A FORTH BASED ROBOTICS LANGUAGE

A FORTH BASED ROBOTICS LANGUAGE

By

TONY DIFRUSCIO, B.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (1991) McMASTER UNIVERSITY
(Computation) Hamilton, Ontario

TITLE: A Forth Based Robotics Language

AUTHOR: Tony O. DiFruscio, B.Sc. (Niagara University)

SUPERVISOR: Professor N. Solntseff

NUMBER OF PAGES: v, 135

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The increase in demand for the analysis of trace organic compounds, for both commercial and research purposes, has prompted chemists to produce new methodologies which increase both the sensitivity and cost effectiveness of analytical techniques. This demand may be attributed to the increasing awareness of environmental concerns and of drug abuse. These concerns led to the development of an automated sample preparation system (ASPS) at McMaster University Pathology Laboratory.

1.1 THE HISTORY OF THE DEVELOPMENT OF THE ASPS

The ASPS was developed to automate the sample preparation of a wide variety of samples using the solid supported technique [Povilonis 1988]. This sample preparation technique was developed to improve the separation of analytes (chemical constituents being analyzed) from a wide variety of sample matrixes (the substance in which the analyte is contained).

The simplicity of this analytical technique, the technique's ability to use small sample and reagent volumes, and the high costs associated with this labour intensive task prompted the exploration of sample preparation automation. Dr. Jack

Rosenfeld who is involved in ongoing research in solid supported sample preparation at McMaster University Medical Centre, spearheaded the development of an automated system [Povilonis 1988].

The main reasons for the design and development of the ASPS are the reduction of human error in repetitive tasks, the improvement of quality control, the reduction of production and research costs, and maximum flexibility and modifiability. The flexibility and modifiability of the system is extremely important since new sample preparation procedures and techniques are currently being developed. It must therefore be adaptable to the highly repetitive commercial use as well as the dynamic environment experienced in the research laboratory [Povilonis 1988].

The design which best fit these criteria is a robotic operated containment system. Sample preparation is automated by utilizing a robotic manipulator arm which is synchronized with a number of devices. Simplicity of operation is effected through a sophisticated software system which allows a laboratory technician to easily program the ASPS.

Upon the completion and testing of the ASPS, it was quickly recognized that the software system required extensive work in order for the ASPS to fulfil the design criteria. It was decided by Rosenfeld and Povilonis that a computer programmer was necessary to design and implement a software system satisfying the aforementioned design criteria.

1.2 CONSIDERATIONS LEADING TO THE DESIGN AND IMPLEMENTATION OF THE ASPS SYSTEM SOFTWARE

A review of the ASPS, the robotic language, and the ASPS software revealed the need for a strategy to determine the best approach for implementing the software system. It was decided that firstly, the ASPS robotic language, RAPL (CRS PLUS robotic language supplied with the robot arm), would be reviewed to determine whether it was adequate for the development of complex robot programs. This review involved a literature search and review of existing robot languages for current developments followed by a comparison of RAPL to these languages.

The next stage is to review the ASPS requirements and explore the possibilities for the enhancement of the system through software. Armed with this information, a decision could be made on the direction of the software development. Figure 1.1 shows the path used to determine the direction in which the software development would ensue. If development of a new robot arm language is required, this project's scope would be limited to that stage; otherwise, this project would go on to the next phase - use of the chosen language to develop user-level ASPS software.

After pursuing the above strategy, it was decided that a new robotics language be designed and implemented for use as the base language for the implementation of the ASPS software. This language is based on the Forth environment and implemented on an IBM-PC. The ensuing text provides an overview of robotics

hardware and software, a literature review of existing robotics languages, a review of the current ASPS hardware and software, and a detailed description of the design and implementation of the Forth based robotics language.

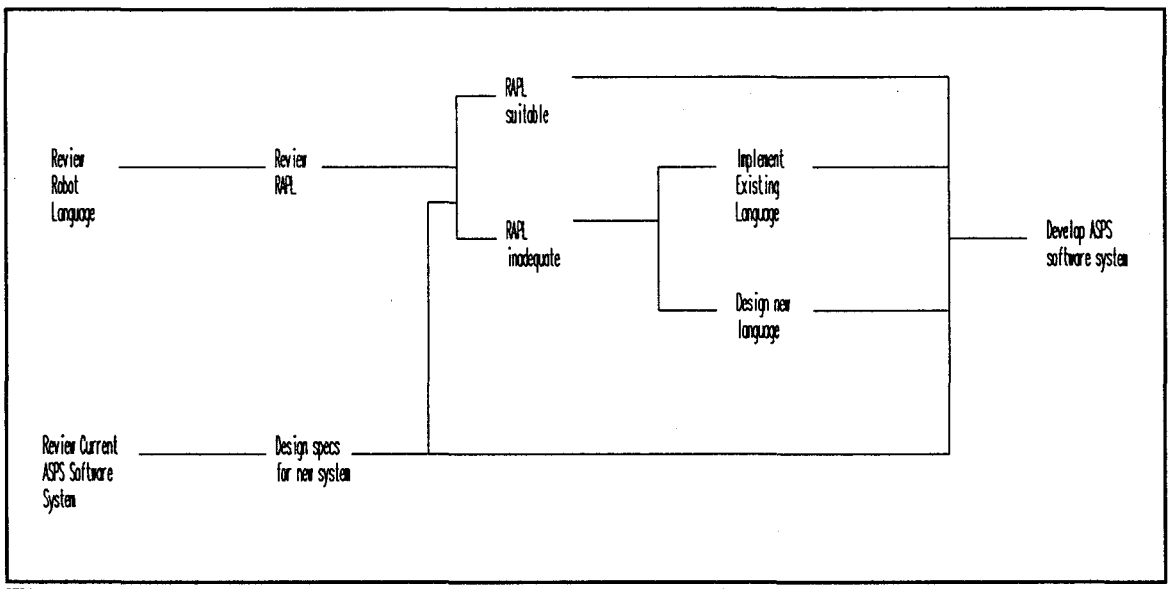

Figure 1.1 Flow diagram of decision strategy for software design

CHAPTER 2

OVERVIEW OF ROBOTICS HARDWARE AND SOFTWARE CONCEPTS

This chapter is designed to give the reader a brief overview of terminology, robotics hardware, common robotics language features and a review of the existing robotic languages.

2.1 ROBOTICS HARDWARE

The term "robot" was introduced by Karel Capek in a play about a society with automated workers [Capek 1923]. Robot is the Czechlozlovachian word for worker. The term was adopted by scientists and engineers who participated in the development of early industrial robots [Engel 1980].

R. Goertz developed manipulators for use in handling radioactive materials in the early 1950's [Goertz 1952] and the first computer controlled robot was developed by Ernst at M.I.T. in 1961 [Ernst 1961].

There is no clear cut definition for the term robot but "typically definitions encompass notions of mobility, programmability, and the use of sensory feedback in determining subsequent behaviour" [Korein 1987]. Robots are used to increase productivity and reduce costs in labour intensive tasks. Programmable robots are extremely useful for tasks which may vary as found in the instance of manufactured

products which may vary over time. The programmable robot arm is extremely adaptable to changes in the automated procedures. Robots are also useful for applications which are not suited to human abilities such as the manipulation of small objects for example, electronic parts, as well as large objects such as turbine blades. Another application is work in environments which are not suited, or dangerous to humans. Clean rooms, furnaces, high radiation areas, and space are among a few examples of these environments [Nevins 1980].

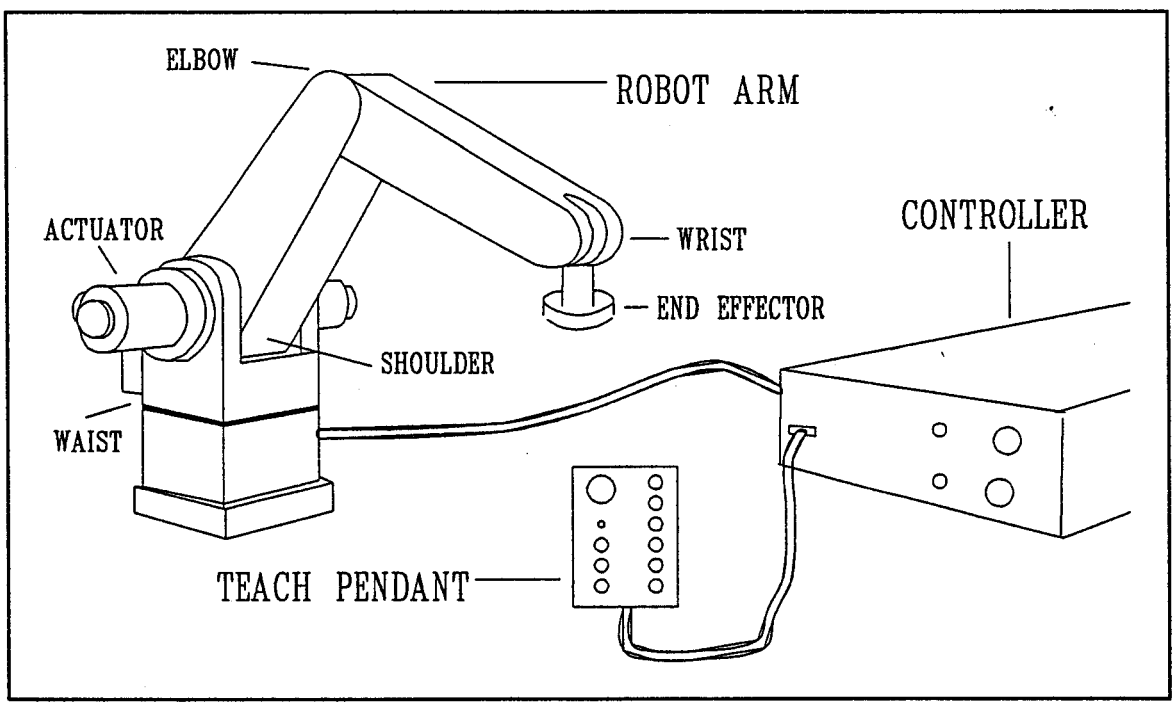

Figure 2.1 Typical robot arm and controller

A typical robot consists of an arm and a controller (see figure 2.1). The controller is the computer system used to control the robot servo control system. The controller is typically a microcomputer which runs the robot language, performs the trajectory calculations and drives the actuators. There are a number of robot arm configurations five of which are illustrated in Figure 2.2 [Korein 1987].

The grasping component or the component which houses the robot tool is called its end effector or tool tip. An actuator is the vehicle by which a robot arm axis is moved. It may be a pulse driven motor, a pneumatic device, a hydraulic device or a solenoid amongst many others.

Many commercially available robots include a teach pendant which allows manual movement of each actuator via an electronic hand held controller. The operator may use the pendant to guide the arm to specific locations.

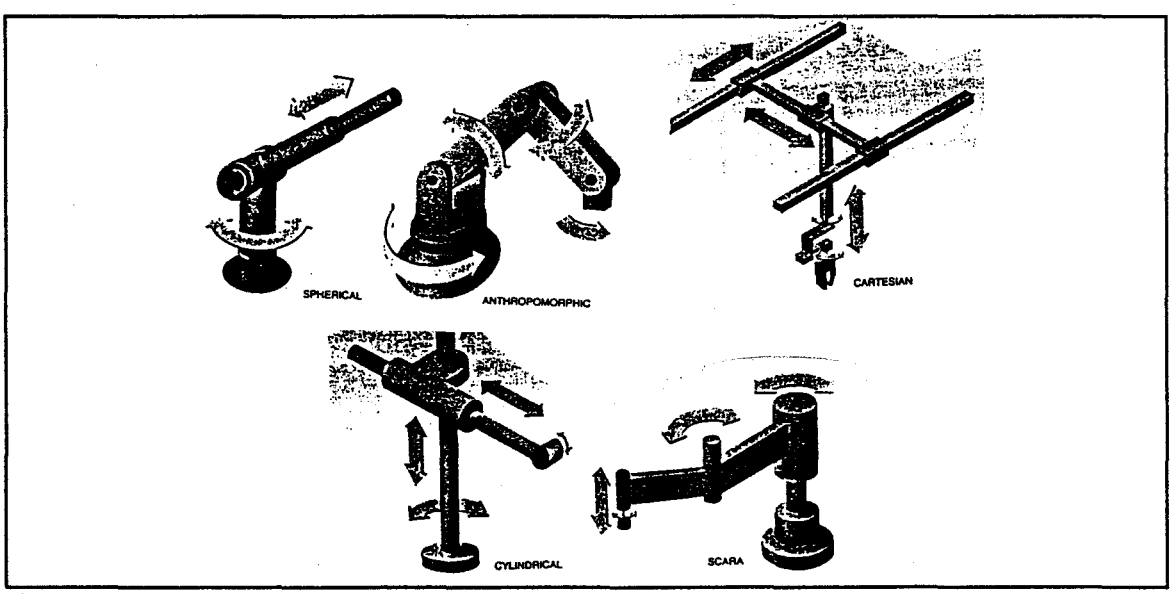

Figure 2.2 Robot arm configurations

2.2 REVIEW OF ROBOTICS LANGUAGES

When useful industrial robots were developed, programming languages were developed to teach them a fixed sequence of moves, i.e., the programmer would guide the arm through movements, teaching positions to the robot which would subsequently blindly follow these instructions [Shahinpoor 1987]. Since this technique was sufficient to develop useful repetitious industrial tasks, there was no need to include provisions for developing complex algorithms or for languages which could express such algorithms. The industrial robot language was simple to use but lacked the flexibility and power required to develop more complex applications.

The original programs suffered from lack of concurrency. A motion could be planned and then executed but during execution no other processing could be performed. Further, the manipulator had to be brought to rest between manouevers in order to perform the calculations for the next move. This problem was resolved by running two concurrent tasks. One task would control the robot and the other would plan the next move which now enabled the robot to go from one manoeuvre to the next without stopping at intermediate positions [Cox 1989].

Unfortunately, there were no operating systems which were available to handle such a task so that special standalone robot-control systems had to be developed to provide for concurrency. Consequently, high level language features such as data structures, input/output control statements, subroutines, structured flow, etc., were not

included. As more complex tasks were attempted, it became apparent that more powerful programming features were required. This need has sparked researchers to develop languages which are suitably powerful and easy to use in the real-world application involving industrial robots [Paul 1985].

Robot language development has proceeded along two main paths. The most obvious one is to build a robot-specific programming system from routines in a general purpose language. The second direction involves the implementation of robot languages *ab initio.*

Both approaches have their advantages and shortcomings. The advantage of the first is a reduction in robot language development time and a reduction in robot programmer training time. The second solution is appealing because the robot programmers need not be computer programming experts since they may develop useful programs by learning only a subset of the language.

The shortcoming of the first approach has been the limitation imposed by the base language used to develop the robot language. Often, the programming environment is not suitable for robot programming due to the lack of an interactive interface which is extremely useful in the development of robot applications. The main drawback of the second solution is that effort is wasted in duplicating features which already exist in standard languages such as Pascal, C, and Forth. [Gini 1985]

2.3 COMMON ROBOT PROGRAMMING LANGUAGE FEATURES

Although many robot languages have been developed, they all have standard features in common. These features include a number of variations on the MOVE command, location templates, cartesian real world coordinate systems, tool manipulation commands, speed control, intermediate points, approach and departure points, complex trajectories, sensor information, and multi-robot operations.

MOVE COMMANDS - There are many variations on the MOVE command. These range from the simplest which allow the user to issue movement commands based on low level, single joint movement to joint interpolated movements which involve the manipulation of the arm in reference to the tool tip. The programmer need only specify where to move in the real world coordinate system and the system subsequently moves all axes in such a manner as to bring the arm with the tool tip to the new location.

The MOVE command in robotics languages is the fundamental method of changing the position of the robot arm and consequently the tool tip. It usually accepts a real world location as a destination point and the language determines the combination of actuator movements to effect the movement of the robot arm tool tip to this location. Two variations of this command involve the method by which the tool tip is manipulated. The standard move operates in a joint interpolated method where all joints start and stop at the same time.

The straight line mode is another variation. This involves effecting the movement with the tool tip moving in a straight line, along a prespecified trajectory, from the source location to the destination. This is normally used to precisely place an object or tool in a confined or restricted location which may be a hole. The APPROACH command is another variation on the MOVE command. This command accepts a real world destination and a distance to which the location is approached. The approach destination is determined by the tool tip configuration. This is useful in moving the tool tip to an approach line for the grasping or placing of an object. Normally, this command places the tool tip in such a manner as to allow the movement to the location in a straight line fashion for high precision grasping. The APPROACH command may also have the straight line feature.

The DEPART command may be considered the opposite to the APPROACH command. This command accepts a distance as a parameter. The arm moves from the current location, along the tool tip configuration, to the specified distance. This is normally used when departing from a location where the object must be moved out of a hole or from between a series of objects. Normally, the straight line version of this command is used so that the object is moved in a straight line along the tool tip configuration and objects are removed in such a fashion as not to interfere with other objects or bind in the hole.

An example of the use of these commands is depicted in the following pseudo-code illustration:

DEPART-STRAIGHT-LINE 2

APPROACH location1 2

MOVE-STRAIGHT-LINE location1

The above procedure would pull the peg from a hole in a straight line, 2" from the hole along the Z axis, move the object to within 2" of the destination location (location2) and place the object at the destination location in a straight line. This is referred to as a PICK and PLACE procedure (see Figure 2.3) [Paul 1985].

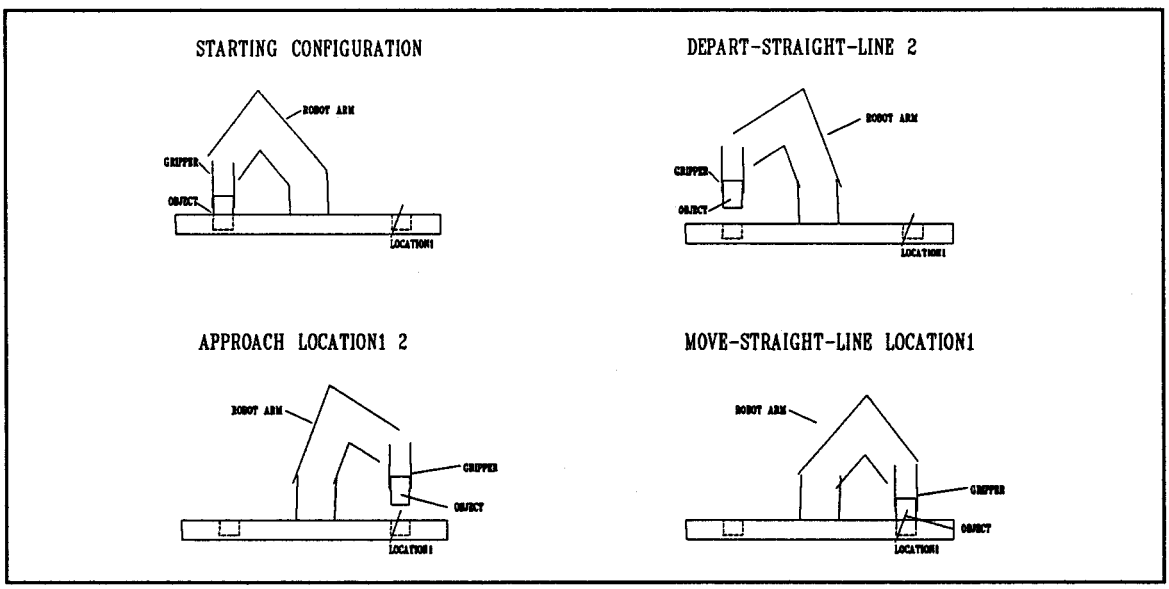

Figure 2.3 Illustration of pseudo-code

CARTESIAN REAL WORLD COORDINATE SYSTEM - The cartesian real world coordinate system is a representation of the robot work space as a three dimensional coordinate system with some physical reference point at the arm. This reference point is normally the base of the arm. This type of system is used to

simplify the programming of robot plans in a system which is universally known.

Most languages allow the programmer to use a 3 dimensional cartesian real world model to specify moves. This relieves the programmer of low level responsibilities, that is, the manipulation of the arm at the actuator level in pulses. The operating system performs all the translations from low level to the real world model [Paul 1985].

LOCATION TEMPLATES OR FRAMES - Location templates (also known as frames) are the representation of the real world coordinate system in the programming language. These templates usually contain, at a minimum, the X, Y, and Z coordinates of the real world system. Other components included in the template are the roll, pitch and yaw of the gripper. This location template is dealt with as a single entity and is used to represent a real world location as well as the tool tip configuration at that location. The X, Y and Z components are normally real numbers representing coordinates in the units of choice and the other components are normally angles in either degrees or radians.

TOOL MANIPULATION COMMANDS - There are generally two types of tools which are frequently used with robot manipulators. Tools which may be operated in a binary mode and others which are operated in the continuous path mode. Binary mode operated tools include such tools as drills, buffwheels, sanders, etc. Continuous path operated tools include robot proportional opening grippers. The user may specify the grip distance and the force which the gripper is to exert.

It is also advantageous to be able to detect whether the grip distance has been effected to detect whether the object has been grasped correctly.

SPEED CONTROL - Most languages offer speed control which is relative to the nominal speed of the manipulator. A more advanced variation of the speed control involves timing. The operator need only specify the amount of time a trajectory should take and the operating system effects the move in such a manner as to satisfy this requirement.

INTERMEDIATE POINTS - The ability to program complex trajectories is often necessary to avoid obstacles. These trajectories may be effected by a linked movement structure which would allow the programmer to specify a number of trajectories which are to be executed in series as one movement. The intermediate coordinates are not reached with great precision. The end location is the only location in the trajectory which is reached in high precision.

It is often necessary to approach an intermediate location before actually moving to the desired location with precision along an unrelated trajectory. This type of command is usually referred to as approach. The programmer need only specify a distance to which the tool tip is to approach the location. The intermediate point is automatically generated by the system. The distance is in reference to the tool configuration/orientation.

TRAJECTORIES - There are different styles of trajectory execution including the free mode, the coordinated actuator mode and the linear interpolation

mode. The free mode involves a low level of precision where locations and paths are calculated in low precision values. This mode is used for trajectories and locations which do not require precise placement. The mode is included to speed up calculations and hence arm movements where high-precision placements are not required for intermediate moves.

The coordinated actuator mode manipulates the arm in a speed controlled manner where all axes are started and stopped at the same time. This is the normal mode used to effect the move command. The Linear interpolation mode is provided for high precision manouevers where the tool set of coordinate axes follows a straight line or circular path. This mode is used to precisely run the tool tip along a straight path to effect the placement of an object in a confined space or along a restricted path. This is usually used for the placement or removal of an object to and from a hole [Coiffet 1983].

A feature which some languages have included involves the linking together of intermediate points to form complex trajectories. This is a useful function which may be used to guide the arm around obstacles to avoid collisions. These trajectories may then be invoked by using one command. This command may be considered a macro.

SENSOR INFORMATION - The ability to cope with sensors has been a feature of robotics since the beginning. Sensors provide a robot with the ability to sense its environment during run time. Sensory data may be used to modify the run

time movement of the manipulator. For instance, a switch may be used to determine if an object has been placed correctly so that the program can use this information to effect an alternate plan when this becomes necessary [Korein 1987].

TEACH PENDANT - As mentioned above, the teach pendant is used to manually guide the arm to specified locations. Many languages use this mechanism to teach the robot arm locations. Once the arm is guided to a location, the operator may specify a location template name which will contain the current arm location.

MULTI-ROBOT OPERATIONS - Research has gone beyond the control of only one robot, as it has become recognized that multiple robotic systems are sometimes needed. These systems allow the programmer to synchronize a number of robots by using either a sequential or parallel approach. [Parent 1985]

2.4 LANGUAGE LEVELS AND PROGRAMMING ENVIRONMENT

There have been several different approaches to the programming environment associated with robots. The compiler method of programming has exhibited shortcomings when applied to robot programming. It has proven more advantageous to provide an interactive interface which allows the programmer to interact with the arm during program development [Gini 1985].

There are four distinct programming language levels. The lowest level of which is referred to as the Actuator Level. This involves programming the displacement of each actuator. The next level is the End Effector Level. The

programmer is only concerned with the coordinates of the tool itself. The Object Level is the next level. The programmer is only concerned with the movement of objects. The highest level is the Objective Level. This level is a generalization of the Object Level. Only the final objective is specified. The details of intermediate stages are generated by the system [Parent 1985].

The majority of current robotic languages operate at the End Effector level. Attempts at implementing Object Level programming languages have proven to be fruitless. It involves the complex modelling of the real world [Halme 1987]. The whole work space must be geometrically represented in three dimensions. The time involved in setting up such a system is much greater than the time required to develop robot plans using the End Effector Level[Korein 1987].

2.5 SURVEY OF EXISTING LANGUAGES

Robot language research has been active since the 1960s and several languages are currently being used commercially. Each of these languages use previously discussed methods and approaches.

PASCAL-STYLE LANGUAGES - There have been many languages developed around the Pascal-style of programming. This includes structured languages such as ALGOL and C. These robot languages include the following research and commercial languages incorporating this style of programming language at the End Effector Level of programming. These languages include AL [Paul 1977],

WAVE [Finkel 1974], AML [Taylor], HELP [GE 1982], LM [Latcombe 1981], MCL [McDonnel 1980], PASRO [Biomatic 1983], RAIL [Franklin 1982], VAL II [Shimano 1984], RCCL [Paul 1985] and SRL [Blume 1984]. Some of these languages have been developed from scratch and others are merely library routines extending an already existing Pascal (or similar language) or compiler [Gini 1985].

One of the most notable languages is AL which has been constantly improved since its inception in 1974. It is one of the first languages to contain most standard, robot language features that have become standard. In order to illustrate the common robotics language features, two languages, SRL and RCCL will be described as they are implemented using different methods. Since SRL implements a large number of robotics language features, it will be described in detail.

SRL - SRL (Structured Robotics Language) was developed by Christian Blume and Wilfried Jakob in West Germany [Blume 1984]. The language is a hybrid of the traditional programming language style like Pascal and high level robot languages such as AL. The major features of SRL are presented in a published proceedings entitled Advanced Software in Robotics [Blume 1984]. Blume and Jacob present such features as data concept, control of program flow, move and effector statements, parallel, cyclic and delayed program execution, input-output and world model.

SRL was designed to be hardware independent, unlike its predecessor AL, which is hardware specific. Data types in SRL include; the pascal types INTEGER,

REAL, BOOLEAN, CHAR and VECTOR; AL types VECTOR, ROTATION, FRAME; and two synchronization data types SEMAPHOR and SYSFLAG. SRL is based on the frame concept as found in many robotic languages including AL. Position and orientation of the robot are described in a frame which consists of a position vector and a rotation.

In order to effect hardware independence, a system specification construct is included. The programmer is able to define specifications for tailoring to different hardware configurations. Programs taking advantage of this construct, are easily adapted to different sensors, robots, and hardware facilities. This facility allows the programmer to refer to specific hardware using meaningful names rather than low level device numbers so that programs are easy to read.

SRL has provisions for concurrency which consists of the SECTION construct. It uses PROCEDURES which are the same as the Pascal PROCEDURE. Compound statements are supported, as well as the traditional control structures of Pascal. An EXIT statement was included for the premature termination of control structures such as loops much like the 'C' EXIT statement.

Assignments and expressions are treated in much the same fashion as in Pascal, data type conflicts being checked at compile time. This concept is expanded to handle the new types VECTOR, ROTATION, and FRAME. SRL also has powerful arithmetic facilities for geometric calculations which facilitate flexible robot moves.

SRL provides several move statements to distinguish between different types of interpolation. Position and orientation of the tool centre point is based on the AL concept of a frame. Sensor conditions are reacted upon by the WHEN or ALWAYS WHEN statement. Table 2.1 lists the moves supplied by SRL.

General specifications (parameters) for these constructs are velocity, duration, acceleration, constant/variable orientation during move, approach/depart points, frames between start and end frame, force and wobble. Complex manouevers may be constructed using the MOVESPEC construct which allows the programmer to define a move as a number of move statements, give it a name and then execute the specified move using MOVEDO given the move name. This is a powerful construct which allows the programmer to build up complex manouevers which may be easily executed any number of times during program execution.

Gripper operations include open and close with specified distance and force. Checking is performed to ensure that the gripper is at the specified distance according to the world model. Error corrective actions may be taken to reduce the discrepancies between theoretical values and actual values.

There is provision for restricted parallel, cyclic and delayed execution of program parts. A WHEN statement is included for handling interrupts and sensor input. This statement may be used for optimal time scheduling which provides for constant checking of the real time clock and the execution of a block of statements when the timing condition has been met.

Table 2.1 Variation on the move commands supplied by SRL.

PTPMOVE	Move without any synchronization between the robot axis. Each axis is moved with maximum acceleration and speed. No general specifications are allowed.
SYNMOVE	Linear interpolation in robot joint coordinates, i.e. all axis will be synchronized. General specifications possible.
SMOVE	Movement on a straight line by linear cartesian interpolation. General specifications allowed.
LANEMOVE	Trajectory calculation by polynoms, similar to the MOVE-statement of AL. General specifications allowed.
CIRCLEMOVE	Movement along circular segment. Specifications: centre point, angular displacement, velocity or duration, fine/rough interpolation positioning.
VIAMOV	Move to a via frame without stopping at the via frame. The interpreter expects a next move statement for continuing the move. Only special specifications of velocity and duration are allowed.
MOVE	Move statements with interpreter or controller dependent parameter specifications. This statement can be used for future types of interpolation, if the controller includes control modules.
DRIVE	Movement of one or more robot axis. Specifications: velocity or duration, force, fine/rough positioning.

The SRL programming environment consists of the SRL-compiler, SRL-interpreter, Frame-editor, Simulator (optional) and Symbolic debugger (optional). The SRL-interpreter contains a crash analyzer (history trace) which gives procedure trace back, line numbers where errors occur along with the run time error message. It also handles hardware configurations, memory management, arithmetic and program flow [Blume 1984].

SRL has been implemented on the CRS plus robot manipulator by Cybofluor based in Toronto, Ontario. This software is proprietary and only a brief demonstration was presented. There were some drawbacks witnessed in this type of system. The compiler converts SRL code into RAPL which is then downloaded to the robot controller and subsequently run. There are major drawbacks to a system of this nature since the software is limited to the amount of memory in the controller, and control is given completely to the controller.

It would be much more desirable to allow the host microcomputer to have control over the robot. This allows the system to take advantage of the PC's powerful features. Even if the controller could be expanded to accommodate larger programs, there is a certain amount of hardware duplication which is unnecessary. Also, run time debugging would be limited to the RAPL interface since the SRL interface is completely removed from the run time system. This does not allow the system to take advantage of a thorough run time debugger which may be incorporated into the SRL system.

There are certain advantages to using the RAPL language as the low level language. As previously discussed, RAPL contains some powerful constructs such as joint interpolated moves, teaching interface, etc. There is no need to reprogram such complex features. A more versatile system would establish a communications link between the host microcomputer and the robot controller. This link would be a two way communication which would allow the microcomputer to completely control the robot hence establishing a microcomputer run time interface. The low level communications would involve the real time issuing of RAPL commands effecting the virtual SRL language. We may now effect a run time debugger as well as maintain complete control of the robot via the microcomputer.

RCCL - RCCL (Robot 'C' Control Library) uses the host language 'C' as the base for developing a robotics language [Paul 1985]. The robot is identified as an input/output device which is manipulated by library routines written in 'C'. This design structure allows the programmer to take advantage of the full power of the 'C' language as well as given high level control of the robot manipulator.

The system was designed to run under the UNIX operating system to take advantage of the concurrency built into this operating system. Four processes run concurrently when RCCL is executed. The lower level controls the torque of each manipulator joint, the setpoint process running at interrupt level which computes cartesian trajectories, a real time communication process, and the user process which makes the RCCL system calls.

RCCL supports a structured location description which is a mathematical construct describing the location of coordinate frames. Relative locations of objects are also supported along with provisions for tool compensation.

Two types of motions are supported by RCCL. Joint mode which is equivalent to the joint interpolated modes previously discussed and Cartesian mode which moves the tool along a straight line.

Sensor integration is also provided which allows programs to adjust robot plans according to sensor information. Locations may be modified during run time. This is done using hold transforms which essentially involve making a copy of the location and allowing the program to update the copy according to user interaction or sensory input. Trajectories may also be modified during run time. This is useful for updating trajectories according to sensory input.

RCCL provides internal sensing for location and force information. If a motion is terminated on a condition, the world model may have to be updated. Joint torques are also obtained from the manipulator state. A mechanism is provided in RCCL which compares the actual forces and torques against expected values. This information may be used to terminate a path segment when a specified limit is reached [Paul 1985].

This is an example of a robot language embedded in an existing high level language. This method of development provides a powerful, proven language base with all the features of a robot language. Development time should be considerably

lower since the programmer is only concerned with the robot routines. Unfortunately, RCCL requires a powerful operating system (UNIX) which supports concurrency, and a powerful machine.

BASIC-STYLE LANGUAGES - There have been a number of languages based on a BASIC style of syntax and structure. Many of these languages are used commercially. The appeal of this style of language-style is the simplicity and ease of training. These languages usually operate at the End Effector Level but tend to be deficient in programming structures and complex data structures. Unfortunately this style of programming lacks the power and flexibility for the development of complex robot plans. Some of the languages developed along this style of programming are ROL, VAL, and as will be discussed in Chapter 3, RAPL [Gini 1985].

OTHER LANGUAGES - Other robot languages are developed around such programming styles as LISP and PROLOG. These languages will not be discussed since the literature search did not yield sufficient data to warrant a discussion.

CHAPTER 3

THE CURRENT SYSTEM HARDWARE AND SOFTWARE

In this chapter, the current hardware and software system which implements the ASPS will be discussed in detail.

3.1 THE CHEMICAL ANALYTICAL SYSTEM

The chemical analytical method which the ASPS automates is referred to as Solid Supported Reaction. The method is a relatively new approach to the preparation of samples for analysis in trace quantities. Analytes dissolved in some matrix (the medium in which the analyte exists, such as blood, water, urine) are removed from the matrix and transferred to a suitable medium in which the analyte may now be detected in trace quantity. This medium is usually a solvent of some type. The analytes are typically quantified using a GC (Gas Chromatagraph) or LC (Liquid Chromatagraph) [Leznoff 1978].

The solid supported reaction involves co-absorption of analyte and reagents on an insoluble resin. The methodology lends itself to automation since it is simpler and more sensitive, in terms of analyte concentrations, than the more classic liquid/liquid extraction methods currently in use in most research and commercial laboratories.

The methodology tends to eliminate errors introduced by sample matrix effects which are errors in the analytical method caused by the medium in which the analyte exists [Leznoff 1978]. This methodology has been tested on the preparation of samples for the analysis of steroids, herbicides, and cannabinoids in body fluids and pharmaceuticals [Rosenfeld 1984a, Rosenfeld 1984b, Rosenfeld 1986, Rosenfeld 1989]. There is potential for the solid supported methodology in applications to environmental, food, pharmaceutical and industrial testing.

The sample preparation stage is labour intensive and, therefore, automation is very desirable in this stage of the analysis. The largest driving force behind automation is the need to increase productivity and reduce costs of analysis [Isenhour 1989]. In addition, automation of analytical procedures offers an increase in reproducibility which would boost the accuracy of the procedure.

Health effects are a concern in the sample preparation stage. The analyst may be exposed to hazardous conditions due to the toxic and/or infectious nature of the sample matrix and reagents. Automation of the methodology should reduce the health risk associated with exposure to samples and reagents since, for the most part, the sample volume is small and the whole preparation process may be carried out under a fume hood [Povolinis 1988].

3.2 THE ASPS HARDWARE CONFIGURATION AND DESIGN

The automated sample preparation system was designed by Povolinis as part of a M. Eng. (Engineering Physics) project at McMaster University [Povolinis 1988]. The system was designed to automate the sample preparation of samples using a solid supported technique.

The ASPS is comprised of a 5 axis robot arm which manipulates reaction tubes, syringes, dispensers, wash probes and vacuum apparatus. Furthermore, there are peripherals which must be operated in conjunction with the robot arm manouevers. These peripherals include vacuum pump valves, dispense valves and syringe plungers.

This automated procedure differs greatly from others in that the system is designed to manipulate a variety of sample types for a number of different analytes. Each sample preparation procedure requires its own robot plan. Compounding the problem is the fact that it is desirable to enable the automated system to manage a number of different sample preparation procedures concurrently.

The ASPS software system utilizes the robotic language (RAPL) supplied by the robot arm vendor CRS PLUS. The original system depended on a FORTRAN program which performs a robot planning role. The FORTRAN program prompts the user for detailed analytical information which is transformed into a number of RAPL subroutine calls. These programs must be loaded into the robot controller before they can be run [Povilonis 1988].

RAPL subroutines called by the planning software are kept in the controller at all times. These subroutines perform several different analytical manouevers based on the order in which they are called and the values of a number of different variables [Povilonis 1988].

This software system was found to contain inherent flaws. It generally did not generate robot plans that worked. During experimentation, it was found that every generated plan required modification of the generated RAPL code which is not an acceptable situation.

The RAPL code generated by the system was extremely cryptic and modification of the code is inordinately difficult. The system generally works on side effects, where control of subroutines are totally effected using RAPL variables. Generally, this system was difficult to use, contains functional flaws and generates code which is difficult to modify.

AUTOMATED SAMPLE PREPARATION TECHNIQUES - Several analytical techniques are required for this particular analytical procedure. Among these techniques is the dispensation of liquids in volume ranges of 1 to 10 ml, syringes for adding minute quantities of reagent in the range of 100 μl, mixing of liquid and resin, temperature control, separation of liquid and resin, liquid rinse of resin and evaporation of organic solvent (see Figures 3.1 and 3.2).

Organic Solvents, water, weak acid, base, buffer and reagents in organic solvent solutions must be dispensed to any test tube on the reaction block. The

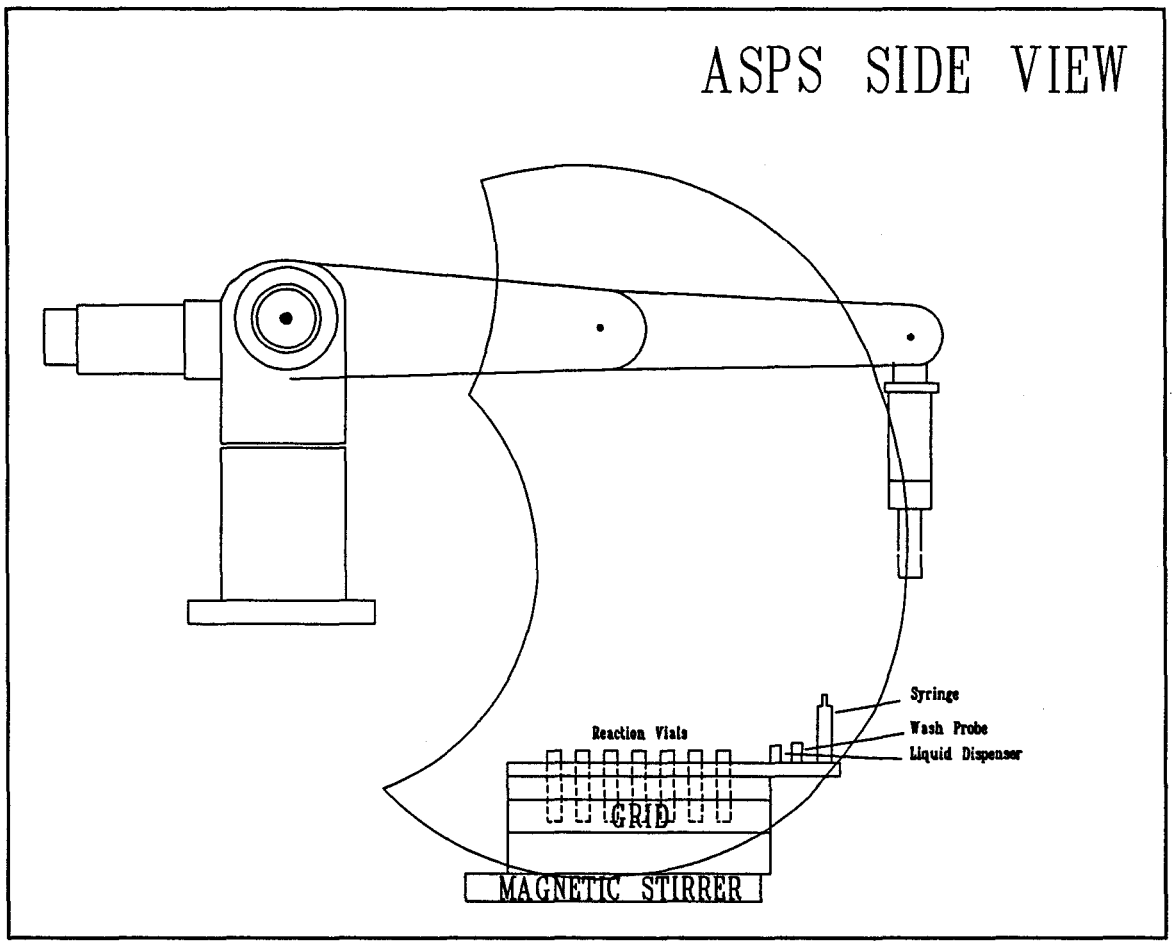

Figure 3.1 Side view of the ASPS.

various liquids are dispensed using liquid dispensers and syringes.

Syringes are solenoid operated syringes which are used to dispense liquid volumes in the range of 100 μl. These are used for the precise injection of solutions. They are operated by moving the syringe, using the robot arm, to a vial containing the solution to be dispensed. The solenoid is fired, sucking up the liquid into the syringe. The arm then moves the syringe to the required testtube and the solenoid is then fired which dispenses the liquid into the solution. There are currently 4

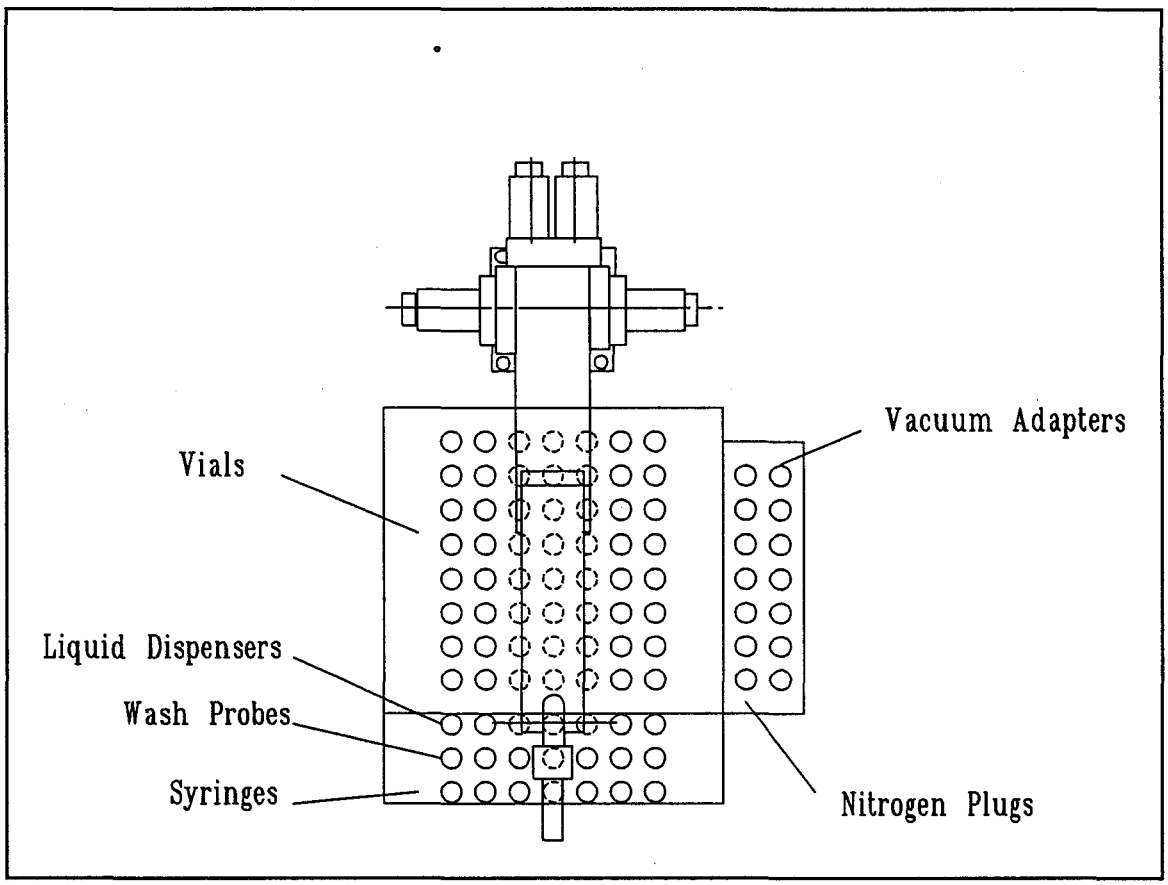

Figure 3.2 Top view of the ASPS.

syringes available (see Figure 3.3).

Liquid dispensers are used to dispense various liquids from reservoirs (containers) using solenoid pumps. These dispensers are operated by using the robot arm to pick up the dispenser from its resting place, move the dispenser to the required vial, sending the required number of pulses (one pulse / ml) to the pump, and moving to the next tube. Each dispenser may dispense one liquid at a time. The current system has provisions for seven of these dispensers (see Figure 3.4).

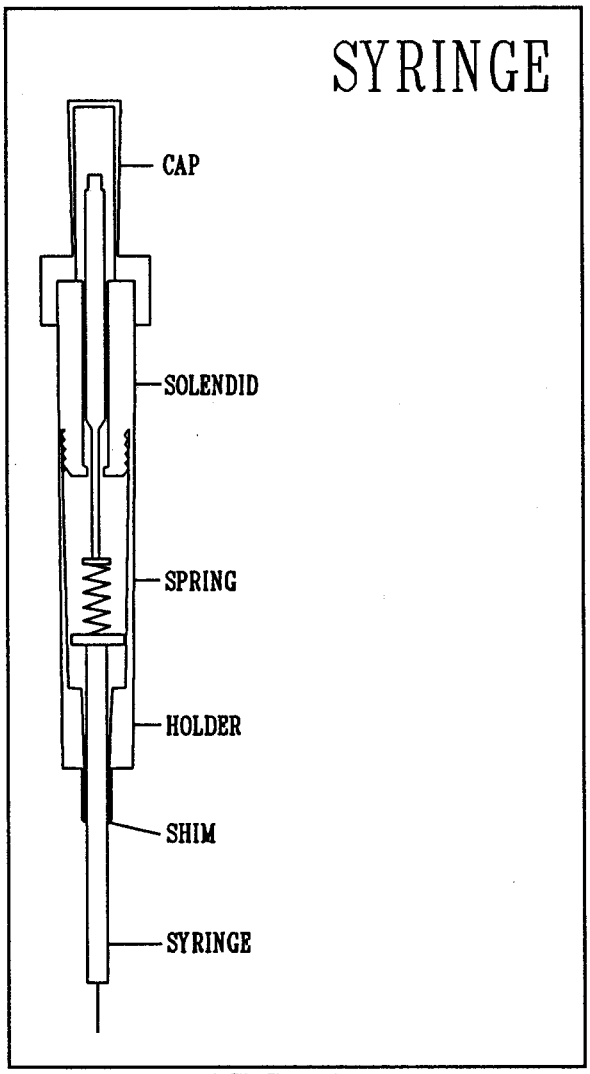

Figure 3.3 ASPS syringe.

The mixing of liquid and resin is accomplished by placing a multipoint stir plate under the sample preparation grid. Magnetic stir bars are placed in each vial to provide stirring during the preparation. Temperature control is effected by using two cartridge heaters which are manually controlled since the temperature remains the same throughout the sample preparation process.

Separation of resin from liquid is accomplished using a classic vacuum separation apparatus which is illustrated in Figure 3.5. The vial is moved from its reaction location to the top of a vacuum separation unit. A valve is triggered which causes a vacuum in the unit pulling the liquid down into an evaporation vial.

The resin must be rinsed by several reagents. This is accomplished using a vacuum nozzle integrated with a liquid dispense unit (see Figure 3.6). This unit is operated by moving it from its resting place to the desired reaction tube using the robot arm. A valve is opened which causes a vacuum in the nozzle removing

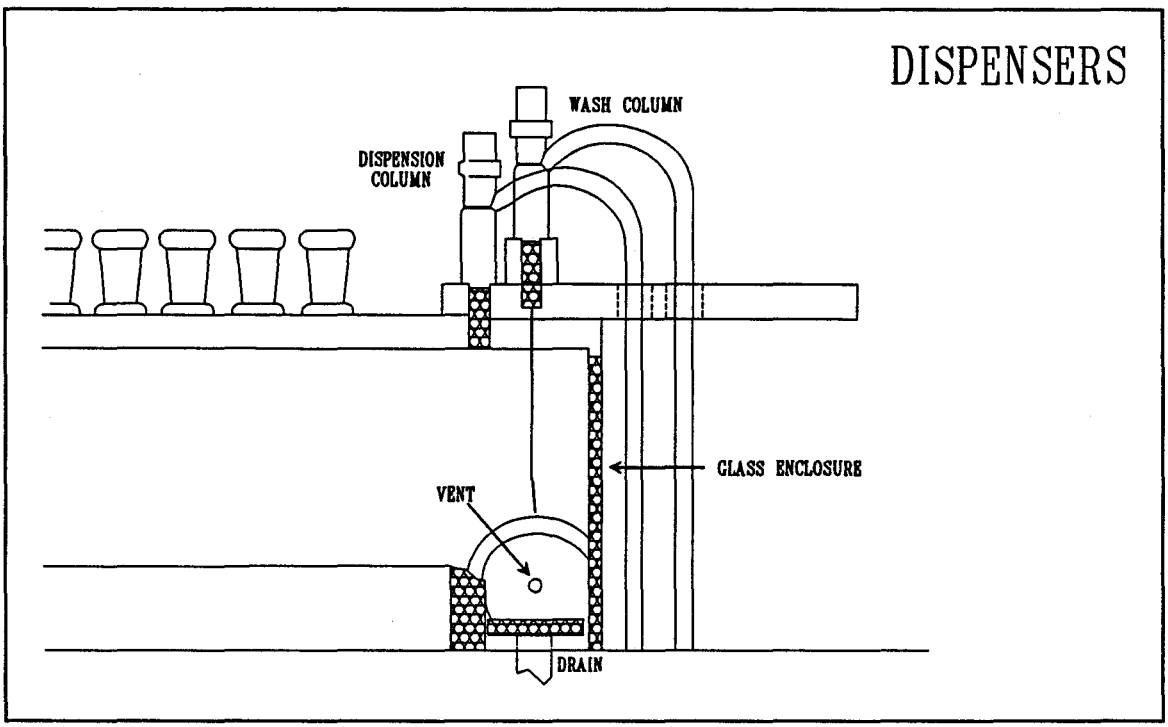

Figure 3.4 ASPS liquid dispenser.

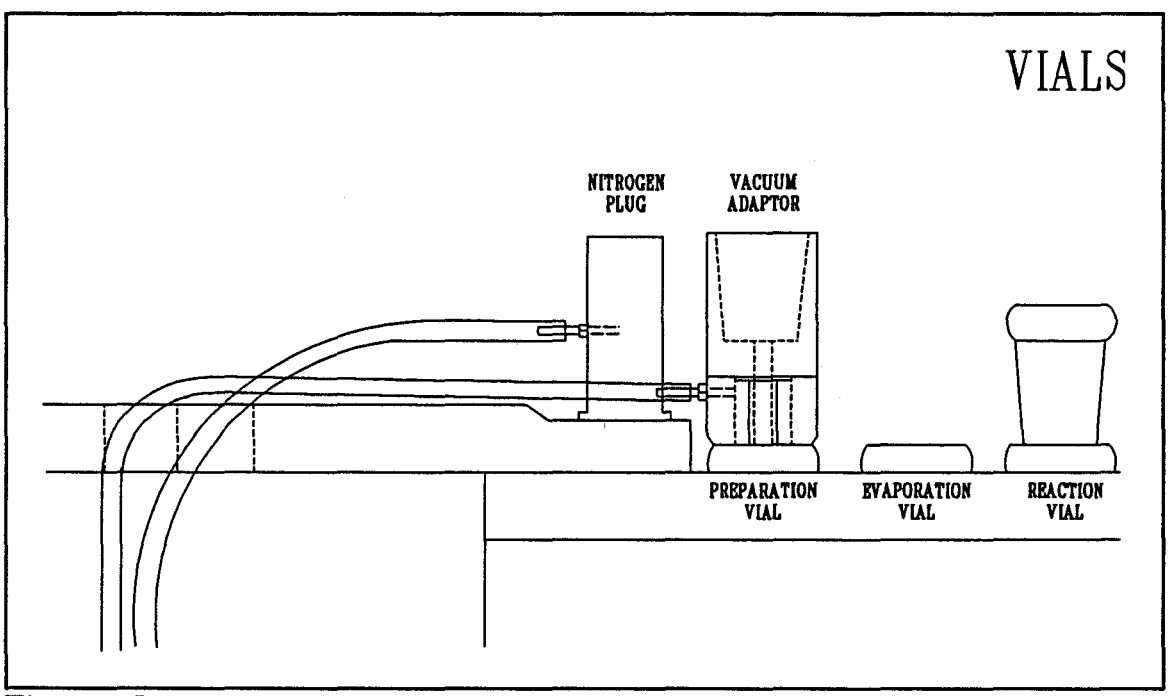

Figure 3.5 Vacuum separation unit and vials

unwanted liquid. A fine screen prevents the resin from being sucked up into the nozzle. Each of these wash units is connected to a liquid dispenser pump and reservoir. A valve is activated routing the liquid to the wash unit. The corresponding solenoid pump is sent the required number of pulses for the amount of wash solution used. The nozzle remains submersed until the wash is finished. The robot arm then removes the nozzle from the reaction tube and either performs the operation on another tube or returns the unit to its resting position.

An evaporation unit is used to evaporate solvent from an evaporation vial. The unit is connected to the vacuum system which is vented. Heating of the tube along with application of vacuum reduce the vapour pressure in the vial enhancing the evaporation stage. An evaporation unit is picked up by the robot arm and placed on top of the evaporation vial. A valve is opened which connects the vial to the vacuum system thus venting the organic vapours and effecting evaporation. Once evaporation is completed the evaporation units are returned to their resting place [Povilonis 1988].

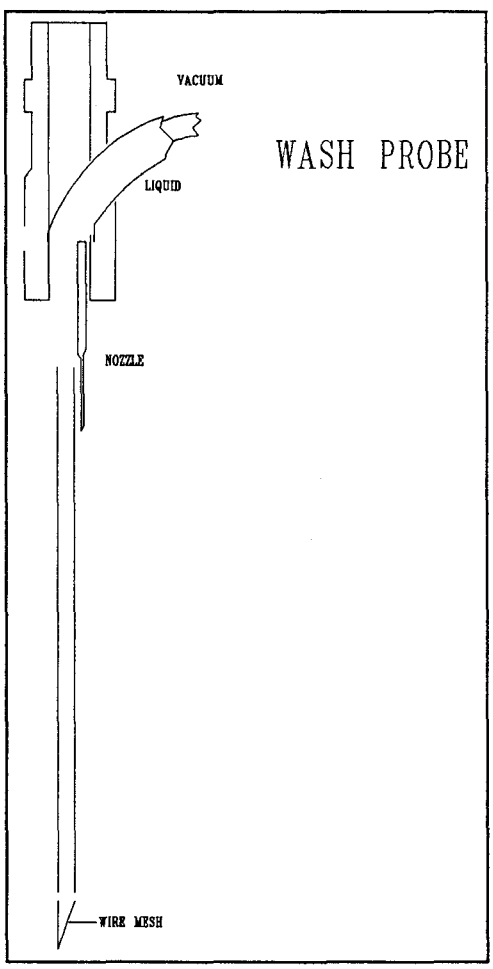

Figure 3.6 Liquid rinse unit.

3.3 THE ROBOTICS HARDWARE (CRS PLUS)

The robot used to manipulate the various objects in the automated sample preparation system is the CRS plus-5 axis manipulator arm. It is manufactured by a Burlington, Ontario based company named CRS Plus. The CRS Plus robot is targeted for the laboratory and manufacturing, table top applications [CRS 1985a, CRS 1985b].

HARDWARE DESIGN - The robotic system used is the SRS-M1 Small Industrial Robot System. This system is a self-contained, five axis, D.C. servo driven robot. Included with the system is an articulated robot arm, a robot system controller (RSC-M1) and a teach pendant. This system requires a video terminal or micro-computer equipped with the CRS-Plus robot communication terminal emulation software as well as a gripper. The ASPS uses a micro computer for terminal emulation and a servo driven gripper. See Figure 3.7 for a general view and configuration of the CRS plus robot system.

The robot arm consists of a shoulder, upper arm, lower arm and wrist. There are five degrees of freedom which include the waist, shoulder, elbow, wrist roll and wrist pitch. The RSC-M1 robot system controller is a 16-bit microprocessor based master controller. It comes with a resident robotics language called RAPL, a teach pendant, 6 D.C. servo amplifiers, arm power supply and voltage regulator, and five servo axis cards. There are 32 input and outputs supplied with the controller which allow control of peripherals [CRS 1985a].

Table 3.2 CRS Plus robot arm specifications

Description	Specification
Structure	Articulated - Five DOF
Payload (Maximum Speed)	1.00 Kilogram
Payload (Reduced Speed)	2.00 Kilograms
Reach (Without Gripper)	0.56 Metres
Workspace Dimensions:	
Base Rotation	+/- 175 degrees
Shoulder Rotation	+110, -0 degrees
Elbow Rotation	+0, -130 degrees
Wrist Bend	+/- 110 degrees
Tool Roll	+/- 180 degrees
Maximum Loaded Speed	0.50 Metres per second
Joint Speeds:	
Base	60 degrees per second
Shoulder	60 degrees per second
Elbow	60 degrees per second
Wrist	180 degrees per second
Tool	180 degrees per second
Repeatability	+/- 0.13 Millimetres
Joint Worst Case Resolution:	
Base	0.0023 inches
Shoulder	0.0023 inches
Elbow	0.0014 inches
Wrist	0.0013 inches
Tool	0.0013 inches
Drive System	DCX Servo Motors with Optical encoders

CONTROLLER TECHNICAL INFORMATION - The robot controller is an Intel 8086 based micro-computer designed solely for the purpose of controlling and driving the robot arm. There are three types of memory utilized by the controller which include

- 4K of low power CMOS memory used for scratchpad use, 8086 stack space, and the interrupt vector space.
- 12K of Battery-backed CMOS memory used for system parameter setup and user memory space expandable to 64K. System parameter setup includes such information as communications parameters, calibration information, arm configuration, etc. User memory space contains variables, location templates, and RAPL programs.
- 64K of EPROM memory used for firmware requirements. This EPROM memory may be expanded up to 256K. The firmware contains the RAPL language, a program editor, communications routines and all low level routines required to drive the robot arm at the actuator level.

The Intel 8087 math co-processor is used for all mathematical calculations. Real numbers are stored as 32 bit numbers giving 9 digit precision.

Table 3.3 CRS Plus robot system controller specifications

Description	Specifications
Control System:	
MicroProcessor Type	8086
Number of axis	Five DC Servo Axis
Teaching System:	
Manual	Teach Pendent
Off-Line	RAPL
Motion:	
Point-to-Point	Yes, with joint Interpolation
Straight Line	Yes, at reduced speed
Position Detection	Digital Optical Encoders
Speed Setting	0 to 100%
Interfaces:	
Communication	Dual RS232C
Expansion	Peripheral Expansion Slot
User Digital Inputs	Sixteen Standard TTL levels
User Digital Outputs	Sixteen Standard TTL levels
Programming:	
Language	RAPL
Preparation	RAPL Editor
User Memory Size	8K
Power Requirements	100-130 VAC, 50-60 Hz, 3A
Operating Ambient Temperature	0 to 50 degrees Celsius
Dimensions	19" x 19" x 17"

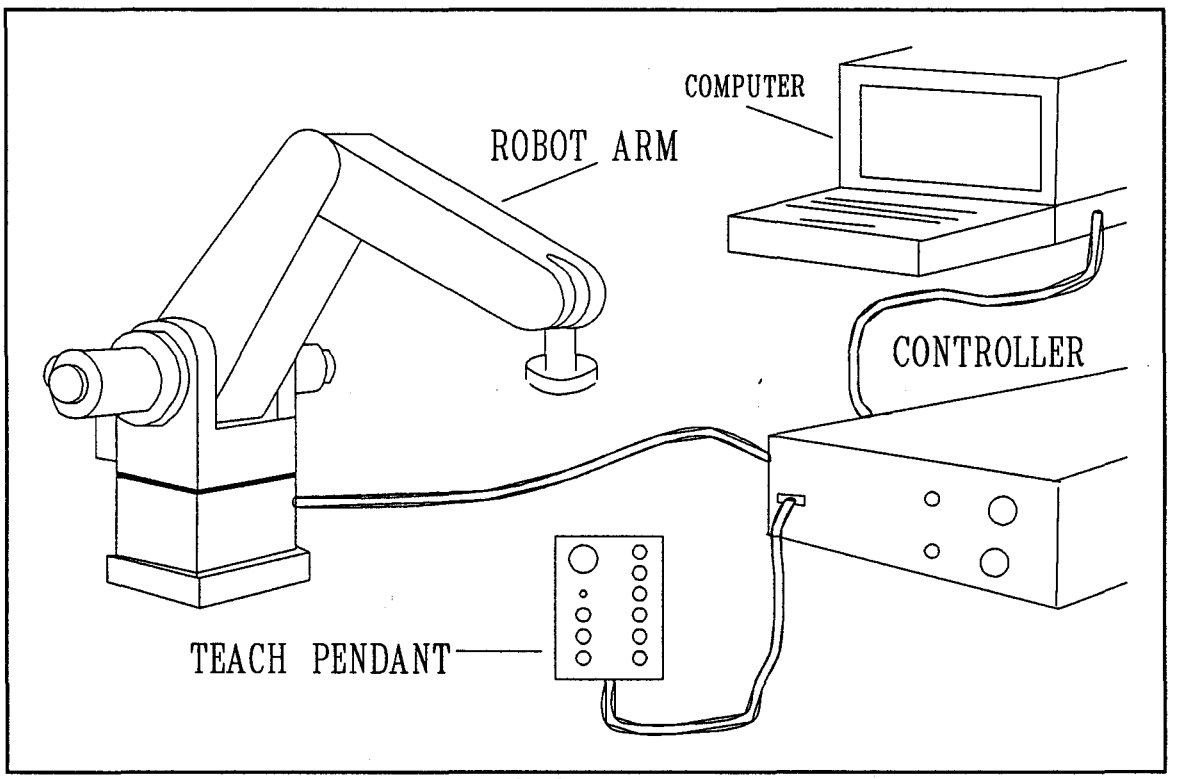

Figure 3.7 Block diagram of the CRS plus control system

The CRS-Plus controller implements the joint interpolated and straight line modes of path control for the robot arm. The joint interpolated mode moves all joints simultaneously starting and stopping all joints at the same time. Straight line interpolation is a 3 axis motion since the wrist axes are not moved. This mode ensures that the tool tip will move in a straight line from the current tool position to the specified location. Digital input/output scanning is executed at 30 milli-second intervals [CRS 1985a].

MEMORY ALLOCATION - Memory visible to the user is the EPROM and CMOS memory. The firmware includes a monitor for hardware definition and the operating system which resides on top of the monitor. The operating system handles the execution of RAPL and the generation of robot path coordinates for motion control. CMOS RAM is used to store user programs, variables and locations.

RAPL programs are stored in a program table which may hold up to 255 eight-byte program names, two-byte indexes to the program buffer and two-byte checksums. Programs are stored as a sequence of ASCII characters. The amount of memory allocated for program storage may be specified by the user using the ALLOCATE command.

A symbol table is maintained for the implementation of variables. The size of this table is also specified by the ALLOCATE command. The symbol table stores an eight-byte name, a four-byte location and a two-byte checksum for each variable.

Locations (precision and cartesian types) are stored in a location table. Each location requires ten fields in the table. The first field contains an eight-byte location name. The next eight fields are four-byte locations used to store each of the eight components for each location. Each component of a cartesian location is stored as a four-byte real number representation of real world coordinates. Each component of a precision location is stored as a four-byte integer which represents the value of an axis motor coordinate (in encoder pulse units). The tenth field is a two-byte checksum [CRS 1985a].

Table 3.4 CRS Plus robot controller memory map

The following is a system memory map of memory allocation

Address (hex)	Description
FFFF:0	Bootstrap Address
F000:FFEF	Upper limit of firmware address space
F000:0000	Standard 64K firmware address boundary
E000:0000	Extended 128K firmware address boundary
0000:FFFF	Optional expansion memory upper limit
0000:3FFF	Standard 16K RAM memory upper limit
0000:2000	Approximate start of user memory (may vary according to firmware release version, and other options)
0000:1FFF	Approximate top of CPU scratchpad space and system parameter space.
0000:1000	Start of CPU scratchpad and system parameter space
0000:0FFF	Top of 8086 stack space
0000:0400	Start of 8086 stack space
0000:03FF	8086 vector interrupt space
0000:0000	Bottom of memory

User memory is allocated according to the following table

Location of:	Variable Table	40:37E0
	Location Table	40:2B80
	Program Table	40:2AC0
Start of User Memory is:	Program buffer	40:1A40

COMMUNICATIONS INTERFACE - The controller has a communications interface which allows external computers to communicate with the system. It may be used to transfer data either to or from a computer. The CRS Plus communications protocol will allow the computer to communicate with any segment of memory in the controller. Error checking and automatic transmission retries are performed by the protocol to establish an error free communication link. The controller is configured to be the slave and the external computer the master device [CRS 1985a].

3.4 RAPL AS A ROBOTICS LANGUAGE

RAPL (robotics applications programming language) is the robot arm language supplied with the CRS Plus robot system. RAPL is a BASIC-style robot language which resides in the controller's EPROM. Like most other BASIC-style robot languages it operates at the End Effector level [CRS 1985b].

VARIABLES - The RAPL language provides only the real variable type. It does support an integer input and output but these values are stored as reals. The normal operators are available for the real type. These being addition, subtraction, multiplication and division. Variables are stored in battery-backed RAM and are therefore stored in memory even when the power is removed.

Mathematical functions supported by RAPL are:

ACOS Arc Cosine

ASIN	Arc Sine
ATAN	Arc Tangent
COS	Cosine
SIN	Sine
TAN	Tangent

Variables may be printed to the screen or to a printer in either a real or integer format. An input statement is also supplied to allow data input from the console [CRS 1985b].

LOCATION TEMPLATES - Location templates contain six real values. These values are the three-dimensional cartesian coordinates X, Y and Z, and the yaw, pitch and roll components. Each of these component's values may be copied to a variable and variable values may be copied to a specific component.

The current position of the arm may be stored in a location template at any time by either using the supplied teach pendant in manual mode or by using the RAPL command ACTUAL. Information contained in location templates may be displayed and edited from the console at any time. Location templates may be assigned to other location templates.

There is a provision for a naming convention for location templates. The last three characters of the location name must be a number. Locations with the same name but different numbers are considered related. The WITH command is used to state which group of related templates are to be used. From this point on, only

the number portion is necessary for addressing locations. This is a crude method of providing a structure which is close to an array of locations. This structure proves to be awkward to use and difficult to read in a program.

Location templates may be listed and deleted from memory. As with variables, location templates are stored in battery-backed RAM and therefore remain in memory even when the controller is shut off [CRS 1985b].

MOVE COMMANDS - RAPL supports a rich set of robot move commands. All high level move commands support both joint interpolated and straight line methods of path control. RAPL includes the following move commands

APPRO	Approaches the given point by the specified amount
DEPART	The arm departs from the present location by a specified amount
MOVE	Moves the arm from the current location to a specified location
JOG	Moves the arm along the X, Y and Z axes by the specified incremental amounts

RAPL also includes commands for the movement of the arm at a lower level of control. The arm may be moved at the actuator level by specifying the incremental distance of movement [CRS 1985b].

INPUT / OUTPUT - Commands have been provided to enable the programmer to operate and monitor input/output devices. This is useful for the implementation of sensory equipment monitoring. Also, other devices may be

controlled by the program to interact with the robot arm.

The ONSIG command is used to monitor an input stream. If the state of this input goes high, the specified RAPL program will take over. When the command is completed, control is returned to the program executing at the time of the input status change.

Unfortunately, the OUTPUT command is very awkward to use. The user is required to operate numbered bits corresponding to output devices. The command is functional, but may require the programmer to produce unnecessarily complicated and convoluted code to operate output devices [CRS 1985b].

THE PROGRAMMING ENVIRONMENT - The programming environment provided by CRS Plus consists of a rudimentary line editor and a RAPL interpreter. The editor is useful only for simple edits to existing programs. CRS Plus obviously recognized this fact and provided a mechanism for downloading programs from a micro-computer. This allows the user to develop programs using any text editor. Text files may be downloaded to the controller via ROBCOM and then run in the controller.

Although this is a better alternative, it is still deficient as a programming environment. The user must create RAPL programs, save them as text files, exit the editor, execute the ROBCOM communications program and download the text file to the controller. The user may now run the program in the controller. The development process requires this sequence to be performed a large number of times

which greatly reduces the efficiency of program development.

The interpreter offers some operating system commands which include the ability to list directories of all programs, and display listings of variables and locations which are currently stored in memory. The user is permitted to delete any of these items as desired [CRS 1985b].

PROGRAMMING STRUCTURES - RAPL is extremely deficient in programming structures. It contains an IF structure which, based on a condition, performs a GOTO to the specified line number. The GOTO and GOSUB commands have also been included. This is the extent of programming structures and flow control. This lack of programming structures makes the programming of complex systems very difficult and promotes the development of spaghetti code which is difficult to read, modify and maintain [Fairley 1985].

RAPL REVIEWED - Many of the deficiencies found in the current robotic system may be attributed to RAPL. The language is similar to unstructured BASIC. Although the language has many powerful robot arm specific constructs, it lacks structure and is deficient in data structures. This style of programming is inadequate for designing a robot program which has a high degree of flexibility. It may be adequate for the programming of a simple, repetitious task but lacks the software engineering characteristics which would allow the programmer to develop complex programs [Fairley 1985].

3.5 OVERVIEW OF THE ASPS

Several test runs of the ASPS have shown both promising analytical results and shortcomings in the current design. In order to get the ASPS to perform the sample preparation sequence, the supplied FORTRAN program was used to create a robot program. The program was loaded into the robot controller and run. Many problems were discovered rapidly during this exercise. Robot program changes were required in order to get the arm to complete the required tasks and object locations required reteaching.

The method of creating the robot program produced RAPL code which didn't work properly and was extremely difficult to modify. Although eventually a program was hard-wired to perform the sample preparation routine, program development time was extremely high and required a programmer to get it to work properly. This makes it impractical for real-world usage. In order for the ASPS to be useful, it must be possible for an analytical technician to program and run the system without the aid of a programmer particularly since the ASPS was designed to perform a number of sample preparation procedures.

The teaching of object locations is a major problem with the ASPS. Removal of the heating block work space for cleaning or repairs is a necessity. It is virtually impossible to place the work station in precisely the same location to the required tolerances each time it is removed. Therefore, the robot must be retaught positions each time the work area is removed. This is a very tedious procedure and requires

knowledge of the ASPS programming system and the RAPL programming language.

Other mechanical problems have been experienced aside from the software problems. The robotic gripper must manipulate round tubes and objects. It is impossible for the current design to meet tolerance requirements. For instance, sample tubes are not held rigidly and many times slip out of position. When the robot arm attempts to insert it into a position, it misses the target. This problem has been experienced with virtually every ASPS object.

CHAPTER 4

FORTH BASED ROBOT LANGUAGE DESIGN CRITERIA AND SPECIFICATIONS

The literature review of existing robot languages together with the ASPS hardware design and desired programming capabilities suggested that a new robot arm language was needed as a base language for implementing the ASPS software system. This new language was designed to incorporate programming features which would allow a flexible ASPS program to be developed.

4.1 ASPS ROBOTICS SOFTWARE REQUIREMENTS

The ASPS process is essentially a pick and place type of operation. That is, objects are basically picked up and moved to other places. Objects must be picked up with enough force to prevent dropping, yet gently enough to prevent damage. Also, care must be taken to avoid disturbing the object during movement and collisions must be avoided [Korein 1987].

The ASPS requires a robotics language which will support the development of a system which allows the integration of robot planning with peripheral control. The robot planning mechanism must be developed in such a manner as to allow a non-programmer user to easily develop ASPS programs. As previously discussed, maximum flexibility must be achieved to allow for the preparation of a varying

number of different sample preparation procedures.

The ability to create a friendly user interface is also very important. This may include support for windowing, menus, and other high level interfacing mechanisms. Several arm movements must be supported including straight line and joint interpolation movement to a location. Tool orientation commands should also be included for easy manipulation of the robot gripper.

Location templates should be supported for easy location of objects. The language should also provide a teach pendant interface for the teaching of object locations. Considering the problems experienced with object locations, a useful feature would be the ability to calibrate surfaces or blocks of objects. This would provide the mechanism for the movement of the surface without reteaching the position of each and every object contained on that base, thus creating a flexible working environment for later developments in the ASPS. This is very useful since the ASPS is still in the research phase and is constantly being modified.

Peripheral control is very important in the development of the ASPS planning system. Constructs must be provided for control of peripherals. These must include timed controls, and synchronized management of a varying number of peripherals. It should be relatively easy to remove or add a peripheral to the system. Peripheral feedback constructs would be helpful to allow for error checking.

Mathematical functions such as trigonometric functions would be an asset since robot planning involves the manipulation of geometric locations. Real time

clock checks are important since sections of the analytical procedures involve timed actions such as reaction times. This becomes critical in the case of coordinating several different preparation procedures at the same time. Although this may not be dealt with directly by the robotics language, it should be considered when selecting the language.

The robotics language should be structured for the development of easily maintainable, readable, and modifiable code [Fairley 1985]. At a bare minimum, looping and logic structures should be provided. The language should utilize the full power of the host computer. This includes the use of all the memory, fixed disk, printer, graphics, mouse, communication ports and other features and peripherals associated with a micro-computer.

Error checking routines are important for the detection of collisions and other potential problems. Corrective actions should be provided for problems which arise more frequently and it should be easy for the programmer to develop corrective actions for any other foreseeable problems. Debugging facilities would also be very helpful for the development of robotics planning programs.

In summary, the specifications for the robotics language are:

- The system must run on the host micro-computer in real time through the communications interface
- A powerful program development environment - with an integrated system which includes an editor, interpreter, compiler and debugger.

- Structured flow constructs
- Standard programming language data structures
- Maximum power and flexibility
- Data storage commands which will accommodate the storage of all data easily
- User friendly environment
- Interactive environment
- Ability to build on the language easily
- The ability to calibrate the system in 3 dimensions allowing the movement of work areas without the reteaching of all positions
- Ability to work with peripheral devices at a high level
- Mathematical functions should be built in directly
- Real time clock accessability
- Standard robot language features including
    - Joint interpolated and straight line movements
    - Location teach feature
    - Location template data structures which are easy to use
    - Data structures for location templates which will enable the user to more easily define and program complex tasks

4.2 CHOICE OF A SUITABLE DEVELOPMENT LANGUAGE AND ENVIRONMENT

After reviewing several styles of development languages, FORTH was chosen as the language for development of the robotics language. The choice of FORTH is based on a number of criterion which were developed during experimentation with the current system and the review of existing robot languages. It was decided that the robot language be developed around an existing high level language to decrease development time and avoid the redundancy of developing a new language with features already found in existing languages.

The development of robot programs is prone to a high degree of trial and error much more so than that of conventional programming. Interpreters have become a necessity for the development of useful programs because of the need for a fast program revision cycle [Korein 1987].

The ideal language would provide an interactive interfacing system, with high level language features and the ability to design the robot language in such a way as to allow the user to use the robot language without having to learn the host language in detail. This provides a sufficiently high level language to program powerful robot plans yet sufficiently easy to allow a non-programmer to write robot programs.

The most obvious choice for a development language would be a Pascal-style language which seems to be the most widely used style in the robot language realm. Unfortunately, the use of such a language would require the robot programmer to

have a knowledge of the host language in order to use the robot language features. Also, the environments of such languages are not conducive to an interactive interface with the manipulator.

FORTH is the language which best fits the aforementioned criteria. The literature search did not reveal any languages utilizing the Forth environment as a base language for a robotics language. The following is a discussion of FORTH features and their applicability to the development of a robot language.

FORTH - Forth has been considered a high-level language, an assembly language, an operating system, a set of development tools and a software design philosophy. Forth has a powerful set of built-in commands and provides a mechanism by which the programmer may develop his own commands. This feature is very powerful and directly applicable to the problem of developing a robot language which is easy to use [F-PC 1988a]. Robot commands may be developed, thus allowing the robot programmer to use these commands without learning to program in Forth.

Forth may be run in an interpretive mode or it may be compiled. This provides the flexibility of an interactive programming environment with the ability to create stand-alone robot programs which may be run without the Forth programming environment. Forth also provides a mechanism for defining words (commands) in assembler [Brodie 1987]. This feature is very useful for the development of low-level routines such as the robot/computer communications interface and the handling of

peripherals.

The program cycle time for Forth is greatly decreased since it is totally integrated with an editor and an interactive environment. This reduces the programming and debugging time required for the development of complex systems [F-PC 1988b].

Since the final programs resemble English-like descriptions of the final application, Forth has been referred to as a meta-application language. It is a language that allows the programmer to develop application-oriented languages.

Since user defined commands can be used to define other commands, the environment is ideal for expansion and further development. After the robot programming language has been developed, changes may be required as changes to the ASPS hardware are made. Forth provides structured control operators which force the programmer to develop a nested program design. Forth is designed to handle small subroutines (words) with virtually no cost in efficiency. This encourages information hiding which simplifies program enhancement. Forth is a very efficient and fast language which is useful for real time applications. Timing may be important if the robot language is to handle sensory information in real-time. Forth is also transportable and has been implemented on a large number of different mini and microcomputer systems.

Forth has been used in the arts, for business and personal computer software, data acquisition and analysis, expert systems, graphics, medical, portable intelligent

devices, process control and robotics [Brodie 1987].

Forth provides an interpreted and compiled language environment. It provides a mechanism to develop a robotics language which is easy to use for the non-computer programmer and also very powerful for the experienced programmer. It provides high level language features as well as low level capabilities. Forth satisfies all the requirements for the development of a powerful, flexible and easy to use robot language.

F-PC DIALECT OF FORTH - The F-PC implementation of Forth was chosen as the implementation language for the development of the robotics language. This version of Forth was implemented by Tom Zimmer and Robert L. Smith [F-PC 1988a]. It is a greatly enhanced version derived from the F83 model for the IBM-PC, XT, or AT by Henry Laxen and Michael Perry. F-PC is a sequential file system as opposed to the standard Forths which use a block file format. The F-PC system advantages are:

- Smaller source files
- Portability and standardization of files. Most programs and editors may read sequential files
- No limitation is placed on the size of a word definition
- No limitation on the number of comment lines in a definition
- Code is more easily modified with sequential files

The F-PC system is designed to address the following objectives:

- A system that is familiar to the large installed base of F83 users
- A system that brings Forth fully into the realm of files
- A Forth system that provides concepts familiar to C and Pascal programmers
- A system which maintains Forth's interactive nature
- A system which is fast to compile
- A DOS FAT (File Allocation Table) system with many tools, which still has room for a large application program

F-PC's major features include:

- Direct threaded dictionary for speed
- Separated lists and heads to increase space
- Prefix assembler to enhance readability
- Assembler supports both prefix as well as postfix assembly syntax for familiarity
- Full DOS access from system and Command level control
- Full Handle/Path based file system
- Full user configurable sequential text editor provided in source
- Full DOS memory management interface
- System based time and date functions

- Full screen editor SED for sequential text files is integrated into F-PC for ease of creating and modifying source code
- Very fast screen I/O is provided for editor and normal text display.
- Paths are fully supported, as entered from the command line, or applied to a file automatically if not specified
- Built in DOS commands such as DIR, COPY, FORMAT, DEL, RENAME, MD, RD, CHDIR, PATH, etc.
- Cursor shape control
- Extended memory access routines are provided
- Function and control keys are supported for program use
- The amount of directly available user space has been greatly increased. Somewhere in the order of 400K for programs and data.
- User interface words are provided including a windowing package.

The F-PC environment provides an ideal interactive system for both the development of the robot language and for the interface which will be made available to the language user. This eliminates the need for developing one specifically for the robot language.

4.3 DESIGN OF THE ROBOTICS LANGUAGE

A balance between power and ease of use must be struck for the implementation of a robotics language which is suitable for useful, commercial

robotics planning. Since RAPL contains the fundamental functions required for a rudimentary robotics language, it was decided that RAPL would be used as a low level language and the Forth environment would generate RAPL commands to implement the higher level language. Forth is used to establish the two way communications link which is used to send RAPL commands to the CRS Plus controller and receive feedback from the controller. This is highly important since this type of link is required for the implementation of an interactive language and the ability to control the robot completely from the host computer.

CHAPTER 5

FBRL DEVELOPMENT

The implementation of FBRL was accomplished in the following sequence:

1) implementation of data structures which would be used to implement the language.
2) development of a communications interface which facilitates the micro-computer control of the robot arm in real time.
3) implementation of a number of words which would form the basis of using RAPL as the low level language.
4) implementation of the robot movement commands.
5) implementation of complex data structures for the manipulation of robot real world coordinates.
6) implementation of higher level constructs such as the calibration system.
7) testing and debugging of the system.

Figure 5.1 is an illustration of the total FBRL system.

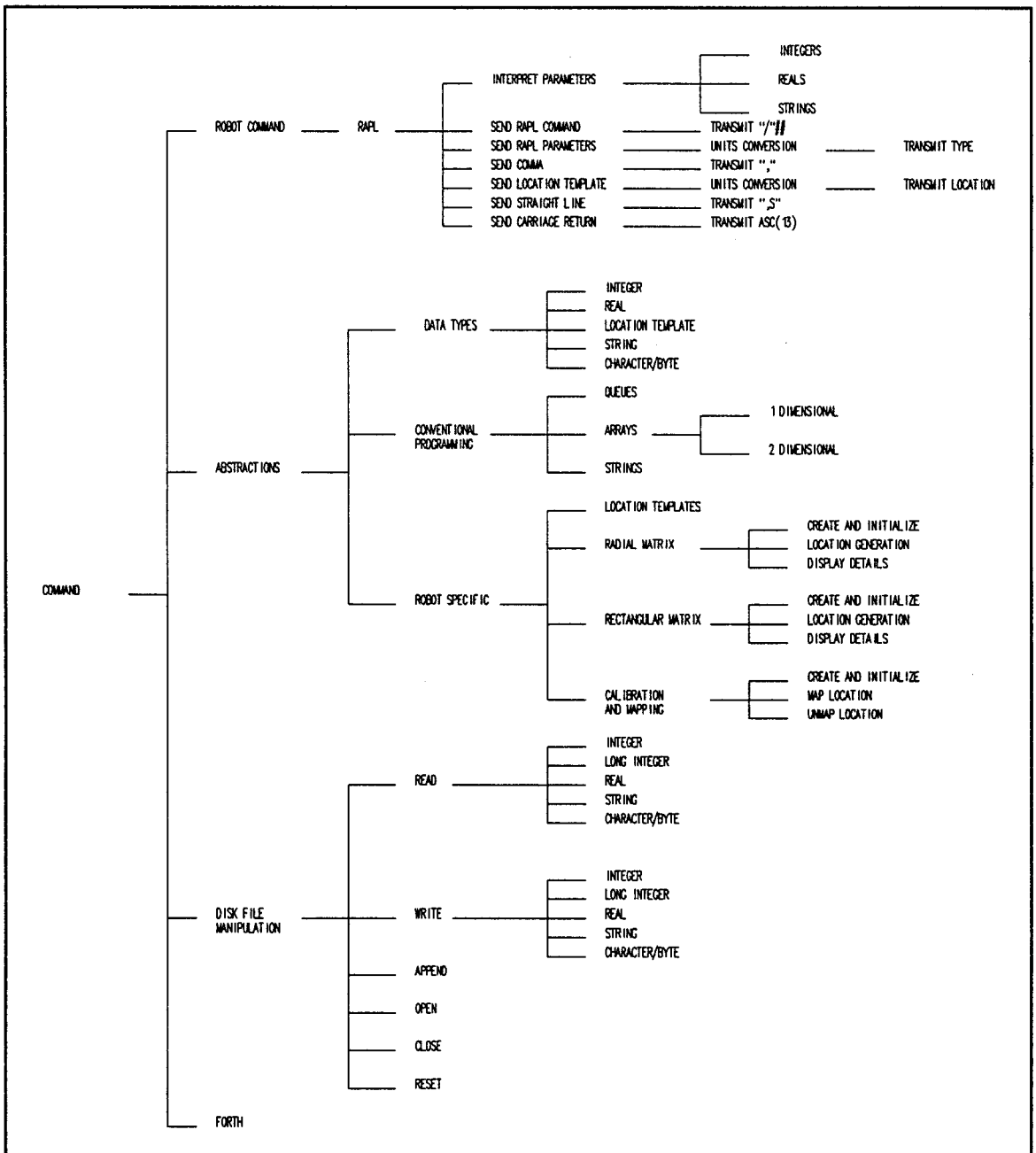

Figure 5.1 Summary of FBRL features.

5.1 DATA STRUCTURES

Several data structures were required for the implementation of a number of the features. These structures include queues, arrays and strings.

QUEUES - A Queue data structure was developed for the implementation of the communications interface. This abstract data structure was designed to provide the user with the ability to create any number of named Queues. This structure was originally implemented in high level Forth. Subsequently, it was decided that several routines should be implemented in low-level code since they were to be used for the communications buffer. Both sets of routines are 100% compatible.

The abstract data structure was implemented to store one-byte characters. It would not be very difficult to implement a generic structure which would store any data type, but since there was no anticipated need for such a structure, the type specific data structure was implemented in order to speed up the routines and cut down on the development time.

The queue structure is implemented by creating an array of one-byte characters which is six bytes longer than the size of the queue. Figure 5.2 shows the implementation structure for the queue.

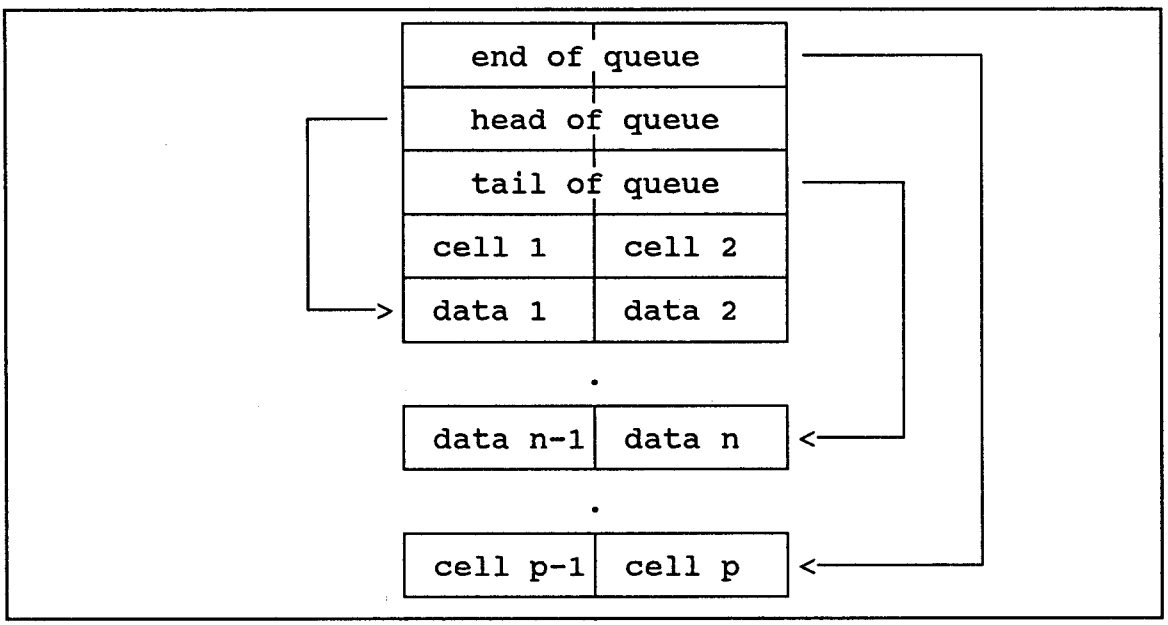

Figure 5.2 Implementation structure for a queue.

ARRAYS - The abstract data type for arrays was included for use in the implementation of the language as well as use as a data structure in the language itself. One and two dimensional arrays were implemented as generic abstractions which may store any data type.

ONE DIMENSIONAL ARRAY - The implementation structure for the one dimensional array is illustrated in Figure 5.3. The run-time code for ARRAY returns the absolute address of a cell given the cell index.

For example, 5 X returns the absolute address of the cell from the array X at index 5.

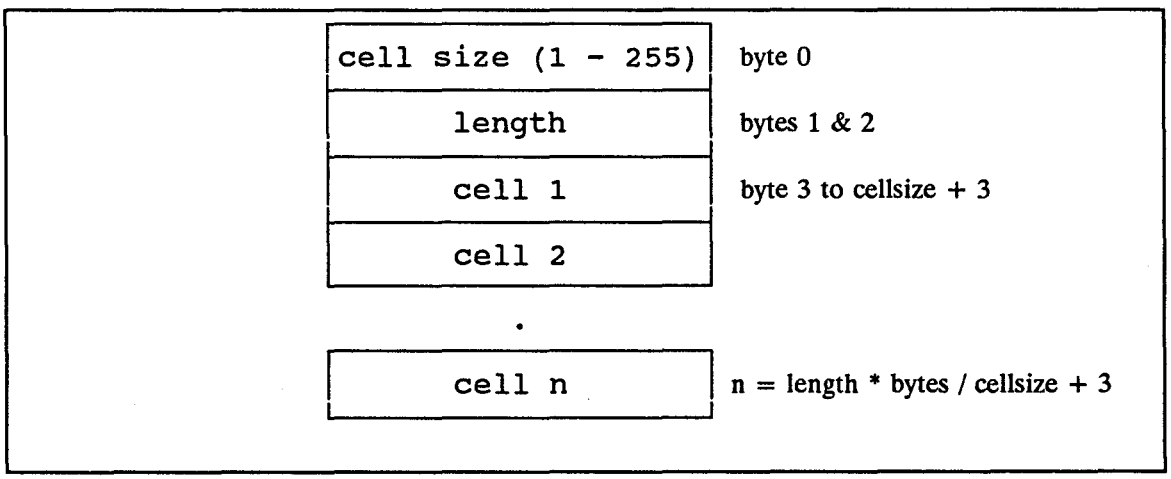

Figure 5.3 Implementation structure for a one dimensional array.

TWO DIMENSIONAL ARRAY - The two dimensional array was added as a vehicle for storing matrices. This is useful in robotics for the representation of a matrix of objects such as the reaction block of the ASPS. The implementation structure is shown in Figure 5.4.

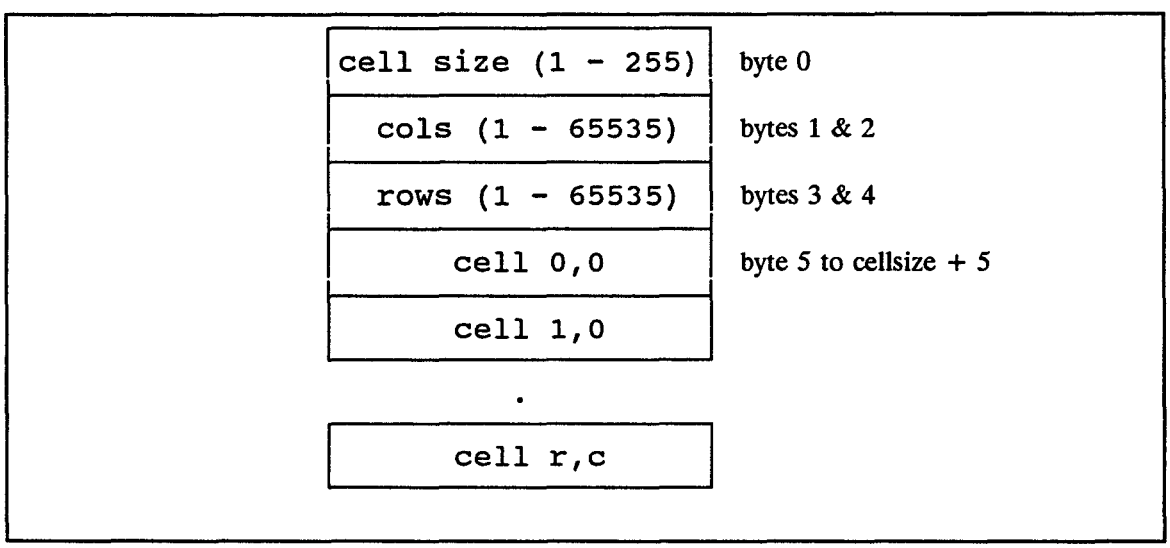

Figure 5.4 Implementation structure for a two dimensional array.

Storage space for a two dimensional array is defined by rows * cols * (bytes per cell)+5. The run-time code for 2ARRAY returns the absolute address of a cell given the row and column of the cell.

5 6 X returns the absolute address of the cell from the two-dimensional array "X" at row 5 and column 6.

STRINGS - String manipulation was necessary for the implementation of the RAPL interface for the language. It also gives the language more flexibility and ease of use. The string structure is shown in Figure 5.5.

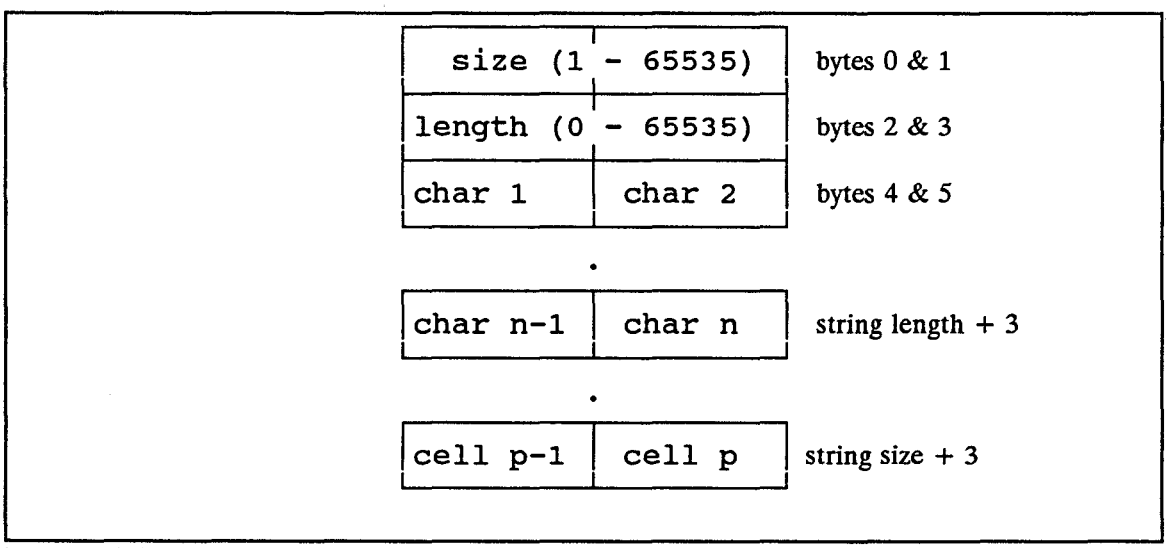

Figure 5.5 Implementation structure for strings.

5.2 COMMUNICATIONS INTERFACE

In order to communicate in real time with the robot controller, a serial communications interface was written for the RS232 interface which could simultaneously transmit and receive data in the background transparent to the routines which require the data. All data is transmitted and received through buffers. These buffers are created using the queue structure defined in section 5.1. In order for a routine to transmit characters it need only store the character in the transmit buffer. To receive, the routine need only get the character from the receive buffer. The XON/XOFF protocol has been adopted as the handshaking mechanism.

The use of a background communications interface serves two purposes. It provides a basis by which the robotics language may simultaneously process data while communicating with the robot controller and hides the details from the user providing a high level interface used to communicate data to the controller.

In order to provide a communications link the following steps must be taken:

1) A read and write buffer must be created
2) The base address for the UART (Universal Asynchronous Receiver/Transmitter) corresponding to the required communications port must be retrieved
3) The UART chip must be programmed to provide the required communications protocols

4) The interrupt vectors must be revectored to jump to our communications routines

5) Communications interrupts must be enabled for serial port read and write

6) A routine must be provided which handles read and write interrupts

READ AND WRITE BUFFERS - Two buffers are provided for writing and reading data to and from the port. The Queue routines are used to implement these buffers. Low-level routines have been provided to be used by the low-level communications routine which are all written in assembler. These queues act as an ideal medium for passing data between high and low level routines.

SETUP ROUTINES - Before any communications may occur, the hardware must be set up to begin background receive and transmit as well as the setup of communications protocols. User defined parameters are word length, number of stop bits, parity and baud rate. Each protocol is implemented by a single Forth word. These words are then used in the implementation of the COMM-SETUP word which provides an easy to use vehicle for the communications protocols configuration. COMM-SETUP hides all the technical details from the user and provides a high level interface by which to configure the communications port.

A high level interface is provided which only requires the communications protocol information passed either on the stack or as parameters to COMM-SETUP. This information is then used to program the UART.

PROGRAMMING THE UART - The UART chip physically transmits and receives characters through the RS232 pin connector. The DOS routine INT 14h is used to accomplish the programming of communications protocols. A two byte variable is set up by these routines to comply with the INT 14h parameter format [Prosise 1989]. Each routine manipulates the lsb of this variable to reflect the corresponding communications protocols with the entry conditions of the BIOS routine. The variable is then passed to the SETUPUART routine which stores it in the DX register and calls the 14h BIOS routine [Tandy 1984].

INTERRUPT REVECTORING AND ENABLING - Background communication is accomplished through the use of hardware interrupts. Interrupt vectors for the communications interrupt must correspond to the location of our read/write routines. This is accomplished by the INSTALL-INTERRUPT routine. This routine uses the DOS service 25h [Tandy 1984] to store the original interrupts before the revectoring. The variables COMMINT-SEG and COMMINT-OFF are used to store the interrupt segment and offset respectively.

At this point the interrupts for communications port read and write must be enabled. This is accomplished in the COMM-INT-ENABLE routine. The routine unmasks the IRQ4 interrupts in the 8259's IRQ mask register and initializes the interrupt enable register [Krantz 1983]. The UART's (8250) registers are then set to enable data receive and transmit.

COMMUNICATIONS INTERRUPT SERVICE - The heart of the communications interface is the receive and transmit interrupt service routine. This routine's address is the target of the interrupt revectoring. When a communications hardware interrupt is detected, the interrupt controller performs the jump to our communications routine (COM).

COM saves the registers on the system stack to preserve the machine state prior to the interrupt. The line control register bit 7 - Divisor Latch Access Bit - is cleared in order to ensure that the Receive/Transmit Hold register is accessed. The interrupt identification register is checked for whether a character was received or whether the transmit register is empty.

If a receive status is detected, the READ-COM routine is called. If the transmit register is empty the WRITE-COM routine is called. The interrupt identification register is checked again to ensure that there is no interrupt pending. That is, no communications interrupt has occurred during the previous interrupt service routine. If an interrupt is pending, a software interrupt is initiated in order to satisfy the pending interrupt. This prevents the loss of data during high speed transmissions. The 8259 is then signalled with an EOI (end of interrupt) code to reestablish the interrupt handling. All registers are restored and an interrupt return (IRET) is issued.

The READ-COM routine is called via the COM communications interrupt service. The character is read from the receive register and stored in the receive buffer by using the low level queue routine ENQ-L. The received character is checked for the XON/XOFF code. If detected and the XON/XOFF protocol is enabled, transmission is halted by setting the XOFF code.

If the receive buffer is full, the XOFF code is immediately transmitted to halt any further transmission. Control is passed to the WRITE-COM routine from the communications interrupt service. The next character in the buffer is extracted and written to the transmit register once the transmit register is deemed empty.

RESETTING INTERRUPTS - Upon exiting the FBRL system, the communications interrupts must be restored to the previous vectors. Since the interrupt service routine no longer exists, an interrupt request would result in a crash.

5.3 ROBOT INSTRUCTION SET

The robot instruction set consists of commands which directly affect the arm status. That is, the arm position or the arm status. The basis of this instruction set is RAPL. As previously mentioned, the RAPL language is rich in robot language constructs so the first stage of the robot instruction set was to implement the RAPL language in FBRL. The required steps for implementing a RAPL command are:

1) interpret command line

2) send RAPL command number to controller

3) send parameters, commas (if required) to controller

4) send location template (if required) to controller

5) send straight line command (if required) to controller

6) send carriage return to controller

In essence, the sequence generates a tokenized RAPL command which is transmitted to the CRS-PLUS controller which in turn runs the RAPL command. A tokenized RAPL command consists of a command number rather than the text version of the command. This process eliminates a majority of the parsing and interpretation in the robot controller. Also, use of the tokenized command slightly reduces the amount of communicated data.

The routines for interpreting the command line may be found in INPUTS.SEQ. This option was included to enable the user to issue RAPL commands in a format which lists the parameters after the issuance of the command. The user may still pass the parameters via the stack in the traditional Forth fashion. The parameter interpreter interprets reals, integers and strings, and places them on the stack. The routine may now work with this data as though it had been passed on the stack by the calling routine.

The RAPL command number must be transmitted to the controller. This command number is an integer which is transmitted in RAPL ("/"##) format by the

command SEND-COM. RAPL parameters are sent via the SEND-INT, SEND-REAL and SEND-COM commands. These commands send integer, real and byte or character values to the robot controller. Commas are used as delimiters in RAPL and may be transmitted to the controller by issuing the SEND-COMMA command.

RAPL commands which require one or more locations as parameters utilize the SEND-LOC command. This command sends a location template in RAPL format as part of the parameter line of a RAPL command. The straight line parameter for RAPL is the same for all RAPL commands. Therefore, a separate routine was included for the transmission of the straight line character sequence to RAPL. This sequence is simply the transmission of ",S".

The SEND-CR routine simply transmits a carriage return (ASCII 13) to the robot controller. It is used at the end of every RAPL command to denote the end of the command and hence instruct the controller to interpret and effect the command.

5.4 ROBOT LANGUAGE DATA ABSTRACTIONS

RECTANGULAR LOCATION MATRIX - The rectangular location matrix is designed for robot applications in which objects are located in a high precision three-dimensional rectangular matrix. The objects are located on a matrix such that they are all placed at equal distances from each other. The data abstraction automatically produces locations by calculating offsets along each of the three axes

and shifting the location from a base address.

In RAPL, the only way to simulate this function is to teach the robot arm each and every location in the matrix. This method requires an initial calibration of the structure by teaching the arm a base location and specifying the offset between objects for each axis.

The matrix is labelled using an ordered triple (x,y,z) for each location. By calling the routine with the specified ordered triple, the location is calculated and returned. The data structure for the rectangular matrix is shown in Figure 5.6.

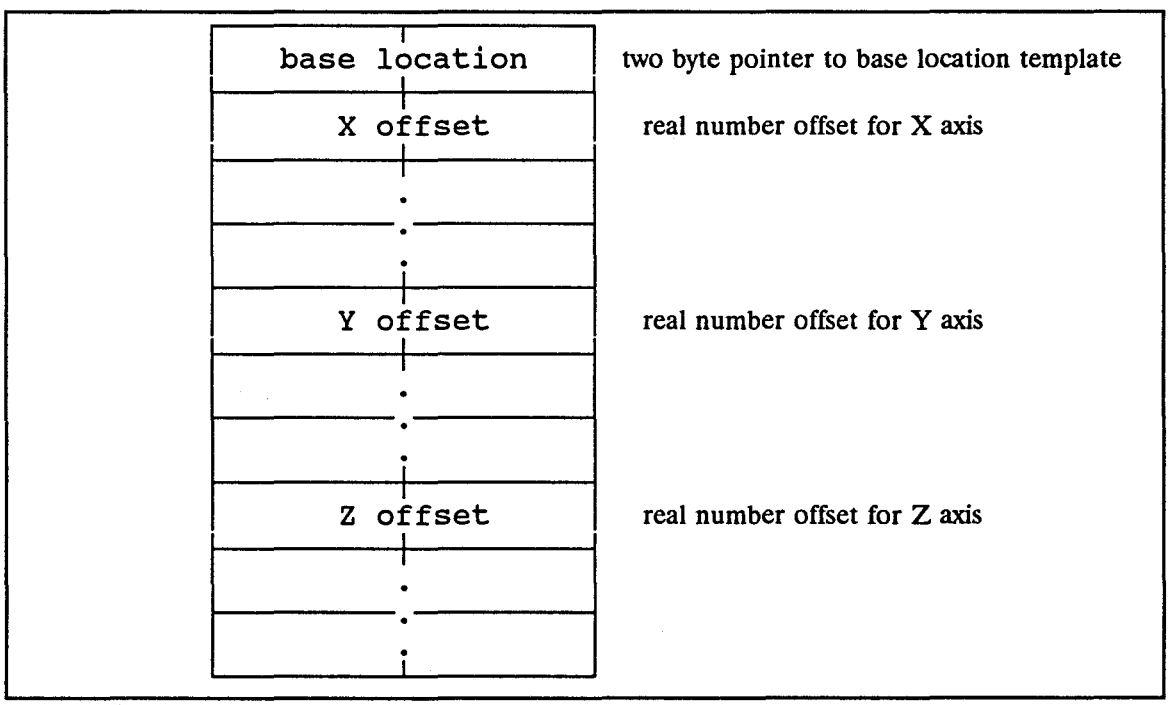

Figure 5.6 Implementation structure for the rectangular matrix data structure.

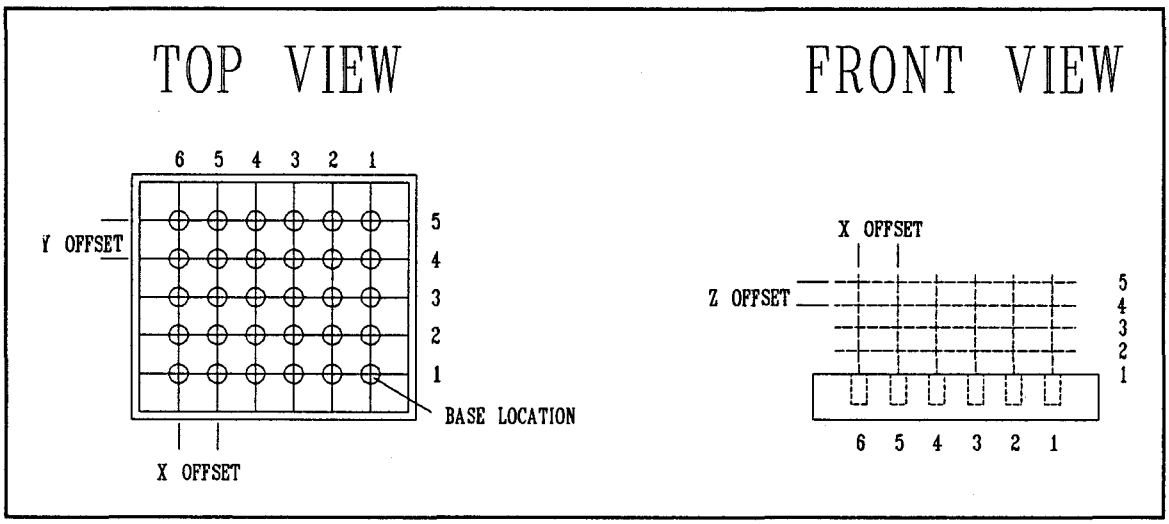

Figure 5.7 Rectangular matrix illustration.

Figure 5.7 illustrates a rectangular matrix configuration. The rectangular matrix provides an easy method of addressing a matrix of locations. Typically, objects are placed in a two-dimensional matrix of locations as found with the ASPS heater block. The third dimension (Z axis) may be used to facilitate objects of varying heights which may be placed in the two-dimensional matrix.

In order to completely understand the rectangular matrix, an example of the set up and use is provided.

LOC 1.0 2.0 3.0 RECTMAT MAT

This particular example sets up a rectangular matrix which has offset spacings of 1.0, 2.0 and 3.0 for the X, Y and Z components, respectively. The base location is LOC and the rectangular matrix name is MAT. The run time portion of the rectangular matrix yields a location based on the previously specified parameters.

1 1 1 <loc> MAT

This example would calculate the X, Y, and Z components of the matrix location (1,1,1) and deposit them in the location template <loc>. Each component is calculated using the offset information as shown in the following example.

<loc>.x = LOC.X + (X-index * X-offset)

By providing this data structure, we eliminate the need to either teach a myriad of locations and hence chew up unnecessary amounts of memory or the need to hard code such calculations for each object matrix. Implementation of the structure as an abstract data type produces an easy to use mechanism for locating objects in a robot workspace which has a number of rectangular object matrices as found in the ASPS.

RADIAL MATRIX - The radial location matrix is designed for robot applications in which objects are located in a high precision three-dimensional circular (radial) matrix. The objects are located on a matrix such that they are all placed at equal distances along the Z axis (height) and between radial rings, and the same incremental angle between radials. The data abstraction automatically produces locations by calculating offsets along each of the three axes and shifting the location from a base address.

As found with the rectangular matrix, the only way to simulate this function in RAPL is to teach the robot arm each and every location in the matrix. The radial matrix abstraction requires an initial calibration of the structure by teaching the arm

a base location and specifying the incremental angle between radials, the distance between rings, and the Z axis offset.

The matrix is labelled using an ordered triple (x,y,z) for each location. By calling the routine with the specified ordered triple, the location is calculated and returned. The implementation structure for the radial matrix is illustrated in figure 5.8.

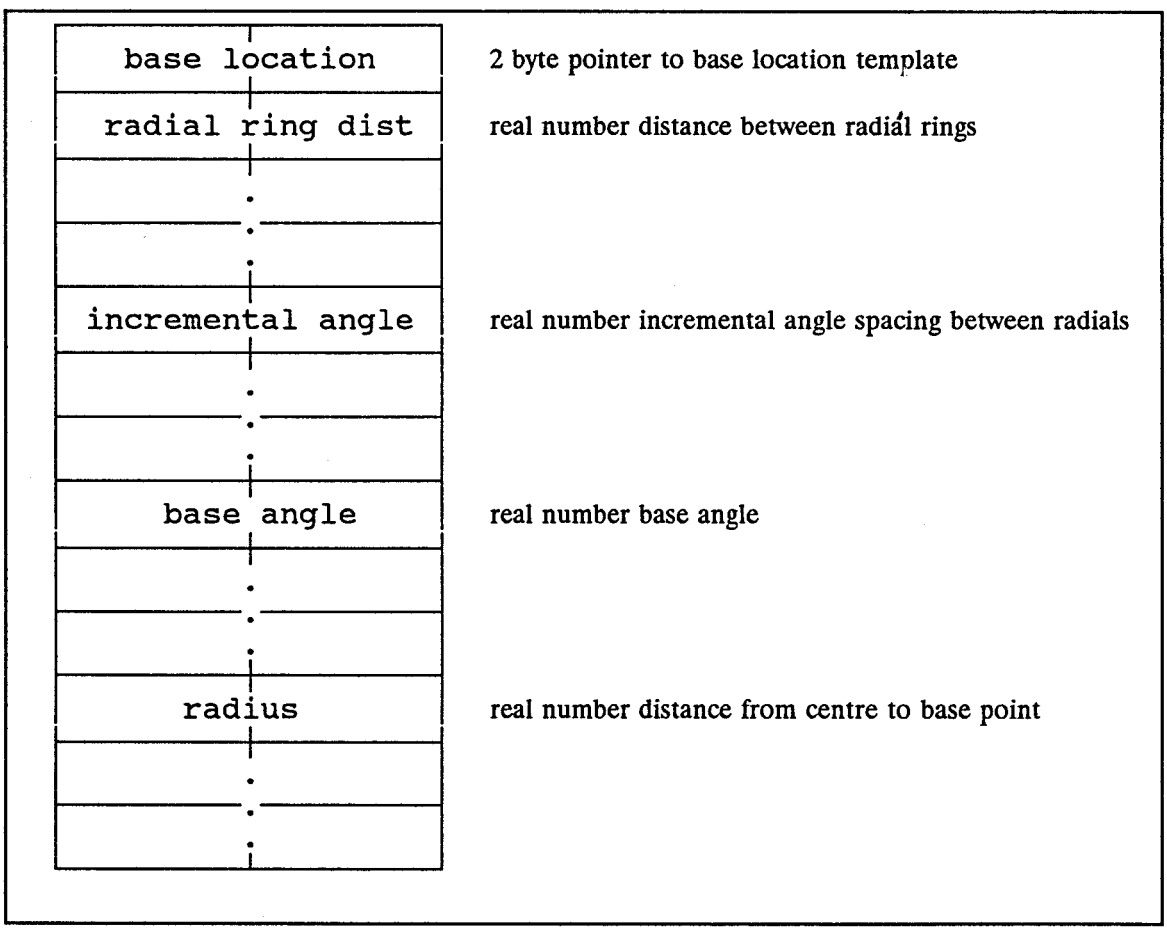

Figure 5.8 Implementation of the Radial Matrix Data Structure.

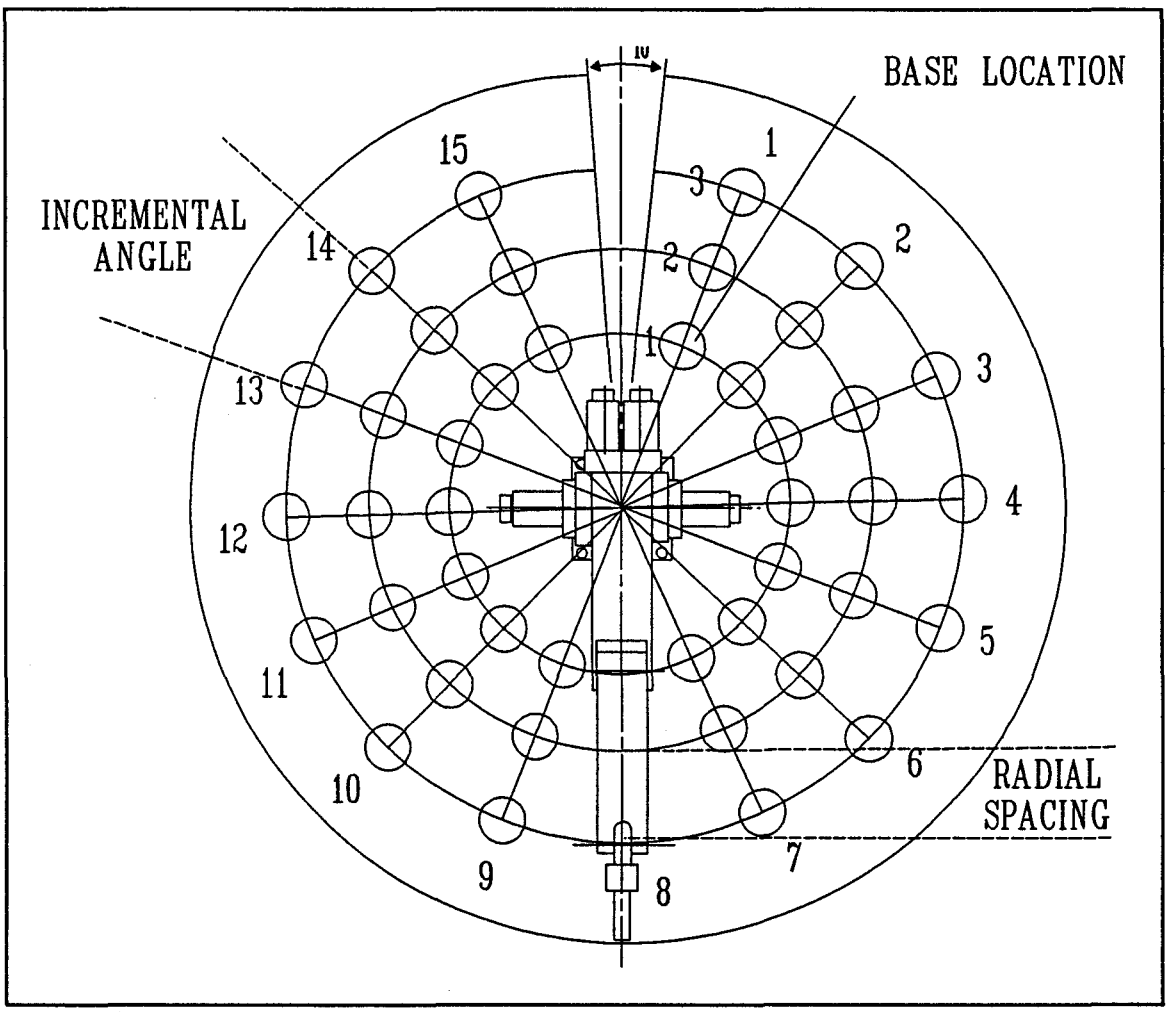

Figure 5.9 Example of a radial matrix.

RADMAT calculates several of the intermediate components and stores them in the data structure for use during runtime. These components are the base angle and the radius. The base angle is calculated by extracting the X and Y components of the base location, and using ANGLE-FROM-ORIGIN to calculate the base angle from the point (0,0). The radius is the distance from the centre to the base location.

The X and Y components of the base location are used to calculate this distance from $R = \sqrt{X^2+Y^2}$.

The runtime version of RADMAT uses the stored information coupled with the indices passed as parameters to calculate the corresponding location. Firstly, the radius to the indexed ring is calculated by multiplying the ring-index by the ring spacing. Next, the incremental angle is multiplied by the increment index to obtain the angle from the base location. At this point, we have the radius and the angle from the origin. By using CALCPT, we may now calculate the X and Y components of the location which is addressed by the specified parameters. The Z component is calculated as an offset as in the case of rectangular matrix calculations.

Radius = Ring-Index * Ring-Spacing

Angle = Radial-Index * Incremental-Angle

X = Radius * COS(Angle)

Y = Radius * SIN(Angle)

Z = Z-index * Z-offset

Figure 5.9 shows an example of the configuration of a radial matrix. This type of configuration takes advantage of the robot arm's total workspace. It provides a natural placement of objects which corresponds with the circular shape of the arm's

workplace. An example of the use of this data abstraction follows:

inc-angle ring-dist Z-off base-loc RADMAT MAT

0.2 0.5 1.0 LOC RADMAT MAT

This example yields a radial matrix which has an incremental angle of .2 radians, radial rings which are at 0.5" spacing and a 1.0" Z axis offset. The base location is LOC and the radial matrix name created by RADMAT is MAT. The runtime portion RADMAT accepts the radial ring index, incremental angle index and Z-index as parameters and returns the calculated absolute location coordinates.

rad-ring-index incremental-angle-index Z-index <loc> MAT

1 1 2 <loc> MAT

This example yields the location at index 1 1 2 and deposits the X, Y, and Z components in the location template <loc>.

5.5 THREE-DIMENSIONAL CALIBRATION / MAPPING SYSTEM

Many of the problems experienced with robotics systems is the precision and accuracy with which the object locations must be placed. With conventional systems, any movement of the placement of objects must be taught to the robot arm. Therefore, a movement of a base supporting many objects requires the programmer to reteach every position. This movement may be caused by the reorganization of the robot workspace or by the inability to reposition a base set of objects to required

tolerances.

This latter scenario was experienced with the ASPS. Since the heating block must be removed for cleaning, problems arise in the placement of the block in the precise location with which each of the objects' locations were taught, consequently, the robot arm must be retaught each and every object location.

A solution to this problem has been implemented in FBRL. A calibration and mapping system has been developed which calibrates the position of a block of objects and remaps the locations when the block is moved. A three-dimensional calibration and mapping system has been implemented for the calibration of a three-dimensional object block.

The system is calibrated by designating three precise locations which represent a plane on the block to be calibrated. The first time an object block is used, it is calibrated by teaching the three locations and storing them in a calibration data structure. Each time the block is moved, the robot arm is guided to the three calibration locations. These locations are then used to form the new plane which defines the orientation of the block. From this point on, all previously taught locations are mapped to correspond to the new orientation and position of the block.

One problem arises with this system when the block is moved from the original position and a new location must be taught. The system has no way of discerning when locations are taught. The way FBRL gets around this problem is by reversing the calibration method for taught locations where taught locations are mapped back

to the block orientation of the original calibration plane. All locations are converted to their corresponding locations on the original calibrated block orientation which alleviates the aforementioned problem.

A data abstraction has been designed to implement the three-dimensional calibration system. Each block of objects may be defined by a different calibration abstraction. The robot work space may now be totally modular and dynamic. Object blocks may be moved freely without restrictions in terms of three-dimensional orientation.

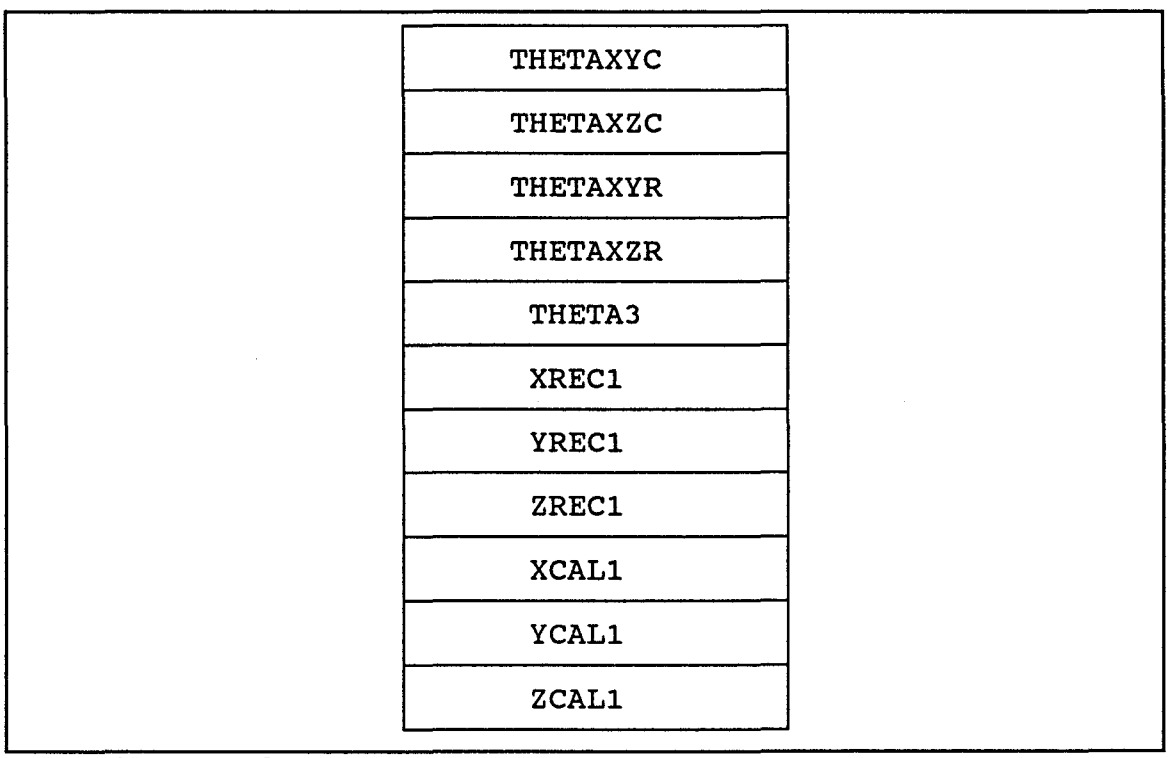

Figure 5.10 3D Calibration data structure.

A block is calibrated by using three locations to calculate intermediate data. This data is then used during runtime to map locations onto new block orientations. A detailed description of the steps and formulas required to perform the three-dimensional calibration system may be found in Appendix II.

This calibration system is incorporated into FBRL as an abstract data type. The user may set up as many calibration surfaces as desired by naming each calibration system. The structure is an array of reals, 11 bytes in length. It contains the intermediate values for the calibration structure. The data structure is illustrated in Figure 5.10.

5.6 DATA CONVERSION ROUTINES

The ability to operate in different units is very desirable in robotics. RAPL units are strictly in imperial units whereas the ASPS and most other engineering systems are based on the metric system. Therefore the ability to readily switch between units should be available to the programmer. Although the task of converting between metric and imperial units is a simple task, the method of implementation determines the effectiveness and ease of use of the conversion system.

It was decided that since RAPL only works in imperial units (inches), all numbers should be stored as such. Therefore we need only worry about the implementation of metric units (centimetres). This is done at the input and output stage providing the user with a metric or imperial interface.

A global flag is set by using the METRIC-ON and METRIC-OFF words. These words are used to switch between metric and imperial units respectively. All input and output of location components are subject to the conversions. Scalar variables are considered to be unitless and therefore are not subject to unit conversions. The programmer is responsible for ensuring that scalar variables considered to have specific units be converted as necessary using the IN->CM and CM->IN words.

5.7 DATA FILE ROUTINES

A set of routines were implemented for the use of disk files for mass storage. These routines include the mechanisms for opening and closing files and reading, writing and appending the various data types. Included in these types are integer, long integer, real, character, and location templates.

Data is stored on disk in binary format. The Forth hwrite and hread words are used for the read and write routines. The routines are set up to perform rudimentary type checking and detect errors if the type does not match.

To use the routines, the user must first create a file by specifying the file path using the CREATE-FILE word. Once a file has been created it may be opened using the OPEN-FILE command. This command requires both the filename and a handle. From this point on, only the handle is required for file access. A file may be referenced from the beginning by using the RESET-FILE routine. This routine

uses the F-PC movepointer word by moving the file pointer to byte 0 of the file.

The user may also append data to a file using the APPEND-FILE command. The command uses the FPC endfile and seek command to set the file pointer to the end of the file. Any data type may be read or written from a data file using the READ-FILE and WRITE-FILE words. These words require the variable name, the handle and the data type. The disk file system was designed to enable the user to easily store and retrieve any type of data to and from a disk file.

5.8 TYPE DEFINITIONS AND CONSTANTS

FBRL includes 5 data types which are listed below.

CHR	Character (1 byte)
INTEGER	integer (2 bytes)
LONG-INTEGER	long integer (4 bytes)
REAL	floating point (F#BYTES where F#BYTES is a constant defined in the F-PC floating point package)
LOCATION	location template type (6 * F#BYTES)

Data types are merely constants which contain the length, in bytes, of the data type. Therefore, programmers may define their own data types by defining a constant which contains the length of the data type. This is a very powerful mechanism which provides an interface to all abstract data structures and routines which access different data types such as the FILE routines.

CHAPTER 6

SUMMARY

6.1 PERFORMANCE CHARACTERISTICS

Testing and debugging of FBRL was limited mainly to mathematical and theoretical comparisons due to mechanical problems with the robot arm and the ASPS. The system requires a mechanical and electronic overhaul in order to operate properly. Furthermore, the hardware system should be modified to incorporate recommendations suggested in the Povilonis [Povilonis 1988] report as well as those listed in this report. Those areas which require immediate modifications are the gripper, the wash probe and the syringe resting area.

The current gripper is not designed to grasp cylindrical objects as found with the ASPS. A new design should enable the gripper to accurately grasp the objects without any slippage, thereby allowing the system to operate to tolerances at which it was designed.

Experimentation with the ASPS wash probe revealed that the resin adheres to the probe. If this resin is removed from the reaction tube, analytical values would err on the low side. Further, there is a high probability of cross contamination between reaction vials.

The current ASPS layout has the syringes located in an awkward position for

the robot arm gripper. The arm must perform two manouevers in order to use a syringe. The syringe must first be gripped with the gripper parallel to the base, then replaced closer to the arm and finally regripped with the gripper perpendicular to the base for dispensing. This is highly inefficient and requires unnecessary robot arm manipulation.

FBRL data structures were tested thoroughly and were shown to be efficient and accurate. Robot language data abstractions were tested mathematically to ensure correctness. All structures operate very efficiently with respect to both memory requirements and operational speed. Data conversion routines were also tested mathematically to ensure correctness.

The communications module was tested by communicating with both another micro-computer and the CRS-Plus controller. The FBRL communications interface works very well at communication speeds as high as 9600 baud. It was discovered, though, that this speed is too fast for the controller as transmitted characters were missed by the controller. There were no errors experienced when communicating with the controller at communication rates of 2400 baud and slower.

The robot instruction set was tested both theoretically and partially on the robot arm. All instructions were transmitted to a micro-computer and all instruction codes received were manually checked for correctness. Instructions tested on the robot arm operated correctly, although the arm was not 100 % functional.

Further work should be performed to test the operation of the language once a fully functional CRS-Plus robot arm is made available. Most instructions not requiring parameters should work correctly as they were already tested on the robot arm. The area which has the highest potential for problems is robot instructions requiring parameters including location templates, variables and constants. The passing of parameters between the controller and the host computer is particularly complex and requires the interaction of FBRL and RAPL programs.

6.2 OVERVIEW

FBRL was designed to be a robotics language which would provide the basis for development of a powerful software system for the ASPS. Incorporated in the language is a set of powerful tools which will make the job of writing such a complex system much easier. FBRL utilizes Forth as the development language and therefore inherits Forth's interactive environment which is essential to the robot programming environment. The language was designed around the CRS-Plus RAPL robotics language to take advantage of the powerful robot manipulator constructs. The FBRL environment provides the maximum power from the micro-computer which enables the user to take advantage of disk storage, large user memory work area and other assorted peripherals. A summary of the FBRL characteristics are described below.

Program development with FBRL is simplified using an environment comprised of an integrated system which includes an editor, interpreter, compiler and

debugger. The structured flow constructs of Forth and standard programming data structures were implemented to provide the programmer with the constructs for developing complex algorithms and provide a mechanism for creating readable, modifiable and robust code. These structures include one-dimensional and two-dimensional arrays, queues and strings.

FBRL provides a user friendly and interactive environment making the implementation of any system as simple as possible. The FBRL language subset may be easily extended as with any Forth system and thus the language may be expanded as new constructs and commands are found to be useful.

FBRL has the built-in ability to work with peripheral devices at a high level which is very desirable in the programming of robotics systems. Such devices as sensors, cameras, switches, etc. are commonly found in robotic systems. Data storage commands accommodating the storage of any data type have been included in the language to facilitate longterm mass storage of system data.

The system runs on the host micro-computer in real time through a background communications interface providing a transparent interface between the host computer and robot arm controller. Mathematical functions are built in directly and real time clock accessability has been provided.

Included in FBRL are standard robot language features including joint interpolated and straight line movements, location teach feature and location template data structures which are easy to use. Robot-specific data structures were

incorporated to provide powerful, yet easy to use, tools for the design of flexible robot plans. These structures include arrays of locations, location rectangular and radial matrix and a three-dimensional calibration system. The calibration system, which calibrates the system in 3 dimensions, provides the ability to move work areas without reteaching all positions in those areas.

The next step in the development of the ASPS software is to utilize the power and facilities of FBRL to develop the ASPS system. Information gathered from the research and use of the ASPS has been used to compile the following specifications. These specifications should be considered during the design and development of the ASPS software system.

A non-programming interface is needed for an implementation of a user-friendly ASPS software system. A menu driven system should be designed to provide the user with a powerful interface which requires little training to use effectively. Programmed manouevers can be incorporated into this type of system. The system must also have the ability to be extended without the need for programming. New manoeuvres should be added through the ASPS user interface. This provides the mechanism for modifying the software system to correspond with new method developments without the need for computer programmers.

Levels of complexity should be built in to the system. This provides maximum ease of use with maximum flexibility. Users who wish only to use already developed methods may do so without learning much about the system. Users who wish to design and implement methods need learn only that required to complete the implementation. Those who wish to program new analytical techniques do so with a more intimate knowledge of the system. The user should only be required to learn enough to use the system at their level. Figure 6.1 illustrates a brief schematic design specification for the ASPS software system.

In summary, FBRL has been designed as a general robotics language which has been designed with the analytical laboratory in mind. The language has been tailored to facilitate the development of the ASPS software system through a powerful and easy to use robot programming environment.

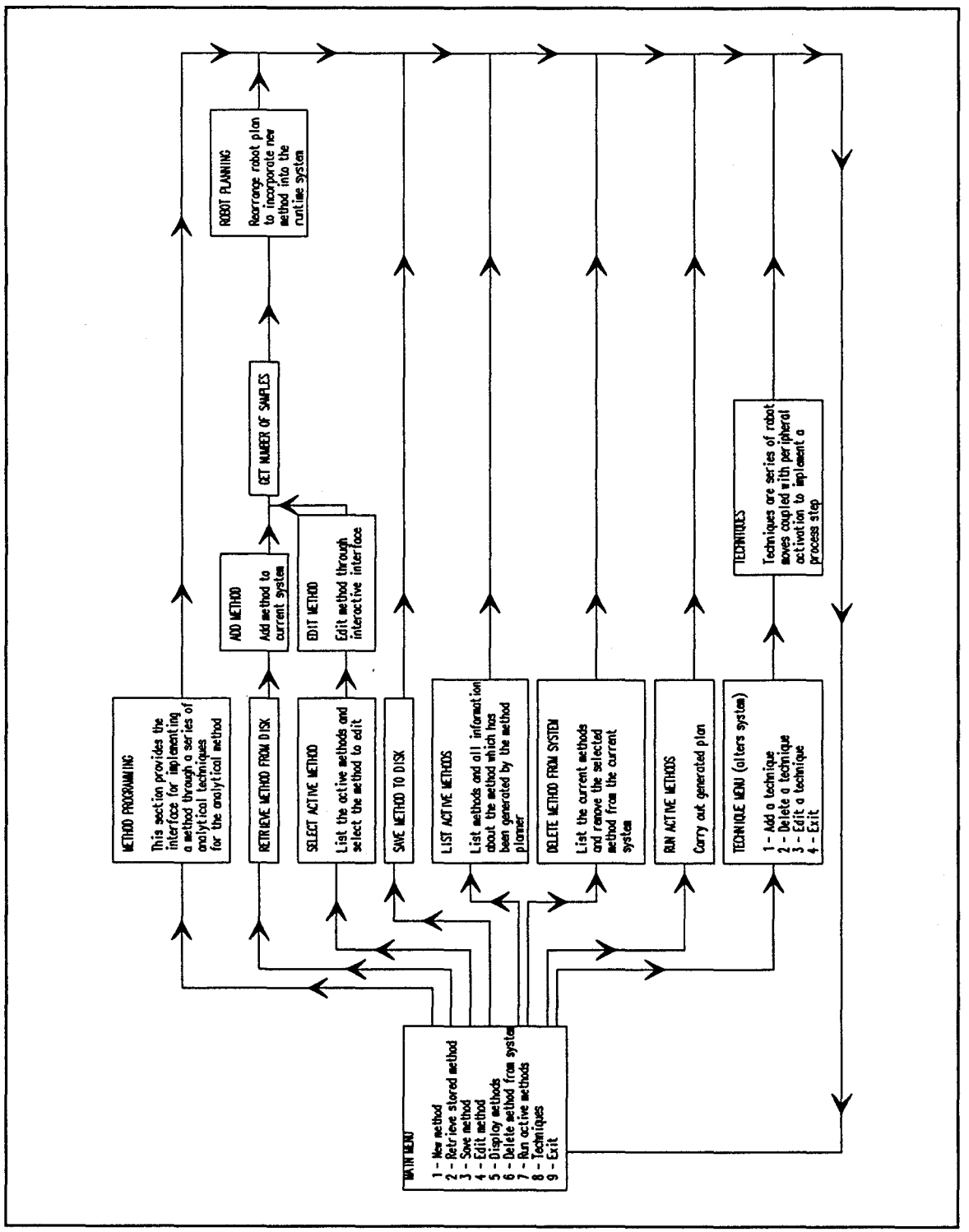

Figure 6.1 ASPS design specifications schematic

REFERENCES

[Biomatik 1983] Biomatik, PASRO - Pascal for Robots, Biomatik Company, Freiburg, West Germany (1983).

[Blume 1984] Blume, C. and Wilfried, J., "Design of the Structured Robot Language (SRL)", Advanced Software in Robotics, Proceedings of an International Meeting, Liege, Belgium, 1983, Elsevier Science Publishers, Amsterdam, The Netherlands (1984).

[Brodie 1987] Brodie, L., Starting Forth, Prentice-Hall, Englewood Cliffs, New Jersey (1987).

[Capek 1923] K. Capek, R.U.R., Doubleday, Page and Co., New York (1923).

[Coiffet 1983] Coiffet, P. and Chirouze, M., An Introduction to Robot Technology, McGraw-Hill Book Company, New York (1983)

[Cox 1989] Cox, Ingemar, J., Narain, Gehani, H., "Concurrent Programming and Robotics", The International Journal of Robotics Research, *8*, No. 2, (April 1989).

[CRS 1985a] CRS Plus, Small Industrial Robot System Technical Manual, Burlington, Ontario (1985).

[CRS 1985b] CRS Plus, Small Industrial Robot System RAPL Programming Language Manual, Burlington, Ontario (1985).

[Engle 1980] Engleberger, Robotics in Practice, AMACOM, A division of American Management Associates, New York (1980).

[Ernst 1961] H. A. Ernst, "A Computer Controlled Mechanical Hand", Sc.D. Thesis, Masachusetts Institute of Technology, Cambridge, MA (1961).

[Fairley 1985] Fairley, R., Software Engineering Concepts, McGraw-Hill, New York (1985).

[Finkel 1974] Finkel, R., Taylor, R., Bolles, R., Paul, R., and Feldman, J., "AL, A Programming System for Automation", Stanford AI Memo, *177*, Stanford University, Stanford, CA (January 1979).

[F-PC 1988a] F-PC User's Manual, San Jose, California (1988).

[F-PC 1988b] F-PC Reference Manual, San Jose, California (1988).

[Franklin 1982] Franklin, J., and Vandenburg, G., "Programming Vision and Robotics Systems with RAIL", SME Robots, *6* (March 1982), 392-406.

[GE 1982] GE, Allegro Documentation, General Electric Corporation, Schenectady, NY (1982).

[Gini 1985] Gini, G. and Gini, M., "Robot Languages in the Eighties, Robot Technology and Application", Proceeding of the 1st Robotics Europe Conference, Brussels, 1984, Spring Verlag IFS, Rubliatron, UK (1985).

[Goertz 1952] R.C. Goertz, "Fundamentals of General Purpose Remote Manipulators," Nucleonics *10*, (November 1952), 36-42.

[Halme 1987] Halme, A., Heikkila, T., "An Interractive Robot Control System", International Journal of Robotics and Automation, *2*, No. 3 (1987).

[Isenhour 1989] Isenhour, T. L. and Eckert, S. E., "Intelligent Robots - the Next Step in Laboratory Automation", American Chemical Society Analytical Journal of Chemistry, *61*, No. 13 (July 1, 1989).

[Korein 1987] Korein, J. U. and Ish-Shalom, J., "Robotics", IBM Systems Journal, *26*, No. 1 (1987).

[Krantz 1983] Krantz, J. and Willen, D., 8088 Assembler Language Programming, Howard W. Sams & Co. Inc., Indianapolis, Indiana (1983).

[Latombe 1981] Latombe, J., C., Mazer, E., "LM: A High-level Language for Controlling Assembly Robots", Eleventh International Symposium on Industrial Robots, Tokyo, Japan (October 1981), 683-690.

[Leznoff 1978] Leznoff, C. C., "The Use of Insoluble Polymer Supports in General Organic Synthesis", Journal of American Chemical Society, *100* (1978).

[McDonnel 1980] Mcdonnel Douglas, Robotic System for Aerospace Batch Manufacturing, McDonnel Douglas Inc., St. Louis, MO (February 1980).

[Nevins 1980] Nevins, J. L. and Whitney, D. E., "Assembly research", Automatica, *16* (1980).

[Parent 1985] Parent, M. and Laurgeau, C., Robot Technology (Logic and Programming), Prentice-Hall, Inc., N.J. (1985).

[Paul 1977] Paul, R. P., "WAVE, A Model Based Language for Manipulator Control", The Industrial Robot, *4*, No. 1, (1984), 25-39.

[Paul 1985] Paul, R. P. and Hayaward, V., "Robot Control and Computer Languages", Theory and Practice of Robots and Manipulators, Proceedings of RO Man Sy 1984, Kogan Page Ltd., Paris, France (1985).

[Povilonis 1988] Povilonis, E., "The Design and Development of an Automated Sample Preparation System", M. Engin. Thesis (Engineering Physics), McMaster University, Hamilton, Ontario (1988).

[Prosise 1989] Prosise, Jeff, "Lab Notes", PC Magazine, (September 26 1989), 307-316.

[Rosenfeld 1984a] Rosenfeld, J. M., "Macroreticular Resin XAD-2 as a Catalyst for the Simultaneous Extraction and Derivatization of Organic Acids from Water", Journal of Chromatography, *283* (1984), 127.

[Rosenfeld 1984b] Rosenfeld, J. M., McLeod, R. A., "Solid Supported Derivatizations in the Analysis of Cannabinoids", Proceedings of the Oxford Symposium on Cannabis, (1984), 151.

[Rosenfeld 1986] Rosenfeld, J. M., Hammerberg, O., "Simplified Methods for Preparation of Microbial Fatty Acids for Analysis by Gas Chromatography with Electron Capture Detection", Journal of Chromatography, *378* (1986), 9.

[Rosenfeld 1989] Rosenfeld, J. M., Moharir, Y., "Selective Sample Preparation for Determination of 11-Nor-9-carboxy-^9-tetrahydrocannabinol from Human Urine by Gas Chromatography with Electron Capture Detection", Journal of American Chemical Society, *61* (1989), 925.

[Shahinpoor 1987] Shahinpoor, M., A Robot Engineering Textbook, Harper & Row Publishers, New York (1987).

[Shimano 1984] Shimano, B., Geschke, C., Spalding, C., Smith, P., "A robot Programming System Incorporating Real-time and Supervisory Control", SME Robots, *8* (June 1984), 103-119.

[Tandy 1984] Tandy 1000 Programmer's Reference, Tandy Corporation and Microsoft Corporation (1984).

[Taylor 1982] Taylor, R., Summers, P., and Meyer, J., "AML: A Manufacturing Language", Robotics Research, *1*, No. 3 (Fall 1982), 19-41.

APPENDIX I

GLOSSARY OF FBRL WORDS

This appendix presents a list of FBRL words, the entry and exit stack conditions and a description of each word's function. The routines described in this appendix include the data structures queues, one and two-dimensional arrays, and strings, as well as the routines which handle the communications with the robot controller, the robot commands, input routines, metric conversion routines, RAPL implementation commands, and file handling routines.

I-1 QUEUES

This section provides a description of the Forth words used to implement the Queue data structure.

CREATE-Q (n -- | PAR: "q-name") queue-size(n) CREATE-Q q-name
Creates a queue structure which will hold n characters (bytes) named "q-name".

ENQ (c addr --) character(c) q-name ENQ
Adds a character (c) to the queue "q-name"

DEQ (Q-addr -- c) q-name DEQ
Removes a character from the queue and leaves it on the stack

EMPTY-Q? (addr-- fl) q-name EMPTY-Q?
Checks if the queue "q-name" is empty and returns a true or false on the stack

FULL-Q? (addr - fl) q-name FULL-Q?
Checks if the queue "q-name" is full and returns a true of false on the stack

The following words have been implemented in assembler and perform the same function as the corresponding high-level words:

Low-level Word	High-level Word
ENQ-L	ENQ
DEQ-L	DEQ
EMPTY-Q?-L	EMPTY-Q?
FULL-Q?-L	FULL-Q?

I-2 ARRAYS

This section provides a description of the Forth words used to implement the One-dimensional and two-dimensional array data structures.

ONE-DIMENSIONAL ARRAY

ARRAY (num-cells cell-size -- | PAR "array-name") m n ARRAY "array-name"
Creates an array "array-name", "num-cells" in length with a cell size of "cell-size" bytes. The runtime portion of this routine returns the absolute address of the cell given the array name and index.

TWO-DIMENSIONAL ARRAY

2ARRAY (rows cols c-size -- | PAR: "array") rows cols c-size 2ARRAY "array"
Creates an array "array" which is a matrix of "rows" rows and "cols" columns with a cell size of "c-size" bytes. The runtime portion of this routine returns the absolute address of the cell given the array name and indexes.

I-3 STRINGS

This section provides a description of the Forth words used to implement the String data structure and the string functions.

STRING (length -- | PAR: "string-name") length STRING "string-name"
Creates the string "string-name" which is "length" characters in length.

STRING-CNT (addr -- cnt) X STRING-CNT
Returns the number of characters in the string "X".

STRING-SIZE (addr -- size) X STRING-SIZE
Returns the number of characters reserved for the string "X".

TYPE-STRING (addr --) n STRING X
Prints the string "X" to the console.

APPEND-CHR (chr str-addr --) c n APPEND-CHR X
Appends the character c to the string X.

APPEND-STRING (targ-addr src-addr --) targ src APPEND-STRING
Appends the string "src" to the string "targ".

COPY-STRING (targ-addr src-addr --) targ src COPY-STRING
Copies string "src" to string "targ".

NULL-STRING (addr --) X NULL-STRING
Sets the string "X" to null.

SUB-STRING (targ-addr src-addr left right --) targ src m n SUB-STRING
Copies the substring in "src" from character m to character n into the string "targ".

STRING->CHR (addr pos -- chr) X n STRING->CHR
Returns the character at position n from string "X".

STRING-LIT (addr -- | PAR: "string") X STRING-LIT "ABC"
Stores the string literal "ABC" into the string "X".

GET-STRING (addr --) X GET-STRING
Receives a string from the console and stores it in string "X".

I-4 COMMUNICATIONS

This section provides a description of the Forth words used to implement the communications interface used for robot controller communications.

INTERNAL VARIABLES

AREG	two-byte variable containing the A register bit code for programming the UART
INCHAR	two-byte variable to hold the latest incoming character from the communications port
INTA00	two-byte variable to hold the interrupt vector segment
INTA01	two-byte variable to hold the interrupt vector offset
COMMINT-SEG	two-byte variable which contains the communications routine interrupt vector segment
COMMINT-OFF	two-byte variable which contains the communications routine interrupt vector offset
UART-ADD	two-byte variable which contains the UART address
XON-XOFF?	Flag for setting the XON/XOFF protocol (TRUE = on)
XOFF?	Flag for determining whether an XOFF code has been received
Q-CHAR	two-byte variable used to retrieve 1 character from the queue

WORDS

SETWORDLENGTH (num-bits --) 8 SETWORDLENGTH
Sets the word length for communications port setup. The example sets the word length to 8 bits.

SETSTOPBITS (numbits --) 1 SETSTOPBITS
Sets the number of stop bits for communications port setup. The example sets the number of stop bits to 1 bit.

SETPARITY (parity-char --) asc E SETPARITY
Sets the parity for communications port setup. The example sets the port to even parity.

SETBAUDRATE (baud-rate --) 2400 SETBAUDRATE
Sets the baud rate for communications port setup. The example sets the baud rate to 2400 baud.

COMMSET (baud parity stopbits wordlen --) 2400 asc E 1 8 COMMSET
Sets up the communications port. The example sets the communications to 2400 baud, even parity, 1 stop bit and 8 bit word length

SETUPUART (comm-port --) 1 SETUPUART
Sets up the specified communications port according to prespecified parameters. The examples sets COMM1 according to the specified parameters.

COMM-BASE-ADD (offset -- addr) offset COMM-BASE-ADD
Returns the base address of communications port specified at the offset

GET-UART (comm-port -- addr) 1 GET-UART
Returns the base address of the specified communications port. The example gets the base address for COM1.

WRITE-OFF
Turns off the transmitter empty interrupt

WRITE-ON
Turns on the transmitter empty interrupt

DIRECT-WRITE-COM (c --) asc A DIRECT-WRITE-COM
Writes the character "c" to the communications port. The example writes the character "A" to the current communications port.

READ-COM
Reads a character from the current communications port and stores the character in the input buffer.

WRITE-COM
Writes a character to the communications port from the output buffer.

COM
Handles communication port interrupts for both read and write functions.

START-WRITING-COM
Starts the interrupt driven communications write routine.

SAVE-VECTORS (n --)
Saves the interrupt vectors for the specified communications port number.

REVECTOR (n --)
Revectors the interrupt of the specified communications port to the FBRL interrupt routine.

COMM-INT-ENABLE
Unmasks the IRQ4 interrupts in the 8259's IRQ mask register and initializes the interrupt enable register.

RESET-INTERRUPTS (comm-port --)
Restores the interrupt system for the specified communications port.

EMIT-COM (char --) ASC E EMIT-COM
Emits a character to the serial port given the character on the stack.

TYPE-COM (chrs addr --) 24 TEST-STRING TYPE-COM
Types a string to the communications port given the number of characters in the string and the starting address of the string on the parameter stack.

.-COM (n --)
Transmits a 16 bit number from the parameter stack to the communications port.

F.-COM (F: r --) REAL-NUM F.-COM
Transmits a floating point number from the floating point stack to the communications port.

GET-COM (-- char)
Retrieves a character from the input buffer and returns it on the stack.

RESET-COMM (uart-address --) ADDR RESET-COM
Reads in all the backed up interrupts which have occurred before the routine was ready to accept them.

COMM-SETUP (pt# bd prty stop-bit wrdlen --) 1 2400 ASC E 1 8 COMM-SETUP
Sets up the specified communications port to the specified parameters and sets up the interrupt vectors to enable the communications interface.

INSTALL-INTERRUPT (addr in# --)
Installs the interrupt vector to the give interrupt number.

REMOVE-INTERRUPT (in# --)
Removes the interrupt vector from the interrupt number and installs a noop service routine.

I-5 ROBOT COMMANDS

This section contains the list of FBRL robot commands.

INTERNAL VARIABLES	TYPE
TEMP1	REAL
TEMP2	REAL

WORDS

APPROACH (loc -- | F: distance --)
Joint interpolated approach command.

APPROACH-STRT (loc -- | F: distance --)
Straight line approach command.

DEPART (F:distance)
Joint interpolated depart command.

DEPART-STRT (F:distance)
Straight line depart command.

MOV (loc --)
Joint interpolated move command.

MOV-STRT (loc --)
Straight line move command.

ACTUAL (PAR: location_name)
Reads the actual position of the arm into the given location template in the controller.

ALIGNTOOL
Quick align of the tool flange.

ARM (boolean --)
Sets the status of the arm power relay.

CLOSE-GRIP
Closes the pneumatic gripper.

CONFIG (dev baud parity #data #stop handshake echo)
Configures the controller serial ports.

FINISH
Finish the last motion command before starting the next command.

FLASH (interval --)
Turns the teach pendant flash light on.

NOFLASH
Turns the teach pendant flash light off.

GAIN (motor# -- | F:value)
Sets the response of the robot arm.

GRIP (torque -- | F: distance)
Opens and closes the gripper to the specified distance and torque.

HALT
Stops the current robot motion that is in progress.

ARM-HERE (loc --)
Returns the current robot position and stores it in the location template passed to it on the stack.

HOME
Initializes the robot position registers.

JOG (F: dX dY dZ --)
Moves the arm by the specified displacements.

JOINT (joint degrees --)
Drives a selected joint by a specified angular displacement.

LIMP (axis# --)
Limps the specified joint.

LOCK (axis# --)

Locks the specified joint.

UNLOCK (axis# --)
Unlocks the specified joint.

MA (F: j1 j2 j3 j4 j5 --)
Joint interpolated move to specified end locations.

MANUAL
Turns on the manual mode which allows the user to manipulate the robot with the teach pendant.

MI (F: j1 j2 j3 j4 j5 --)
Joint interpolated move by the specified incremental amounts in radians.

MOTOR (axis# #pulses --)
Drives the specified motor by the specified number of pulses.

NOHELP
Turns off the RAPL syntax building feature.

NOLIMP
Re-establishes closed loop servo control after a limp command has been issued.

NOMANUAL
Cancels the manual mode.

NOTEACH
Cancels the teach mode.

BASE-OFFSET (F: dX dY dZ dO --)
Sets an offset from the base of the arm.

SHIFT-RIGHT (n -- n b)
Shifts the value on the stack right by 1 bit and pushes the value of the bit onto the stack.

OUT2BYTE (n --)
Sends a 2 byte value to the specified output port.

OUTPORT (output# --)

Sets the specified output port.

READY
Moves the arm to the ready position.

SERIAL
Displays the controller serial port information.

SPEED (speed --)
Sets the speed of the arm.

ROBOT-STATUS
Displays the robot status.

TOOL (loc-template --)
Sets the tool transformation.

TYPEINT (string --)
Types a controller variable in integer format. Used by the Forth language for interaction with commands requiring variables.

TYPEREAL(string --)
Types a controller variable in real format. Used by the Forth language for interaction with commands requiring variables.

W0
Displays the current robot position in the motor, joint and world coordinate systems.

W1
Continually displays the actual robot position in motor coordinates.

W2
Displays the current actual robot position in the motor, joint and cartesian coordinates.

W3
Continually displays the commanded robot position in motor coordinates.

W4
Displays the current path end point in motor, joint and cartesian coordinates.

W5
Continually displays the robot velocity.

WE1
Continually displays the actual position of the extra axis.

WE3
Continually displays the command position of the extra axis in motor coordinates.

WE5
Continually displays the velocity commands to the extra axis.

WAITPORT (input# --)
Waits for the specified port to match the state.

WGRIP (string --)
Reads the value of the servo gripper and places it in the RAPL variable.

XCAL (axis# --)
Stores the home position for any extra axis.

XHOME (axis# --)
Homes a single extra axis.

XZERO (axis# --)
Zeros out the position register for the specified axis.

RAPL (PAR: string)
Send the specified string directly to the communications interface. This allows the user to send RAPL commands directly to the controller.

SET-RAPL-REAL (PAR: variable-name value)
Sets a variable in the controller to the specified floating point value.

SET-RAPL-INT (PAR: variable-name value)
Sets a variable in the controller to the specified integer value.

RAPL-ANALOG (PAR: input# variable-name)
Reads an analog input into the specified RAPL variable.

RAPL-HERE (PAR: template-name)

Reads the arm location into the RAPL location template.

RAPL-RUN (PAR: program-name)
Runs the specified RAPL program in the controller.

RAPL-SET (PAR: loc1 loc2)
Sets a location equal to another in the robot controller.

RAPL-SHIFTA (PAR: loc1 loc2)
Shifts a location by the location components in the robot controller.

RAPL-POINT (PAR: rapl-loc fbrl-loc)
Defines a point in the controller and changes it to the components specified by the FBRL location template.

I-6 INPUTS

This section contains the FBRL words used to interpret parameters as reals, integers or strings from a Forth parameter line.

INTERNAL VARIABLES

PARSTACK

WORDS

GETREAL
Interprets a floating point literal or variable from the TIB.

GETINT
Interprets a floating point literal or variable from the TIB.

GETSTRING
Reads a string from the TIB.

I-7 METRIC CONVERSION

This section contains the words that implement the metric conversion system for FBRL.

INTERNAL VARIABLES	TYPE
METRIC	INTEGER

WORDS

METRIC-ON
Turns on the metric flag.

METRIC-OFF
Turns off the metric flag.

METRIC?
Returns true if metric on and false if not.

CM->IN (F: cm -- in)
Converts a real value from centimetres to inches.

IN->CM (F:in -- cm)
Converts a real value from inches to centimetres.

METRIC-OUT (F: F -- F)
Checks the metric flag and returns the appropriate converted value.

METRIC-IN (F: F -- F)
Checks the metric flag and returns the appropriate converted value.

I-8 RAPL IMPLEMENTATION COMMANDS

The following is a list of words used to implement RAPL commands in FBRL.

WORDS

SEND-COM (command# --)
Sends a RAPL command number to the communications port.

SEND-REAL (F: real --)
Sends a floating point number to the communications port.

SEND-INT (n --)
Sends an integer to the communications port.

SEND-CR
Sends a carriage return to the communications port.

SEND-STRT
Sends the RAPL straight line sequence ",S" to the communications port.

SEND-LOC (loc-addr --)
Sends a location template to the communications port.

SEND-COMMA
Sends a comma to the communications port.

I-9 FILES

This section contains the FBRL words used for disk file manipulation.

INTERNAL VARIABLES	TYPE
TEMP-TIB	BUFFER 80 CHARACTERS
HNDL	HANDLE
TEMP->IN	INTEGER

WORDS

SAVE-TIB
Saves the type input buffer to TEMP-TIB.

RESTORE-TIB
Restores the type input buffer from TEMP-TIB.

CREATE-FILE (PAR: <filename>)
Creates a file using the specified name. If the file exists, it set zero length.

OPEN-FILE (PAR: <hndl filename>)
Opens a file for read or write given a handle and filename. If the handle is not defined, it is automatically created by the routine.

RESET-FILE (hndl --)
Sets an open file specified by hndl to the beginning of the file.

APPEND-FILE (hndl --)
Sets an open file specified by hndl to the end of the file for appending.

CLOSE-FILE (hndl -- return_code)
Closes the file specified by hndl. A DOS error code is returned on the stack.

WRITE-FILE (addr type hndl --)
Writes a variable to the file specified by hndl. The address and type of variable are required.

READ-FILE (addr type hndl --)
Reads a variable from the file specified by hndl. The address and type of variable are required.

APPENDIX II

THREE-DIMENSIONAL CALIBRATION

This appendix contains the implementation strategy and theory for the three-dimensional calibration as well as a list of FBRL words, the entry and exit stack conditions and a description of each word's function.

II-1 CALIBRATION ABSTRACTION

INTERNAL VARIABLES	TYPE
MAP-FLAG	INTEGER
MAP-VECTOR	INTEGER

CONSTANTS	VALUE
THETAAXYC	1
THETAAXZC	2
THETAXYR	3
THETAXZR	4
THETA3	5
XREC1	6
YREC1	7
ZREC1	8
XCAL1	9
YCAL1	10
ZCAL1	11
NUM-CAL-CELLS	11

WORDS

MAP-ON
Switches to "mapping on" status.

MAP-OFF
Switches to "mapping off" status.

MAPPING? (-- boolean)
Returns the status of the mapping switch -- true = mapping on.

WITH-CAL
Revectors the calibration vector to use the specified calibration template structure.

CREATE-CAL (PAR: cal-name)
Creates a calibration data structure which stores 3 location templates.
CAL-ADDR (cal-name cal-index -- loc-addr)
Retrieves the address of allocation in a calibration structure and leaves it on the parameter stack.

SET-CAL (cal-name cal-index loc-name --)
Sets the specified calibration location slot to the specified location.

GET-CAL (cal-name cal-index loc-name --)
Retrieves the specified calibration location slot to the specified location.

CREATE-CAL-STRU (PAR: cal-str-name)
Creates a named calibration structure.

CAL-STRU@ (cal-str-addr index -- | F: -- value)
Retrieves a calibration value.

CAL-STRU! (cal-str-addr index -- | F: value --)
Stores a calibration value.

GET-CAL-POINTS (cal-addr --)
Physically sets up a calibration structure.

II-2 CALIBRATION TRANSFORMATION

INTERNAL VARIABLES (ALL VARIABLES REAL)

X1
X2
Y1
Y2
DX
DY
ANG
COSANG
SINANG
MAP-LOC
UNMAP-LOC

CONSTANTS

PIDIV2 PI/2.0

WORDS

ANG-FROM-ORIGIN (F: X1 Y1 X2 Y2 -- angle)
Calculates the angle from the positive X axis given 2 sets of points which define the line in a plane. The angle is returned in radians.

ROTATE (F: X1 Y1 angle -- X2 Y2)
Rotates a point through the specified number of radians and returns the new point on the floating point stack.

CALIBRATE (recal-addr cal-addr cal-stru --)
Performs a 3 dimensional calibration of a real world system given 2 planes defined by 3 points each.

(MAP) (cal-stru loc-addr --)
Performs the runtime transformation (mapping) of a point from one plane to another in a 3 dimensional, real world coordinate system. The location is altered to reflect the new mapped location.

UNMAP (cal-stru loc-addr --)
Performs the runtime transformation of a point from one plane to another in a 3 dimensional, real world coordinate system in reverse. That is, the transformation is from the recalibration plane back to the base calibration plane.

II-3 THEORY AND IMPLEMENTATION STRATEGY FOR THREE-DIMENSIONAL CALIBRATION

CALIBRATION OF THE ORIGINAL PLANE

1) Get 3 calibration points

PC1 (XC1,YC1,ZC1) PC2(XC2,YC2,ZC2) PC3(XC3,YC3,ZC3)

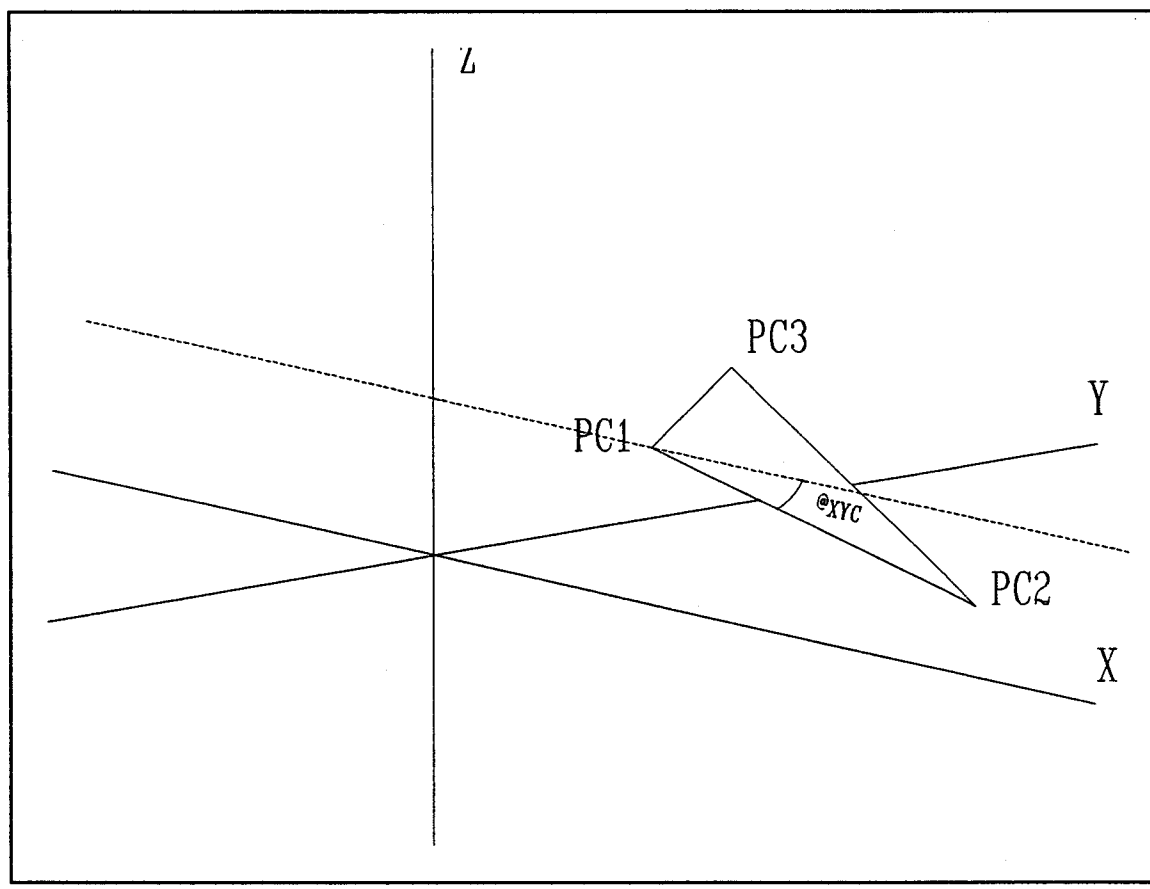

Figure II.1 The calibration plane.

2) Calculate the angle formed by the line PC1,PC2 and the X-axis in the XY plane

$$\theta XYC = \arctan\frac{YC2 - YC1}{XC2 - XC1}$$

3) Rotate line(PC1 , PC2) in the XY plane to the X-axis (Y=0)

$$PC4 = (XT, YT) = \text{ROTATE}((XC2 - XC1), (YC2 - YC1), -\theta XYC)$$

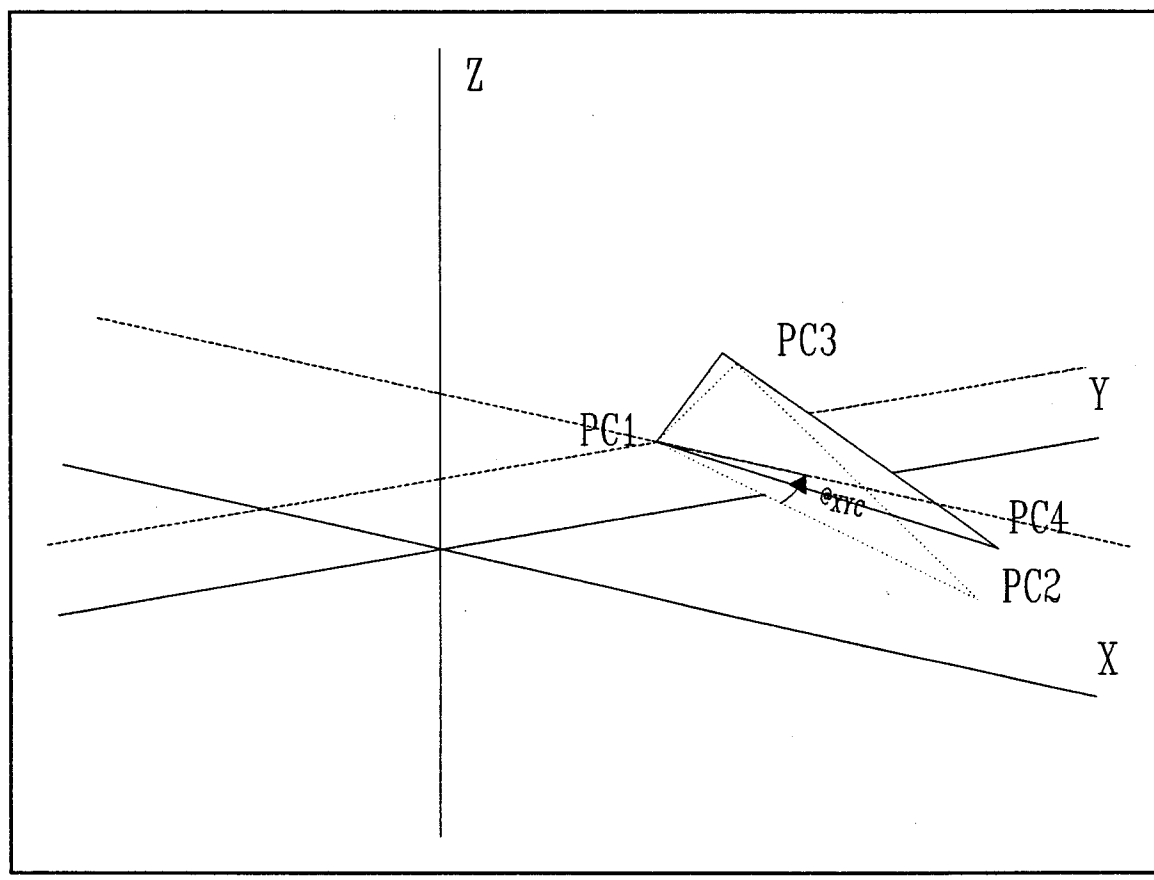

Figure II.2 Rotation of the calibration plane along -θXYC in the XY plane.

4) Calculate the angle formed by the line (Origin, PC4) and the X-axis in the XZ plane

$$\theta XZC = \arctan\frac{ZC2 - ZC1}{XT}$$

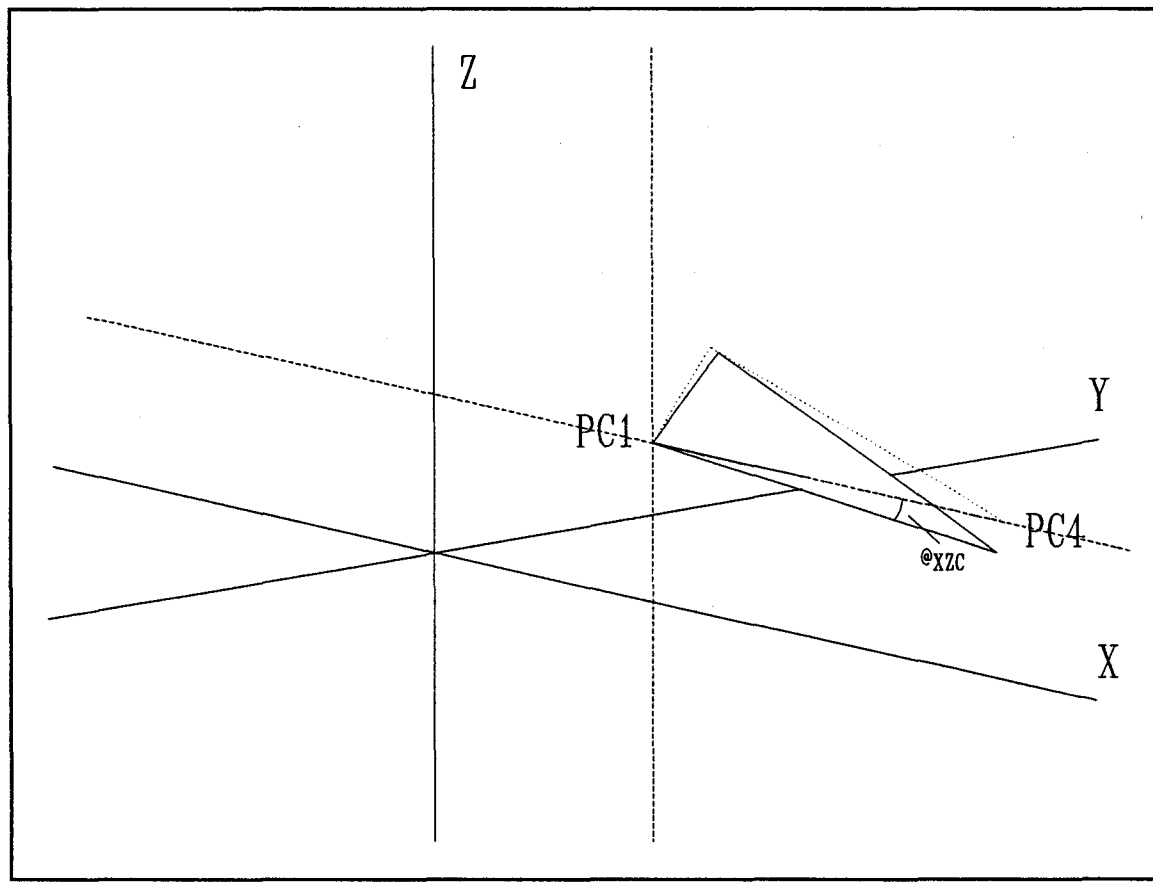

Figure II.3 Rotation of calibration plane along θXZC in the XZ plane.

5) These intermediate points are stored in the calibrate structure for use in the recalibration routines

$$\theta XYC, \quad \Theta XZC$$

The above steps set up the original plane for recalibration in the future. The next step is the recalibration. This step involves acquiring the 3 calibration points and calculating the information necessary for the mapping of the calibration plane onto the recalibration plane.

CALIBRATION OF THE NEW PLANE

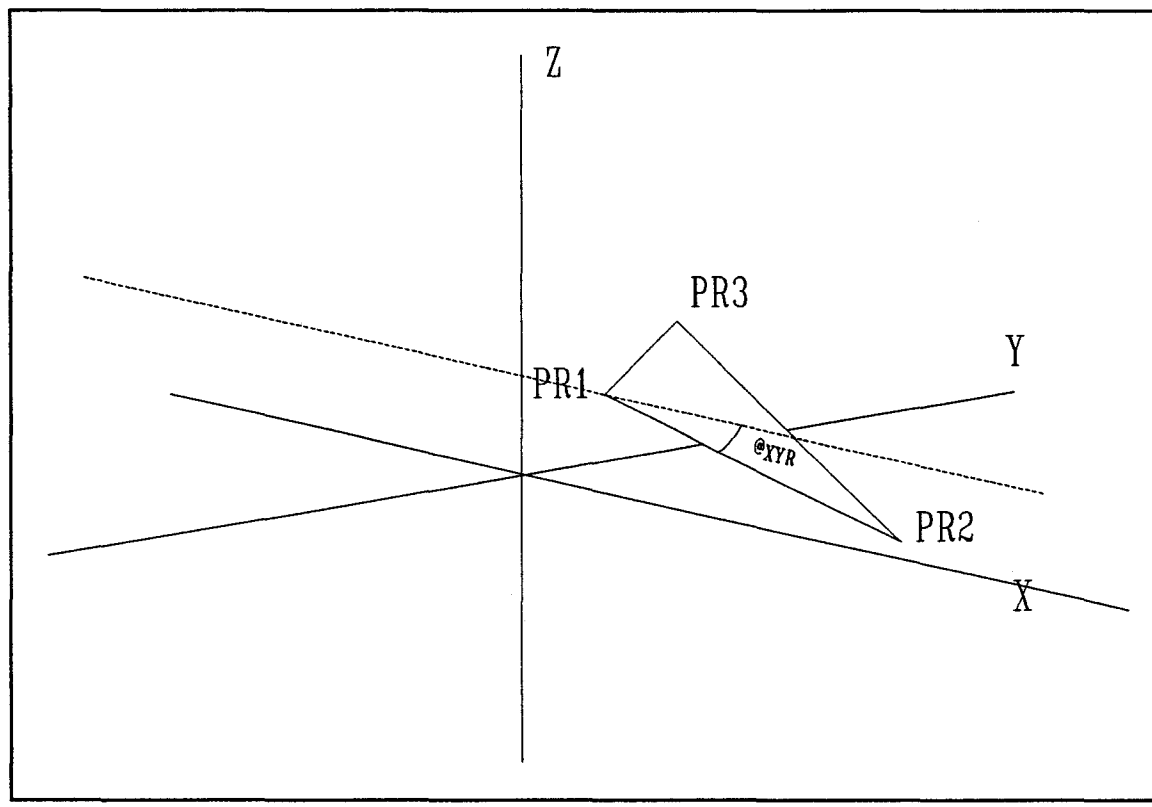

Figure II.4 New plane which is to be calibrated from calibration plane.

1) Get 3 recalibration points
PR1 (XR1,YR1,ZR1)
PR2(XR2,YR2,ZR2)
PR3(XR3,YR3,ZR3)

2) Calculate the angle formed by the line PR1,PR2 and the X-axis in the XY plane

$$\theta XYR = \arctan\frac{YR2-YR1}{XR2-XR1}$$

3) Rotate line(PR1 , PR2) in the XY plane to the X-axis (Y=0)

$$PR4\text{-}(XT,YT)\text{-}\mathrm{ROTATE}((XR2\text{-}XR1),(YR2\text{-}YR1),\text{-}\theta XYR)$$

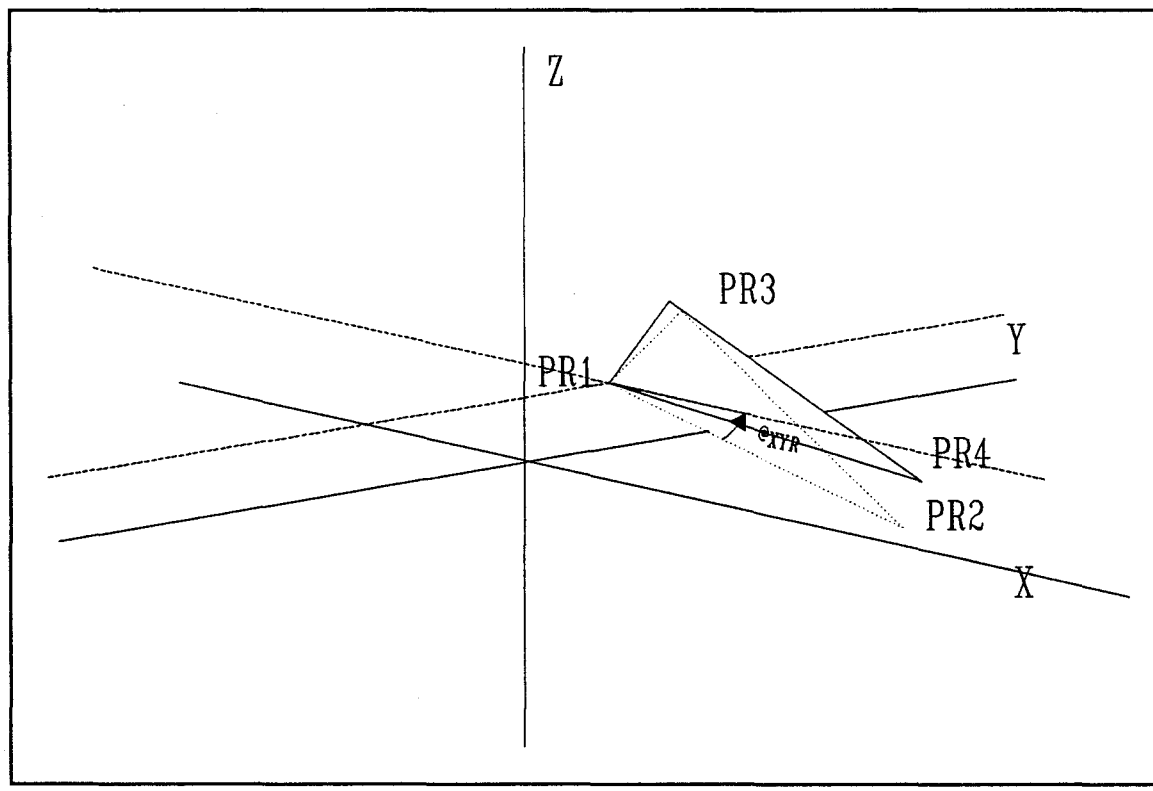

Figure II.5 Rotation of the recalibration plane through -θXYR in the XY plane.

4) Calculate the angle formed by the line (Origin, PR4) and the X-axis in the XZ plane

$$\theta XZR\text{-}\arctan\frac{ZR2-ZR1}{XT}$$

5) Rotate the calibration third point PC3 to the X-axis in the XY plane (Z=0)

$$(XCT1,YCT1)\text{-ROTATE}((XC3,XC1),(YC3,YC1),-\theta XYC)$$

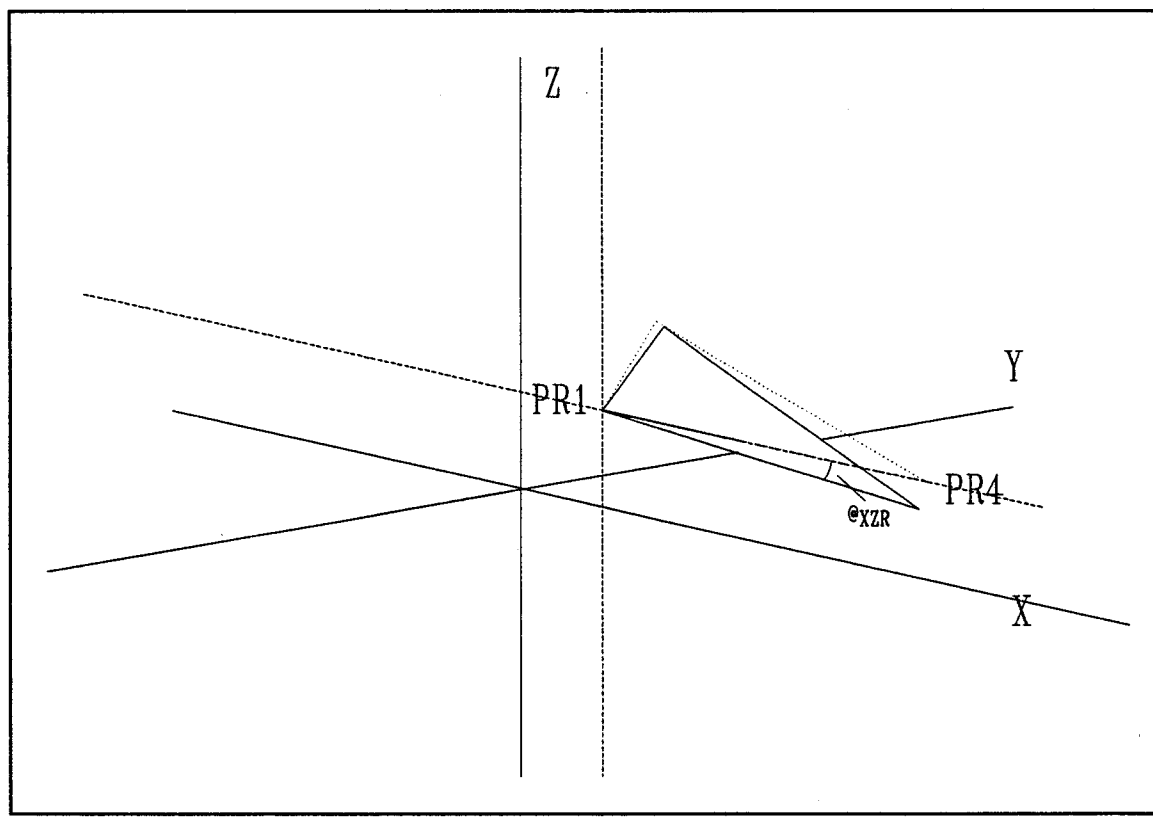

Figure II.6 Rotation of the recalibration plane through -θXZR in the XZ plane.

in the XZ plane (Y=0)

$$(XCT,ZCT)\text{-ROTATE}((0,XCT1),(ZC3,ZC1),-\theta XZC)$$

6) Rotate the recalibration third point PR3 to the X-axis in the XY plane (Z=0)

$$(XRT1,YRT)\text{-ROTATE}((XR3,XR1),(YR3,YR1),-\theta XYR)$$

in the XZ plane (Y=0)

$$(XRT,ZRT)\text{-ROTATE}((0,XRT1),(ZR3,ZR1),-\theta XZR)$$

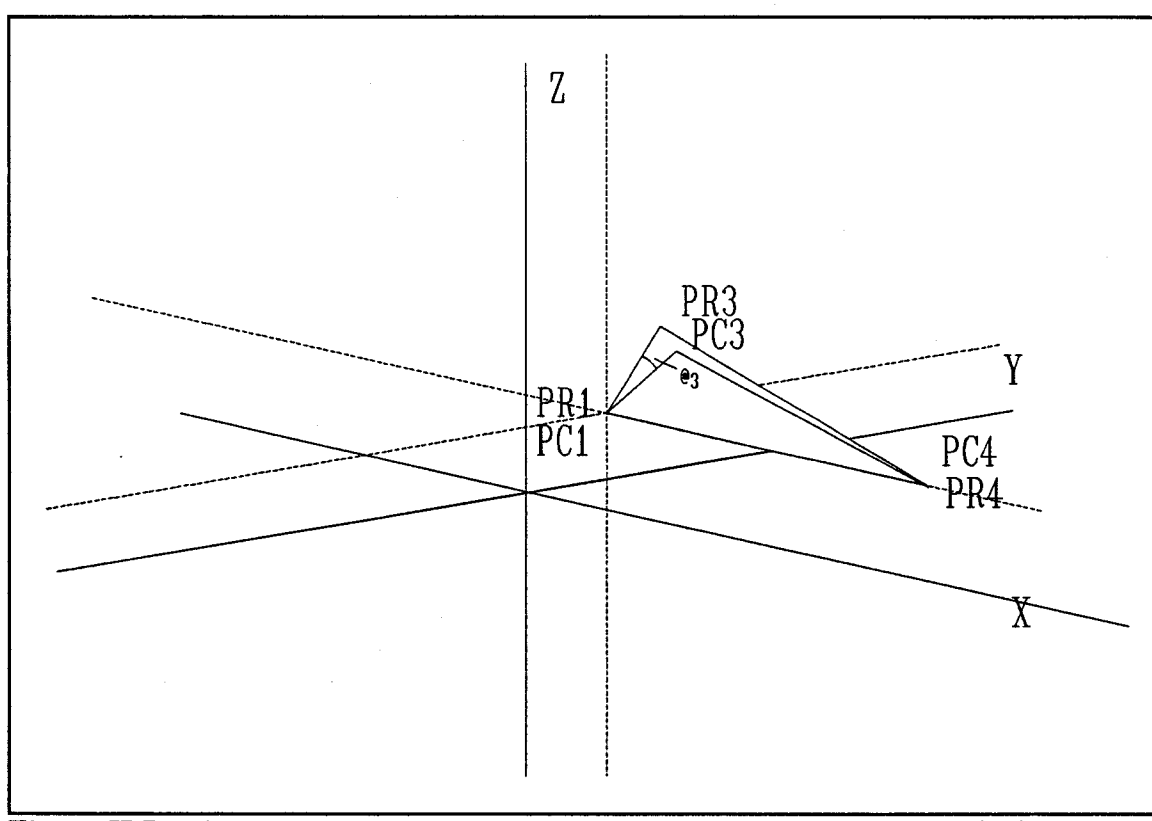

Figure II.7 Angle between the calibration and recalibration planes (θ3).

7) Calculate the angle between the Calibration and recalibration plane

$$\theta R\text{-ANGLEFROMORIGIN((0,0),(}YRT,ZRT\text{))}$$

$$\Theta C\text{-ANGLEFROMORIGIN((0,0),(}YCT,ZCT\text{))}$$

$$\Theta 3\text{-}\Theta R\text{-}\Theta C$$

MAPPING

Maps a point from the calibration plane to the recalibration plane

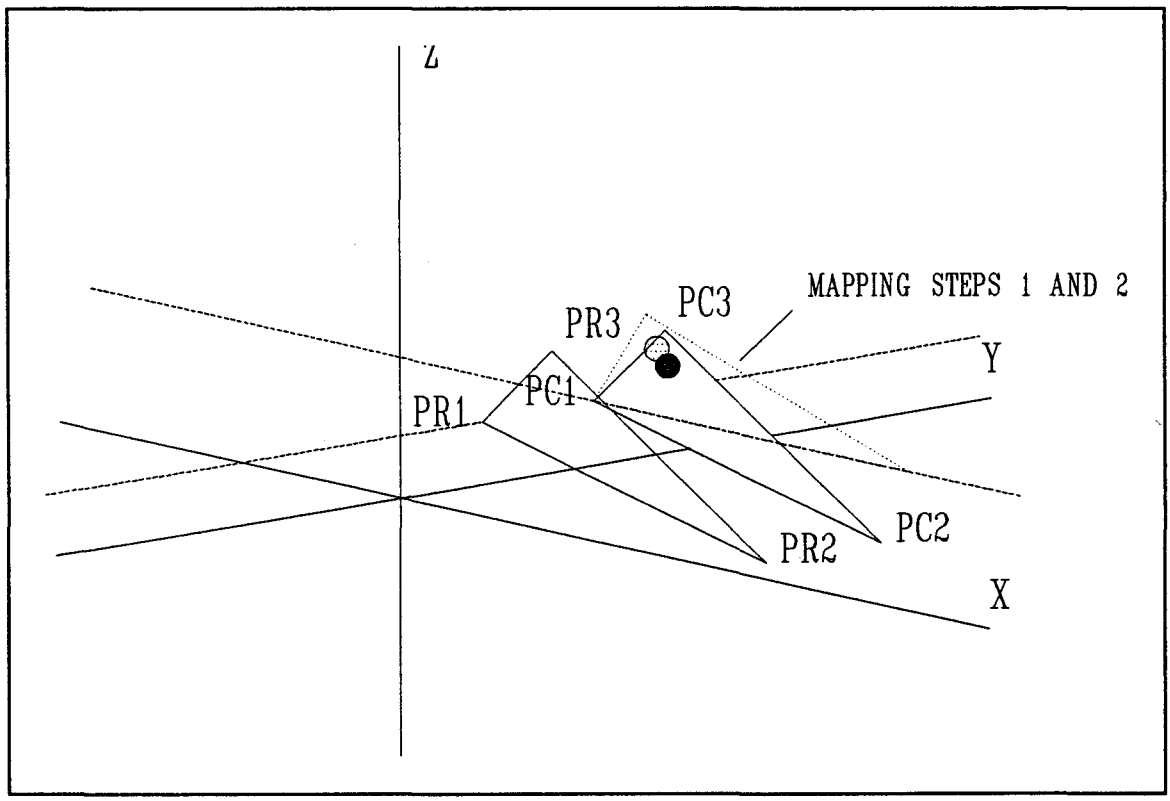

Figure II.8 Mapping steps 1 and 2.

1) Rotate the point from the calibration plane to the X-axis

in the XY plane

$$(XM1,YM1)\text{-ROTATE}((XM,XC1),(YM,YC1),-\Theta XYC)$$

in the XZ plane

$$(XM2,ZM2)\text{-ROTATE}((0,XM1),(ZM,ZC1),-\Theta XZC)$$

2) Rotate the third point in the YZ plane

$$(YM2,ZM2)\text{-ROTATE}((0,YM1),(0,ZM1),\Theta 3)$$

3) Rotate back along recalibration angles to achieve the recalibration plane in the XZ plane

$$(XM3,ZM3)\text{-ROTATE}((0,XM2),(0,ZM2),\Theta XZR)$$

in the XY plane

$$(XM4,YM3)\text{-ROTATE}((0,XM3),(0,YM2),\Theta XYR)$$

4) Perform translation back to origin of recalibration plane (PR1)

$$XM\text{-}XM4\text{+}XR3$$

$$YM\text{-}YM3\text{+}YR1$$

$$ZM\text{-}ZM3\text{+}ZR1$$

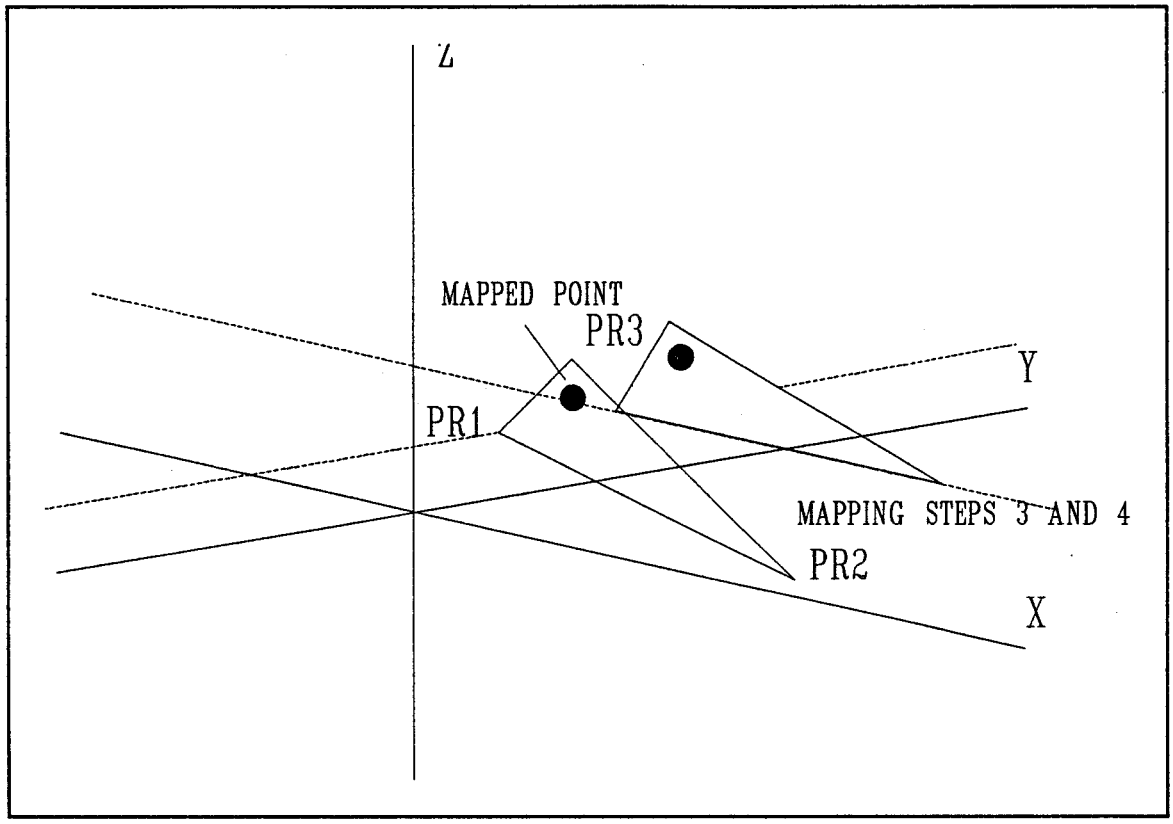

Figure II.9 Mapping steps 3 and 4.

Mapped Point-(XM,YM,ZM)

UNMAPPING

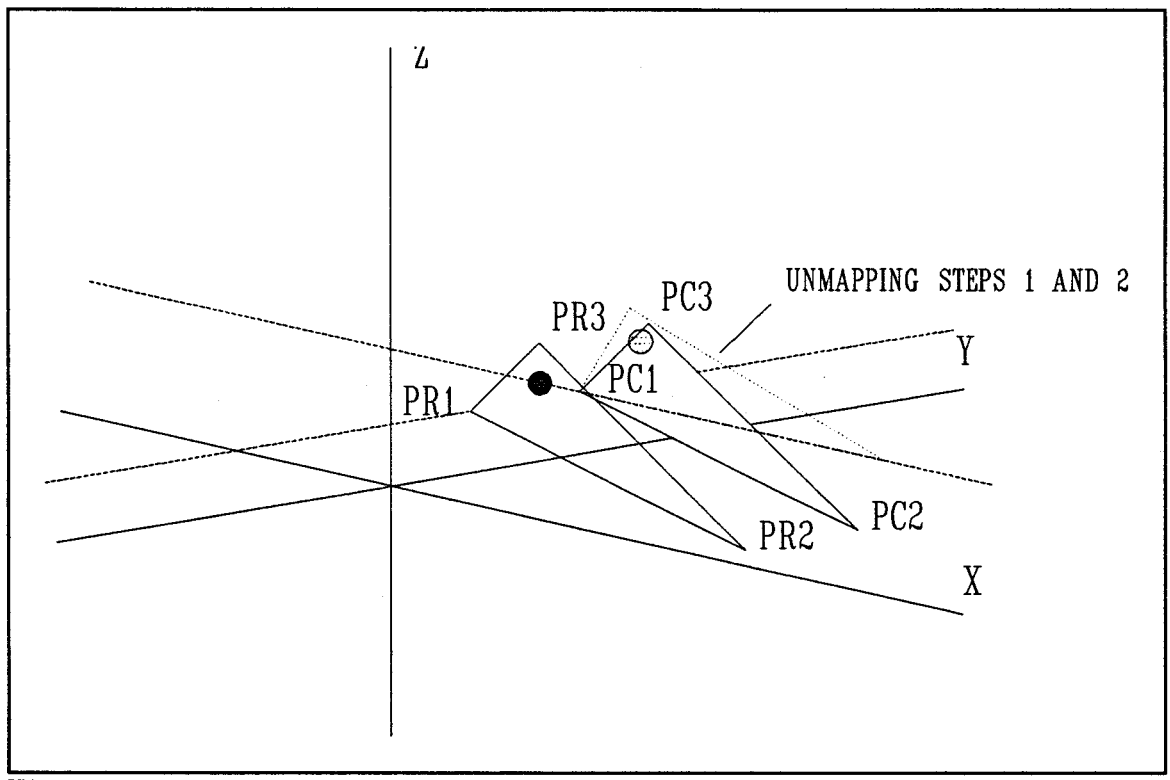

Figure II.10 Unmapping steps 1 and 2.

Maps a point from the recalibration plane to the calibration plane

1) Translate to origin from recalibration plane

$$XM1 - XM - XR1$$

$$YM1 - YM - YR1$$

$$ZM1 - ZM - ZR1$$

2) Rotate along recalibration angles to x-axis
in the XZ plane

$$(XM2,ZM2)\text{-ROTATE}((0,XM1),(0,ZM1),-\Theta XZR)$$

in the XY plane

$$(XM3,YM2)\text{-ROTATE}((0,XM2),(0,YM1),-\Theta XYR)$$

3) Rotate third point in the YZ plane

$$(YM3,ZM3)\text{-ROTATE}((0,YM2),(0,ZM2),-\Theta 3)$$

4) Rotate from the X-axis to the calibration plane
in the XY plane

$$(XM4,YM4)\text{-ROTATE}((0,XM3),(0,YM3),\Theta XYC)$$

in the XZ plane

$$(XM5,ZM5)\text{-ROTATE}((0,XM4),(0,ZM3),\Theta XZC)$$

5) Translate origin to calibration origin (PC1)

$$XM-XM5+XC1$$

$$YM-YM4+YC1$$

$$ZM-ZM4+ZC1$$

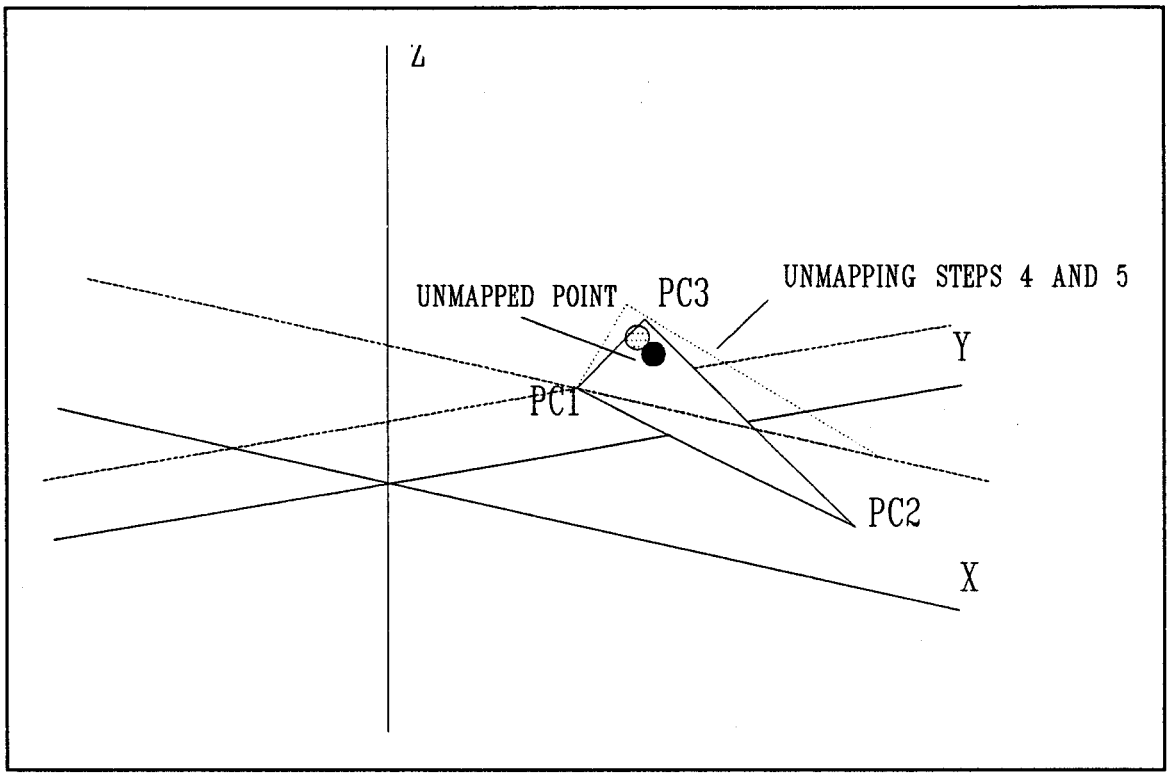

Figure II.11 Unmapping steps 3, 4 and 5.

unmappedpoint-(XM,YM,ZM)

The ROTATE function has been used throughout the preceding text. This function rotates a point with the X-axis as the base point through theta degrees.

FUNCTION ROTATE(X,Y,Θ)

Rotates the point (X,Y) through θ degrees

$$Yrotate = X \times \sin(\theta) + Y \times \cos(\theta)$$

$$Xrotate = X \times \cos(\theta) - Y \times \sin(\theta)$$

APPENDIX III

ROBOT SPECIFIC DATA STRUCTURES AND ABSTRACTIONS

This appendix contains a list of FBRL words, the entry and exit stack conditions and a description of each word's function for the robotic data structures and abstractions.

III-1 LOCATION TEMPLATES

This section contains the words for routines associated with location templates.

VARIABLES	TYPE
TEMP-LOC	LOCATION TEMPLATE

WORDS

SETX (loc-addr -- | F: x --)
Sets the X component of the location.

SETY (loc-addr -- | F: y --)
Sets the Y component of the location.

SETZ (loc-addr -- | F: Z --)
Sets the Z component of the location.

SETO (loc-addr -- | F: O --)
Sets the O component of the location.

SETA (loc-addr -- | F: A --)
Sets the A component of the location.

SETT (loc-addr -- | F: T --)
Sets the T component of the location.

GETX (loc-addr -- | F: -- x)
Gets the X component of the location and stores it on the floating point stack.

GETY (loc-addr -- | F: -- y)
Gets the Y component of the location and stores it on the floating point stack.

GETZ (loc-addr -- | F: -- Z)
Gets the Z component of the location and stores it on the floating point stack.

GETO (loc-addr -- | F: -- O)
Gets the O component of the location and stores it on the floating point stack.

GETA (loc-addr -- | F: -- A)
Gets the A component of the location and stores it on the floating point stack.

GETT (loc-addr -- | F: -- T)
Gets the T component of the location and stores it on the floating point stack.

SET-LOC (loc-address1 loc-address2 --)
Sets all six components of the location variable by copying the contents of location 2 into location 1.

DEFINE-LOC (loc-addr -- | F: dX dY dZ dO dA dT --)
Sets all six components of the location template to the specified values.

CREATE-LOC (PAR: location name)
Creates a location template structure.

SHIFTA (loc-addr1 loc-addr2 --)
Shifts all six components of the location 1 variable by the amount specified in location 2.

PRINT-LOC (loc-addr --)
Prints all six components of a location to the communication port - calibration mapping and metric conversion is included.

DISPLAY-LOC (loc-addr --)
Prints all six components of a location to the screen - calibration mapping and metric conversion is included.

III-2 LOCATION MATRIX

This section contains the words associated with the Radial and Rectangular location matrix data abstractions.

RADIAL MATRIX

RADMAT (base-loc-address -- | F: inc-angle ring-dist)
Creates a radial matrix structure for holding locations - the runtime portion calculates a new location given the radial ring number and the radial index.

SHOW-RADMAT (radial-matrix-address --)
Displays the contents of a radial matrix abstraction.

RECTANGULAR MATRIX

AUTOMAT (base-loc-address -- | F: x-space y-space z-space --)
Creates an automated matrix structure for holding locations - the runtime portion calculates a location given the matrix indexes.

SHOW-AUTOMAT (matrix-address --)
Displays the contents of an auto-matrix abstraction.