

### 1.1.2 Description of Motion

Specification of three dimensional movements, characteristic of assembly operations, is a very difficult task. The problem is that three dimensional movements are very difficult to conceptualize, because of their associated spatial complexity. While loose oral descriptions, with a set of drawings, may be given in a factory environment to describe an assembly procedure, such simple descriptions are of no use in programming the robotic movements, which are complex in nature

Description of the motions required to be performed, depend on the capacity of the programming language to make accurate statements about the geometric and physical nature of assembly. Geometric terminology is necessary to describe the shape of an object and its location relative to other objects. Physical support or attachment arrangements and stability of the objects, while relatively easy to describe in static conditions, become difficult to describe under conditions in which the physical relationships are changing, as in robotic manipulator fastening operations.

Therefore, the programmer must be able to function in an environment,which provides immediate feedback through observation of the robot's movements, as they are being programmed, while permitting immediate modification of commands as may be desired.

Efforts to solve these difficult description problems are being made by the development of systems such as the world model data base, [Lieberman and Wesley 2] to provide and update information in order to reflect changes related to specific commands. The work of [Ejiri et al 3] as described by [Will and Grossman 4] has solved in principle, most standard assembly problems. The challenge now, is to handle real-life industrial situations in which parts, objects and assemblies are varied in design, and to develop convenient methods of programming manipulators which may be imperfect.

## 1.2 Types of Robots

There are many different types of manipulators or robots characterized mainly by their joint configurations. Two types of joints are used in robots, rotary or revolute joints and

prismatic or sliding joints Multi-motor configurations are required to rotate the machine, to articulate the joints and to open and close the grippers. Sensors and potentiometers measure motor positions and the signals from these sensors are used to control the trajectory and position of the manipulator. To achieve any arbitrary position and gripper orientation there must be at least three revolute joints in the manipulator.

In order to describe various movements by which the robot can be actuated, the terms "axis" and "degree of freedom" are used. The number of movements possible in a given system is called the degree of freedom of the system while the mechanical devices causing these movements are called axes. Each axis does not necessarily correspond to a degree of freedom and in fact the maximum degree of freedom is six. Additional axes only serve to increase the geometric dimensions of movement, if they are primary axes, or increase the orientation possibilities, if they are articulated axes. The many types of robots can be divided into three categories defined by the coordinate system the robot operates in.

The first type of robot is the cartesian robot. This robot moves in the three basic axes ( X, Y, Z) and exhibits a cubic work space located to one side on the robot, see figure [1.1]. The cartesian robot has the advantage of great precision in the ( X, Y, Z) coordinate frame but has the disadvantage of more limited orientation possibilities for its tool tip. The cartesian robot is mainly used for its precision in areas such as precision assembly.

The so called articulated robot can be thought of in terms of polar coordinates and exhibits a spherical work space that surrounds the robot, see figure [1.2]. This type of robot overcomes the disadvantage of limited work space and orientation possibilities of the cartesian robot but gives up its greater precision in the plane. The articulated robot because of its flexibility is preferred in applications such as arc welding and spray painting.

The final category is the cylindrical robot, see figure [1.3]. The cylindrical robot keeps two of the axes of the cartesian robot and one of the rotary axes of the articulated robot. In this way the precision in the cartesian axes typically (Y and Z) are retained and the work space is expanded by the ability to rotate about one of the axes typically ( Z ). This combination type of robot finds use in applications such as precision assembly.

McGill currently has three robots which all exhibit different joint configurations and fall into a different robot class. The PUMA 260 is an articulated manipulator with six revolute joints. The IBM 7565 robot is a cartesian robot with three prismatic and three revolute

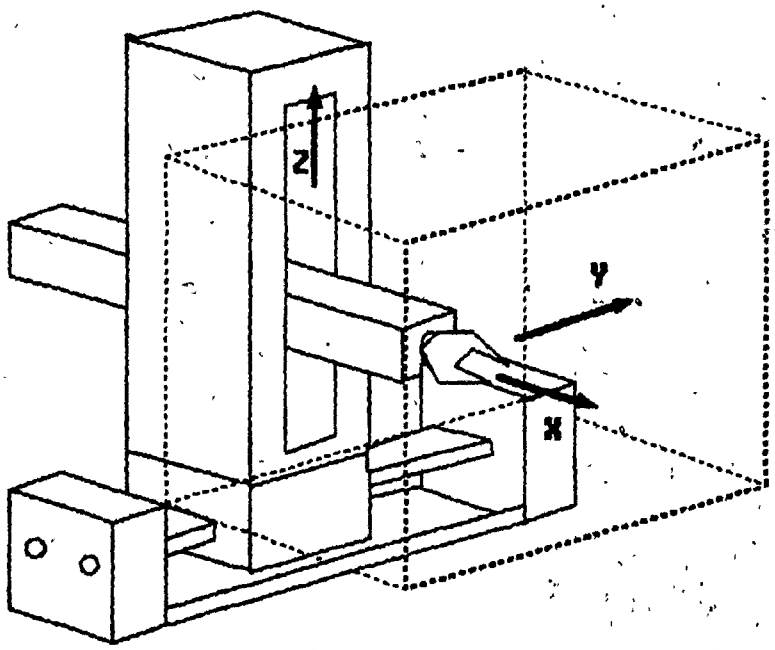


**Figure 1.1** A Cartesian Robot Configuration

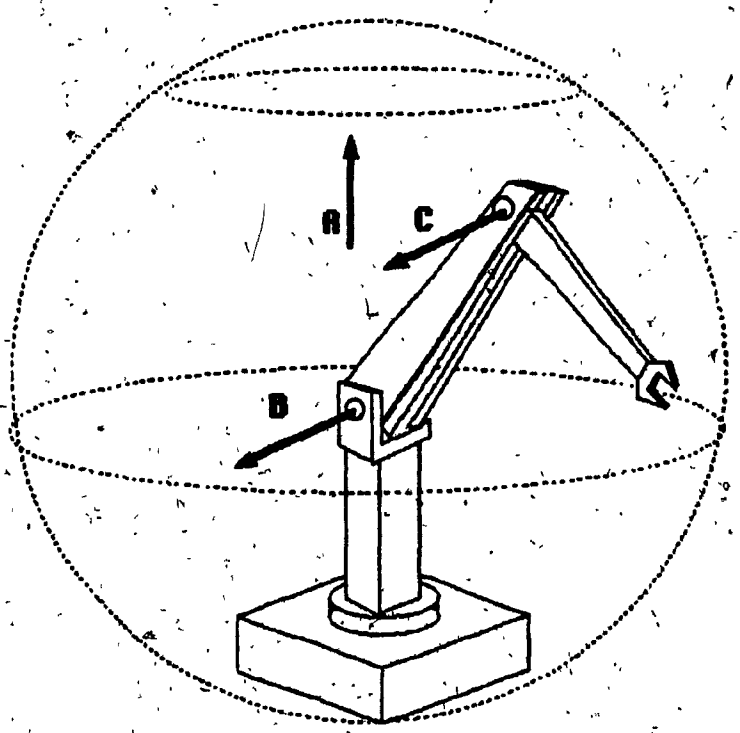


**Figure 1.2** Articulated Robot showing Spherical Work Space

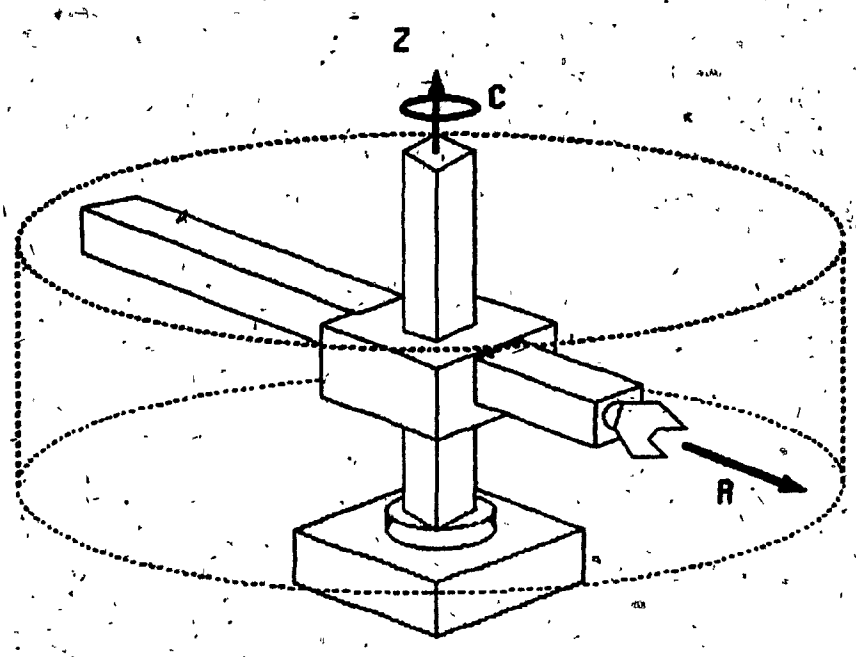


**Figure 1.3** Robot showing Cylindrical Work Space

joints, and the Microbo is a cylindrical robot with two prismatic and four revolute joints

The control of these various robots as can be imagined is a very complex task Typically each motor has a dedicated controller and these separate or slave controllers are supervised by a main coordinator which takes request from a user or application program and then relays commands or tasks to the slave processors Each slave processors is responsible for the monitoring and control of one of the servo joint motors. The division of the tasks in this way is vital as the control of the robot is needed in real time and thus speed of processing is a major concern The supervisor and slave processor architecture also allows for a very modular design to which more slave controllers may be added as needed.

As might be expected the many types of robots and controllers in addition to the many tasks and applications the robot is to be put, has led to many different languages and environments for the control of these robots. Some of the languages which have evolved will be summarized in the next section.

## 1.3 A Survey of Robot Languages

The earliest method of programming a robot was to guide it manually through a sequence of motions. These motions could be recorded and then played back to achieve repetitive tasks This pioneering form of robot programming was simple to implement and could be done without the use of general purpose computers The method, though simple, lacked several important features. For example, programs could not react to sensory inputs. Further, there could be no looping or branching, neither could computations be made.

The need for these missing features, essential to create more "intelligent" robots, was partially satisfied by the use of general purpose computers The fall in price and the rise in power of these general purpose computers allowed them to be incorporated into robotic packages, leading to the development of robot level programming languages.

### 1.3.1 Robot Level Languages

Robot level programming languages have the advantages of incorporating sensory data from such devices as vision systems, force sensors and external sensors into the control of the robot. However, in order to use the robot level languages, it is necessary for the

user to become a programmer and become involved in the details of robot motions and sensory based motion strategies. A variety of robot level languages have been developed, as outlined here below

The Stanford University Artificial Research Laboratory developed the language WAVE [Paul 5] in 1973, to program robots for research into the limitations of robotics theory. While this is a low level language, it served to pioneer important mechanisms in robot programming These included the description of end-effector positions in cartesian coordinates, the coordination of joint motions to achieve continuous velocities and accelerations and the specification of compliance in cartesian coordinates In the following year, the laboratory developed AL [Finkel et al 6] a higher level language, to facilitate programming assembly operations In addition to the manipulation capabilities of WAVE, AL is capable of executing concurrent processes and offers data and control structures similar to ALGOL.

IBM has developed a number of languages at their Watson Research Center The first of these, EMILY [Evans Garnett and Grossman 7] and ML [Will and Grossman 4], have been used in assembly tasks. AML [Taylor, Summers and Meyer 8] which was offered commercially in 1982, for robot programming work, does not support cartesian motion in a tool frame, compliant motion or multiprocesses However it offers a system environment, in which, robot programming interfaces may be built

SRI international developed RPL [Park 9] for use in facilitating control of machines in a work cell. It is implemented as a set of subroutine calls and may be viewed as a LISP type language cast in a FORTRAN syntax JARS [Craig 10] has been developed by Jet Propulsion Laboratory to control robots assembling solar cells The JARS language is basically PASCAL, with many robot specific types, variables and subroutines added Jet Propulsion Laboratory has also developed the language TEACH [Ruoff 11] which deals with concurrency in a systematic way.

The RAIL language [Franklin and Vanderbrug 12], which is an interpreter loosely based on PASCAL, has been developed by Automatix to control visual inspection and to carry out robotic assembly and arc-welding. HELP [G E.C 13], also an interpreter similarly based on PASCAL, is a robot programming language, announced by General Electric Company in 1982. It is best suited for use with cartesian arms as motions are expressed in terms of actual robot joints

VAL [Shimano 14] is a commercially available language which was developed by The Unimation Corporation. It is the programming language upon which this thesis is based and is described in detail in Chapter 2 A new version of this language, VAL II [Shimano et al 1], offers facilities for local area networking, real time control of trajectory, concurrent processing and synchronization and general sensory interfaces as discussed in chapter 2

The PAL [Takase and Paul 15] system, developed at Purdue University, represents tasks in terms of structured cartesian coordinates Motions are achieved as a side effect in solving position equations RCCL [Hayward 16], a continuation of this work, is a set of C subroutines used for controlling the robot

There are other languages which may be found in the literature. MCL [McDonnell Douglas 17] was developed for the US Air Force by McDonnell-Douglas. It is an extension of the numerical control machine tool programming language APT for the off-line programming of robots using a CAD data base

MAPLE [ Darringer and Blasgen 18], was developed by IBM, based on the capabilities of ML, but has never been much used. SIGLA [Salmon 19], developed at Olivetti, for the SIGMA robots is comparable to ML in its syntactic level and supports pseudo multi tasking and simple force control

MAL [Gini et al 20] was developed in Italy at the Milan Polytechnic. It is a basic like language and supports multiple tasks and task synchronization LAMA-S [Falek and Parent 21] is a VAL like language developed in France at IRIA LM [Latombe and Mazer 22] was also developed in France at IMAG It provides many of the facilities of AL, but does not support multi processing.

### 1.3.2 Task Level Programming

As the power of computers and the complexity of robot functions have expanded, a new set of user friendly languages has started to emerge. This new set of languages concentrates on specifying robot actions at a task level The task actions take the form of specifying goals for the positioning of objects As such, they may be robot independent but require extensive geometric data bases to model the world or the environment in which they are working These task level languages, described below, are very sophisticated in concept, but due to their complexity, they are not yet developed to the same extent as the

more simple robot level programming languages. No doubt they will develop and mature, as work advances in the field.

AUTOPASS [Lieberman and Wesley 2] is a task level language developed at IBM for semi-automatic programming It uses English like statements to specify assembly objectives rather than mechanical movements An extensive data base is required to relate the assembly objectives to the necessary mechanical motions. The data base is updated to reflect the state of the world at each assembly step.

RAPT [Popplestone Ambler and Bellos 23] uses the APT language, as a syntactic basis. It transforms symbolic geometric specifications into a sequence of end-effector positions. The language's main focus is task specification and does not deal with obstacle avoidance, automatic grasping or sensory operations.

The LAMA [Lozano-Perez 24] system was designed at M. I. T. The language is intended to formulate the relationship between task specification, obstacle avoidance and error detection Other advanced concepts have also been defined, but the language has not yet been developed

LM-GEO [Mazer 25] is a task level extension to LM It incorporates symbolic specification of destinations and with the use of ROBEX [Weck and Zuhlke 26], has the ability to plan collision free motions. However the full blown ROBEX system has not yet been implemented.

HIROB [Bork 27] is a high level hierarchical robot command language which features the use of simple English phrases to specify robot procedures. These phrases are parsed to determine what portion of the phrase is known or already defined Those portions which do not exist must be defined using either existing HIROB procedures (English phrases) or by using the primitive commands of the low level robot command language LOROB, which is part of HIROB An intermediate language MIDROB is also incorporated and used to control the logic flow in the HIROB phrases.

As indicated, the goal of these task level languages is to provide high level control of the robot, using English or English like phraseology. While this is very desirable from a user point of view, it is extremely difficult to design such a language, due to the ambiguity of the spoken word and the need to describe, in finite detail, every single physical feature likely to be encountered by the robot. It is not surprising, therefore, that task level languages

have been only partially implemented and the subject is a challenging topic of research in the field.

In summary it can be seen that there are many developed and developing languages for the control of robots The basic robot level languages in general offer control of whatever direct features the robot has to offer The more advanced offer some additional processing to achieve features the robots do not offer directly (for example cartesian motion on a cylindrical robot) and increased access to the outside world. The tasks level languages try to take the control of a robot to the extreme in that they try to allow the user to program a robot in English like language with out having to know any of the details of the control of the robot. This abstraction of the language from the robot offers the potential for creating robot independent programs and even task description that are almost independent of the parts they are to manipulate. Clearly these languages are striving for an ideal situation and is thus no surprise to see that many are only in their design stages.

It is the authors opinion that a compromise should be sought between the primitive low level control of the robot and the unimplemented high level task oriented languages. While the programmer should not be expected to know the intricacies of the control of the robot he should be expected to know that it is a robot that he is programming.

## 1.4 The Project

The repair project, at McGill, involves the automated repair of hybrid circuit boards. There are many aspects associated with this project Areas such as robot control, vision and sensory feedback, task description, tooling and material handling, inspection and classification of defects, are all essential in the goal of automating the repair of such boards.

Robotic control of intricate mechanical assembly processes is a technique quite unlike anything found in business computing The repair of hybrid circuit boards, presents the opportunity for exploration of assembly-directed programming, with the long term goal of bringing parts, fixtures and tools together in a natural manner to carry out assembly operations, with robot decision making capability

The actual repair project, at McGill, may be considered to be a logical subgoal, with the objective of discovering how parts are grasped and placed in fixtures, how components may be inspected, how general purpose feedback may be used in robot programs, what is

to be done and the extent of the repair necessary. The description of the motions required for such assembly and repair operations is a complex task with many subtleties, requiring extreme attention to detail. The project field is therefore appropriate for the examination of the problem of robotic motion description and the development of user friendly programming systems needed for precise robotic control.

This thesis is based on the development of an integrated robot programming environment. The following chapters organize the material by specific topic. Architecture and system hardware together with the work being done at McGill is presented in Chapter 2. In Chapter 3, the features of the environment are outlined and the commands required for motion, editing, debugging and vision processing are described. A discourse on the implementation aspects is given in Chapter 4. This includes details of the communication interface, parsing and execution of commands.

The system is demonstrated in an example program which implements a repair function in the hybrid integrated circuit board repair process. The features and the limitations of the system are discussed in Chapter 5. Finally, conclusions are drawn in Chapter 6.

# Chapter 2 Robotic Work at McGill

## 2.1 Introduction

The purpose of this section is two fold. First to describe the work being done in the Department of Electrical Engineering at McGill University in the area of assembly and repair of hybrid circuits and printed circuit boards. Second, to give an overview of the program environment developed by the author.

A Computer Vision and Robotics Laboratory (CVaRL) was started at McGill University in 1982. Since then the laboratory has become involved in the repair and assembly of printed circuit boards and hybrid integrated circuits, thereby developing the techniques of distributed processing, vision, world modeling and collision avoidance.

### 2.1.1 Distributed Processing

The concept of distributed processing is one in which separate agents are employed to carry out specialized discrete tasks. In order for the separate agents to function harmoniously, there must be information feed back to a central command post, from which commands are issued in a precise and reliable manner.

The importance of multi robot operation has been well recognized in the laboratory and in industry. While most tasks may be performed by employing one robot and some external holding fixtures, advantages of flexibility, speed and reduction in the number of external fixtures may be achieved with multi robot configuration. Moreover, the bottlenecks characteristic of a single robot design may be eliminated with distributed processing.

The current state of the art is to employ several slave processors designed to perform specific tasks being controlled by a master computer. There are several advantages to this configuration. These include greater fault tolerance, inherent with redundant elements in design, as well as ease of maintenance; modification and expansion due to modularity. These features all enhance performance.

Distributed processing networks are limited by the primitive communication systems presently available. [Gauthier et al 28]. Some research laboratories are addressing the deficiency. At the McGill CVaRL, work is being done to develop the communications technology needed to carry out inspections of hybrid integrated circuits, to identify defects and to make repairs in a three dimensional environment, abstracts on all of the current projects can be found in the CVaRL progress report [CVaRL 29].

The McGill CVaRL is developing a Session Layer for a local area network based on an ethernet. This Session Layer, which will be compatible with any programming language which can be linked to UNIX, will enable the end user to create links and end points between processes for the purpose of passing messages between processors. The object of this work is to make the network transparent to the user, thus promoting its easy use for distributed processing.

### 2.1.2 Vision

Computer vision is the process of converting an image or scene, into elements from which information can be obtained in an appropriate format for use by a computer. This has the potential to provide the best sensory input to any device which requires feedback for it to function.

Its vast potential is difficult to exploit because of the complexity of the conversion process involved. Nevertheless, because of the many development possibilities, a great amount of research is being done in this field.

One of the main focuses of research at McGill in this area, is inspection of hybrid integrated circuits and printed circuit boards as well as robot hand eye coordination. Research in inspection involves examining soldered joints and capacitor alignment on hybrid circuit boards.

In addition McGill has a rich library of vision software including the HIPS [Y. Cohen 30] and SPIDER [J.S.D. 31] software packages. HIPS is a set of C functions for image processing that was developed at the New York University. The package provides many filters for changing the image in the spatial-frequency domain. The SPIDER package provides a set of FORTRAN routines for performing similar functions on an image.

A line detection algorithm has been developed [Mansouri 32] which converts an image into a list of vectors which can then be matched to a model of a capacitor to obtain its location and orientation.

Work in hand eye coordination involves making end point corrections when inserting components into printed circuit boards [Mansouri 33]. For example, errors in obtaining components can be found and translated into correction factors for final placement on the circuit board.

### 2.1.3 Force Sensing

Force sensing involves joint, wrist and pedestal activities, which may be employed to supplement vision in providing local information for such tasks as grasping components and inserting them into sockets and circuit boards. Force sensing may also be used for contour tracing and in grinding functions.

In joint sensing, forces are monitored at each pivot or prismatic point in a robot mechanism and these may be transformed into other forces relative to a coordinate frame. Wrist sensing incorporates a sensor between the robot's last joint and an end-effector from which the forces in the wrist can be measured. In pedestal force sensing, the stand registers the forces applied to it. The capability to sense, interpret and utilize such information is critical to the smooth execution of tasks involving delicate handling. Further improvement may be achieved by the use of sensing pads attached to the robot's fingers (as in the IBM's strain gauges) to detect forces while grasping objects.

McGill is investigating the use of a force sensing device which can be mounted in the robot's wrist and used to resolve forces and torques in and around the X, Y and Z directions. The device may also be mounted under a platform and used in a similar manner, but with the advantage of freeing the robot of the weight of the device.

Experiments have been carried out in joint force feedback by monitoring the currents flowing through each of the joint's motors. This has been achieved through the use of the force primitives in the Robot Control C Library ( RCCL) language.

### 2.1.4 World Modeling

The successful performance of robots, in an artificial intelligence environment, depends on a versatile and dynamic learning system supported by a rich knowledge base, which has the capability to expand its knowledge without external intervention. Such a world modeling system, learns by proving rules, analyzing the proof, refining them and storing them in the knowledge base automatically. [Xu and Chen 34].

The use of a data base to model the robotic environment has the advantage that it may be manipulated by computers far easier and safer than the actual environment. Thus it may be used in areas such as collision avoidance and assembly procedures.

Depending on the use of the model, different representations may be desired. For collision avoidance, objects may be simply modeled by their bounding cubes, while for assembly or mating operations, more detailed representation is required.

### 2.1.5 Collision Avoidance

There are a number of techniques available to the researcher to implement collision avoidance systems. Obstacles may be detected by the robot with vision, radioactive, capacitive, magnetic or ultrasonic sensors. Of these, ultrasonic sensors are favored, [Mack 35] because they are lightweight and inexpensive, being most suitable for most industrial assembly situations.

### 2.1.6 The Robot Languages at McGill

An overview of various methods of programming robots was discussed in the introductory chapter of this thesis and some of the existing languages were described.

At McGill, there are three robots, each with its own native language. The PUMA 260 runs VAL, the Swiss Microbo runs IRL and the IBM runs AML.

At present, the PUMA and the Microbo robots have overlapping work spaces and may be used together, on separate projects in the same work space, or in cooperation with each other on a single project. While some low level form of cooperation may be achieved using I/O ports as synchronization lines, this technique is not considered adequate for sophisticated cooperation. A more structured form of message passing would be needed so that more than the go/nogo type of synchronization could be achieved.

A more powerful language RCCL is being investigated for use by the PUMA 260 and Microbo robots. This language has routines capable of processing force sensing information in real time, which offers greater control over the robot. Additionally, time varying functionally defined transforms can be used to control the robot's trajectory. This language has already been installed, at McGill, on the PUMA robot by [Llyod 36] and on the Microbo by [Kossman 37].

### 2.1.7 Project Goals

As discussed under distributed processing, the system of slave processors controlled by a master computer offers substantial advantages in the management and control of work tasks. With the diversity of equipment available to the programmer, as described above, it is apparent that the environment should be one in which the programmer is free to be creative, concentrating on the tasks required to be performed by the robot rather than being involved in repetitive routines.

Accordingly, creation of a superior environment was enthusiastically selected by the author as the project to be covered by this thesis. The desired goal is to encourage development of technology to speed up operations, minimize errors and simplify debugging routines with the objective of making dramatic advances in robot programming productivity. More specifically the following are the main goals or features set for the environment.

(1) Rapid turn around time was considered necessary to replace the laborious technique of first editing, then compiling and finally running the program, with associated inherent opportunities for error.

(2) The ability to control all of the available equipment from a central programming environment. The user should be able to control and synchronize the robot, stage and vision processes.

(3) Increased debugging facilities to simplify program development The programmer should have the ability to test any command individually and to single step through a program in order to observe its performance, as well as to change and modify existing programs.

(4) The elimination of syntactic and spelling errors through immediate parsing of command lines was also considered to be important for quick and easy development of programs.

(5) The environment should be modular and easy to add to and adapt to specialized situations.

The rest of this thesis will deal with the description of the implemented environment that achieves such goals.

## 2.2 System Overview

One of the major advantages of robots is their great versatility. They may be used in a number of different tasks without many modifications. This great versatility is mainly due to the generality of the robots physical structure and control [Sanderson 38]. However much of this flexibility is lost due to the difficulty in programming the robots. Often robots are designed as stand alone systems, are low cost and offer primitive facilities to develop and modify programs as well as the ability to deal with the external world. In this thesis an interface to such a stand alone system has been developed. The system is a Unimation PUMA 260 robot running VAL. The interface offers an enhanced environment well suited to both the on line as well as the off line development of robot programs.

A block diagram of the experimental repair station can be seen in figure [2.1]. The repair station integrates a PUMA 260 robot running VAL, an X-Y stage, microscope, Grinnell monitor and a VAX 11/750 host computer. The user sitting at the host computer can control any one of these devices. Images obtained from the microscope can be displayed on the Grinnell and processed by the host computer. Storage for images, robot positions and programs is provided by the host's disk drive

Communications to the PUMA is provided by a RS-232 link, with the host replacing the PUMA terminal. Access to the I/O module of the VAL controller can also be obtained via this link. The X-Y stage and microscope are controlled by a stepper motor controller

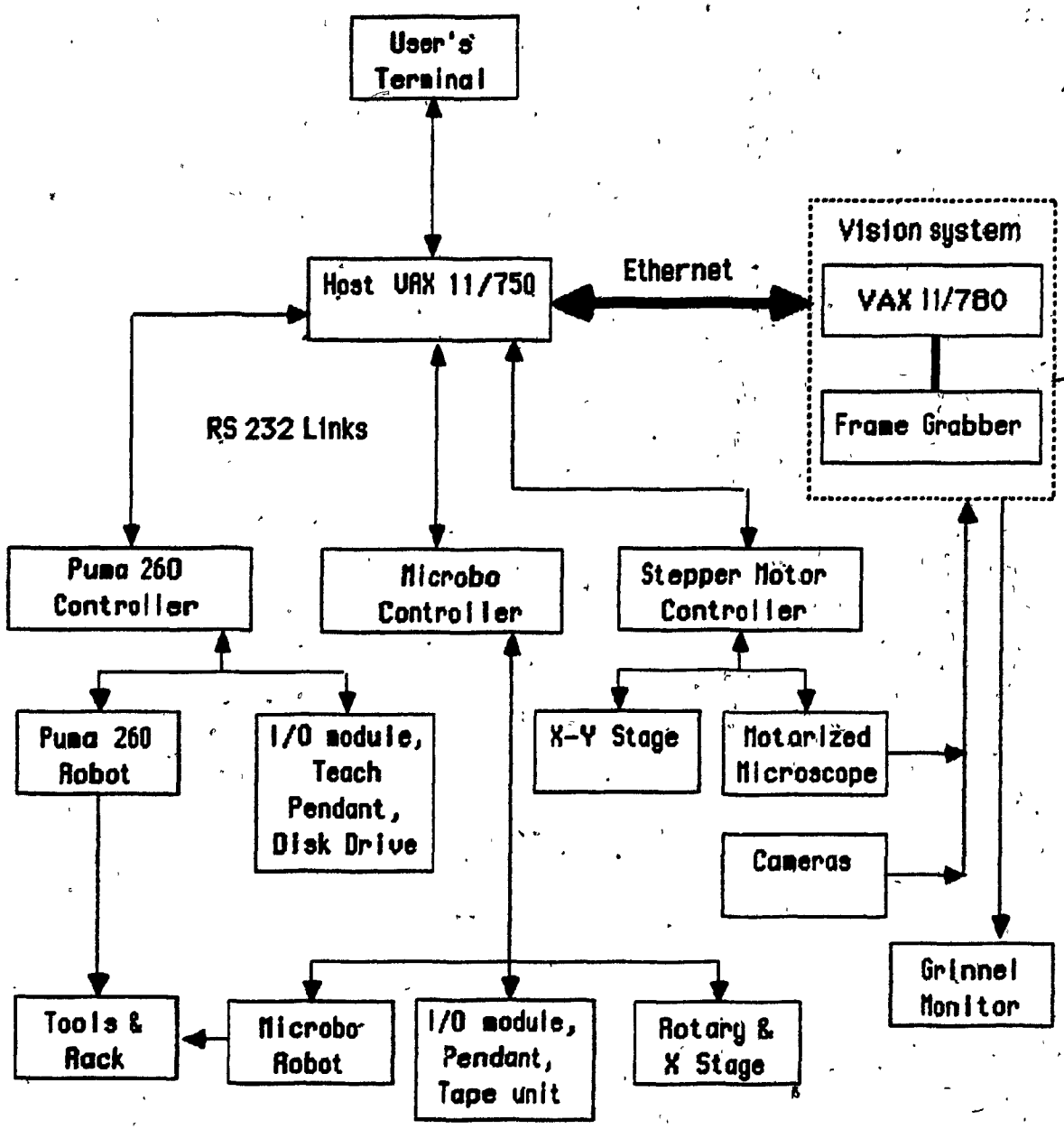


**Figure 2.1** Block Diagram of Repair Station

which also interfaces via a RS-232 link. Further development to include a Microbo robot is underway. The Microbo robot has already been interfaced to the host computer and all that remains to be done is to incorporate it into this programming environment.

### 2.2.1 The PUMA Controller

The PUMA 260 robot is controlled by an LSI 11 control computer which is also used to control, the binary I/O sensor and relays, the PUMA teach pendant and disk drive. This robot and controller package, which is manufactured by Unimation, is also used to run the VAL language. This package offers no external computer interface, therefore for a host computer to talk to the PUMA with out modifying Unimation's hardware it must do so through the terminal port The terminal port is a RS-232 port normally connected to the user's terminal. The host computer was placed as shown in figure [2 1] between the user's terminal and the LSI 11 controller. A set of C subroutines was then developed on the host to relay commands and messages back and forth between the user's terminal and LSI 11 controller so that the host computer terminal can act as if it was connected to the PUMA controller.

This method of interfacing the host and LSI 11 is somewhat awkward but it should be noted that the package was designed as a turn key stand alone system and as such was considered to be sufficient to satisfy the prevailing need. Furthermore there is no standard interface available from any other manufacturer. It is expected that this deficiency will be overcome as the industry matures

The use of the host computer provides useful extensions to the VAL language For example, the host can be used for enhanced numerical calculations, processing of points and trajectories, as well as superior processing of sensory inputs from the I/O module In addition, complex data from such devices as cameras and microscopes may be processed and incorporated into the robot program

The host computer used in this project is a VAX 11/750 running UNIX 4.2 BSD. This computer was acquired by McGill for robotic research and is being used on a number of projects.

The UNIX environment makes the programming language " C " a natural choice for program development. C is a general purpose programming language developed on the

UNIX system [Kernighan & Ritchie 39] It is a relatively low level language, that can be learned quickly. The language is quite efficient and can be ported to other machines with little difficulty. It provides the constructs for creating structured programming, and functions or procedures may be written and compiled in separate files and linked later, allowing for a degree of modularity in program development.

### 2.2.2 The PUMA Robot

The PUMA 260 Robot is an articulated manipulator with six degrees of freedom manufactured by Unimation Inc. [Unimation 40], a diagram of the PUMA can be seen in figure [2.2]. Each joint is controlled by a DC servomotor, each containing an incremental encoder mounted on its shaft The encoders provide position information relative to a known initial absolute position The "nest" position of the robot is used as the initial reference position. The controller obtains data from the encoders and calculates velocity from them The work space of the PUMA is roughly spherical as shown in figure [2.3]. This configuration of totally revolute joints allows for very flexible movements of the arm and a stated accuracy of repeatability of 20 microns. However due to wear of the robot through much experimentation the accuracy has diminished somewhat.

### 2.2.3 The Stepper Motor Controller

The stepper motor controller is used to control two devices, an X-Y stage and a microscope. The Controller actually controls four motors; Two are used to move the stage and the other two control the focus and zoom of the microscope [Mansouri 41].

The X-Y stage as the name implies is a platform that may be moved in the X and Y directions. This stage may be moved in increments of six and a half microns, thus allowing for very precise positioning in the plane. This is ideally suited to the assembly and repair of electronic components

The microscope's zoom and focus may be controlled by a user program through the stepper motor controller The microscope's height above an object may be changed, to achieve focusing, and the zoom control knob rotated by the two motors The use of this microscope provides necessary visual feedback in tasks dealing with small electronic components.

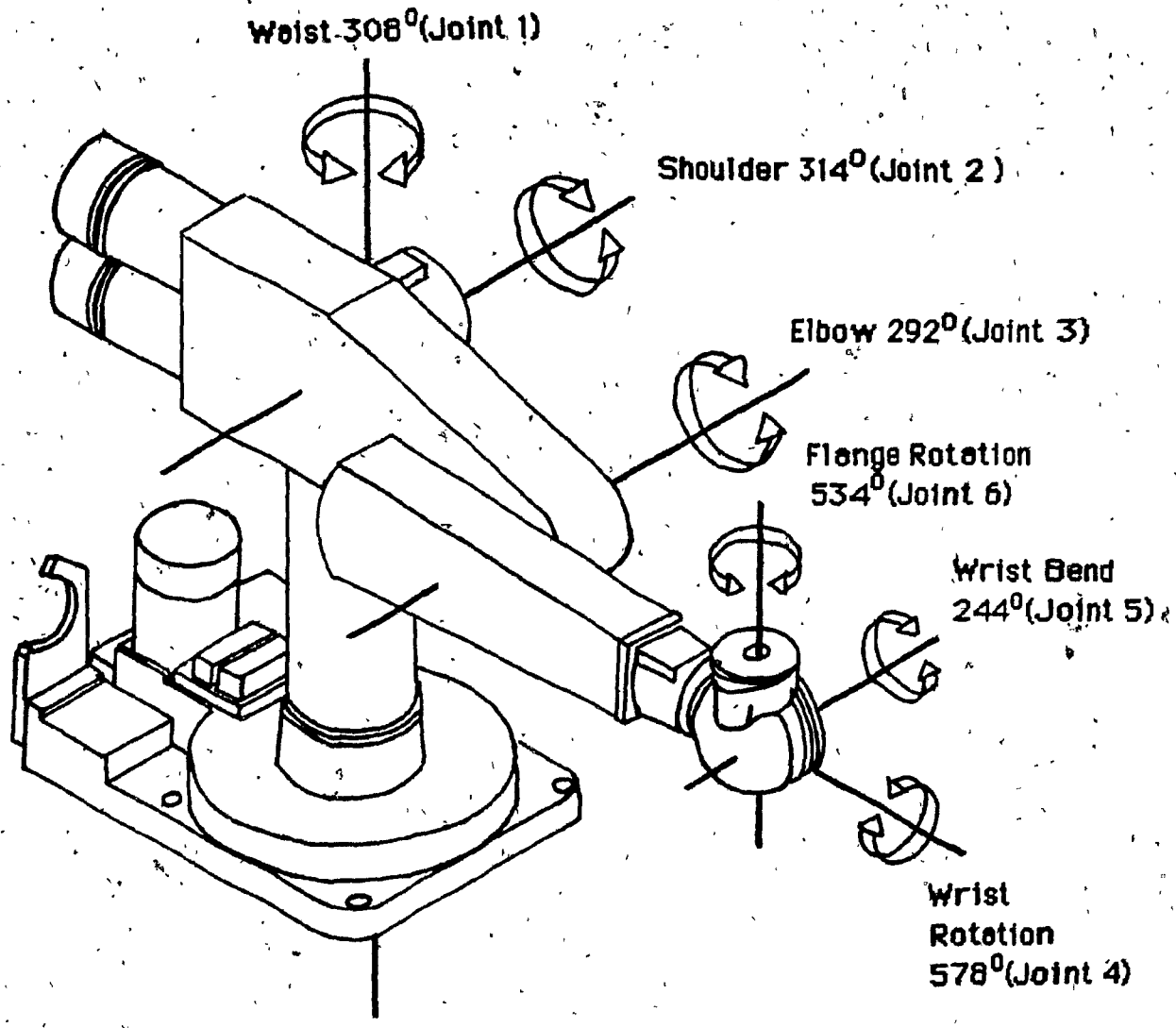


**Figure 2.2** The PUMA Robot

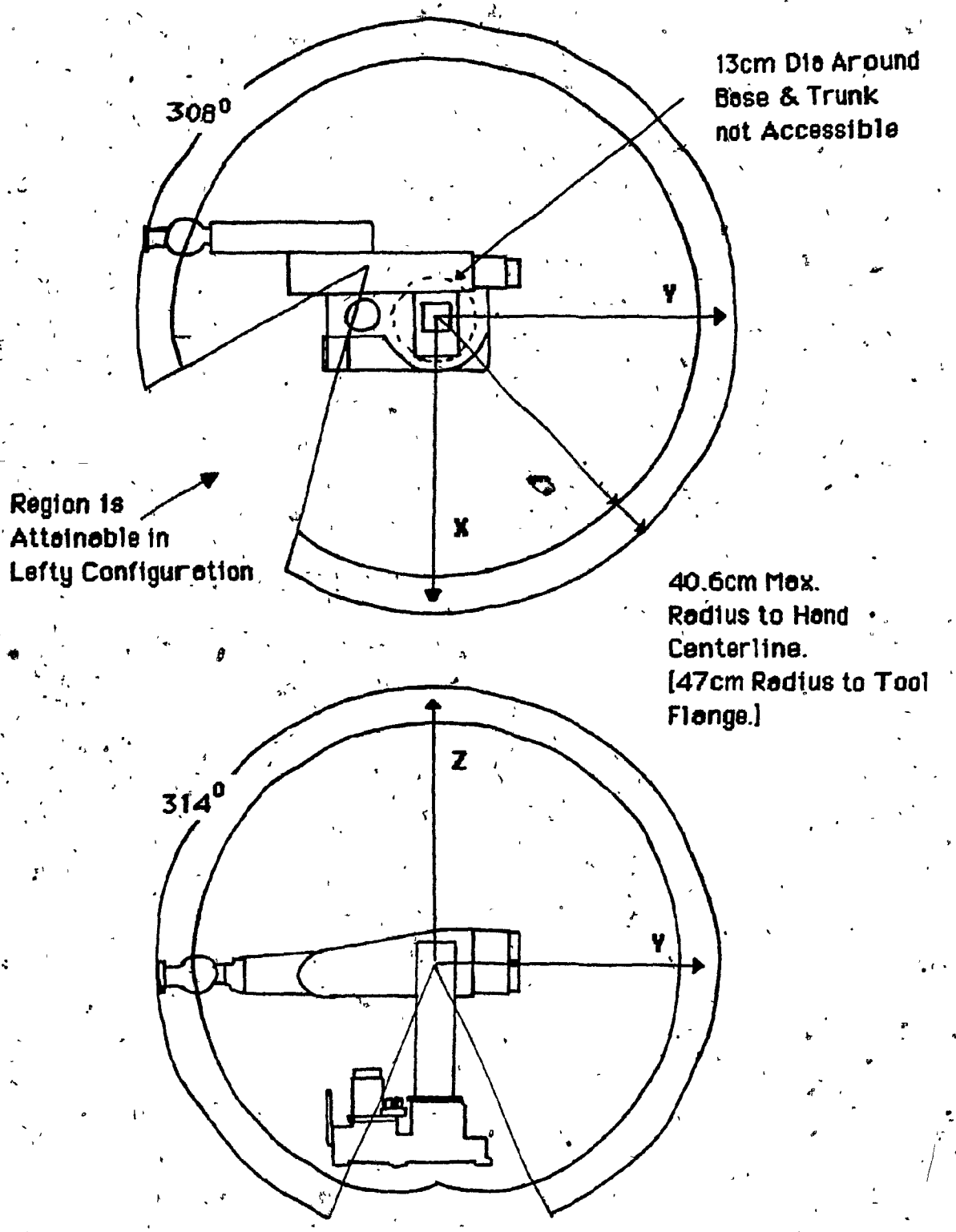


**Figure 2.3** The PUMA's Work Space

### 2.2.4 The Grinnell Monitor

The Grinnell monitor is a 256 by 256 color monitor used to display images obtained from the microscope This monitor is controlled by a VAX 11/780 host computer. Images must then be sent over a network from the VAX 11/750 to the 11/780 to be displayed on the monitor. This transfer of data is slow and so limits the use of the vision feedback to non real time uses Work is being done to interface a dedicated 512 by 512 Matrox frame_grabber and color display monitor to a dedicated Intel system 310 with multiple vision processors. This system in turn will be linked to the VAX 11/750. The Matrox/Intel system will offer higher resolution and faster execution times, in the order of a few seconds.

### 2.2.5 The Microbo Robot

The Microbo robot is Swiss made [Microbo 42]. Like the PUMA, it has six degrees of freedom, but differs from the PUMA in its architecture While the PUMA is an articulated manipulator comprising solely of revolute joints the Microbo has four revolute and two prismatic joints, as shown diagrammatically in figure [2 4]. The prismatic joints are structured in such a way as to allow considerable improvement in the precision of motions, in the radial and vertical directions, compared to the PUMA. (5 microns in the case of the Microbo and 20 in the case of the PUMA) However its work area is restricted to a dough-nut shape, see figure [2 5], which limits its use compared to the larger and more comfortable spherical work area associated with the PUMA

Like the PUMA it runs an interpreted language called IRL (Intuitive robot language), thus Interfacing this robot to the host was achieved in a manner similar to that of the PUMA. Incorporation of this robot into the authors environment is thus straight forward. However, routines will have to be developed to implement necessary cartesian and other motions, essential for effective use of the robot to be achieved from the environment.

### 2.2.6 Interchangeable Tools

Having the right tool for the job is vital if meaningful work is to be performed by the robots. A variety of tools have been developed by the author and others in the lab for use with the PUMA and Microbo robots. These tools all feature a standard mechanism for

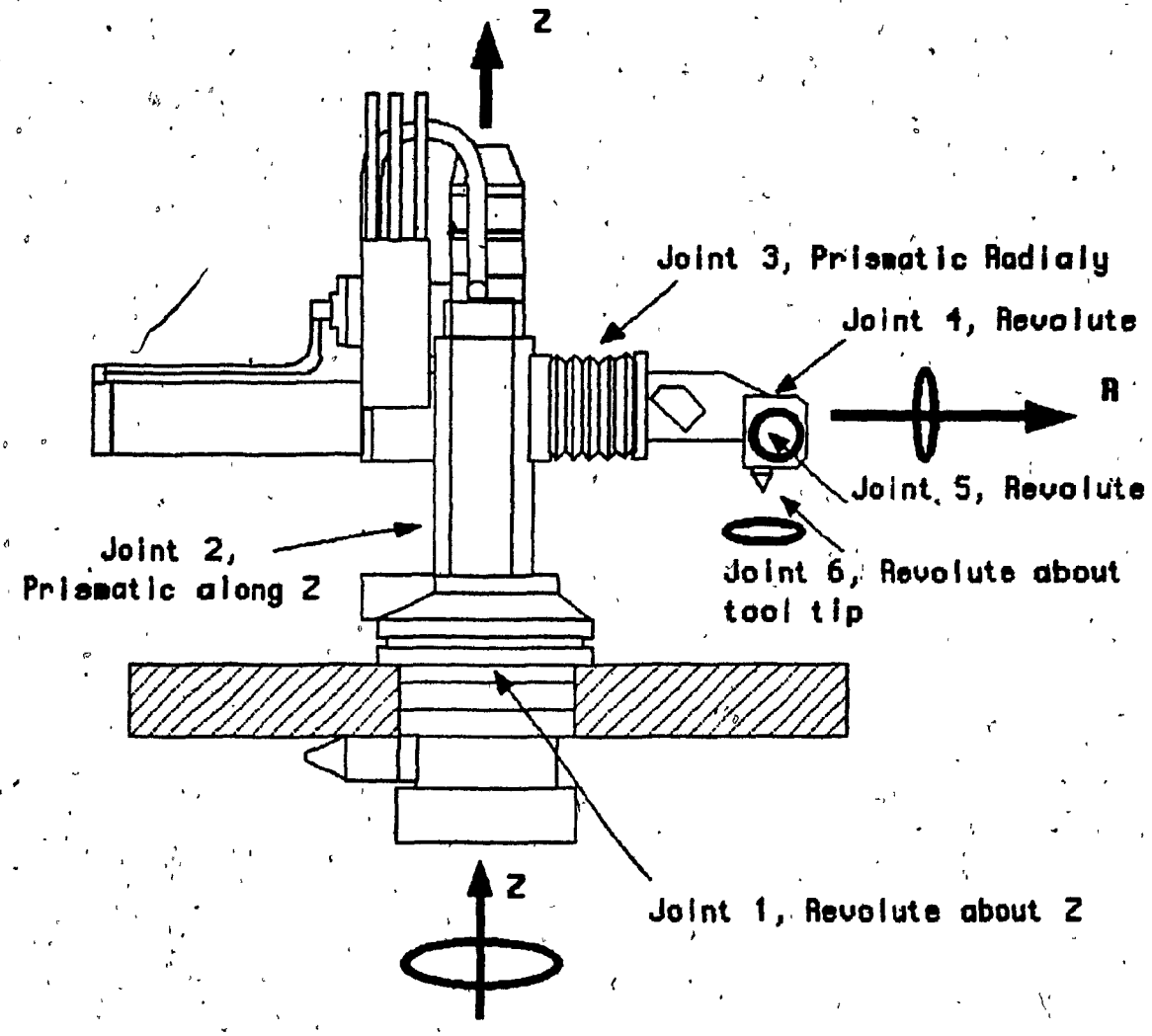


**Figure 2.4** The Microbo Robot

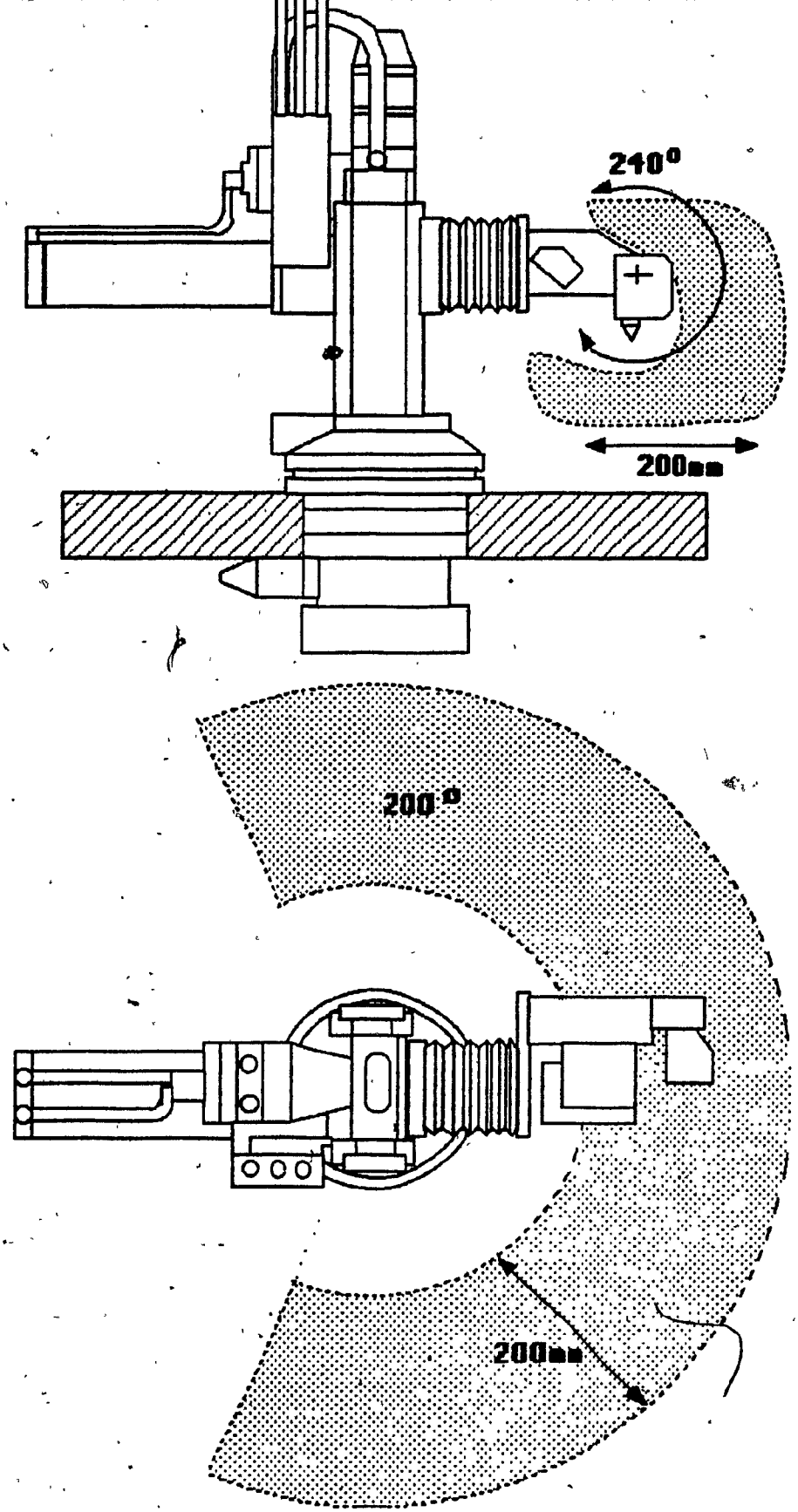


Figure 2.5 The Microbo's Work Space

attachment to the manipulators and are thus interchangeable. Each tool has been designed with its own stand, which assures correct alignment of the tool when it is not in use. Specialized tools for grasping capacitors, hybrid integrated circuits and chips etc., have all been developed. Tools have also been designed and made to carry out such tasks as grinding, continuity checking and solder paste dispensing. These are fully described, in chapter 5 with diagrams accompanying those designed by the author.

## 2.3 The VAL language .

VAL is a rudimentary programming language designed for industrial robots. It has the capability to interactively edit, interpret, debug, execute and store user programs. It has been designed primarily for operations involving predefined robot positions and is ideal for pick and place operations with human interaction. Where feedback is required, VAL cannot be used without addition, because it lacks the facilities to process complex sensory input.

The basic capabilities of the VAL language, are listed by Tomas Lozano-Pérez, in his paper on Robot Programming [Lozano-Pérez 43], and described by Bruce Shimano [Shimano 1]. It is necessary for the reader to understand the value and limitations of these capabilities in order to appreciate the improved environment developed in this project. For full details of VAL the reader is referred to the VAL user's manual [Unimation 44].

VAL is an interpreted language, that is, commands can be run with out the need for timely compilation to an executable format. A variety of commands are available, which may be run directly from the monitor or stored and edited in a program file The editor stores the commands in an internal format for quick interpretation and compact storage

Point to point motion commands are used in situations where only the final position of the end-effector is important and where the path taken by the manipulator may be disregarded. This type of motion on the PUMA is joint interpolated motion, that is, all joints complete their actions simultaneously This is achieved by interpolating the control variables between the initial and final positions of the joint. The time of motion is set according to the time required for the slowest joint to complete its motion. The advantage of this type of motion is that it provides the fastest controlled trajectory, but the tool tip often moves along a complex space curve, which is a limiting factor in operations requiring motions relative to an external object.

To overcome this limitation, cartesian motion commands are available to move the tool tip along specified paths. This kind of motion is useful for generating straight line paths and reduces the number of positions that would otherwise have to be taught, by utilizing motions relative to a coordinate frame. However, to achieve such paths, joints may have to be moved more than is necessary for the task. Moreover, since a constant velocity is implemented, joints may have to be frequently accelerated and decelerated, an unnecessary waste of energy.

In addition to these two types of motion commands, described above, the manipulator may also be moved in various other ways:

- The manipulator may be moved incrementally to perform departures and relative motions.

- It may be moved to a position relative to a defined point, useful for approaches

- Individual joints may be driven by a specified number of degrees.

- The opening of the manipulator's hand may be controlled.

- Parameters used to control the trajectories, including the speed of each motion, may be changed.

VAL has the capability of specifying coordinate frames and motions relative to these frames in addition to some manipulation of the frames. Most motions are relative to the world coordinate frame, situated at the base of the robot, which may be offset or rotated about its z axis. It is frequently more important for the end point of the tool to be precisely positioned than the manipulator itself. VAL accommodates to this need by defining a tool transform which can be used to change the description of tools being used by the manipulator as they are changed. However, a need to move in the tool coordinate frame, while supported by the teach pendant, is limited to tool Z motion in the VAL language.

Integer arithmetic is accommodated by the normal operators of addition, subtraction, multiplication and division. This together with the facilities of branching, labels, and comparison tests provide the mechanism for looping and indexed operations such as palletization. Subroutines are also provided but they lack any argument passing thus making them somewhat limited in their application.

VAL has several ways to allow the teaching of points or transforms to be used in programs. The teach pendant may be used to manually steer the robot to the desired

position and that place then stored with the HERE command. Alternately any of the robots joints may be freed so that they may be manually pushed and moved to a desired position. VAL also provides a semi-automatic method of teaching a series of points to the robot and have the robot automatically generate move statements to move between the taught points, thus a path may be taught in a fairly automatic way.

Finally the binary inputs and outputs can be controlled and monitored, providing a means for synchronizing external devices with program execution. Interrupt service routines may also be set up to start execution when one of the binary input lines is activated. This allows for guarded moves, that is, motions may be stopped or modified by the activation of a sensor.

As can be seen VAL offers control of many of the PUMA's features and as a stand-alone system performs rather well. However if the Puma is to be used in a more complex environment extensions to the language are needed.

### 2.3.1 VAL-II

VAL-II a successor to VAL offers many important extensions needed for a more complex environment. The main improvements over VAL are as follows. VAL-II has a formal communications capability, flexible path control, general sensory interfaces, and improved computational facilities [Unimation 45].

As has been seen robots in todays applications seldom operate in isolation, making the need to communicate with external devices and computers essential. The facilities offered by VAL were of the simple on/off go/nogo type of binary interface mechanism. In VAL-II communication can be achieved via a formal network. This interface allows for the complete supervision of the robot system by a remote computer. The network protocol is based on Digital Equipment Corporation's DECNET communications network system. This allows for error detection, retransmission of faulty data, and the ability to use inexpensive RS-232 serial lines for communication. The remote computer can issue all VAL-II commands normally available to the user, communicate with user programs to provide and collect data, as well as upload and download programs, and monitor the status of the system.

VAL-II offers a much more flexible control of the robot's path than did VAL. Besides the standard VAL joint interpolated, and cartesian motions VAL-II has the ability to generate

functionally defined motions, such as circles and arcs, by having procedures calculate many short motion request, from which VAL-II will smooth using its continuous path feature. In addition to this procedural motion VAL-II also has a real time ability to accept motion corrections from some external device and alter its intended motion by these correction factors. The corrections are given in terms of their X,Y, and Z components and may be used by the controller in a cumulative or non-cumulative fashion. The corrections may also be specified relative to the world or tool coordinate systems.

The real time ability to alter robot paths leads directly into the ability to accept or control sensors. The sensory interface consists of the 19,200-baud serial line for the access of corrective data from an external device. This line can also be used to monitor the robot's position in real time. In addition to this line the binary and analog inputs and outputs can be controlled and monitored not only by single instruction commands, but by process control programs which run concurrently with the main user program. Such process control programs have access to all the VAL-II commands that do not cause motion. They may monitor and control I/O lines and modify program variables or halt the robot This allows for more complete control of sensory data.

The teach pendant can now be accessed and used by user programs. This allows the user to create flexible systems using the teach pendant to guide the positions needed by a program. This is done through what is called detached motion control. In detached motion control the users program can release control of the manipulator while still executing. This enables all manual control modes to be used from the teach pendant while the user's program runs and prompts for positions to be taught. There are even facilities to redefine the effect of the buttons on the teach pendant.

The computational power of VAL-II has also improved, integer arithmetic has been replaced by floating point, and real valued functions and predefined system constants added. The increased ability to handle computations adds greatly to the procedural motion subroutines.

All of these improvements over VAL make VAL-II a very interesting system to interface to a host computer. Unfortunately VAL-II was not available at the time of this thesis research, and so the interface was done with VAL. However it will be noted that many of the enhancements of VAL-II can be found in the implemented system.

# Chapter 3 The System Environment

This chapter describes the system environment for programming the PUMA 260 robot as seen by the end user. The user is basically concerned with the functionality of the environment and how a program may be created in it. Accordingly, the various commands available to the user are described. These include commands for motion, vision; and movement of the X-Y stage and microscope. Techniques for file management, debugging and editing are also described.

This chapter also indicates the types of problems likely to be encountered by users, the level of help available in each case and the methods of teaching transforms and positions used by the robot.

## 3.1 Programming Menu Concepts

It is recognized that most users of robots desire a friendly environment, which can best be achieved by menu programming. With such a facility, the user is in a position to be guided by prompts in selecting commands from a menu. Every executed command results in syntactically correct inputs and desired changes may be made with ease.

Work on menu programming has been done by [Gomaa et al 46] to achieve an interactive programming environment for all stages of program development. Their system, as does this system, permits the user to select commands while having access to all functions at all times. The user may switch between executing commands, debugging and editing. Commands may be selected by working along branches of a menu tree, which may be accessed by selecting the appropriate group of commands and working through submenus.

For example, the motion group of commands contains a submenu which defines various arm motions, such as joint interpolated, and straight line motions.

The author's work also resembles that being done by [Kirschbrown & Dorf 47] on KARMA (A Knowledge based Robot Manipulator System). Their system is a menu driven system which uses the menu and graphic capabilities of the Apple Macintosh to provide a pleasant user interface. The system has an associated knowledge base to aid the user in creating programs. If the system can not find the necessary information in the data base the user is prompted for it. While the system is not connected to a real robot, robot actions are instead simulated graphically on the screen. A combination of this graphic interface and simulation along with the actual control of a robot and its associated devices is the eventual goal of any high level robot programming environment.

### 3.1.1 System Features

As previously intimated, a system comprising of diverse equipment for robotic applications depends on proper supervision and the collection and interpretation of data. The advantages of a centralized environment for supervisory control and data acquisition are reflected in better management and improved productivity. The system developed by the author is a successful implementation of such an environment.

The system has been entitled the RAP (Robotic Applications Programming) system to indicate its ease of communicating with the PUMA robot in VAL, the X-Y stage and the microscope. Additionally routines providing vision primitives permit data from a vision system to be obtained for use in a robot program.

RAP permits the user to see the progress of the robot's movements as the program is developed. That is, in one of its modes, RAP will execute every command as it is entered, this is especially helpful in developing sequential portions of a program as the robot is always only one step away from the next step to be programmed. The user then only has to visualize one program step at a time. This means that the user is free to concentrate on the task of optimizing the robots movements without the distraction of programming details, or visualizing where the robot might be after a long sequence of commands.

### 3.1.2 Command Features

Commands may be ordered from the menu by typing command names as desired. These typed commands are parsed by a key tree matcher, whenever a space or end of line character is typed. On comparing the command ordered with those available in the menu, the matcher will automatically complete the desired command, as soon as it can be uniquely identified. This means that most commands may be ordered by typing only their initial letters and a space. This minimizes the typing task, while simultaneously eliminating spelling errors.

Additionally, the matcher will help the user at any stage of the proceedings. Whenever a question mark character is typed, all subsequent possibilities which are then available will be listed on the screen. This means that a list of all commands may be generated by typing a single question mark character, or all commands starting with a particular letter may be listed when that letter is typed and followed by a question mark.

Some commands cannot be completed without user specified arguments In such cases the matcher will wait for the arguments and ensure that they are of the correct type and appropriate range before executing the command. In a manner, similar to that described above, a question mark character may be typed to obtain help with the argument being awaited

Various modes of operation exists in RAP. Commands may be run immediately, on line, with the option for recording the commands simultaneous with their execution. Alternatively, commands may be recorded in an off line mode This latter alternative frees the equipment while programs are being developed, which is an advantage under conditions in which the work station equipment is being utilized The former option offers ready verification of each command as ordered This is useful in sequential work, because observation of the robot's movement simplifies the programming task.

Recording commands, as they are executed, provides a number of interesting features. If the user is not satisfied with the result of a command executed, a change in the robot's actions may be desired This possibility is provided for by allowing the user to backup the robot as far as may be needed before commands are reissued. The incorrect commands may then be completely overwritten or supplementary commands may be inserted to achieve the desired modifications.

After recording a set of commands into a program file the user may wish to observe the execution of this command file in its entirety. This may be done in a number of ways. The user may run the program at normal speed or single step through the program line by line. At any point in execution the user may abort, pause or even backup through the program. Pausing a program is very useful when runing an old program in which points may require updating. These programs may be paused as they are about to reach the old point. the new one taught and the program resumed Backing up a program although having no logical significance is extremely useful in debugging a program as the robot may be backed up for a command or set of commands to be retried.

### 3.1.3 Debugging and Editing

In addition to these quite powerful tools for controlling the flow of a program, points to break the flow of a program may be set This feature is useful in debugging, and for adding demonstration pauses to a program. These points called break points can be activated or deactivated before or during a program run

Programs need not be independent They can be created to be subroutines or procedures These subroutines unlike VAL can take arguments A typical subroutine may be created to pick up an object The object to be picked up, the method of pick up and any special approach to be used in picking up the object may all be specified via arguments to the pick up subroutine. This provides the opportunity to create quite complex and powerful subroutine libraries

In addition to subroutines, branching and conditional testing are provided for These additional facilities may be used to create loops, to branch on user input and sensory data or to handle error conditions. Sensory data may be acquired from the PUMA's I/O module, the microscope or the camera system.

Editing of recorded programs can be done after or indeed during their development The editing is line oriented That is the user has the ability to change, insert, delete or list command lines In the cases of deleting and listing command lines a range of command lines may be specified so that repetitive commands to delete a series of lines need not be given When inserting a line, RAP automatically changes to insert mode where all further commands are inserted until the user terminates this mode with the "insert off" command

as described later.

## 3.2 System Commands

Having introduced the main features of RAP the commands will now be described. The commands are sorted according to their function and grouped with other commands which offer similar actions

The first set of commands to be described are those used to control the communication channels to the various devices The commands that define locations for the robot are discussed next, followed by commands which cause motion of the PUMA to these locations. Next are the commands for moving the X-Y stage and microscope. Following the motion commands are commands for debugging and editing, conditional branching, and vision processing, as well as other implemented VAL commands The last section of commands are a special group of commands developed to tailor the environment to the task of hybrid circuit board repair The commands are listed by section in alphabetical order and are shown in **bold** type, arguments to the commands are given in *<italics>* and enclosed in angle brackets The values that the argument fields may take on are also give in the text in *italics*.

### 3.2.1 Channel Commands

**OPEN** *<channel>*

Opens a channel for communication. *<channel>* may be *val* or *stepper* If for some reason the channel can not be opened the user is informed of the failure by an error message printed on the user's console The VAL channel is used to talk to the puma running VAL. The OPEN *VAL* command must be issued before the robot will respond to any robot command or a channel "unopened" error will be generated. Similarly the stepper motors will not respond to any stepper command until the OPEN *STEPPER* command is given

| | |
|---|---|
| OPEN *VAL* | The robot channel is opened. |
| OPEN *STEPPER* | The stepper channel is opened. |

**CLOSE** < *channel* >

Closes a communications channel, as in the OPEN command the <*channel*> can be *val* or *stepper*. Any errors encountered in closing a channel will be reported to the user.

**EXIT, QUIT**

The system is exited. The exit and quit command do the same thing, both are provided for the user's convenience.

**START_VAL**

This command starts up the PUMA controller runing VAL. It should be given after the OPEN_*VAL* command. It prompts the user to turn on the controller if it is not already on. It then responds to the VAL start up prompts and prompts the user to turn on the power to the robot arm and set the robot to computer mode. The PUMA robot is then calibrated and placed in its "ready" position.

**STEPPER_INIT**

The stepper motor controller is initialized and the stage and microscope driven to their home positions. It should be used after the OPEN *STEPPER* command. It sets such parameters as the step size (half or full), the acceleration and deceleration speeds and the maximum cruise speed of the motors. This command will also reset the stages coordinate frame to its home position. The home position of the stage is when the stage hits both its X and Y limit switches which is considered its (0,0) position. The home position of the microscope is when the height of the scope is greatest and the zoom factor is least. The command may be used at any time to reset the stages coordinate frame if its accuracy is in doubt.

**VISION**

The Vision command takes the user into the vision subtree where all the vision commands can then be executed. Once in the vision subtree all editing functions must be performed with their control key equivalents as explained in the debugging and editing section.

### 3.2.2 Point and Location Commands.

These commands deal with the teaching of points and transforms Points and transforms are VAL's way of representing locations in the robot's space While a transform and a point may represent the same physical location their internal representation is different. A point is defined by the six joint angles of the robot at the desired location. A transform is represented by the world X, Y, and Z coordinates and the three wrist orientation values O, A and T of the desired location As the point representation stores the actual joint angles it is more accurate than a transform However the gain in accuracy is offset by the limitations of manipulating points A transform can be shifted and have computations performed on it before it is used in any motion command, there is no facility for this on points

**DEFINE** <*type*> <*name*> <*data*>

Allows the manual creation of a *point* or *transform* according to the <*type*> specified. The name of the *point* or *transform* is taken from the <*name*> field. The <*data*> field is a six element list in joint angles for a point or in X,Y,Z,O,A,T format for a transform

DEFINE *POINT FEEDER 30 20 40 0 90 -90*

The point feeder is defined according to the joint angle data.

**POINT** <*name*>

The current position of the robot is stored under the the name given by the <*name*> field in precision point format.

POINT *FEEDER*

The position of the robot is recorded as the feeder position as a point.

**TRANSFORM** <*name*>

The current position of the robot is stored under the the name given by the <*name*> field in transform or X,Y,Z,O,A,T format.

TRANSFORM *FEEDER*

The position of the robot is recorded as the feeder position as a transform.

### 3.2.3 Motion Commands

**ACTIVATE** <*what*>

Activates the hand of the PUMA or a suction pump as specified by <*what*>. If <*what*> is given the value *hand* the robots hand will be closed. Suction can also be turned on by setting <*what*> to *suction*. The DEACTIVATE command has the opposite effect as described later.

| | |
|---|---|
| ACTIVATE *HAND* | The robots hand is closed. |
| ACTIVATE *SUCTION* | The suction pump is turned on. |

**ALIGN**

Causes execution of the VAL alignment command. This causes the tool of the robot to be rotated so that its Z axis is aligned parallel to the nearest axis in the world coordinate frame.

**APPROACH** <*how*> <*where*> <*distance*>

Approaches to a distance given by <*distance*> the location specified by <*where*>. The type of motion is specified by <*how*>. The argument <*how*> may take on the values of *transform*, *point*, *straight transform*, *straight point*, or *tool*. The first two allow for joint interpolated motion to transforms and precision points, while the second two allow for cartesian or straight line motion to transforms or points. These four methods of approach all use the tool Z axis as the axis of approach.

APPROACH *TRANSFORM VACUUM 60*

An approach is made by the robot in a joint interpolated mode to a position 60 millimeters in the tool Z direction from the point vacuum.

APPROACH *STRAIGHT_POINT VACUUM 50*

An approach is made by the robot in a straight line motion to a position 50 millimeters in the tool Z direction from the point vacuum.

The *tool* mode allows a vector to be given instead of the regular distance argument. This allows for an approach to be then made from any direction by supplying an appropriate vector. It should be noted that this last *tool* approach is not offered from VAL Its implementation is discussed in chapter 4.

APPROACH *TOOL VACUUM 10 0 0*

An approach is made by the robot in a joint interpolated mode to a position 10 millimeters in the X direction from the point vacuum.

**CALIBRATE_PUMA**

The VAL calibrate command is executed. The robot must be in the nest before this command can be executed This command need only be used after the robot is jimped and placed in the nest, as when the system is started by the START VAL command the robot arm is automatically calibrated and put into the ready position as described previously

**DEACTIVATE** <*what*>

performs the action opposite to ACTIVATE. The robots hand is opened if <*what*> is set to *hand* and the suction is turned off if <*what*> is given the value *suction*.

| | |
|---|---|
| DEACTIVATE *HAND* | Opens the robots hand |
| DEACTIVATE *SUCTION* | Turns the vacuum pump off |

**DEPART** <*how*> <*distance*>

The arm performs a motion relative to its current position, by an amount given in the argument <*distance*>. The type of motion is specified by <*how*>. The argument <*how*> may take on the values *joint*, *straight*, or *tool* The departure is carried out in the direction of the tool's negative Z axis in joint interpolated motion or straight (cartesian) motion as specified

DEPART *STRAIGHT 20* — The robot moves in a straight line 20 millimeters in the direction of the tool's negative Z axis.

The *tool* option does not allow the specification of a <*distance*> argument instead the departure vector is taken as the negative of a preceding APPROACH or TOOL motion command. If an alternate departure is sought the TOOL command can be used instead. Thus departures can occur along a vector rather than always along the tool Z axis. This is an extension of VAL and its implementation is discussed in chapter 4.

| | |
|---|---|
| DEPART *TOOL* | The robot moves in the opposite direction to the most recent APPROACH or TOOL command. |

**DRAW** <*dx*> <*dy*> <*dz*>

Causes the VAL draw command to be executed. This moves the tool along a straight line, a distance *dx* in the X direction, *dy* in the Y direction and *dz* in the Z direction. The tool orientation is maintained during this motion.

| | |
|---|---|
| DRAW *10 0 0* | The tool is moved 10 millimeters in the X direction |

**DRIVE** <*joint*> <*degree*> <*speed*>

The indicated robot <*joint*> is driven by the number of degrees given by <*degree*> at the specified <*speed*>. The joint is given by an integer *1* to *6*. The degree can be a negative or positive real number and the speed a percentage of the current monitor speed.

| | |
|---|---|
| DRIVE *1 -20 75* | The robot's first joint is driven in the negative direction by 20 degrees at seventy five percent of the monitors speed. |

**LIMP**

Executes the VAL limp command. This causes all of the PUMA's joint to become free. It is used when there is a need to manually place the robot into its nest. Caution should be taken to support the PUMA when this command is used. The PUMA will be left uncalibrated after this command.

**MOVE** *<how> <where>*

Moves the robot to a location and orientation specified by *<where>*. The type of motion is specified by the *<how>* argument. *<how>* may take on the values of *transform, point, straight_transform,* or *straight_point.* This allows for joint interpolated and cartesian (straight) motions to both transforms and precision points.

| | |
|---|---|
| MOVE *TRANSFORM VACUUM* | Moves the robot to the transform point vacuum, using joint interpolated motion. |
| MOVE *STRAIGHT_POINT VACUUM* | Moves the robot to the precision point vacuum, using straight line motion. |

**NEST_PUMA**

This instruction can only be used after a READY_PUMA instruction which is enforced by VAL. The command will place the PUMA arm in its nest. The speed should be less than 20 when placing the arm into or bringing the arm out of the nest.

**READY_PUMA**

Moves the robot to a ready position above the workspace. This forces the robot into a standard configuration regardless of its location. This command must be used before the robot can be nested.

**SPEED** *<what> <percentage>*

Sets the speed of the PUMA or X-Y stage as indicated by *<what>* to a percentage of top speed. If *<what>* is given the value *puma* the PUMA's speed is set. The speed of the X-Y stage is set by giving *<what>* the value *stepper.* This sets the speed of all four motors controlled by the big stepper, but by using a speed command just before a command to the stage or microscope the speed of the individual motors can be controlled. However if a speed command is issued while one of the motors is moving the speed of that motor will change as well.

| | |
|---|---|
| SPEED *PUMA 20* | Sets the PUMA's speed to 20% of its maximum. |
| SPEED *STEPPER 20* | Sets the stage's speed to 20% of its maximum. |

**TOOL** <*dx*> <*dy*> <*dz*>

This command is an extension not originally supported by VAL. It offers straight line motion along a vector relative to the tool coordinates. The robot tool will be moved along a straight line, a distance *dx* in the tool X direction, *dy* in the tool Y direction and *dz* in the tool Z direction. The orientation of the tool is maintained during the motion. The implementation details are given in chapter 4.

### 3.2.4 Motion Commands for the Stage and Microscope

**CHECK** <*what*>

The CHECK command provides information about the current state of a motor controlled by the stepper motor controller. The user can find out if a motor has been stopped, or is still moving. If the motor is still moving the amount still remaining to be moved may be determined. The motors that can be checked are specified by the <*what*> argument which may be given the values *tablex*, *tabley*, *zoom* or *height*. Tablex and tabley are used to check the stage's progress in a motion while zoom and height refer to the microscope's zoom and height adjustment motors respectively.

| | |
|---|---|
| CHECK *ZOOM* | Reports to the user the current state of the microscope's zoom control motor. |

**FREE** <*what*>

This command is used to halt the motion of the stage or microscope motors by disconnecting their power supply. The stage may be halted by setting <*what*> to *table* while *scope* may be used to halt the microscope. It is recommended that after issuing a motion command that the free command be used to isolate the motors, while they are stationary, from their power supply in order to avoid over heating problems. A new motion command will automatically restore power to the motor or motors affected by the command.

| | |
|---|---|
| FREE *SCOPE* | The two microscope motors are disconnected from their power supplies until the next motion request. |

**SCOPE** *<motor> <distance> <speed>*

This is the command to adjust and focus the microscope. The *<motor>* specification can be *zoom* or *height* to change the microscope's zoom or focus respectively. The *distance* is an absolute focus or zoom setting. The focus can be thought of as the height of the microscope below its home position in millimeters. The zoom can be thought of as a percentage of the scope's maximum zoom capabilities. The *<speed>* argument is given as a percentage of the top speed of the selected motor. Speeds must be kept relatively low (below 50) when moving the microscope to avoid motor overload and the consequential loss of calibration. The *speed* field is used to update the selected motor's speed from its initial value or a previous SPEED command.

| | |
|---|---|
| SCOPE *HEIGHT 50 20* | Move the microscope to 50 millimeters below its home position at 20% of its top speed. |
| SCOPE *ZOOM 30 20* | Adjust the zoom to 30% of its maximum at 20% of its top speed. |

**STAGE** *<sub command>*

This is the command for moving the stage around. There are three sub commands that are offered for this purpose. The sub commands are *move*, *where* and *to_cap*. The *move* option takes an additional X and Y argument. The X and Y arguments can be given in millimeters or either may be left as the literals X and Y in which case the values for X and Y will be taken from some global variables. These variables can be set by some of the vision commands to achieve feedback from vision routines. The setting of these variables is discussed later in the vision commands AREA and SET_DISTANCE. The stage performs an absolute move to the given X and Y values.

| | |
|---|---|
| STAGE *MOVE 10 20* | The stage is moved to the coordinates 10 20. The stage will not move if the stage is already at this location. |
| STAGE *MOVE X 20* | The stage is moved to the coordinates X 20, where X will be taken from the global variable set by a vision command. The stage will not move if the stage is already at this location. |

The *where* option can be used to find out where the stage is in the stage's local coordinate frame.

| | |
|---|---|
| STAGE *WHERE* | Reports on the stage's position in its local coordinate frame. |

The *to_cap* command is an example of a specialized command and it will be described later in the section on specialized commands.

**TABLE** *<motor> <distance> <speed>*

The stage is moved by a relative amount as specified by the *<distance>* argument which may be positive or negative. The speed is controlled by the *<speed>* argument and is given as a percentage. Motion may be in the X or Y directions as give by the *<motor>* parameter values *tablex* or *tabley* respectively. The stage's relative motion is used to update the stage's absolute position so that the STAGE commands will still work correctly.

| | |
|---|---|
| TABLE *X -20 30* | Move the stage by 20 millimeters in the -X direction at 30% of its top speed. |

This concludes the section of commands that deal with moving a piece of hardware. The next section will present the commands that deal with editing running and debugging of programs.

### 3.2.5 Debugging and Editing Commands

The following commands form the heart of the system. These commands are a little different from the other commands in that they can be called in two different ways. Firstly they can be invoked like the other commands from the menu by typing them in at a command line prompt, and secondly they can be called by issuing a control character. When the system is at its top level, that is, not in a separate command tree such as the vision tree, all the editing and debugging commands are available from the menu. However when the system is in a subtree the commands are no longer available from the menu. This is because it would be a waste, and inhibit the modular development of subtrees if every subtree had to duplicate the menu entries of the top level menu tree. Instead a method was

devised to catch control characters and have them interpreted as top level menu entries. This means that a subtree can be developed with total disregard of the top level menu entries, and yet, when completed and linked into the system, still have full access to all of the editing and debugging features that make the system what it is.

The following commands may then be thought of as global commands available at all times, be it via the menu or corresponding control key. It was found that when using the system it was often more convenient to use the control keys for the editing functions than typing its menu entry. The control key equivalents are given next to their corresponding command below

**^H HELP** *< what>*

The help command may be used to get information on the purpose and use of a command. It will read in a help file and using the UNIX function *more* will display it on the screen. The user may do HELP *HELP* to get a list of topics on which help is available.

**^B BACKUP** *<for what> <line number>*

This command may be issued while recording a program to cause the robot to retrace its path back to the specified program line. The *<for what>* argument indicates that the user wishes to over write the following code or wishes code to be inserted before the specified line number. Thus the *<for what>* argument can take on the values *to_overwrite* or *for_insert*. If a backup is done to insert some code then the user must indicate when this insertion is to stop by issuing the INSERT *OFF* command as discussed later.

| | |
|---|---|
| BACKUP *TO_OVERWRITE 10* | The program counter and robot are backed up to line 10. Any recorded code at, and after line 10 is over written. |
| BACKUP *FOR_INSERT 10* | The program and robot are backed up to line 10. Commands are then inserted before line 10 until the INSERT *OFF* command is given. |

**BREAK**

This command may be placed anywhere in a program. If the break is activated by the SET *BREAK ON* command then execution of this instruction will cause the program to halt and the user will then be asked how or if to proceed. The same set of options are then available at this point as when single stepping or interrupting a program as described below. If the break command is not activated (SET *BREAK OFF*) then the BREAK instructions are simply ignored.

**^C INTERRUPT**

Control C is taken as a user Interrupt to stop whatever is happening. A prompt is issued and the user may then type a single letter to indicate what action should be taken. The options are "a" to Abort a running program, "b" to Backup one step in running program, "p" to Pause a running program "s" to start Single step mode, "r" to restore normal Run mode, and "E" to perform the Error recovery or system reset function. When in single step mode each instruction is listed to the screen and the user given the same options as described above. Any other key will cause execution of the next instruction in the program as if in single step mode. Thus a user can step through a program line by line by entering single step mode and then simply hitting the space bar for each line to be run. If a program is aborted or paused the user is placed back in the top level of the key tree matcher If the pause command was used the user can change whatever may be needed and the program resumed with the CONTINUE command described below. The system reset option provides a fail safe mechanism for error recovery. Any open channels will be closed and running programs aborted. The user is then placed in the top level of the key tree matcher with the current program ready for addition, change or execution as the user may see fit.

**CONTINUE** *<where>*

This command is used after pausing a program. The program may be restarted from any command line. To restart it from where it left off *<where>* should be set to *run* or *from* if a new continuation point is to be specified.

CONTINUE *RUN* — A paused program continues its execution.

CONTINUE *FROM 10* — A paused program resumes its execution from line number 10.

**^D DELETE** *<what>*

This command is used to delete command lines or VAL locations. If *<what>* has the value *command_lines* then a range of command lines must be given and will be deleted. If instead *<what>* is set to *place* then a position name should be given to delete a location.

DELETE *COMMAND_LINES 2 9* — The command lines two to nine inclusive are deleted from the program.

DELETE *PLACE FEEDER* — A point named feeder is deleted.

**^E EDITLINE** *<line number>*

The command line pointer is moved to the given *<line number>* whereupon up to ten lines of commands are displayed to provide relevant context. Any previously recorded commands after the specified *line number* will be replaced line for line by any subsequent commands The user may use this command to move around and change things in a program file but this should be done only in an off line mode as the robot does not follow as the command line pointer is moved around, as happens for the BACKUP command The user may return to the end of a file by giving a large command line number and the system will automatically set the pointer to the end of file.

EDITLINE *20* — The command pointer is set to line 20 and lines 10 to 20 are listed for the user to view.

**^I INSERT**

This command is used to enter and leave insert mode. To enter insert mode the user must specify the line which the insertion is to precede. The insertion is then made and the mode is exited using the INSERT *OFF* command. This command can be used only if the user is in record mode. Note that the robot does not follow the command line pointer as it does with the BACKUP *FOR INSERT* command.

| | |
|---|---|
| INSERT *BEFORE_LINE 10* | Commands are inserted before line 10. |
| INSERT *OFF* | The insert mode is left and the command line pointer is set to the end of the program file. |

**^L LIST** *<what>*

This command is used to list command lines and VAL locations The *<what>* parameter may be set to *command_lines*, *all_places* or *named_place* If command lines are to be listed then a range of line numbers is expected while a point name must be given in the named_place option. The all_places option will give a listing of all currently defined points.

| | |
|---|---|
| LIST *COMMAND_LINES 10 20* | All existing command lines between lines 10 to 20 are listed |
| LIST *NAMED PLACE FEEDER* | The location feeder is shown if it exists. |
| LIST *ALL_PLACES* | All defined locations are listed. |

**^R RECORD** *<mode>*

This command switches the user between the record and non-record modes. The *<mode>* may be set to *on* or *off*.

| | |
|---|---|
| RECORD *ON* | Enter record mode. |
| RECORD *OFF* | Leave record mode. |

**RUN** *<start number> <end number>*

The run command is used to run a program that has been loaded into memory, or is being recorded and edited. The program will start execution at the line given by *<start number>* and end either at *<end number>* or the last program line depending on which is smaller This means that the user does not have to remember the exact number of the last line of a program, instead he may just give a large number in the *<end number>* field.

| | |
|---|---|
| RUN *1 99* | The program in memory is run from line 1 to line 99 or the last line of the program whichever comes first. |

**SAVE** *< what > < file name >*

This command is used to save a command or position file. Accordingly *< what >* may be set to *command_file* or *position_file*. A command file will be saved under the given file name with the extension of .PLA added to indicate that it is a playback file. The extension of POS will be added to all position files

| | |
|---|---|
| SAVE *COMMAND FILE DEMO1* | The current command file will be saved under the name demo1 pla |
| SAVE *POSITION_FILE DEMO1* | All currently defined points will be saved under the name demo1 pos |

**SET** *< what >*

This command is used to set various environment variables. The argument *< what >* may be set to *break, single_step* or *edit_only*. The *break* option is used to activate or deactivate the BREAK command. Likewise the single step mode may be switched on or off. The edit only mode allows for the direct writing of programs with out the actual execution of robot motions. This is very useful if a set of working subroutines has already been developed and is ready to be used in a larger program The subroutines can be quickly recorded without having to wait for any lengthy execution which might otherwise be involved The lengthy execution is not a result of a slow programming environment but the result of the inherent slowness of mechanical equipment such as the robot and stage or slow vision processing.

| | |
|---|---|
| SET *BREAK ON* | The break command is activated. |
| SET *SINGLE_STEP OFF* | Single step mode is exited. |
| SET *EDIT ONLY ON* | Edit only mode is entered. |

**STATUS**

The STATUS command can be given to list what the present modes are That is single step, break ,record and edit only as well as the state of the VAL channel are shown to have states of on or off.

**USE** *< type > < file name >*

This command looks for the specified file and loads it into memory The types of files available are *command files*, *position files* and *capacitor files* *Capacitor files* are treated later in the special commands section

USE COMMAND_FILE DEMO1 — The command file demo1.pla is loaded into memory if it exists and error messages are given otherwise.

### 3.2.6 Conditional Commands

The following set of commands are used to perform branching and and flow control of a program.

**GOTO** *<line number>*

An unconditional branch is made to the *<line number>* specified. This can be used for infinite user terminated loops or for a branch after an IF command.

**GOSUB** *<name> <argument list>*

The subroutine specified by the *<name>* field is called with the *<argument list>*. The *<argument list>* is comma separated and may contain less arguments than the routine is expecting. In such a case the default strings in the subroutine are used in place of the missing arguments. The argument substitution occurs in the order that the arguments are given in the list, therefore, if arguments other than the last in the list are to be omitted commas must be used to fill their place.

GOSUB *PICKUP "FEEDER,,HAND"* — The subroutine pickup is called with two arguments *feeder* and *hand* The second and final arguments have been skipped and will take their values from the default strings in the subroutine. An example of such a subroutine can be seen in Chapter 5.

**IF** <*flag*> **THEN** <*line number*>

A conditional branch is made to <*line number*> if the <*flag*> is true, otherwise the following instruction is executed. The presently implemented flags are *answer*, *c1* - *c4* and *error*. *Answer* is set true if the user has answered the most recent prompt in the affirmative. *C1* - *c4* are set true if the corresponding VAL input channels are high, and the *error* flag is set true if VAL has reported an error.

IF *ANSWER THEN 20*

If the user has answered the previous prompt in the affirmative a branch is made to line 20, otherwise the following instruction is executed.

**WAIT** <*condition*>

The program waits until the condition is satisfied. The presently implemented conditions are *answer*, *capacitor*, and *stage*. The program will wait for the user to answer yes or no to a prompt if *answer* is set as the condition, or for a capacitor number if *capacitor* is set as the condition. In both cases the wait will continue until a valid argument is entered by the user. The final condition *stage* allows a user program to synchronize actions between the robot and stage by waiting for the stage to complete its motion.

WAIT *STAGE*

The program waits until the stage has come to a halt, whereupon execution of the program continues with the next instruction.

**TYPE** <*prompt*>

The string specified by <*prompt*> is typed at the user's terminal. This command is used to prompt the user for any terminal input needed by the program.

TYPE *"Would you like to continue the demo (y/n)"*

TYPE *"Please enter a capacitor number"*

### 3.2.7 Vision Commands

The next set of commands deal with the control of the vision system. These commands are an example of a separately developed tree of commands. These commands are accessed by issuing the VISION command.

**ALLOCATE GRINNELL**

This command is used to gain access to the grinnell monitor which is controlled by the VAX 780. It should be the first vision command issued.

**CAMERA** <*frames*> <*channel*>

Grabs a camera's image and loads it into the camera image buffer. The <*frames*> argument specifies how many frames are to be averaged into the final image buffer. The <*channel*> argument is used to select which of the three image planes (red, green or blue) the image is to be stored on Channel 1 is red, channel 2 is green and channel 4 is blue Images can be grabbed onto multiple image planes by giving a channel number that is a sum of the desired channel numbers That is channel 6 2+4 will give the green and blue channels.

| | |
|---|---|
| CAMERA *3 1* | Three frames from the camera are averaged onto the red channel |

**DISPLAY** <*what*>

Displays an image specified by <*what*> on the monitor. <*what*> may be *camera image*, *file image* or *work image* The camera image is the image buffer associated with the CAMERA command The file image is associated with the READFILE command described later and work image is the image that can be manipulated by vision commands such as AREA, as described in the special commands section.

| | |
|---|---|
| DISPLAY *CAMERA IMAGE* | The camera image is displayed on the grinnell monitor. |

**ERASE**

Erases a selected channel, as selected by the SELECT CHANNEL command described later.

**EXIT**

Exits from the vision commands back to the main command tree.

**FREE**

Prints a free message on the grinnell so that other users may be notified that they may now use the grinnell.

**HISTOGRAM**

Performs a histogram of the work image so that thresholds may be determined.

**LOAD** <*what*>

Loads a *file_image* or *camera_image* into the work image as specified by <*what*>.

| | |
|---|---|
| LOAD *CAMERA_IMAGE* | The camera image is copied into the work image. |

**READFILE** <*filename*>

Reads into the file image a stored file as specified by <*filename*>

| | |
|---|---|
| READFILE *TEST PIC* | Reads a file called test.pic into the file image buffer. |

**SELECT CHANNEL** <*channel*>

Selects a channel or channels that will be used by subsequent commands. As explained in the camera command the channels are numbered *1*, *2*, and *4* for red, green and blue respectively, and channels may be combined additively.

**WRITEFILE** <*filename*>

Stores the work image into a file given by <*filename*>.

WRITEFILE *TEST.PIC* — The work image is stored in a file test.pic.

**VIEW**

This command displays continuously the image seen by the camera on the monitor. It can be used to set up the camera and lighting by providing immediate feedback to changes made

### 3.2.8 More VAL Commands

This next set of commands are other VAL commands that have been included in the key tree matcher. It should be noted that not all of the VAL commands have been implemented. It was found that for the work cell at hand the commands implemented were sufficient. However should the user wish to have a more complete selection of the VAL commands two options are provided. The user may write VAL code and then down load it and then run it, or they may enter VAL directly and run any VAL commands as if directly connected to VAL as explained in the VAL command described below.

**BASE** <*dx*> <*dy*> <*dz*> <*zrotation*>

Causes the execution of the VAL base command. This changes the origin of the VAL reference frame.

BASE *100 0 0 90* — The robots world reference frame is shifted by 100 mm in the X direction and rotated by 90 degrees around the Z axis.

**CLEAR_VAL**

This command is equivalent to the VAL zero command. It causes all VAL programs and positions to be erased.

**DELAY** <*time*>

Causes the PUMA to wait for a specified time in seconds.

**DOWNLOAD** <*program*>

This command can be used to down load a VAL program stored on the host to the VAL controller. In effect the floppy disk drive of the VAL controller is replaced by the host's hard disk. This command is also used to download interrupt service routines as these must run on the VAL controller to be activated at interrupt rates.

| | |
|---|---|
| DOWNLOAD *TEST* | A VAL routine called test is down loaded from the host to the VAL controller. |

**EXECUTE_VAL** <*program*>

Starts a VAL routine that has been previously down loaded running. control will not be returned to the key tree matcher until the VAL routine has stopped.

| | |
|---|---|
| EXECUTE_VAL *TEST* | The routine test is run on the VAL controller. |

**OUTPUT** <*line*>

This command causes one of the VAL controller's output lines to be set or reset. If <*line* > is positive the corresponding line will be turned on, and turned off if <*line*> is negative. This command allows the control of the output lines so that communications with other devices is possible.

| | |
|---|---|
| OUTPUT –3 | The output line 3 is turned off. |

**READ** <*line*>

This command is used to test the status of an input line. This is useful for synchronizing of devices or for simple sensory input. The global flags C1 - C4 are correspondingly set to true if the *line* is found to be high

| | |
|---|---|
| READ *3* | If the output line 3 is high C3 is set true. |

**WHERE**

Causes the execution of the VAL where command. This displays the robots' position in joint angles and X,Y,Z,O,A,T notation.

**VAL**

The user is connected in a "pass all" mode to the VAL controller. This is as if the user's terminal was connected directly to the VAL controller. The user may return to the key tree matcher by typing CONTROL-C.

### 3.2.9 Specialized Commands

The following set of commands gives an example of how commands can be added to the system to tailor it to a particular situation. In this case the system has been modified to deal with the problems of hybrid circuit board repair.

**AREA** *<threshold> <window size> <plot>*

This command will run a mask of size *<window size>* over an image and label all connected regions above the specified *<threshold>*. The sizes of the five largest regions found will be returned along with their centers. The last argument *<plot>* is a flag to specify whether or not the labeled regions are to be displayed on the monitor.

| | |
|---|---|
| AREA 20 10 1 | The areas and centers of the five largest regions above a threshold of 20 are returned. The window size is 10 X 10 pixels and the regions found are displayed on the monitor. |

**DISTANCE**

This command can be used after the AREA command to find the distance of a fixed target from the camera being carried by the robot. The target is a set of white on black circles of known size. The distance is found by relating the area of the circles found with the AREA command to a distance area function of the camera being used. This function was experimentally determined by taking a number of distance area readings and fitting a curve to the data points. The curve function was then used to determine distances from areas.

**SETDISTANCE** *<object> <Xsize> < Ysize>*

This command can also be used after the AREA command. It sets the global X and Y distances, in millimeters, of an object labeled in the AREA command from the center of the screen. This command is used to get these X and Y distances so that an image can be centered under a camera by the robot or X-Y stage. *<object>* is a number from one to five specifying which of the labeled objects from the AREA command should be used in the calculation. *<Xsize>* and *< Ysize>* are the respective X and Y sizes of the object in question. These sizes are needed so that correct scales may be calculated.

| | |
|---|---|
| SETDISTANCE *2 20 20* | The X and Y distances of object 2 are found in milimeters from the center of the image and the corresponding global variables X and Y set. The actual object having dimensions of 20 X 20 milimeters. |

**STAGE** *to_cap*

The function of the STAGE *to_cap* command is to move the stage so that a capacitor as specified in a data base will be positioned over a defined location. This defined location is specified in the data base on the first line of the file, and is given as an absolute position in the stage's local coordinate frame. The number of the capacitor must have been already defined with the WAIT *CAPACITOR* command and a capacitor data base must have been loaded in by the USE *CAPACITOR_FILE* command. These commands are discussed in more detail below.

| | |
|---|---|
| STAGE *TO_CAP* | Moves a prespecified capacitor to a fixed location. |

**USE** *capacitor_file <name>*

The file specified by *name* will be loaded as a capacitor data file. The format of this file is as follows. On the first line the desired absolute location of the stage in its local coordinate system should be given. Following this each subsequent line should have the capacitors x,y, and z offset from the desired location as given on line one. Normal the z offset is zero as the stage is located horizontally.

**WAIT** *capacitor*

The program will wait for the user to enter a capacitor number. A capacitor's number corresponds with the order that the capacitors were specified in the capacitor file. That is, the first capacitor in the file is numbered one and the second two etc. The WAIT command is usually given after a TYPE command requesting capacitor data.

This concludes the sections on commands available on the system. It can be seen that a variety of commonly used VAL commands can be accessed. Commands exist for manipulating the stage, microscope and vision system. The set of vision commands is an example of how a modular set of commands can be developed and used in the system. The only draw back of such modularity, as implemented, is the need to explicitly issue commands to enter and leave the vision environment. Besides the basic set of commands to manipulate the various pieces of equipment, examples of specialized commands have been given. These specialized commands deal mostly with the vision section as this area of vision processing is quickly growing and changing and must therefore be able to be customized to the desired environment. Examples of specialized commands were also seen in the specialized commands section showing how the environment can be tailored to the hybrid repair process.

# Chapter 4 System Implementation

## 4.1 System Software

In this chapter the more technical aspects of the system will be discussed. These include interface of the PUMA to the host, the associated communications routines and inherent limitations of the chosen interface method. The function of various elements of the software and their interaction in the environment is described, as well as the methods used to back up through a program, perform error recovery and implement tool motions

A block diagram of the system software can be seen in figure [4.1]. The software takes user input and parses it using the key tree matcher This matcher may also be fed from program files by a spooler to run in an automatic mode The key tree matcher passes the user's parsed commands onto the interpreter which determines whether direct or editor commands are to be executed. Commands are forwarded by the interpreter to the appropriate module for execution. The editor routines deal with the program, position and capacitor files. The direct commands are handled by the command modules which communicate with the PUMA, stepper motor controller and vision system The error handler module routes errors to the user's terminal or performs error recovery functions as needed.

## 4.2 The Communications Interface

As stated previously the PUMA has no method of communicating with an external host computer except through its terminal port. This method of communications means

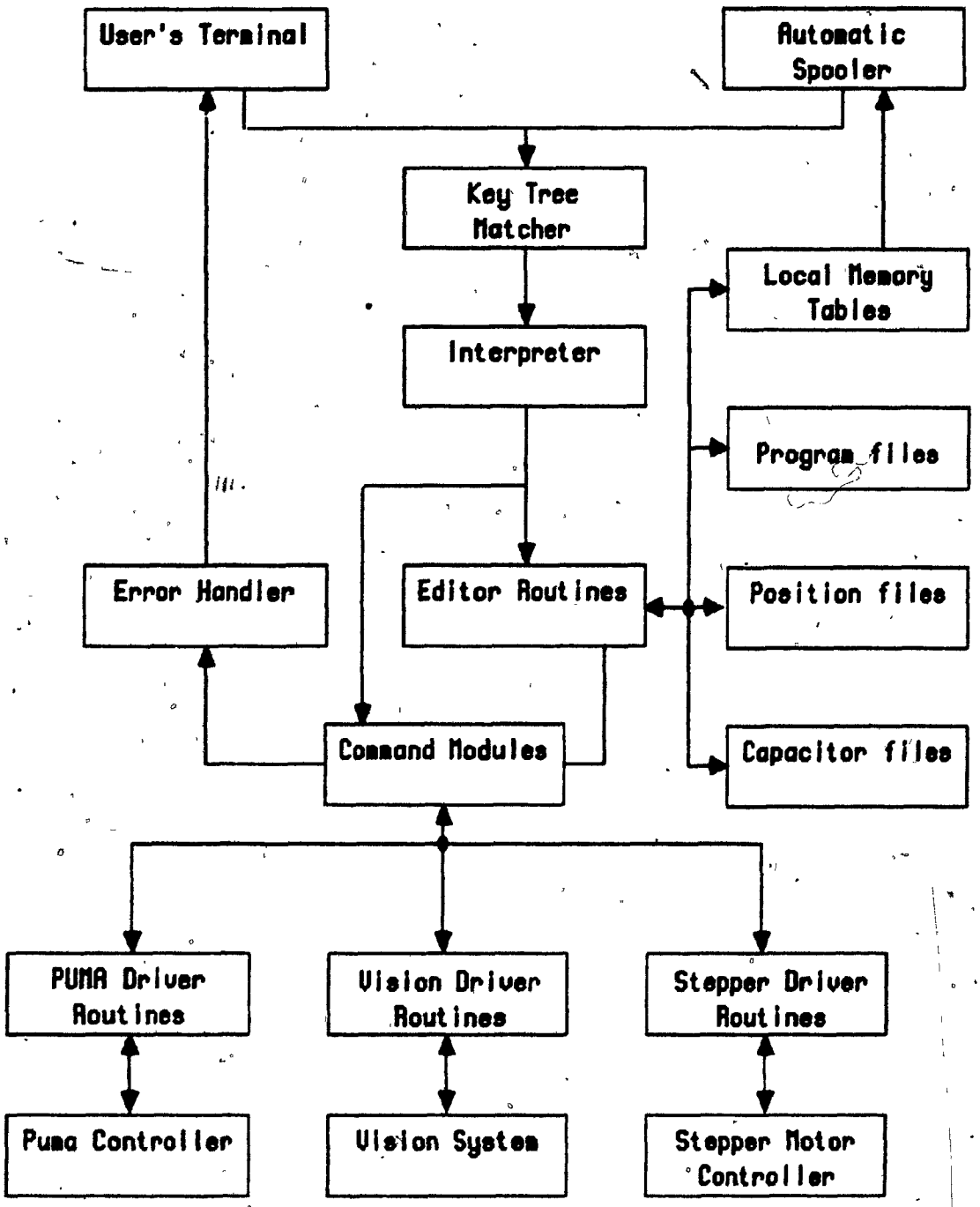


**Figure 4.1** System Software

that the host must emulate a user typing at the PUMA's terminal This technique of simulating a user's input terminal commands may appear to have some limitations [Lechtman et al. 48]. Firstly, the communication speed is limited to typical terminal rates, less than 9600 baud Secondly the PUMA controller generates a great deal of output in order to be user friendly This data must be processed by the host and either relayed to the user or discarded depending on its usefulness Thirdly there is no standard handshake protocol normally associated with communication, thus there is an inherent risk of transmission errors. Finally, and perhaps most importantly, any commands communicated to the controller must be routed via a VAL command. Therefore, no direct control of the servo joint controllers can be achieved

On careful examination, most of these limitations are not of particular importance, while others are not difficult to overcome. The slow communication speed is not of great importance as the robot's motion usually accounts for a far greater portion of time than the relative command communication

As the communication is slow, routines may be easily developed on the host to deal with, and categorize any messages returned by the PUMA controller These messages may be used to implement a form of handshaking between the host and PUMA controller However this is not the most reliable way to achieve handshaking as the communications lines may hang as one end waits for a continuation signal that may never come if it has been lost in a transmission error. This is very infrequent and presently a manual reset is implemented but a more automatic method could be implemented using a time out routine to detect the loss of transmission

The limitation of having to implement all commands based on existing VAL commands is seen as the most serious drawback of this method of controlling the PUMA For example there is no direct way of doing any force sensing as VAL does not allow for this. Also as commands are being sent through the VAL monitor no continuous path motion may be obtained Another problem area is control of the trajectory as the user is limited to the types of motion specified by VAL

Work on overcoming this final restriction is being done at McGill by using RCCL as a control language and modifying the PUMA's controller by adding a parallel link to it [Lloyd 36]. This link allows direct control of the joint servo controllers of the PUMA by the host

computer. Although adding the power of direct control, in its present state this language requires the user to become involved in the details of the control which is a limitation when used for anything but research. Accordingly more work in building an user interface layer to RCCL would have to be done before it could be offered to an end user in an acceptable format. However as RCCL is made up of a set of C subroutines it is possible that the authors environment can offer just such an interface. An appropriate subset of these routines could be linked into the environment and entries made in the menu so they could be called. This would be in the same spirit that the vision algorithms were developed as C subroutines and then linked in when complete.

### 4.2.1 The PUMA Driver

A communications package called Routines.c has been developed in the C language to allow communications between the PUMA and host [G. Carayannis 49]. This package which was developed before the author started work on this project consisted of a set of C routines that could be called from a main program to send VAL commands to the robot.

The routines.c package incorporates many subroutines. There are subroutines for transmitting commands, character by character, to the PUMA. VAL echoes each character it receives. This echo is checked by routines.c to ensure that no transmission error has occurred. In running VAL commands run time errors may occur. The routines constantly monitor transmissions from the PUMA controller and return with an error code or a signal that 'all is ok'. These errors can be obtained in their raw form or encoded into numerical form as may be desired.

To write a robot program, a user would have had to write a C program to call these routines, then compile, link and debug the code and then finally run the program before any motion of the robot could be realized. There was no way for the user to interact with the running program save aborting it and no way to debug it, without stopping it and repeating the edit compile, link and try again routine. Clearly this represented a very cumbersome and time consuming way to add the power of host control and complex sensors to robot programs.

The author's alternative was to take these routines and incorporate them into an environment whereby the user would regain the more interactive programming method pre-

viously enjoyed in VAL and still enjoy the benefits of host control and complex sensor interface.

### 4.2.2 The Key Tree Matcher

To provide an interactive programming environment user commands must be parsed, interpreted and executed with feedback indicating their success or failure. The parsing is done by making use of a superb key tree matcher package developed at McGill by [Parker 50]. This package provides mechanisms for setting up a tree of key words (commands) against which user commands may later be matched. Hence the name key tree matcher.

The key words in the tree may be set up to take additional arguments. The matcher can be made to check the range and type of the arguments before they are assigned to programmer specified variables. The common types of variables, integers, reals and strings, are all supported. The two main routines used are called tree_match_parse and tree_match. Tree_match_parse takes a tree specification and returns a pointer to an internal tree structure. This pointer should be treated as a char * pointer. Tree_match_parse needs to be called only once per key word tree which may be required in a program.

The tree_match routine takes the pointer returned by tree_match_parse along with a prompt and pointer to a key word buffer as its parameters. The prompt can be any string specified by the programmer and will be displayed when user interaction is being sought. The key word buffer is where the key word or command is actually returned after parsing. This key word buffer is stripped of any arguments and may be examined to see what command was typed by the user. The tree_match routine then should be called whenever user interaction is required.

In its present state of development, commands may consist of any characters with the exception of white space (space,tab,new line), control characters and the characters ! " # % & ( ) ? [ ]. The characters * and @ should be avoided as they have a special meaning when typed at the beginning of a line. The * will list the entire tree which is useful in debugging a tree that is being set up. The @ followed by a file name will cause input to be taken from that file instead of the user's terminal and may be used to run a batch of commands previously prepared.

A character which has a powerful effect when typed at the start of a line is !. When

followed by a system command it will cause a new shell to be started and the command run before control is passed back to the programmers routine This is most powerful as directories may be listed and files found from within a program without any need to exit or stop the running procedure

The following example shows how a tree containing the key words "quit", "set" and "show", can be set up and then used as a reference for matching commands.

```
char *tree;
char *tree_match_parse();
char *buffer;
float sigma;

. . . .

tree = tree_match_parse("(quit show set(sigma (%r",&sigma)));

. . .

tree_match(tree, "Enter command ?" ,buffer);
```

The above example also shows how a variable argument may be used. The real variable sigma can be set to a specific number, say 20, by typing "set sigma 20". The %r signifies that a real argument is expected, while %s and %i may be used for strings and integers as appropriate

### 4.2.3 Feeding the Matcher

The above has given a flavor of what can be done using the key tree matcher The matcher will normally handle the fetching of characters as the user types them in However as program files had to be stored and then replayed through the matcher, a method was found to feed the matcher commands in a controlled and automatic manner The new extension allows for a programmer to specify a routine that is to used by the matcher to get input A "spooler" routine was developed by the author for this purpose

The "spooler" routine will normally just return whatever the user types However if the user wishes to run a program file the spooler will read in this file and then proceed to feed a character at a time to the matcher This character by character feeding allows for some convenient features to be added to allow quite a bit of control over the flow of

commands being fed to the matcher It is this control of the flow that allows for branching and looping to be implemented as well as argument passing. Branching can be implemented by simply changing the spooler pointer in the file being spooled, and argument passing by the interception and replacement of marker characters by the spooler, as discussed later. This also facilitates the debugging capabilities of single stepping and backing up through a program

## 4.3 Adding a Subtree

A subtree can be added to the system quite easily This allows for modular development of packages to enhance the capabilities of the system, as was done with the vision package. Once a set of subroutines have been developed independently they must be linked into the system so that they can be used, This linking procedure is explained below.

The procedure of turning a set of subroutines into a subtree starts with the creation of a tree of commands that will call the subroutines as described in the key tree matcher section and more fully in [Parker 50]. Once this has been done an entry must be made in the main key tree menu for calling the sub tree as shown below

```
include ''subtree.h''
char buf[80],
char *tree_match_parse;
...
tree = tree_match_parse('' (exit          \
                            help          \
     ..                     vision        \
     ...                   )'');
     ...
while (strcmp (buf, ''exit''))
 {
   record_command();
   if (!strcmp (buf, ''help'')) help_routine();
   ...
   if (!strcmp (buf, ''vision'')) vision_tree();}
   ...
 }
```

The above shows how the entry was made in the main menu for the vision subtree The include file "subtree.h" contains the necessary declarations for the saved command array and line pointer variables It should be included in the user's subtree as well. In addition to the entry in the main menu the subtree routine, given here as vision_tree(), should include in the same place as shown in the main program the record_command() routine. This routine saves the commands entered by the user if the record flag is on.

The subtree should also of course include an exit command so that the main tree can be returned to after any subtree commands have been executed

Finally once the subtree and main routines have been modified as indicated above they must be recompiled and linked in the normal UNIX manner

It can be seen then that this is quite a modular expansion as only one menu and calling routine entry must be made in the main program While an include file and the addition of the record_command() routine need be made in the subtree

## 4.4 Debugging

Debugging of a program can be achieved at any time in the program development. A command may be tried to see if its effect is desirable before it is recorded The robot may then be backed up and the command recorded or another tried for its effect. Alternatively commands may be recorded off line and then run on the robot These recordings may have break points set so that programmed pauses may be made at appropriate times or the program may be single stepped

If the user is running a program in single step mode each command is fed to the matcher and the user is then prompted with options for executing the command, pausing or aborting a program, as well as backing up through the program. If the program is paused, the user may change anything in the environment and then continue the program from where it was paused

## 4.5 Subroutines and Argument Passing

Subroutines which take arguments may be developed as easily as normal programs These subroutines are stored in separate program files and may be treated like new, user defined commands This provides the ability to develop high level commands which the user may then call to simplify a task description

Argument passing is handled in a quite unique way Special characters are interpreted as markers where arguments are to be inserted. These arguments are taken from the parameter list specified when calling the subroutine As a marker may be followed by a descriptive string the user may use this string to provide a default argument in case the

subroutine is called without arguments. An example of such a subroutine that performs the action of picking up an object is given below.

```
MOVE TRANSFORM $ready
APPROACH TOOL !vacuum 40 0 0
MOVE TRANSFORM @vacuum
ACTIVATE #hand
DEPART TOOL
MOVE TRANSFORM $ready
```

If the routine is stored in a file named PICK, it may then be called to perform its default action by giving the command with no arguments as shown below.

```
GOSUB PICK " "
```

This will cause the robot to move to the ready position then approach the position "vacuum" by 40 millimeters in the tool X direction. The robot will then move to the vacuum tool and close its hand. It will finally depart in the reverse direction to its approach and return to the ready position If the command is given with arguments, totally different robot actions may be produced for example the call.

```
GOSUB PICK "feeder 0 60 0, feeder, suction."
```

Will cause the robot to pick up a board from a feeder as opposed to picking up the vacuum tool

There are several things to be noted here. First only three arguments are given namely "feeder 0 60 0", "feeder" and "suction". These arguments correspond to the first three markers respectively That is, the first argument will be mapped onto the ! marker (shift 1), the second to the @ marker (shift 2) and the third to the # marker (shift 3) The missing fourth argument which would normally map onto the $ marker (shift 4) is missing and will thus be taken from the default string after the $ marker This default string as given in the example is "ready" which is a safe position to which the robot may return in between subroutine calls

A second point of note is that only strings are passed as arguments Thus to pass an approach vector such as 0 60. 0. it must be preceded by a string namely "feeder". This results in the string "feeder" having to be passed twice. Once to specify a point

and approach vector and again to specify and end point. This limitation is due to the simplicity of the replacement algorithm used to perform the marker substitution and could be overcome with a better algorithm.

A third point to notice is that the method of picking up the object was changed from using the robots hand to using a suction pump controlled by one of the PUMA's output ports. This shows the power of such a small subroutine. By specifying only three values (approach vector, point and method), virtually any thing may be picked up. This same concept may also be applied directly to other routines such as those involved in placing objects.

## 4.6 Status Recording

To allow the robot to backup through a program two methods may be used. The program may be analyzed and a reverse instruction generated, or the previous state of the robot recorded and returned to upon request. The author has chosen to implement the second method.

The first method while requiring reduced storage facilities, presents a complex task of generating reverse instructions. For example, if the robot has executed an absolute move to an object, then the robot's previous location can not be determined from this instruction alone. It would be necessary to look not only at the last "move" command, but also at the previous motion commands to see what route was taken by the robot. The method of analyzing previous commands may become complex as the user tries to move farther and farther back through a program

The alternative method is relatively simple As each command is executed the state of the environment is saved Not only is the robot's position and hand opening saved, but also the state of the output lines and positions of the stage and microscope Then all that is needed, to back up, is to restore the states in reverse order The state is not restored by the brute force method of issuing commands to set every recorded state variable Instead all present states are compared to the saved states to which the user wishes to return, and commands are issued to change any states that may differ

This restoration of state variables allows the user to backup one or any number of commands. The system will ensure that the user does not backup further than the first

recorded state. The user may, by this method of backing up, even retrace through a conditional statement, then pause the program, change the conditional's argument and restart the program. This allows for easy tracing of program branches.

## 4.7 Error Recovery

As commands are executed, they are monitored and any errors found are reported to the user. Besides reporting an error to the user an error variable is set and may be tested for in a user's program. These errors can usually be classified into two areas, fatal and non-fatal.

A fatal error is one which requires restarting VAL or the recalibration of the PUMA. Such errors occur unpredictably, for example when the robot hits an obstacle. In such an event, the user will not lose any work that may have been completed. All that is necessary, is for VAL to be restarted, if it has crashed, or for the PUMA to be renested and then recalibrated. All user programs and position data are preserved in the system environment and may be reloaded after the PUMA has been restarted. This is a great improvement over VAL as it avoids a situation where, after much laborious programming and position teaching, all may be lost because the user did not take the precaution to save all data before trying out a program.

In the case of non-fatal errors, such as forgetting to return the teach pendant to computer mode after teaching a point, the user may have the program do something in an effort to recover. The user may for example test that computer mode has been restored following a programmed teach sequence by issuing a DELAY command and testing the resulting error code until all is ok.

In the event that the system hangs, such as when a communications link is broken or VAL crashes while the system is expecting a return message, the user may still recover. To recover, the user need only hit control-C to create a "user interrupt". The interrupt handler will enquire whether the user wishes to perform a system reset or stop a running program. If a system reset is desired, all communications channels will be closed and system variables reset. A system reset can be used to recover from virtually any error condition without loss of programming data. Programs being run or edited will be preserved as well as any

taught points. This recovery has been implemented as a fail safe feature to ensure a reliable program development system.

Another use of this "user interrupt" is to change states as a program is running. A running program may be interrupted and paused, aborted, backed up or caused to run in a single step mode. This is a particularly convenient feature when developing large programs as tested parts of a program may be run at full speed and then interrupted and traced in single step mode as an untested section is about to run.

The aforementioned recovery is implemented via the use of some UNIX system functions. Namely signal handling and longjump commands.

## 4.8 Implementation of Tool Motion

The author has developed three commands which use motions relative to the tool coordinate frame, because it is often easier to express motions in this way rather than in the world coordinate frame. This is especially true when grasping or placing tools from or into a stand, or using tools which do not operate parallel to a world or the tool Z axis, or when data from a data base is used for palletization operations.

The method of moving in the tool coordinate frame as implemented is to convert the tool motion request from tool coordinates to world coordinates and then use VAL DRAW commands. To achieve this a tool to world transformation matrix is necessary. To determine the transformation matrix the method similar to that described by [Bazerghi et al. 51] was used.The reader is also referred to [Paul 52]. The orientation of the tool frame is specified by the three VAL variables O, A, and T. This leads to a break down of the problem into the four following parts.

(1) The rotation matrix relating the tool frame to a reference frame.

(2) The rotation matrix describing the rotation due to O

(3) The rotation matrix describing the rotation due to A

(4) The rotation matrix describing the rotation due to T

The first matrix was found by placing the robot in its ready position and then driving each of the tool joints so that the values of O, A, and T were zero. This done the transform from this configuration to the reference frame was found. The transform is given by the equation below.

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} X_z \\ Y_z \\ Z_z \end{pmatrix} = \begin{pmatrix} X_w \\ Y_w \\ Z_w \end{pmatrix}$$

The above equation represents the transform from the tool coordinate frame when O, A, and T have been set to zero $(X_z, Y_z, Z_z)$ to the world reference frame $(X_w, Y_w, Z_w)$. The effects of O, A and T were then observed from this initial state. The results were that O represents a rotation about the negative X tool axis, A a rotation about the new Y tool axis and T a rotation about the final Z axis. These rotations can be represented by the following matrices.

Matrix O Matrix A Matrix T

$$\begin{matrix} -X \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & C_o & S_o \\ 0 & -S_o & C_o \end{pmatrix} \end{matrix} \quad \begin{matrix} Y \\ \begin{pmatrix} C_a & 0 & S_a \\ 0 & 1 & 0 \\ -S_a & 0 & C_a \end{pmatrix} \end{matrix} \quad \begin{matrix} Z \\ \begin{pmatrix} C_t & -S_t & 0 \\ S_t & C_t & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{matrix}$$

where $S_o$, $S_a$, and $S_t$ are the sines of O, A, and T and $C_o$, $C_a$, and $C_t$ the cosines respectively. Having found these matrices they can now be multiplied together to give the total transformation as done below.

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & C_o & S_o \\ 0 & -S_o & C_o \end{pmatrix} \times \begin{pmatrix} C_a & 0 & S_a \\ 0 & 1 & 0 \\ -S_a & 0 & C_a \end{pmatrix} \times \begin{pmatrix} C_t & -S_t & 0 \\ S_t & C_t & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} C_oS_t - S_oS_aC_t & C_oC_t + S_oS_aS_t & S_oC_a \\ S_oS_t + C_oS_aC_t & S_oC_t - C_oS_aS_t & -C_oC_a \\ -C_aC_t & C_aS_t & -S_a \end{pmatrix}$$

This result can then be used in the following equation to find the world X, Y, and Z coordinate components from the given tool components. The world components are labeled $X_w$, $Y_w$, and $Z_w$ and the tool components $X_t$, $Y_t$, and $Z_t$.

$$\begin{pmatrix} C_oS_t - S_oS_aC_t & C_oC_t + S_oS_aS_t & S_oC_a \\ S_oS_t + C_oS_aC_t & S_oC_t - C_oS_aS_t & -C_oC_a \\ -C_aC_t & C_aS_t & -S_a \end{pmatrix} \times \begin{pmatrix} X_t \\ Y_t \\ Z_t \end{pmatrix} = \begin{pmatrix} X_w \\ Y_w \\ Z_w \end{pmatrix}$$

This equation is implemented on the host and is used in the APPROACH, DEPART, and TOOL commands, which all issue VAL DRAW commands with the converted components. This type of command while extending the flexibility of control of the robot uses the host to make the calculations. It would seem better if VAL had the ability to handle the real number calculations required so the host could be relieved of this relatively low level conversion process. These types of calculations are supported in VAL-II.

At this point the concepts, features, commands, and implementation aspects of the system have been seen. The next chapter will give some examples of the system at work in robot applications.

# Chapter 5 A Repair Function an Example

This chapter presents an example of a repair function programmed in and carried out by the developed environment. The tools developed by the author in this repair experiment are described and illustrated. The chapter will begin by presenting the repair objective and the physical layout of the robot environment The tools will then be described and the program for performing the repair presented.

## 5.1 The Repair Function

As an example of hybrid integrated circuit (IC) board repair it was shown how capacitors may be desoldered from such a board. There are many reasons why a capacitor may have to be removed in the process of repairing a hybrid IC. The capacitor in question may be misaligned with the pads on which it has been placed, the solder contacts between the pads and capacitor may be bad, or the capacitor may be of the wrong type for the pad. Any of these defects may occur in the manufacturing process [Bauks 53] Typically these repairs are carried out by hand. The circuit boards are inspected and grouped into classes of boards with similar defects These boards are then passed on to repair persons who do the actual repair.

Industry is looking at ways to automate both this inspection and repair process. As the repairs are usually varied in nature there is a need for flexible robots and vision processors to carry out the repair task. It must be easy to program such tasks on robot controllers in spite of the task complexity, precision and distributed or concurrent nature of the problem.

## 5.2 The Physical Layout

The repair station in this example is laid out as shown in figure [5.1]. As can be seen, the station configuration used consists of one robot, an X-Y stage, a microscope, a flame heater, a hybrid IC stand and a rack of tools needed to perform the task, as well as the controllers supervised by the host The robot is located centrally so that it can reach all areas of the environment. In this experiment it is designed to reach the hybrid IC feeder, the X-Y stage, the tool rack and the microscope

It is not suggested that this is an optimal layout for the task at hand. Rather it is an experimental layout, best suited for research The robot, X-Y stage and microscope illustrated in the diagram have all been previously described, therefore the tools, feeder and rack need to be described before the example is presented

### 5.2.1 The Tool Rack

The repair task was found to be complex enough to require a variety of tools These tools are required to be used at various times by the robot and must be interchangeable in an automatic fashion The tools must therefore be stored in a position where the robot will be able to find them as needed The tool rack was designed for this purpose The rack offers a set of cones onto which tools may be placed (see figure [5.2]) The cones assure correct and consistent tool alignment by using the forces of gravity to advantage This arrangement has proven to be very successful in automating tool changes It is planned to add sensors to the work station for several uses, such as, the verification of tool exchanges for even better reliability

### 5.2.2 The Vacuum Tool

The vacuum tool has been developed to perform two tasks It can be used to suck up integrated circuit chips and also to lift whole circuit boards by an attached chip A schematic of this tool can be seen in figure [5 2] The tool was developed primarily for carrying and inserting chips into printed circuit boards, but its use for transporting entire hybrid circuit boards is also proving very successful

Other methods of carrying these circuit boards have been developed However, none of these are considered suitable for this repair case, as they all involve grasping the hybrid IC

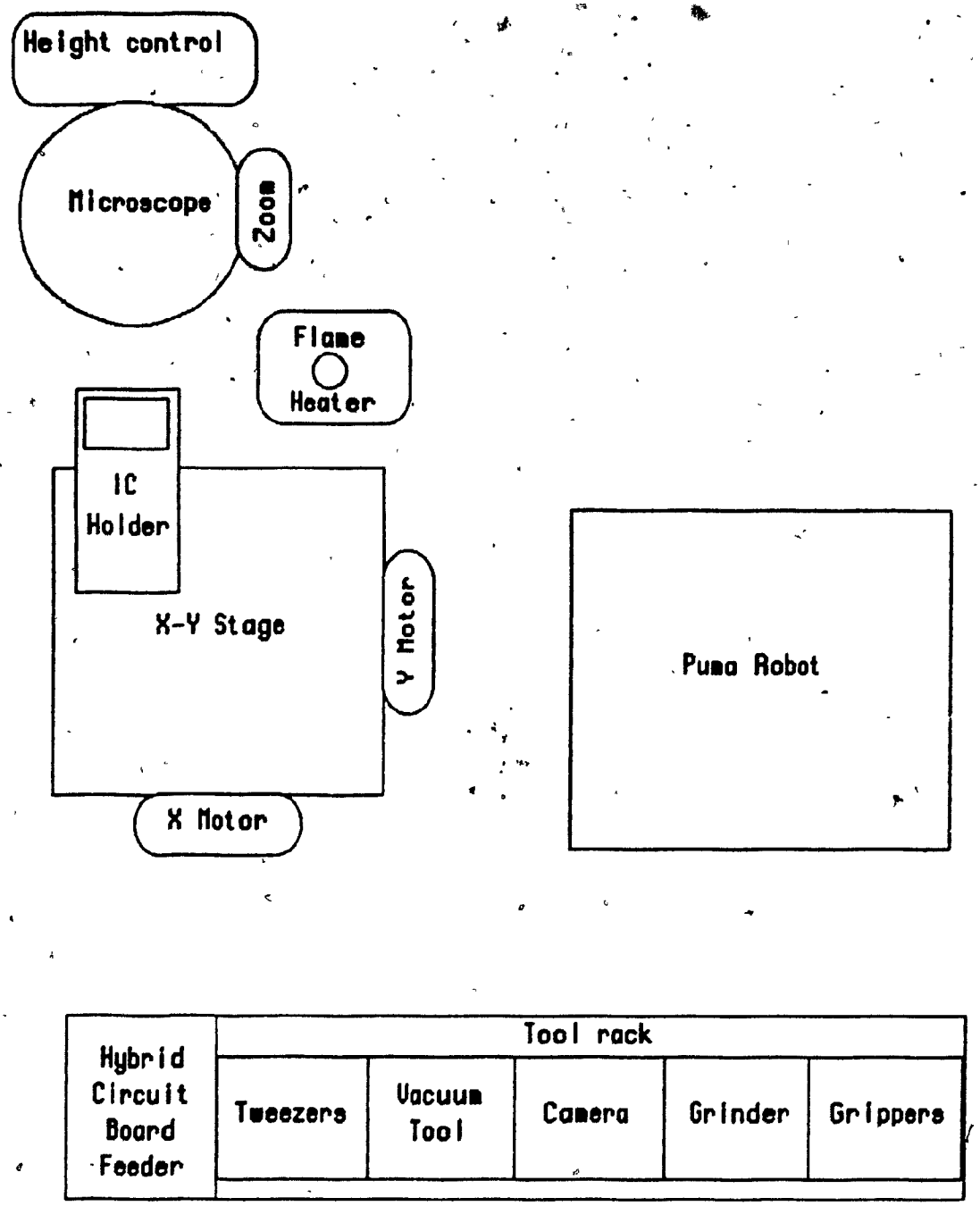


**Figure 5.1** The Physical Layout of the Repair Station

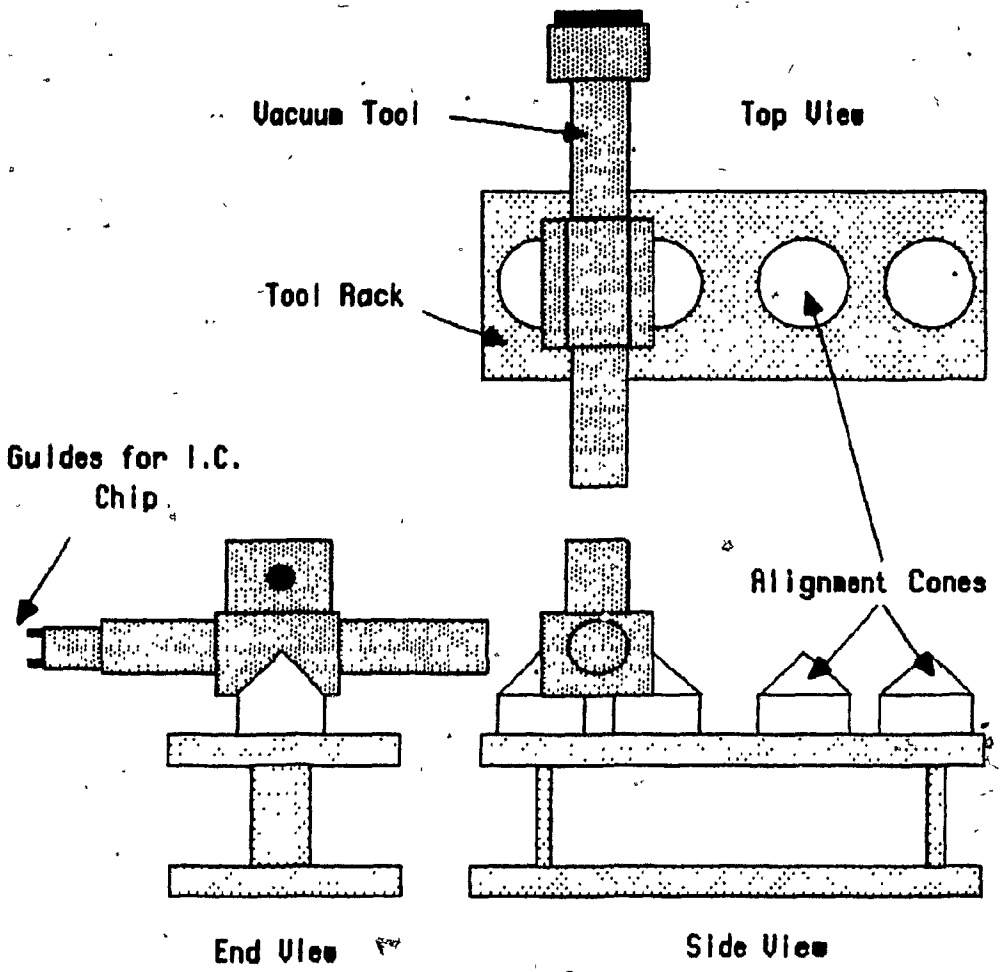


**Figure 5.2** The Tool Rack and Vacuum Tool

by its edges. This is inappropriate in this case because the edges are not free for grasping since they were used for stationary support in order to keep the upper and lower surfaces free for repair work to be carried out

### 5.2.3 The Hybrid I.C. Stand

A temporary stand has been provided for the experiment at hand. In the future an automated feeder could replace it. In this demonstration the stand serves only as an initial and final resting place for a board being worked on. Nevertheless it has been designed to present a circuit board in a consistent orientation. This is achieved as shown in figure [5.3] which again uses gravity to advantage This slanted plane of the stand also provides an opportunity to illustrate how movement in the new tool coordinate frame can be used to simplify the programming task

### 5.2.4 The Tweezers

This last tool was developed to handle the exacting precision needed in dealing with components as small as the capacitors to be desoldered. As can be seen from figure [5.4], it is made from a tube and common home tweezers. It has been designed to automatically compensate for errors in capacitor positioning as would be common in a repair task The jaws of the tweezers are forced shut around a capacitor as the robot pushes them against the surface of a hybrid IC board The capacitor may then be lifted off by them To release the capacitor from the tweezers the robot must further push against a hard surface, such as a disposal bin. The tweezers can then be reset by pushing the plunger against a surface This method of activating a small gripper allows for movements of the robot to be less precise as the tweezers are made of spring steel and slide within the tube, thus allowing for compliance in the tool's operation

### 5.2.5 The Flame Heater

For the purposes of this research, the heater consists of simple alcohol burner that has been modified by passing a stream of air through its flame. The air stream has the effect of creating a directed fine flame A flame was chosen over an electrical hot plate because

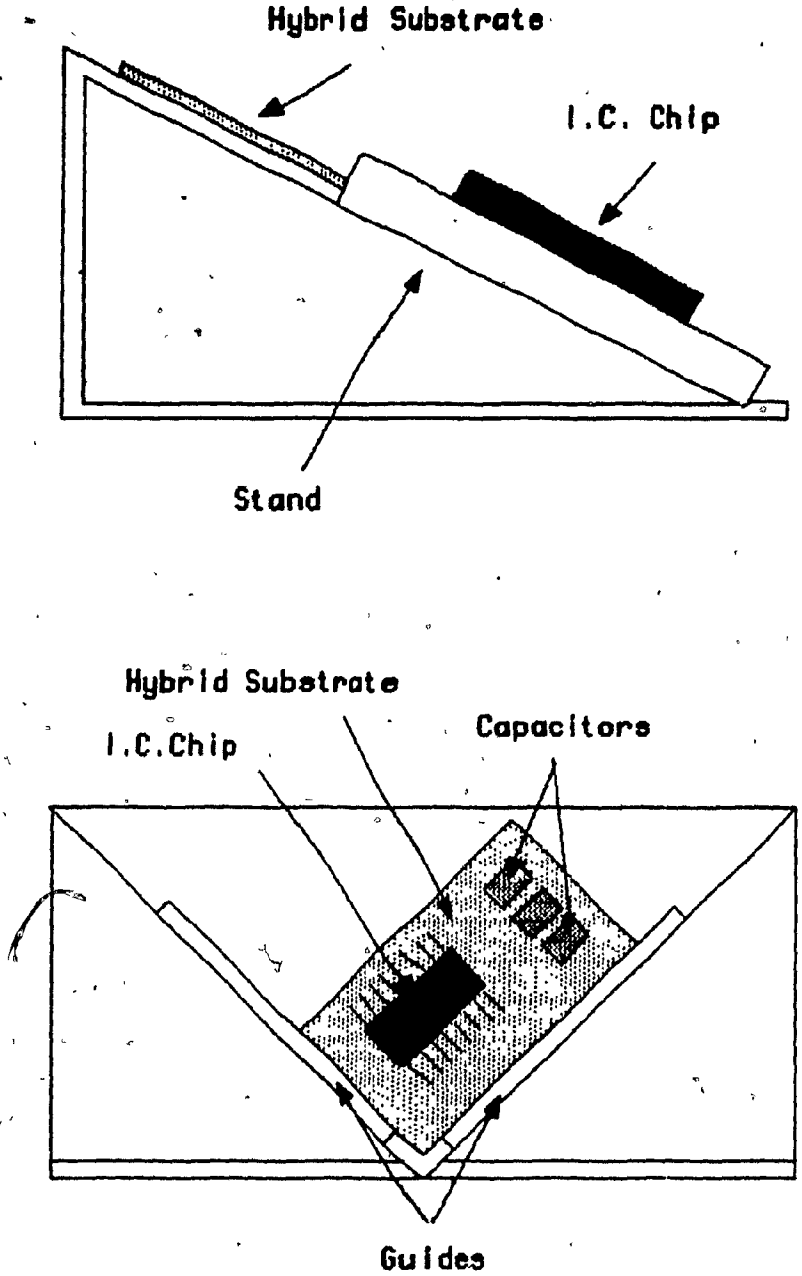


**Figure 5.3** The Hybrid I.C. Stand with I.C

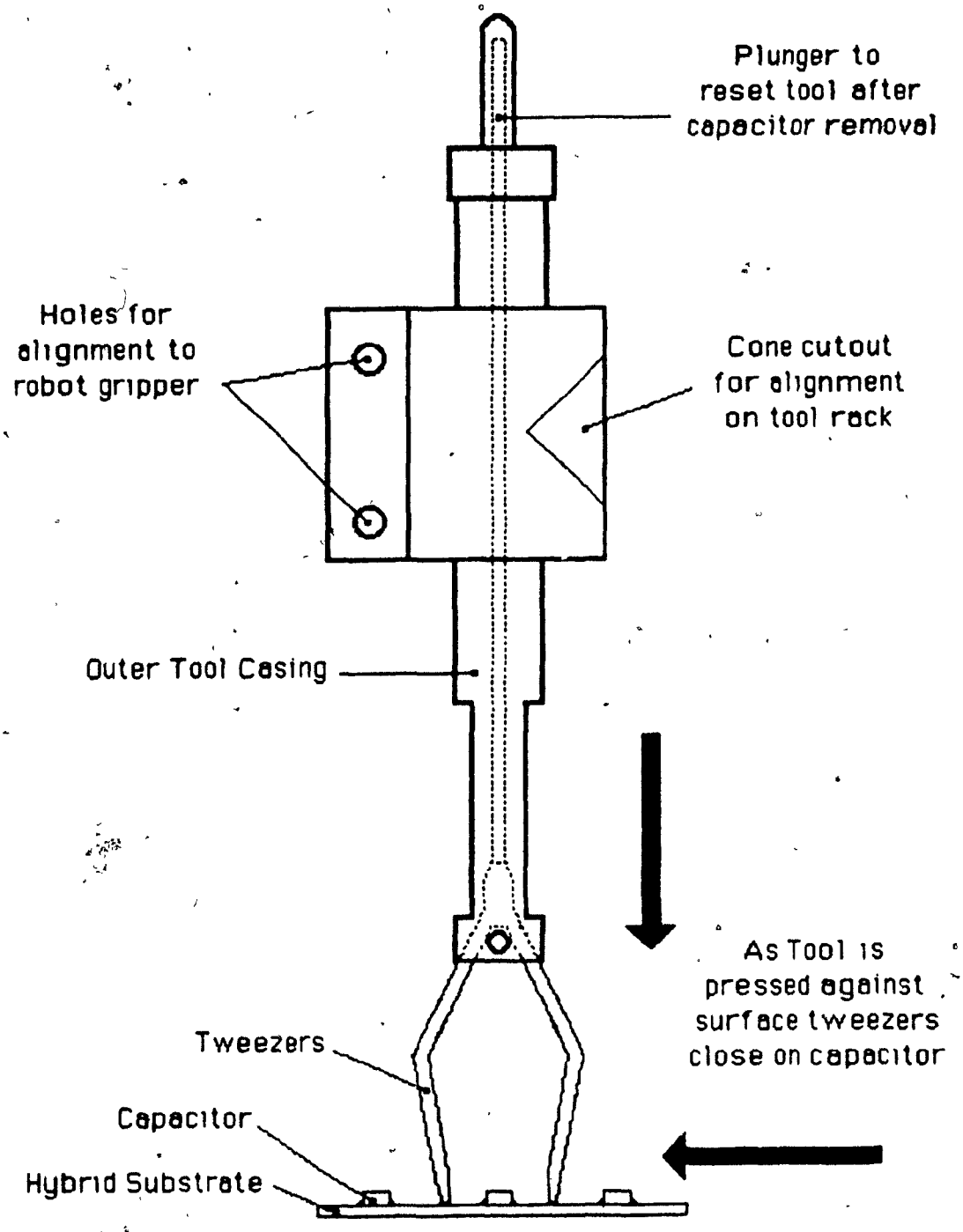


**Figure 5.4** The Tweezer Tool

heat could be transferred to the hybrid IC in a selected spot without having to make a physical contact with the board The advantage of this is that the board may be placed on the X-Y stage and moved over the flame with precision, while at the same freeing the robot to work on the top side of the board

## 5.3 Sample Program # 1

```
[1]     STAGE MOVE 0 90
[2]     GOSUB PICKUP "VACUUM.VACUUM 0 0 80.HAND"
[3]     GOSUB PICKUP "FEEDER.FEEDER 10 20 10.SUCTION"
[4]     STAGE WAIT
[5]     GOSUB PUT "STAGE.STAGE 10 0 0.SUCTION"
[6]     STAGE MOVE 40 40
[7]     GOSUB PUT "VACUUM.VACUUM 0 0 80.HAND"
[8]     GOSUB PICKUP "TWEEZERS.TWEEZERS 0 0 80.HAND"
[9]     TYPE "PLEASE ENTER THE CAPACITOR TO BE REMOVED"
[10]    WAIT CAPACITOR
[11]    STAGE TO.CAPACITOR
[12]    STAGE WAIT
[13]    GOSUB PICKUP "CAPACITOR.CAPACITOR 10 0 0.NONE"
[14]    STAGE MOVE 0 90
[15]    GOSUB PUT "GARBAGE.GARBAGE 0 0 -60.NONE"
[16]    TYPE "DO YOU WISH TO REMOVE ANOTHER CAPACITOR (Y/N)"
[17]    WAIT ANSWER
[18]    IF ANSWER THEN 9
[19]    GOSUB PUT "TWEEZERS.TWEEZERS 0 0 80.HAND"
[20]    GOSUB PICKUP "VACUUM.VACUUM 0 0 80.HAND"
[21]    GOSUB PICKUP "STAGE.STAGE 10 0 0.SUCTION"
[22]    GOSUB PUT "FEEDER.FEEDER 10 20 10.SUCTION"
[23]    GOSUB PUT "VACUUM.VACUUM 0 0 80.HAND"
[24]    TYPE "OUR LITTLE DEMO IS ALL FINISHED"
```

Sample program # 1 represents a typical robot program developed in this environment.

When run it will desolder capacitors from a hybrid circuit board. The program will now be explained. As can be seen the program calls two subroutines "pickup" and "put", these two subroutines will be described before the overall action of the program is described.

**SUBROUTINE PICKUP**

```
[1]    APPROACH TOOL @VACUUM 80 0 0
[2]    MOVE TRANSFORM !VACUUM
[3]    ACTIVATE #HAND
[4]    DEPART TOOL
[5]    MOVE TRANSFORM $READY
```

**SUBROUTINE PUT**

```
[1]    APPROACH TOOL @VACUUM 80 0 0
[2]    MOVE TRANSFORM !VACUUM
[3]    DEACTIVATE #HAND
[4]    DEPART TOOL
[5]    MOVE TRANSFORM $READY
```

As can be seen the two routines are identical except for line 3. The routine pickup will approach a given point and activate the hand of the robot or the suction tool to pick up an object The Put routine will do the same as the pick up routine except that it will deactivate the hand or suction and so put down the object it is carrying As described in chapter 4 the symbols !, @, #, and $ act as markers for the arguments 1, 2, 3, and 4 respectively, being passed to the subroutine Note that in the example program an argument to fill the $ marker was never passed thus the default argument READY was used This ready position is situated centrally in the work area and is clear of all obstacles This simplifies the use and linking of subroutine calls as the robot will always return to a safe and ready position between calls

The sample program # 1 will now be described The first line of the program will move the stage to the coordinates 0 90 This is the position at which the robot will later place a board to be repaired The next command line calls the subroutine "pickup" with

arguments "vacuum", "vacuum 0 0 80" and "hand". These arguments will cause the robot to pick up the vacuum tool with the its hand or gripper attachment, as can be seen in plate 1..The vacuum tool is approached for pick up from 80 mm in the Z direction as specified. Notice at this point the stage and the robot will be moving concurrently. After the robot has picked up the vacuum tool command line 3 will cause the same subroutine "pickup" to be called but this time the arguments specify that it is the feeder that is to be approached and the suction activated instead of the robots hand As the robot already has the vacuum tool this will cause the hybrid circuit board in the feeder to be sucked up, plate 2.

Now before the robot can place the board onto the stage it must be sure that the stage has reached its final destination and, if not, wait for it This is the purpose of the next command "stage wait" Once the robot has synchronized with the stage it will then proceed to "put" the circuit board onto the stage, plate 3 This is achieved by approaching the stage by the specified approach vector and deactivating the vacuum which is holding the circuit to the vacuum tool Once the circuit board is on the stage it is moved to a location, near the heater ready for heating While the stage is moving into the new position the robot will replace the vacuum tool to its holder with a call to the "put" routine and then pick up the tweezers which it needs to remove the tiny capacitors, plate 4

The robot now has the tool for removing capacitors and the stage is poised to heat the solder joint for removal The user is now prompted in line 9 as to which capacitor is to be removed This is simply for demonstration purposes as normally repairs would come from a list or from the vision system Once the capacitor to be removed has been entered, it is placed over the tip of the flame for heating by moving the stage by an offset corresponding to that capacitor, The offset is determined from a data file As before the stage is now checked to ensure that it has completed its motion before the robot proceeds to pick up the desoldered capacitor, plate 5 The capacitor is picked up by the tweezers which move according to the specifications in the "pickup" command, plate 6 The tweezers are activated (closed) by pushing them against the surface of the circuit board The "none" argument of the statement in line 13 is used since no activation of the robots hand or suction is desired, in effect causing line 3 of the "pickup" subroutine to have no effect

After the capacitor has been removed the stage is moved away from the heat and the bad capacitor is placed in the garbage The motion of placing the capacitor in the garbage

(line 15) also has the effect of reseting the tweezers for use in subsequent removals. At this stage the user is prompted with the "type" command. If the user continues with removals by answering yes then the sequence is resumed from line 9 with the user being asked for the next capacitor to be removed. If however the user answers no then the program will continue from directly after the "if" command at line 19. These last commands will return everything to its starting position. First the tweezers will be returned to their stand and the vacuum tool will then be picked up. The robot will then use the vacuum tool to pick up the hybrid circuit board from the stage and return it to the feeder. Once it has placed the circuit board in the feeder the robot will return the vacuum tool to its holder and a final message informing the user that all has been completed will be printed.

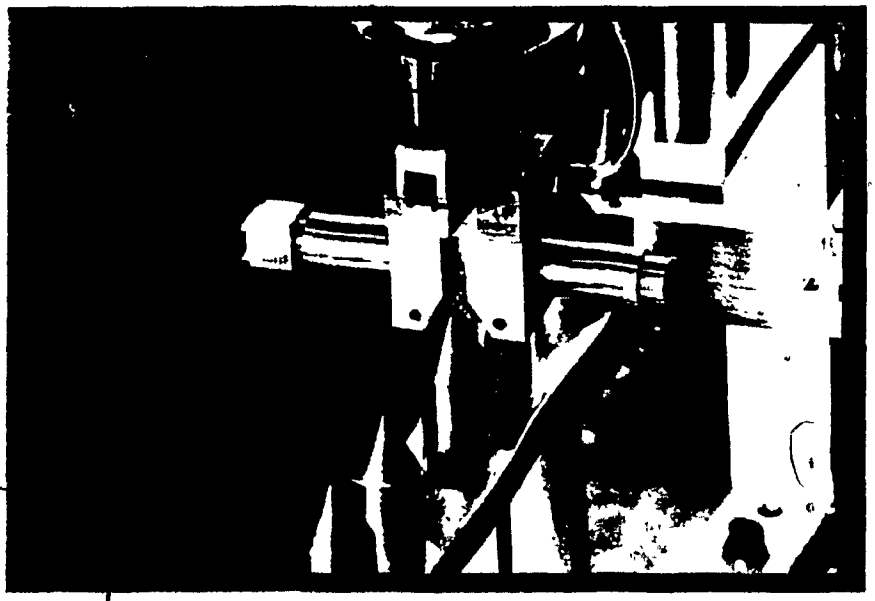

**Plate 1.**
**The Puma Robot Picks Up The Vacuum Tool**

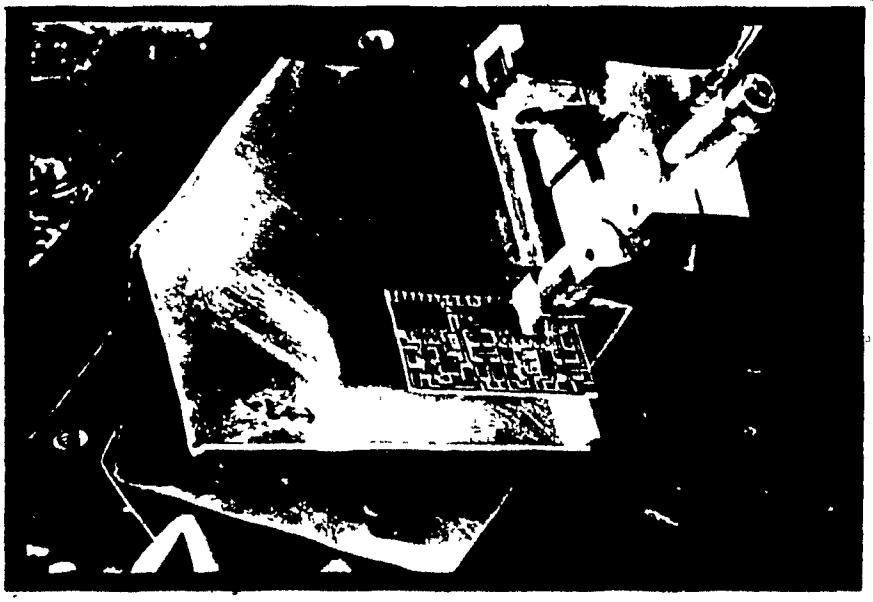

**Plate 2.**
**The Hybrid Circuit Board Is Picked Up From The Feeder**

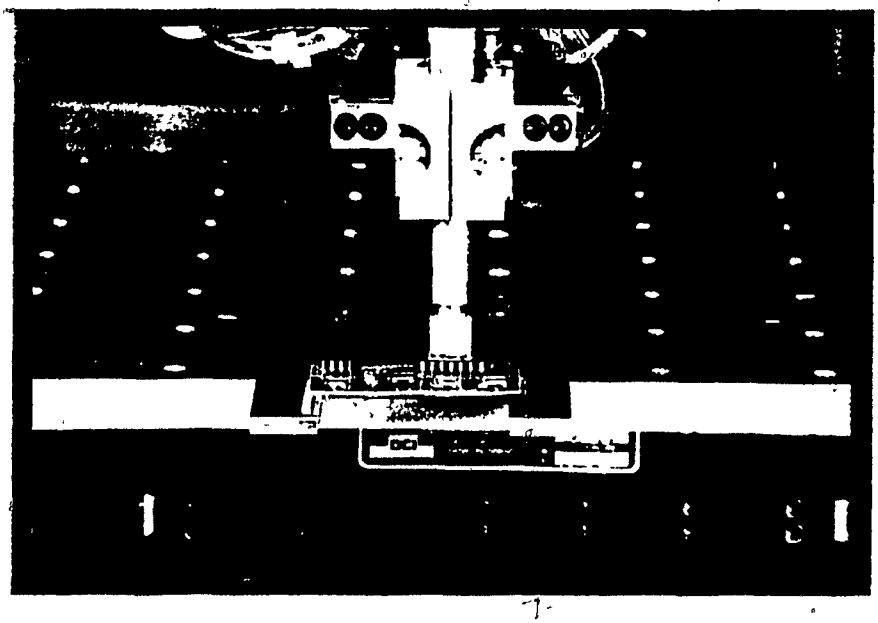

**Plate 3.**
**The Hybrid Circuit is Placed On The Stage**

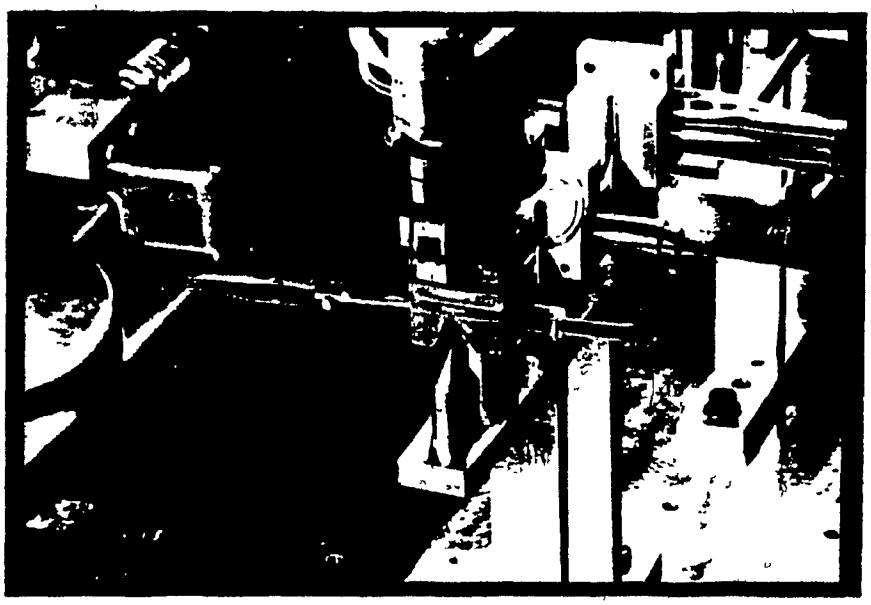

**Plate 4.**
**The Robot Picks Up The Tweezer Tool**

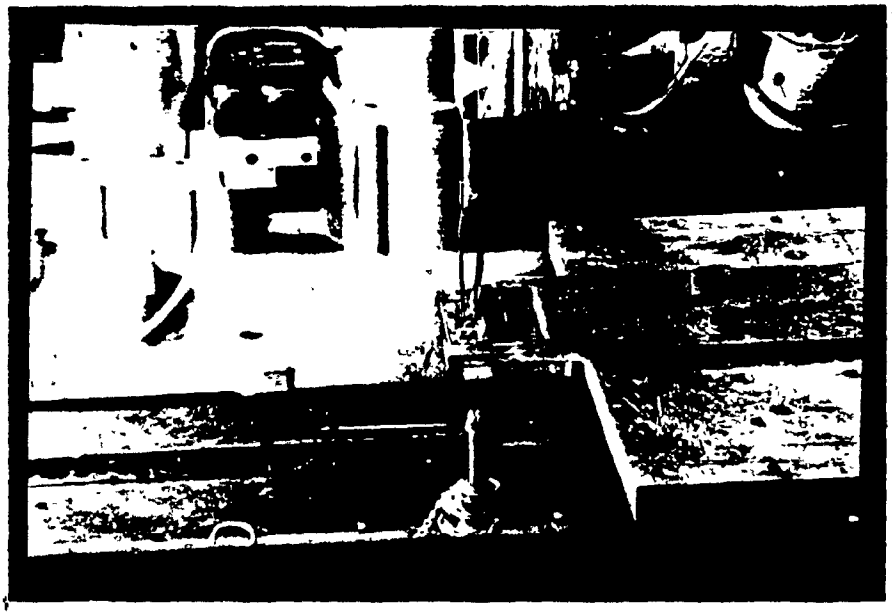

**Plate 5.**
**The Robot Is About To Remove A Heated Capacitor With The Tweezer Tool**

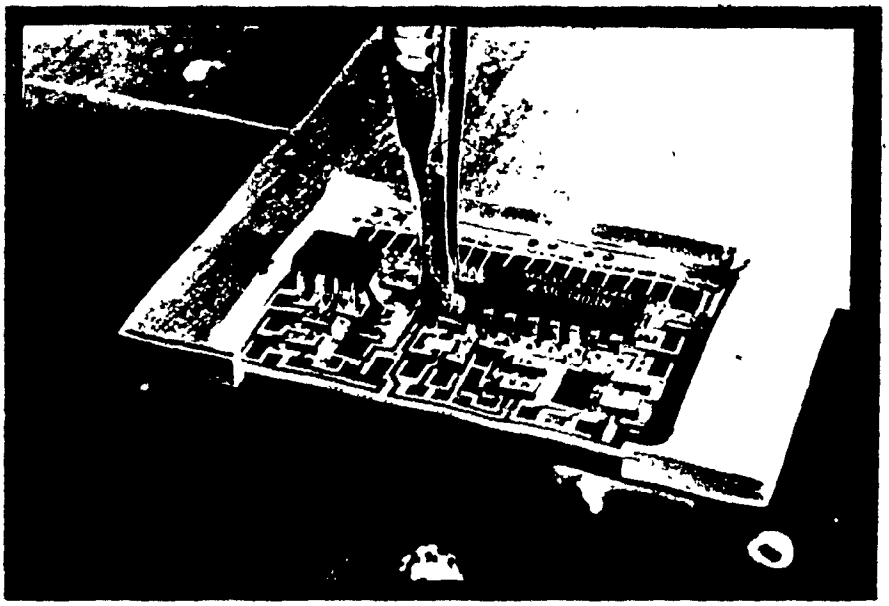

**Plate 6.**
**A Capacitor Has Been Successfully Removed**

### 5.3.1 Sample Program # 2

The second example given below shows the use of some of the vision commands and how they can be used to obtain information that the robot can then use.

```
[1]     MOVE STAGE 20 20
[2]     GOSUB PICKUP "VACUUM,VACUUM 0 0 80,HAND"
[3]     GOSUB PICKUP "TARGET,TARGET 10 20 10,SUCTION"
[4]     GOSUB PUT "STAGE,STAGE 10 0 0,SUCTION"
[5]     VISION
[6]     CAMERA FRAMES 2 CHANNEL 1
[7]     LOAD CAMERA_IMAGE
[8]     AREA 20   10   1
[9]     SET_DISTANCE 1   5   5
[10]    EXIT
[11]    GOSUB PICKUP "STAGE,STAGE 10 0 0,SUCTION"
[12]    APPROACH TRANSFORM STAGE 10
[13]    TOOL X Y 0
[14]    TOOL 0 0 10
[15]    DEACTIVATE HAND
[16]    DEPART TOOL
[17]    MOVE TRANSFORM READY
[18]    TYPE "IMAGE IS NOW CENTERED UNDER MICROSCOPE"
```

In this example the robot starts by picking up the vacuum tool and then using the vacuum tool to pick up the target object, consisting of a black rectangle containing white circles. The target is then placed on the stage under the microscope. The vision area of the command tree is now entered with the vision command. The first vision command is "camera frames 2 channel 1". This command will take two frames of the image under the microscope and average them together into the red channel. the next command loads the image from the camera image into a work image area. Once the image is in the work area it can be analyzed by the area command. This command will find the area and the

centers of the largest five regions that are above the threshold of 20 and use a window size of 10 as given in the command line. The results will also be plotted on the grinnell as specified by the 1 in the plot field. After the area command has been completed the set distance command is used to find the distance in millimeters of the largest object from the center of the image. This information is stored in the global variables X and Y that the tool command of line 13 will use later. Thus the vision processing is now complete and the vision subtree is left with the exit command. The robot is now moved to pick up the target from the stage and then repositions it using the tool command. by the distance obtained from the vision process Once the target has been repositioned. it is released onto the stage and the robot is returned to the ready position and a message that the target has been centered is printed.

## 5.4 Discussion

As can be seen from the example programs. development of useful programs can be achieved in a relatively few lines of code. The use of subroutines for the functions of pick and place have substantially reduced the amount of code. It has also been seen how easy it is to develop specialized commands tailored to the needs of the environment. The first program gave as an example a function that would be needed in the repair of hybrid integrated circuit boards namely the removal of capacitors from the boards. The second example has shown how a vision system can be used to give feedback to the robot so that a target could be centered under the microscope. This second example would find uses in the automated inspection of hybrid boards were a defect or flaw might need to be centered so that analysis can be done on that area of the board.

Extensions to the existing environment are easy to make and can be done in a modular fashion. Either a single command may be developed in C and linked into the environment or indeed a whole tree of commands or packages may be developed and linked in. These packages can then be used just like any of the other commands with all the editing and debugging power of the system at hand. This has the advantage of allowing independent development of packages or algorithms. Once these algorithms have been developed they may be incorporated into the system and tested with all the available equipment of the station. This also allows for the station to be tailored to special situations by the addition

and removal of packages of specialized commands.

The environment as mentioned previously is implemented in the C programming language and is approximately six thousand lines of code. The size of the executable module is some two hundred kilobytes. On the VAX 750 this code runs without any noticeable delays to user commands. However when the system is loaded, pauses between successive robot motions can be seen. These pauses can happen if the RAP system has been swapped out of core between motion request to the VAL system. Pauses such as these could be avoided by running the system at a greater priority, but for the research at hand such pauses were not a major concern.

The size of the executable code is quite modest considering the variety of options available to the user, and on a VAX presents no problems. In comparison a typical RCCL program compiles to at least four hundred kilobytes. For the code to be ported to a smaller system such as a micro computer, the executable could be trimmed down, if a size limitation problem arose. Size could be reduced by including only those packages which would be needed in the experiment at hand, another advantage of a modular design.

Future expansions to the system could include a graphics interface, improved numerical processing, and a more comprehensive vision package.

A graphics interface could be implemented on one of the SUN work stations recently acquired by the lab or even on a micro with graphics capabilities such as the Apple Macintosh. The Interface should offer both mouse and keyboard activated menus in addition to the present keytree matcher command entry system. Running programs could be displayed with a line pointer to indicate what line was executing instead of simply printing the current line to the screen as is now done. Editing capabilities should be expanded to support full screen editing instead on the line oriented type of editing presently supported. A graphics interface would also present the opportunity to move the code from a time shared VAX to a smaller dedicated processor on which the interface was being written.

Improved numerical processing would be needed if the robot is to be used in an industrial type environment as opposed to a research one. This is because in industry functions are usually repeated many times thus creating the need for complex loops and palletization type operations. In research on the other hand, many types of different functions are tested, as opposed to one function being tested over and over. It is realized that any commercial

system would have to support complex numerical processing, and as the implementation is in C such an addition would not present any major problems.

A more comprehensive vision package is the third area where expansion is needed. The basic facilities for frame grabbing, viewing, storing, and simple processing of images need to be augmented with image analysis and pattern recognition algorithms McGill's work in this area has lead to many research topics in themselves, and such additions while still in the development stages are progressing rapidly.

Shortcomings of the system arise mainly from the method of controlling the robot, through VAL. This means that no matter how sophisticated the vision algorithms may become, the robot will still be limited to motions and trajectories that VAL can produce. This situation is made worse by using VAL in a terminal mode since features such as continuous path smoothing, which VAL can normally provide, must be forfeited. The smoothing feature of VAL must be given up because commands are now being sent one at a time so VAL cannot calculate interpolations between motions as it does not get the next motion until the previous one is complete An instantaneous stop in the motion of the robot can be seen between such commands.

Such limitations can be eliminated by using RCCL control instead of VAL This switch would mean that the greater trajectory control of RCCL and its force control primitives could be accessed and used for better control of the robot As RCCL operates on a motion queue strategy, more than one motion command may be sent at a time thus allowing smooth transition between motions The switch to RCCL would not be difficult to do and indeed none of the system features would need to be given up. All that would be needed would be a set of C subroutines to be written to call the appropriate RCCL routines. Entries for these subroutines would then be made in the key tree matcher and the tree of commands incorporated into the system.

When the system is viewed like this, it is seen to be a really convenient way of specifying the order that a set of subroutines should be run. It is like having a main program that is compiled and linked but the order of execution can be determined on the fly and recorded for later play back. This novel way of programming is the heart of the flexibility and expandability of the system

At the present it can be seen that the types of operations, although a step above what

could previously be done. are still on the primitive side. This is not a fault of the system It is due to the lack of sensors in the work station, and lack of tools which can provide feedback as they perform their job.

The cameras require very involved and time consuming algorithms to extract even the simplest feedback for the robot. In addition the cameras must be calibrated to the robot, a very complicated procedure in itself Also the cameras are rather large and must be mounted in areas where their range of view is limited or obscured by the moving robot. Thus although the cameras have the potential to provide much feedback they are still many problems faced when using them.

Tools at the present simply perform grasping, sucking, or grinding actions. There are no sensors associated with the tools. This has meant that the tools have been designed to use mechanical stops, guides and alignment features to try and eliminate any uncertainties in the operations they are to perform. While this is a credit to the ingenuity of their designers it leaves the robot at the mercy of the tool's accuracy and repeatability with no way of knowing if any thing has gone wrong with its use. In the future, tools must also incorporate sensory feedback that can confirm their failure or success in carrying out an operation. The system has been designed with this in mind as can be seen from its ability to test input lines which could be tied to the appropriate sensors on the tools

In summing up it can be seen that the system offers a great improvement in terms of turn around time and ease of programming, verifying, and debugging of a task However it is also seen that for the tasks to become more sophisticated there are many stumbling blocks still to be overcome. Efficient vision algorithms as well as dedicated hardware processors are needed to both improve and speed up the feedback that can be obtained from the cameras Tools must be built with feedback capabilities even if only binary, to confirm an action they have carried out In addition proximity sensors and force feedback sensors could play a vital role in avoiding obstacles and locating parts. However like the cameras these sensors will need dedicated processors so that data from them can be analyzed and compressed into a form that the robot can use. Here it is seen that the system described can easily be expanded to accommodate input from these sensory type devices as was done with the camera system. This flexibility, expandability, and ease of use are the main features of the system designed

# Chapter 6 Conclusion

This thesis has presented an environment for Robotic Applications Programming on a VAX 11/750 under UNIX 4.2 BSD. The environment RAP allows for the control of a PUMA 260 robot, an X-Y stage, and a vision system.

RAP features a full range of interactive programming and debugging tools. A user may compose and watch programs run an instruction at a time or program in an off line mode as needed. Program files may be run in a step by step fashion in both a forward or backward direction. Debugging may also be achieved by the use of strategically placed break points. Running programs may be stopped or paused, to allow the user to modify any positions or equipment in the environment, and be then restarted at any point.

The control of the PUMA is achieved by interfacing a VAX 11/750 to the LSI/11 PUMA controller and building on the VAL programming language. As the interface makes use of the controller's RS-232 terminal port no hardware modification need be made to the Unimation robot controller package. Despite having to route all control commands through VAL, the author has developed extended commands for the movement of the robot in its tool coordinate frame and argument passing to subroutines

A stepper motor controller was similarly interfaced to the host computer This stepper motor controller was used to control four motors Two of the motors were used to control the movements of an X-Y stage The remaining two motors were used to automate the focusing and zoom of the microscope.

Algorithms for doing vision processing were incorporated and can be called by user programs. The vision routines are modular and can be easily expanded to include routines from the rich libraries at McGill such as the HIPS and SPIDER packages. This opens new

avenues for the use of complex sensory feedback in robot programs. It has been seen that such feedback is vital in all but the most simple pick and place type applications.

A program for the desoldering of capacitors from a hybrid circuit board has been given as an example to show off some of the system features. The program demonstrated a typical hybrid circuit board repair function that would be needed in the automation of the repair process The use of subroutines, movement in the tool coordinate frame, and vision feedback have all been incorporated into the examples. The performance of the system was evaluated and recommendations presented.

All tools developed in the course of presenting the example have also been fully described and schematic diagrams given where appropriate. These tools include, a vacuum tool for the transportation of the hybrid circuit boards, a tweezer type tool for the grasping and removal of capacitors, and a rack for the holding of these tools while not in use.

The project as a whole has exposed the author to many areas in the robotics field Topics such as distributed processing, vision and sensory feedback, world modeling, and robotic programming languages all had to be addressed. The project is seen as such, not only to be of a use to industry in the repair process, but also as a vehicle for the research of this dynamic and fast growing field.

# References


[1.] **B.E. Shimano, C.C. Geschke, C.H. Spalding III,** "VAL-II: A New Robot Control System for Automated Manufacturing," *Proceedings IEEE Conference on Robotics*, pg. 278 - 292, 1984

[2.] **L.I. Lieberman and M.A. Wesley,** "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly," *IBM Journal of Research and Development*, Vol. 21, no. 4, pg 321-333, July 1977.

[3.] **M. Ejiri, T. Uno, H. Yoda, T. Goto, and K. Takeyasu,** "A Prototype Intelligent Robot that Assembles Objects from Plane Drawings," *IEEE Transactions on computers*, Vol C-21 pg 161 - 170, February 1972.

[4.] **P.M. Will, D.D. Grossman,** "An Experimental System for Computer Controlled Mechanical Assembly," *IEEE Transactions on Computers*, Vol C-24 pg. 879 - 888, 1975

[5.] **R. Paul,** "WAVE - A Model Based Language for Manipulator Control," *The Industrial Robot*, Vol. 4, No. 1, pg 10-17, March 1977

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

[7.] **R. Evans, D. Garnett, and D. Grossman,** "Software System for a Computer Controlled Manipulator," *IBM T.J. Watson Research Center*, RC 6210, May 1976.

[8.] **R. Taylor, P. Summers, and J. Meyer,** "AML: A Manufacturing Language," *Robotics Research*, 1.3, 1982

[9.] **W. Park,** "Minicomputer Software Organization for Control of Industrial Robots," *Joint Automatic Control Conference*, San Francisco, 1977.

[10.] **J.J. Craig,** "JPL Autonomous Robot System," *Jet Propulsion Laboratory*, Pasadena, CA, 1980

[11.] **C.F. Ruoff,** "TEACH - A Concurrent Robot Control Language," *Proceedings of the Third International Computer Conference*, The Palmer House, Chicago, Illinois, pg. 442-445, November 1979.

[12.] **W. Franklin, and G. Vanderburg,** "Programming Vision and Robotics Systems with RAIL," *SME Robots*, VI, pg 392 - 406, March 1982.

[13.] **General Electric Company,** "Allegro Operators Manual (A12 Assembly Robot)." *General Electric Company*, Bridgeport, CN, 1982

[14.] **B.E. Shimano,** "VAL - A Versatile Robot Programming and Control System." *Proceedings of the Third International Computer Conference*, IEEE Computer Society, The Palmer House, Chicago, Illinois, pg 878-883, November 1979.

[15.] **K. Takase, R.P. Paul, and E.J. Berg,** "A Structured Approach to Robot Programming and Teaching." *IEEE Transactions SMC*, Vol. SMC-11, no. 4, pg. 274-289, April 1981.

[16.] **V. Hayward,** "RCCL User's Manual Version 1.0." *Technical Report*, TR-EE 83-46, School of Electrical Engineering Purdue University, West Lafayette, Indiana, 47907, October 1983.

[17.] **McDonnell Douglas** "Robotic System for Batch Manufacturing, Task B High Level User Manual." *Technical Report*, AFML-JR-79-4202, Wright Patterson Air Force Base, OH, October 1981.

[18.] **J. Darringer, and M. Blasben,** "MAPLE: A High Level Language for Research in Machine Assembly." *IBM T.J. Watson Research Center*, RC 5606, September 1975.

[19.] **M. Salmon,** "SIGLA: The Olivetti SIGMA Robot Programming Language." *Eight International Symposium on Industrial Robots*, Stuttgart, West Germany, June 1978.

[20.] **G. Gini, M. Gini, R. Gini, and D. Giuse,** "Introducing Software Systems in Industrial Robots." *Ninth International Symposium on Industrial Robots*, Washington D.C., pg. 309 - 321, March 1979.

[21.] **D. Falek and M. Parent,** "An Evolutive Language for an Intelligent Robot." *The Industrial Robot*, pg. 168-171, 1980.

[22.] **J. Latombe, and E. Mazer,** "LM: A High Level Language for Controlling Assembly Robots." *Eleventh International Symposium on Industrial Robots*, Tokyo, Japan, October 1981.

[23.] **R. Popplestone, A. Ambler and I. Velos,** "RAPT: A Language for Describing Assemblies." *The Industrial Robot*, pg. 131-137, September 1978.

[24.] **T. Lozano-Pérez and, P. Winston ,** "LAMA: a Language for Automatic Mechanical Assembly". *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, MIT, Cambridge, Massachusetts, pg. 710-716, August 1977.

[25.] **E. Mazer,** "LM-Geo: Geometric Programming of Assembly Robots." *Laboratoire IMAG*, Grenoble, France, 1982.

[26.] **M. Week, and D. Zuhlke,** "Fundementals for the Development of a High Level Programming Language for Numerically Controlled Industrial Robots." *AUTOFACT West*, Dearborn, Michigan, 1981.

[27.] **P. Bork,** "Controling Robots with an English-like High Level Hierarchical Command Language (HIROB)." *IEEE*, pg. 404-412, 1984.

[28.] **D. Gauthier, G. Carayannis, P. Freedman and A. Malowany,** "A Session Layer for a Distributed Robotics Environment." *IEEE Proceedings Compint;* pg.459-465, September 1985.

[29.] **CVaRL,** "Progress Report." *Technical Report 85-10R*, McGill University, Computer Vision and Robotics Laboratory, September 1985.

[30.] **Y. Cohen, M.S. Landy,** "The HIPS Picture Processing Software Reference Manual." Psychology Department, New York University, Janurary 1983.

[31.] **J.S.D.,** "SPIDER User's Manual." *Joint System Development Corp.*, Tokyo, Japan, December 1983.

[32.] **A. Mansouri, A. Malowany, M. D. Levine,** "Line Detection in Digital Pictures: A Hypothesis Prediction/Verification Paradigm", *Technical Report 85-17R*, McGill University, Computer Vision and Robotics Laboratory, 1985.

[33.] **A. Mansouri, A. Malowany,** "Using Vision Feedback in Printed-Circuit Board Assembly", *Proceedings of the 1985 IEEE Microprocessor Forum*, Atlantic City, April 1985.

[34.] **L.XU, and J. Chen,** "AUTOBASE: A System which Automatically Establishes the Geometry Knowledge Base." *IEEE Proceedings Compint*, pp 708-714, September 1985.

[35.] **B. Mack and M.M. Bayoumi,** " An Ultrasonic Obstacle Avoidance System for a Unimation PUMA 550 Robot." *IEEE Proceedings Compint*, pg. 481-483, September 1985.

[36.] **J. Llyod,** "Robot Control Interface under UNIX." *M. Eng. Thesis*, Dept. of Electrical Engineering, McGill University, Montréal, Canada, (Spring 1986)

[37.] **D. Kossman,** "A Multi - Microprocessor - Based Control Environment for Industrial Robots." *M. Eng. Thesis*, Dept. of Electrical Engineering, McGill University, Montréal, Canada. (Spring 1986).

[38.] **A.C. Sanderson and G. Perry,** "Sensory - Based Robotic Assembly Systems Reasearch and Applications in Electronic Manufacturing." *Proceedings of IEEE.* vol. 71 no. 7, pg. 856-871 July 1983.

[39.] **B.W. Kernighan and D.M.Ritchie,** "The C Programming Language." *Prentice-Hall Software series.* 1978.

[40.] **Unimation,** "Unimate PUMA Robot 200 Series Volume 1 Equipment Manual." Unimation Inc., Danbury, CT. August 1983.

[41.] **A. Mansouri,** "A report on the Stepper Motor Controller Interface to the VAX." *Technical Report,* McGill University, Computer Vision and Robotics Laboratory, March 1985.

[42.] **Microbo,** "I.R.L. Intuitive Robot Language." Version 3.2, Microbo S.A., Beau-reguard, Ch-2035, Corcelles, Switzerland, 1984.

[43.] **T. Lozano-Pérez,** "Robot Programming." *MIT AI Memo,* no. 698, 1982.

[44.] **Unimation,** "Unimate PUMA Robot 200 Series Volume 2 User's Manual." Unimation Inc., Danbury, CT. August 1983.

[45.] **Unimation,** "Users guide to VAL-II." Unimation inc., Danbury, CT, April 1983.

[46.] **H. Gomaa, R. Captenter, and J. Popelas,** "Menu Programming - An Environment for Programming Robots." *IEEE Proceedings Compint,* pg. 466-470, 1985.

[47.] **R.H. Kirsehbrown and R.C. Dorf,** "KARMA a Knowledge-Based Robot Manipulation System." *Robotics,* vol. 1 no. 1, pg. 3-12, May 1985.

[48.] **H. Lechtman et al.,** "Conecting the PUMA Robot with the MIC Vision System and Other Sensors." *Robots VI Conference Proceedings,* pg. 447-466, March 1982.

[49.] **G. Carayannis,** "Controlling the PUMA 260 Robot from a VAX." *Technical Report,* 83-3R, McGill University, Computer Vision and Robotics Laboratory, April 1983.

[50.] **M. Parker,** "A Key Tree Matcher Tutorial for User's." *Internal report,* Electrical Engineering Department, McGill University Montréal, Canada.

[51.] **A. Bazerghi et al.,** "An Exact Kinematic Model of PUMA 600 Manipulator." *IEEE Transactions on Systems, man, and Cybernetics,* vol. smc-14, no. 3, pp 483-487, May/June 1984.

[52.] R.P. Paul, "Robot Manipulators. Mathematics, Programming, and Control." *MIT Press*, Cambridge, Mass., 1981.

[53.] D.Z. Bauks, "Automated Hybrid Assembly Systems for the Electronic Factory of the Future." *International Journal for Hybrid Microelectronics*, Intrnational Microelectron Symposium, Philadelphia, Pa, USA, pp 40-42 October 1983.