STUDIES IN ROBOT PROGRAMMING

Balbir Singh Bal

Doctor of Philosophy

UNIVERSITY OF ASTON IN BIRMINGHAM

September 1990

DEDICATION

To the loving memory of my

Father, **Mr. Ajit Singh**

and

Mother, **Mrs. Amar Kaur**

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and thanks to the following people:

Dr. B. Gay, Head, Department of Computer Science and Applied Mathematics, who directed this study, for his invaluable encouragement, guidance and support during this research.

My beloved wife, Jaspal, to whom I am deeply indebted for typing this thesis and constantly offering tremendous encouragement, support and co-operation without which this work could not been successfully completed.

Mr. A. Cole, a colleague at Sandwell College of Further and Higher Education for his excellent help and invaluable suggestions in the final preparation of this thesis.

Mr. A. Low, Head, School of Computing , Sandwell College of Further and Higher Education, for his encouragement and understanding.

Sarbjit Singh, a research colleague and friend, for offering valuable help and suggestions in the preparation of the thesis.

The technical staff of the Department; in particular Mr. A. Sutton, for developing the hardware needed in the project.

Sandwell Metropolitan Borough Council for financing part of this research

Finally, I am deeply grateful to my son Andeep and daughter Rubinder, for showing immense patience and understanding during these difficult years.

contents **page**

figures..page

tables.. page

CHAPTER-- 1 INTRODUCTION

1.1 HISTORICAL BACKGROUND

The word robot was used by a Czechoslovakian playwright Karel Capek in his play entitled "R.U.R." In Slav. languages, the word robot indicates a machine that resembles human workers. Initially, the robots were manufactured as slaves to replace human workers but towards the end, they turned against their creators and anhilated the entire human race. Since 1926, robots appeared in films such as METROPOLIS and STAR WARS have given people a conception of robots as humans like machines with intelligence and individual personalities. Some of these imaginative expectations of robots created in such films, are much more advanced than are the actual robots which are available at present time.

The definition of robot is vague, since it depends upon the human perception of them. Webster's dictionary defines a robot as :---

"an automatic device that performs the functions ordinarily ascribed to human beings ". Such a definition is far too broad since devices like washing machines and dishwashers may also be considered as robots.

The concise Oxford dictionary has defined robot as:---

" apparently human automation, intelligent and obedient but impersonal machine". Clearly this definition does not accurately define present day robot nor does it differentiate from other types of automation. The Robot Institute of America has defined industrial robot like this, "a robot is programmable multi-functional manipulator designed to move materials, parts, tools, or specialized devices, through variable programmed motions for the performance of a variety of tasks."

This definition appears to imply that the robot is a general purpose device with external sensors to provide some intelligence to the machine to perform various tasks. The term manipulator is not defined. It is not clear whether a single axis mechanism qualifies as a manipulator. The Japanese industrial robot association (JIRA) in 1980 has defined robot as:---

"versatile, flexible mechanisms with displacement functions similar to those of human limbs or with displacement functions controlled by sensors and means of recognition."

Even this definition, appears to be vague and includes maximal effort manipulators and remote control manipulators. Finally The British Robot Association has defined robot as:-

"an industrial robot is a reprogrammable device designed to manipulate and transport parts, tools or specialized manufacturing implements through variable programmed motions for the performance of specific manufacturing tasks."

From the various definitions of the robot, it is evident that there is no internationally agreed definition of a robot. However, robots still have features which distinguish them from hard automation. Whatever our perception of robots is, they must be capable of being :---

(a) Reprogrammable and hence will be computer controlled.

(b) Able to show multiple movements in three dimensional space i.e several degrees of freedom.

(c) Provided with external sensors, if they require interaction with the environment.

(d) Able to support different grippers.

(e) Able to take decisions by comparing data received from many sensors and reacting accordingly in a preprogrammed way

The development of robots has occured along two paths: domestic and industrial [STAUGAARD, Jr. 1987]. Domestic or personal robots have been primarily developed for the home hobbyist market. These robots can have features like voice synthesis (speaking), sensing light levels, detecting motion, moving around and sonar type navigation systems. Industrial robots have been developed to perform various manufacturing tasks such as welding, spray painting, pick and place and assembly operations. Most of the present robots still have very limited sensing ability, consequently they cannot alter their sequence of operations when obstructions or unexpected circumstances arises.

Like all other fields, development of robot technology has occured in distinct phases or generations. The first generation robots were able to repeat certain fixed sequence of operations. They were like 'dumb' robots as they did not have any idea of the robot environment, whether an object to be grasped is present or not. These robots are more commonly known as pick and place robots. They were mainly used in welding, paint spray and simple assembly operations. These robots were mainly programmed by limit switches, resetting stops on indexable drum or cams.

Second generation robots may be called clever robots. They are equipped with range of sensors and are controlled by computers. The integration of sensors in the control of the robot confers an ability to interact with their environment. The main method of programming these robots is by writing textual program using high level languages.

The third generation robots are referred to as intelligent (SMART) robots. These robots are still in the research stage. They make extensive use of data bases, sensors, complex software and Artificial Intelligence (AI) techniques. The aim is to give robots an instruction similar to a human worker. The robot will automatically plan the sequence of operation, trajectory, collision avoidance etc. and will perform the desired task automatically. This is still an active area of investigation and there are numerous problems which must be resolved before robots are able to perform jobs like human beings.

The early development of robots is summarised in the following table:

YEAR	REFERENCE	NAME	COMMENTS
1945	Goetz 1963	Teleoperator	handle radioactive substances
1948	Goetz 1963	Handyman	Teleoperator by GE
1959	FU et al. 1987	-----	First industrial robot by Unimation
1961	Ernst 1962	MH-1	Computer controlled mechanical hand
1962	Tomoric and Boni 1962	--	Hand with pressure sensor
1963	FU et al. 1987	Verstran	Commercial robot by AMFC
1968	Mc Carthy 1968	--	Robot with hand, eyes and ears
1972	Bolles and Paul 1973	---	Assembled water pump
1972	Ejiri et al. 1972	--	Assembled machinery from drawing
1973	Will 1975	---	IBM arm
1974	FU et al. 1987	T3	Computer controlled robot by Cincinnati Milacron
1978	Faverto 1978	---	Polar 6000 for welding
1979	Borrowman 1979	RMS	Duplicate of human arm for NASA

Table 1.1 Early developments of robots

Currently, there are numerous industrial robots available in the market which are almost invariably computer controlled. They vary in size, shape, joint type and driving mechanism and are described in chapter 2.

1.2 ROBOTICS

With lowering cost of computer hardware, robots have attracted considerable interest within industrial and academic research establishments which has led to the development of a completely new field known as robotics. It is a highly interdisciplinary field that ranges from the design of mechanical and electrical components to sensor technology, computer systems and artificial intelligence. Robotics is a generic term which embraces the following main subject areas :

(a) Robot arm kinematics

(b) Robot arm dynamics

(c) Planning of trajectories

(d) Control of robot manipulators

(e) Robot sensors

(f) Robot programming languages

(g) Robot intelligence and task planning

Since the field is so vast, it is beyond the scope of this work to describe each of these subjects in detail. Robot programming languages (f) is one of the most active areas of research in robotics and will be considered in detail in this dissertation.

1.3 ROBOT PROGRAMMING

The robot programming provides the essential interface between the robot and human operator. Robot programming requires the use of a programming language. Methods of robot programming and robot programming languages are discussed in chapter 3. There are numerous programming languages some of which are new and others are modified versions of existing languages. General and specific requirements of robot programming languages are described in chapter 4. Two extreme approaches in robot programming can be identified. On one hand, there are low level programming languages which requires the user to specify every minute detail in programming and majority of them suffer from a lack of portability since they were devised specifically for a particular robot type. At the other extreme, are task-oriented programming languages/systems which are mainly used by AI researchers. This type of programming is very complex, relies heavily on sensor feedback, the use of expert systems and is very active area of research, as yet only in its infancy. No commercially available programming systems are available at this level. Clearly, there is a huge gap between the two approaches. For robots to become more accessible to industry in the near future, they must be easy to program, preferably in high level language perhaps with limited number of sensors. Other ideal features of the language include portability, interactivity, extensibility, modularity and commercial availability. Such a language should be designed to cater the needs of a non-specialist programmer. In order to bridge the gap between these two extreme programming strategies, it was decided to investigate a hybrid approach towards the programming of a robot which constituted the original contribution to the use of Forth for controlling a robot.

It was decided to use Forth as a programming language as it was specifically written for process control applications. The language and those features which render it an ideal language for robot programming are described and discussed in chapter 5. Forth is extensible and allows the user to define data structures suitable for specific application.

The programming methodology of Forth allows the user to define very abstract levels of command using English like syntax appropriate to task execution. For example, to pick up an object in Forth the word may be defined as follows:

```
object1  GRAB
```

The user is provided with a vocabulary of task-oriented commands for various purposes. The user is not expected to learn language or write any program in formal sense because all the information and code is well hidden from the user. What the user is expected is to learn is which command to use and Reverse Polish Notation (RPN). For example, to move shoulder, the command would be :

```
125   shoulder   move
```

It was also decided to extend the data structure of Forth to develop software tools to facilitate robot programming both for the ease of understanding and for programming purposes. Forth uses reverse Polish notation and is a stack oriented language. This feature makes it suitable for developing data structures because actual data is not required at the time of defining new data structures. A development system consisting of a Motorola 6809 microcomputer, Smart arm, simulator and ultrasonic transducer is described in chapter 6. The development of Forth software along with a method of calibration of robot joints and a solution to the inverse kinematic problem for Smart arm is described in chapter 7. The conclusion and recommendations from the studies are described in detail in chapter 8.

CHAPTER -- 2 ROBOT ARM AND ROBOTIC COMPONENTS

2.1 ROBOT ARM

A robot arm, like the human arm, is composed of links held together by joints. A joint is the part of the arm which connects two adjacent links and allows relative motion between them as shown in figure 2.1 below :

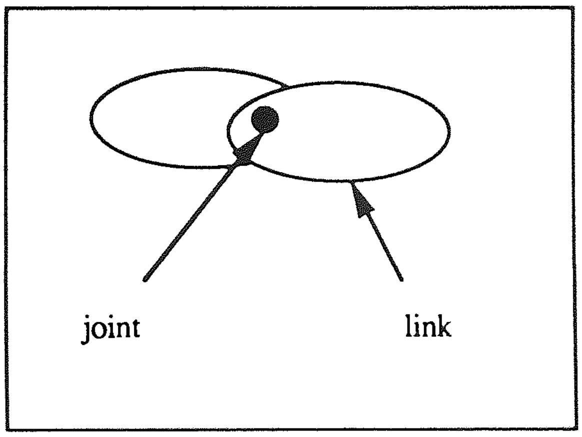

Figure 2.1 showing relationship between links and joints.

Thus a robot arm can be considered to be an articulate chain of links connected in series by suitable joints. Each robotic arm has a base, which is usually attached to the fixed surface, or floor. The first link of the arm is attached to the base and other links are in turn attached to the next link by a joint. The last link is relatively free and is commonly attached to a specialised tool which is more commonly known as hand or end-effector or gripper. The joint motions of a robot are generated by actuators that are either hydraulic, pneumatic or electrical. These actuators are either active at the joints or produce an action at a distance which is transmitted by belts, pulleys, and chains etc.

2.2 TYPES OF JOINTS

Most common types of joints are revolute and prismatic and less common one is known as ball and socket.

2.2.1 PRISMATIC JOINT

This type of connection allows the prismatic or linear motion between the two adjacent links, as shown in figure 2.2 below :

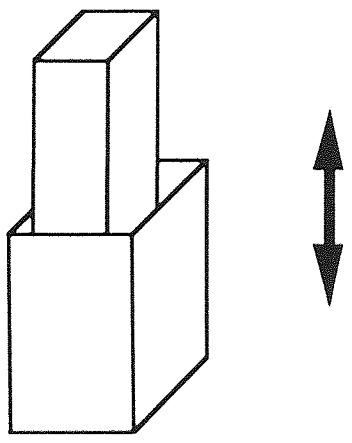

Figure 2.2 showing prismatic joint

It is composed of two nested links, or of one link sliding along another. In other words, one part can move (slide) on a line straight outward or inward in relation to the other part.

2.2.2 REVOLUTE JOINT

This connection permits revolute, or rotary, motion between two adjacent links. The two links are joined by a common hinge, so that one part can move in a swinging motion in relation to the other part, as shown in figure 2.3 below :

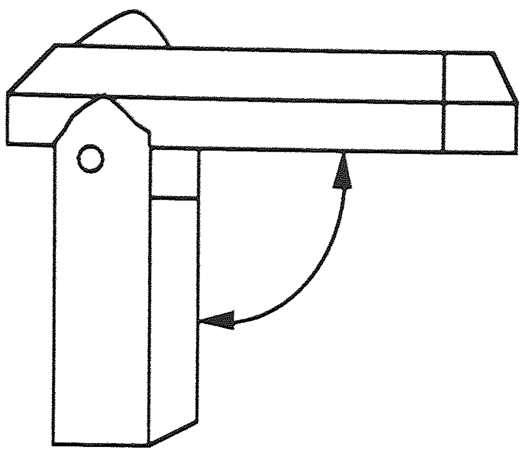

Figure 2.3 showing revolute joint.

This joint rotates along in a single axis.

2.2.3 BALL AND SOCKET JOINT

This type of connection is composed of more than one joint and enable motion in more than one axis as shown in figure 2.4. This type of joint exist between shoulder and fore-arm and pelvis and thigh. This type of joint is not commonly used in robots for difficulty of activating joints of this type. However, many robots include 3 separate revolute joints whose axis of motion intersect at one point as shown in figure 2.4 below :

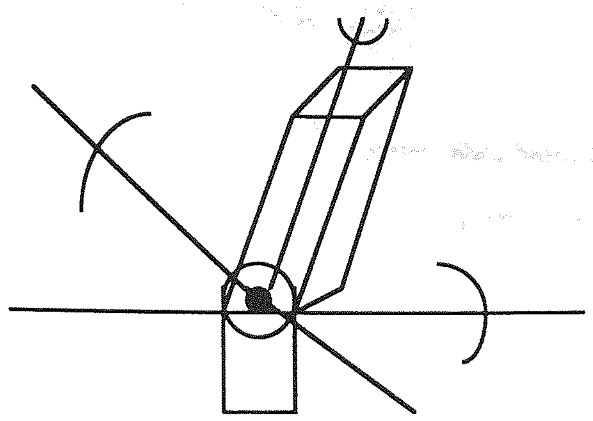

Figure 2.4 showing ball and socket joint and its axis of motion.

2.3 DEGREES OF FREEDOM (DOF)

Generally speaking every joint in a robotic arm enables a relative motion between two links and allows the links one degree of freedom. When the relative motion occurs along or around a single axis, the joint has one degree of freedom. When the motion is along or around more than one axis of motion the joint has two or 3 degrees of freedom. Thus the number of joints in a robot arm is also referred to as numbers of degrees of freedom. Most robots have between 4 and 6 DOF.

2.4 CLASSIFICATION OF ROBOTS BY TYPE OF JOINT:

Robots can be classified by joint type in to 5 groups:

(i) Cartesian

(ii) Cylindrical

(iii) Spherical

(iv) Horizontal articulated

(v) vertical articulated

The usual convention for comparison is by 3 joints closest to the robot base, and letter R is used for revolute and P for prismatic, in the order they occur, beginning with the joint closest to the base. For example, RPP indicate a robot whose base joint is revolute and whose 2nd and 3rd joints are prismatic.

2.4.1 CARTESIAN ROBOTS

These robots have 3 prismatic joints and hence are referred to as PPP. The characteristic features of these robots are :

(a) have a small work envelope (volume),

(b) high degree of rigidity ,

(c) capable of greater accuracy,

(d) control is simple due to linear motion of the links and fixed inertial load caused by fixed moments of inertia throughout the work envelope.

The robots of this type has three linear axis.

This type of robots are ideally suited for closely calibrated tasks and machining. For example, IBM's RS-1 and SIGMA from Olivetti.

2.4.2 CYLINDRICAL ROBOTS

These robots arms consist of one revolute joint and two prismatic joints, hence are referred to as RPP. The characteristic features of these robots are:

(a) Larger than Cartesian work envelope.

(b) Mechanical rigidity is slightly lower than Cartesian robot.

(c) Control is bit more complicated than in cartesian models, due to the varying moments of inertia at different points in the work envelope and to the revolute base joint.

It has two linear and one rotary axis. The robots of this type are suitable for pick and place applications. The example of this type of robot include Versatran 600 robot made by Prab.

2.4.3 SPHERICAL ROBOTS

These robots have two revolute and one prismatic joint and are commonly referred as RRP. The robots of this type have the following characteristics:

(a) Larger working envelope.

(b) Lower degree of mechanical rigidity than cylindrical models.

(c) Control is more complicated than in cylindrical robots, because of the rotary motion of the first two joints.

These robots have one linear and two rotary axis. The robots of this type are best suited for long and straight reach operations. The examples of this type of robots include Unimate space 2000B from Unimation Inc.

2.4.4 HORIZONTAL ARTICULATED ROBOTS

Arms of these robots have two revolute joints and one prismatic joint hence are referred as RRP. The robots of this type have following features:

(a) Work envelope is smaller than those of spherical robots but larger than Cartesian cylindrical robots.

(b) Robots of this type are appropriate for assembly operations due to vertical linear motion of the 3rd axis.

2.4.5 VERTICAL ARTICULATED ROBOTS

This type of robots include 3 revolute joints hence referred as RRR. Robots of this type are similar in structure to human arms which also have only revolute joints. The characteristic features of these robots are:

(a) Work envelope is larger than any other type.

(b) Mechanical rigidity is lower.

(c) No great degree of precision can be achieved in locating the end effector.

(d) Control is complicated and difficult, because of 3 revolute joints and because of variances in the load moment and moments of inertia throughout the work envelope.

This is most popular type of robot and avoids prismatic joints. Examples of this type of robots include PUMA from Unimation Inc. and T3 from Cincinnati-Milacron.

2.5 ROBOT ACTUATORS

Robot actuator is a device for producing mechanical movement. Robotic manipulators commonly use one of the three basic drive systems: hydraulic, electric or pneumatic.

2.5.1 HYDRAULIC ACTUATORS

Hydraulics is a Greek word for water. However, in robot drive systems oil is used instead of water. The oil is placed under pressure so that the energy from the oil is used to move the manipulator. A typical hydraulic actuator consists of hydraulic piston, hydraulic servo valve, electric motor and hydraulic power supply.

Hydraulic actuators are useful in robots which carry high payload of several tons, as it is possible to generate an extremely high force in a small volume.

2.5.2 ELECTRIC ACTUATORS

The electric actuation is most common method among the three methods of actuation in industrial robots. Electric motors are used to lift and position medium to heavy weight objects. They are easily controlled with a computer. Electric motors can be used on direct current (DC) or alternating current (AC). These motors are equipped with positional information systems that continuously feed positional information back to the control. The motors used for robotic control have speed control so as to control the speed of the joints while appproaching the destination. Another important consideration is the availability of the braking system so that when the robot has reached the desired position the motor must stop.

The main disadvantage of this form of actuation is that the power-to-weight or torque-to-weight ratio is smaller than that of hydraulic motors. Also, the current density is high, which causes problems of power loss and overheating. Another common method of electric actuation is the use of stepper motors. The description of each of these motors is beyond scope of this work. The readers can refer to TODD [1986], LHOTE et al. [1987], MILLER [1988] and MALCOLM Jr. [1988] for further detail.

2.5.3 PNEUMATIC ACTUATORS

Pneumatic actuators use air to drive a piston. The main difference between hydraulic and pneumatic systems is that former transfers fluid under pressure while the latter transfers air under pressure. Pneumatic systems are used when light loads have to be manipulated by the robot. A pneumatic system consists of an air compressor and air storage tank. The compressor is usually driven by an electric motor. An electrical signal controls a valve which, in turn, control the flow of air in to a cylinder. Pneumatic systems are limited to pick and place manipulators as the compressibility of air makes it difficult to design servo systems. In this mode of operations, the valves are either fully open or closed and each actuator stops only at the end of its travel. A pneumatic actuator is commonly used for the robot gripper of an electric or hydraulic robot.

2.6 ROBOT SENSORS

In order to perform tasks like humans, a robot must have a sensing capability so that it can interact with its environment. This is very desirable feature, if the robot has to have some degree of intelligence, so that it can receive information about its working environment and take some decisions on the basis of information received i.e collision avoidance, shape recognition, robot guidance, object identification and handling and grasp failure. Hence sensors in robotics have drawn considerable interest in research in the last 10 years or so. The basic sensing devices used in robotics are sensors and transducers. A transducer is a device which is capable of converting a physical non-electrical input quantity in to an electrical output quantity . A sensor is defined as a device , usually based on a transducer, capable of converting a physical non-electrical input quantity in to an electrical output quantity and of processing it according to a given algorithm, so that its output can be sent to a device such as computer. These electrical signals are processed by the robot controller, which instructs the robot to perform the desired task. Robot sensors varies considerably in complexity from simple limit switches to highly sophisticated vision systems.

2.6.1 CLASSIFICATION OF ROBOTIC SENSORS

Robotic sensors can be broadly classified in to two main groups: internal and external. The internal sensors usually forms integral part of the robot such as potentiometers, tachometers, etc,. and are used for the movement of the robot. The external sensors are mainly used to sense the environment of the robot. External sensors can be further subdivided in to contact and non-contact sensors. As the name implies, the contact sensors makes physical contact with the objects being manipulated and noncontact get information without making such contact. The contact sensors include pressure, tactile, force and torque sensors. The noncontact sensors include vision, proximity sensing and sonar ranging,temperature sensors and chemical detectors. The sensor classification is shown in figure 2.5 below :

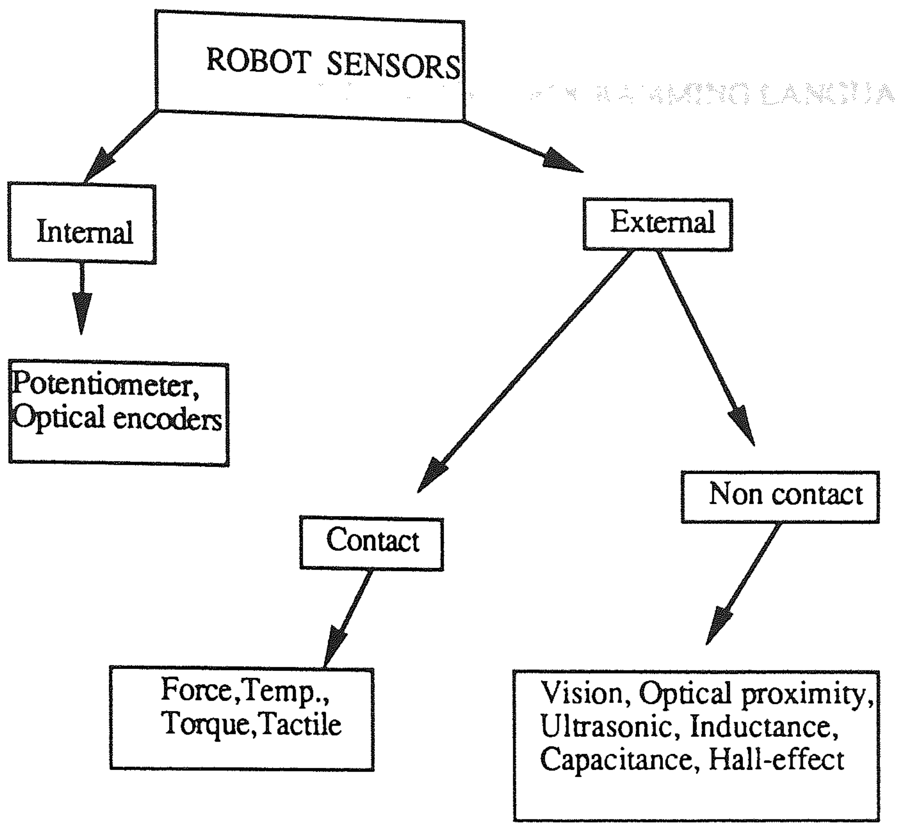
ROBOT SENSORS
Internal
External
Potentiometer, Optical encoders
Contact
Non contact
Force,Temp., Torque,Tactile
Vision, Optical proximity, Ultrasonic, Inductance, Capacitance, Hall-effect

Figure 2.5 showing classification of robotic sensors

Detail description of each type of sensor is beyond the scope of present study. Further information about each of these sensors is provided by REBMAN and TRULL [1983], SNYDER [1985], HILL and SWORD [1973], St. CLAIR and SNYDER [1978], DARIAO et al. [1983], LEATHAM-JONES [1987] and RUOCCO [1987].

CHAPTER--3 ROBOT PROGRAMMING AND PROGRAMMING LANGUAGES

3.1 INTRODUCTION

Since their development in the sixties all industrial robots have had some means of being programmed. It is this feature, being programmable, which distinguishes robot manipulation from pure automation. Robot programming is the essential means of interfacing the human with an industrial robot to perform various prescribed tasks. A major obstacle in the use of manipulators as general purpose assembly machines is the lack of a suitable and efficient means of communication between the user and the robotic system, so that the manipulator can perform the desired tasks. Consequently robot programming has become a major focus of attention in the present decade. At present unfortunately there is no standard universally acceptable definition of what constitutes a robot programming language. A NATO report [1986] of the working group on robot programming languages formulated this definition :

"a robot programming language is a means by which programmers can express the intended operations of a robot and associated activities."

The group emphasised that a robot language in addition to expressing the motion of a robot , must also allow the programmer to interface with a large number of items such as sensors, geometric modelling, systems planners and robot control systems etc.

At present there are nearly as many types of robotic languages as there are types of robot. The methods used in programming commercially available robots varies from the simplest, teach by hand approach through high level programming languages to complex task-oriented problem solving techniques.

For assembly and more complex handling operations teach by hand methods are inadequate . Furthermore, it is essential to integrate sensors into the software of the robot controller.

3.2 TYPES OF ROBOT PROGRAMMING

In order to execute a task by the robot, it is necessary to be able to give it appropriate instructions on how and in which order to approach and execute the task. These instructions to the robot can either be given in the form of textual programming or by manually teaching it the appropriate sequence required for the execution of a task. Robot programming as shown in figure 3.1 and 3.2 can be achieved in one of two ways: on-line programming and off-line programming.

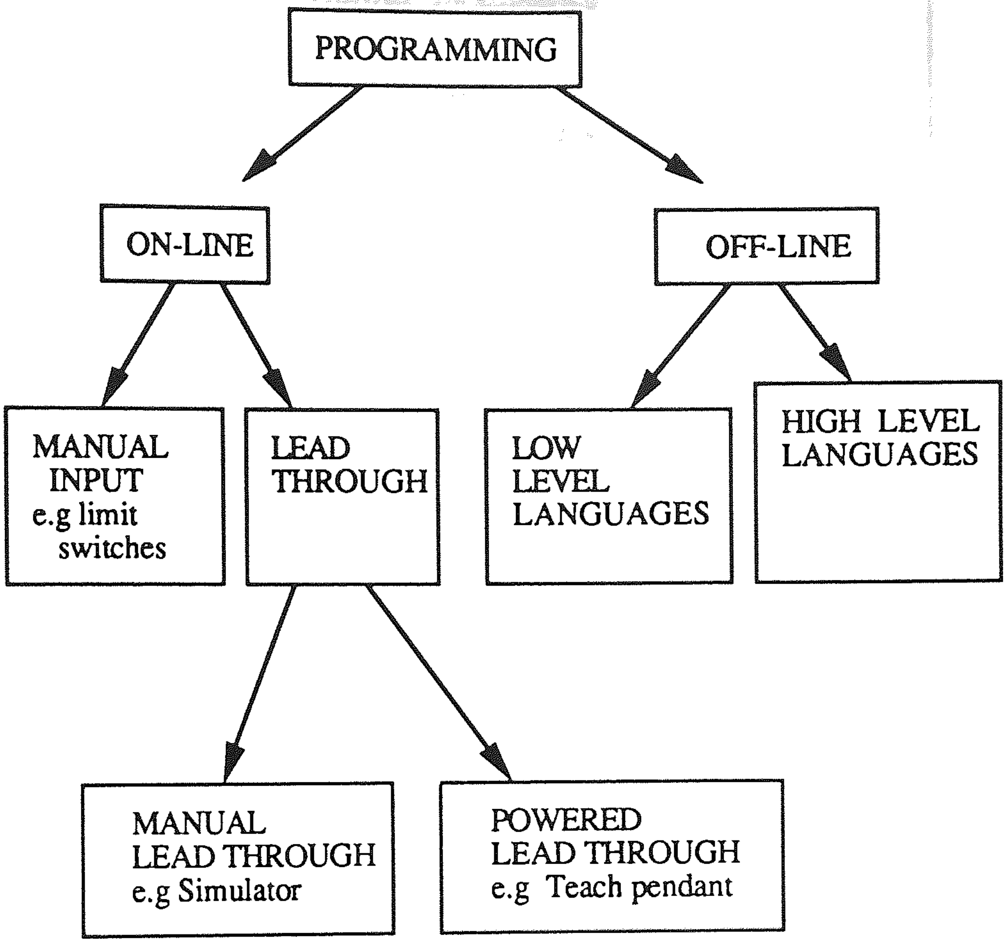
PROGRAMMING
ON-LINE
OFF-LINE
MANUAL INPUT e.g limit switches
LEAD THROUGH
LOW LEVEL LANGUAGES
HIGH LEVEL LANGUAGES
MANUAL LEAD THROUGH e.g Simulator
POWERED LEAD THROUGH e.g Teach pendant

Figure 3.1 showing types of robot programming.

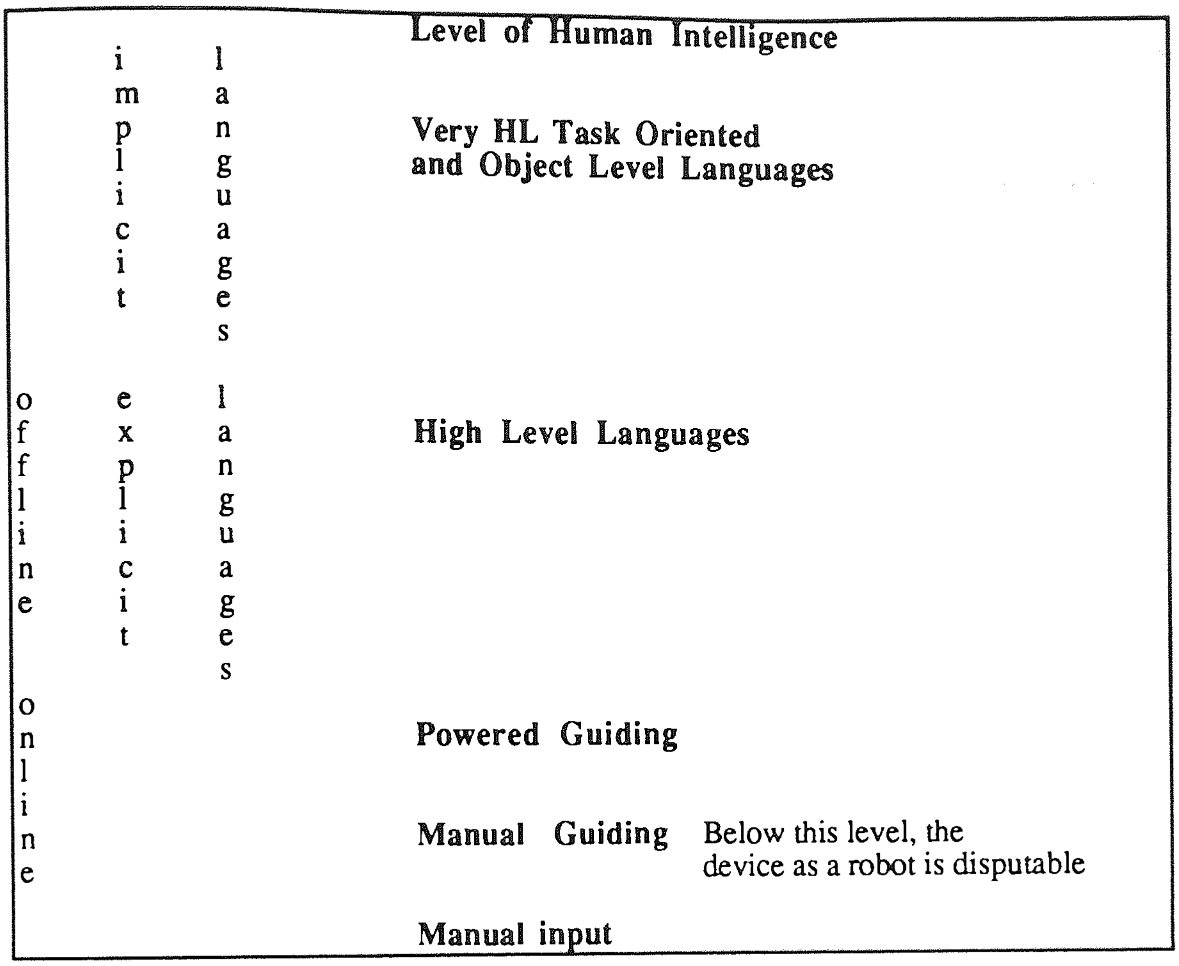
Level of Human Intelligence
implicit languages
Very HL Task Oriented and Object Level Languages
offline
explicit languages
High Level Languages
online
Powered Guiding
Manual Guiding
Below this level, the device as a robot is disputable
Manual input

Figure 3.2 showing types of robot programming and language levels

3.3 ON-LINE PROGRAMMING

By this method the robot is programmed directly by the human operator the main feature being that the robot is actually used during the programming process. The appropriate data such as positions and orientations of joints are read directly from the robot controller and is saved in memory. The saved data can be read and sent to the robot controller, later on, to repeat the taught task. Hence on-line programming essentially involves using the robot first in teach mode and later in repeat mode. This type of programming is also referred to as teach and repeat mode of programming which can be carried out in one of two ways viz :- manual input and lead through programming.

3.3.1 MANUAL INPUT

Manual input [MAIR 1987] uses limit switches or mechanical stops to start and terminate the movement of the robot. Robots using these type of programming are simple, low cost and non-servo controlled. However, because of the lack of sophistication and of complex electronics it is disputable whether such devices should be classified as robots.

These robots are used mainly in pick and place type of applications such as with machine tools, press servicing and the loading of injection moulding and die-casting machines. In simpler devices, the motion of robot is controlled by bolting metal blocks to the axes of the robot which function as stops. In more sophisticated robots, cams or pegs are mounted on a cylindrical drum, which rotates and makes contacts with micro-switches. These switches activate solenoids which open and close the hydraulic or pneumatic control valves resulting in the activation or deactivation of robot motors.

3.3.2 LEAD THROUGH PROGRAMMING

Lead through programming involves moving the robot by the operator through a series of locations or points to execute a task. The operator has complete control over the motion path. In order to teach a task, the operator allows the robot to move through its working space in the desired motion path and record the relevant joint angles of each joint in its memory. This sequence of positions saved in memory can be retrieved during the playback mode so that the taught task can be repeated. This method of programming is sometimes referred to as guiding since the operator is essentially guiding the robot and has complete control on the path and sequence of operations performed by it. Teaching the robot in this way can be achieved by means of a joystick, a set of push buttons or using a master slave system. This method of programming was principal means of programming first generation robots in the early 60's when robots were first being used for industrial applications and still is the most commonly used method of programming industrial robots. The main advantage of this type of programming is that it is simple to learn and the user is not expected to have any programming experience and is quite adequate for repetitive applications such as spot welding, paint spraying and pick and place applications. The main drawback is that the teaching is done on-line using the robot, so that if the robot is part of a complex industrial assembly system, the whole production line may have to be stopped, which is normally unacceptable. Furthermore, a taught task cannot be modified in any way, so that even a slight change in component size, position or orientation will require repeating the whole sequence of teaching and saving.

This simplest level of programming is also inadequate for complex applications which may require sensor integration such as part recognition using vision system. This type of programming can be achieved by one of the two ways: manual lead-through (simulator) and powered lead-through (teach-pendant).

The main advantages and disadvantages of on-line programming are summarised in table 3.1 below:

	ADVANTAGES	DISADVANTAGES
1	Easy to learn by the operator and quick to implement.	It can be tedious and time consuming for complex applications.
2	Collisions with other objects and equipment can be avoided by the operator by observing during the teaching session	Since it involves using the robot, which has be disconnected from executing its present task during teaching session thus loosing valuable production time.
3	Operator can transfer his/her manual skill during teaching session	Program modification is not possible, thus a slight change in task will require complete teaching of the task.
4	No additional testing or debugging is necessary.	Sensors can not be integrated.
5		Programs are robot specific hence are not portable.
6		Synchronisation with other devices is not possible.

Table 3.1 comparing advantages and disadvantages of on-line programming.

3.4 OFF-LINE PROGRAMMING

Off-line programming implies, in robotics, writing a program off-line with respect to the robot and on-line with respect to the computer using a computer terminal. Thus, the robot is not used during the program development stage but is needed at the final stage to test the program with the parts to be manipulated.

From computer science point of view off-line programming is true programming in robotics applications. Evidently, the off-line programming requires a programming language which may be assembly language or a high level language. The use of assembly language in robot programming is rarely found in literature because of the obvious advantages of high level languages over assembly languages and machine codes. Robot programming using a high level language (off-line) has the following advantages over teach by hand methods (on-line):

(a) Programs are easy to develop to high level constructs.

(b) It is possible to describe a task more naturally using a high level language.

(c) Programs are more portable.

(d) The existing program can be easily modified when robot task is changed.

(e) Provide decision making facilities

(f) Allows mathematical operations to be performed.

(g) Allows the development of structured and textual programs which are readily understood by the programmer.

(h) Supports the development of software off-line and subsequent down loading it to the robot thereby, saving the valuable time and cost in software development.

(i) Allows integration with other systems such as CAD/CAM facilities.

(j) Supports modularity.

During the last two decades various high level languages have been developed either as extensions to existing languages or by writing new customised languages for robot programming which are reviewed in the next section (3.5).

3.5 REVIEW OF ROBOT PROGRAMMING LANGUAGES

The field of robot programming has been a highly active area of research in the last two decades. A number of robot languages have been reported and newer ones are constantly being added to the list. Most of these languages were developed for a particular type of robot to run on a specific hardware. BONNER [1982] has described and compared 14 robot programming languages. GINI and GINI [1984] have also described languages which were mainly developed in Europe. GRUVER et al. [1983] has also carried out a comparative evaluation of 8 commercially available robot programming languages. Thus it appears that perhaps there are as many robot languages as robot types. It is not possible to describe each of these languages in significant detail, however, some of these languages will be briefly described in order to highlight their important features. In this section table 3.2 summarises the background information and table 3.3 carries out a comparative account of these languages. Further information about these languages is contained in appendix A.

Language	Year	Origin	Computer Hardware	Robot arm	No. of arms	Flexibility to other devices	References
WAVE	1970-75	Stanford	PDP-10 PDP-6	Stanford	1	no	Paul a 1976 Paul b 1983
AL	1974	Stanford	PDP-10 PDP-11	Stanford Puma	up to 4	yes	Finkel et al. 1975, MC Lellan 1981
HELP	1975	General Electric	PDP-11	Allegro Pragma A3000	1-4	yes	Gini and Gini 1984, Donato and Camera 1980
LAMA-S	1975	France	-----	------	-------	--------	Falex and Parent 1979
SIGLA	1975	Olivetti	Mini computer	Sigma	1-4	yes	Banzano and Buronzo 1979
RAPT	1975-80	University of Edinburgh	----	------	--------	------	Ambler 1982
VAL	1975-78	Unimation	PDP-11/45	Puma	1	no	Shimano 1979, Shimano et al. 1984
MAL	1977	Milan Poly. Italy	Mini computer	Milan Poly.	2	-----	Gini et al. 1979
AML	1977-84	IBM	IBM 370, 4360, 5530, Motorola 68000	IBM RS-1, 7535	1	yes	Grossman 1985, Grossman and short 1985
AUTOPASS	1977 +	IBM	Mainframe	IBM	1+	yes	Liberman and Wesley 1977
LM	1979	----	-------	Robitron, Renault, Kremlin	1	yes	Hendy and Braley 1986, Mazer 1984
MCL	1979	Mc Donald Douglas	Main frame	no specific	1+	yes	Wood and Fugelso 1983
LRS	1979	Spain	PDP-11/73	Scara robot	---	-----	Puente et al. 1986
ROBEX	1980	Aachen, W. Germany	----	---------	1+	yes	Weck et al. 1984

PASRO	1981	Biomatic, W. Germany	----	----	---	-----	Gini and Gini 1984, Biomatic 1983
FA-BASIC	1982	---	-------	-------	----	-----	Mohri et al. 1985
RAIL	1982	Automatix Inc.	---	Robovision, Cybervision	-----	-------	Franklin and Vandenburg 1982
AR-BASIC	1983	American Robot Corp.	---	-------	1	yes	Gilbert et al. 1984
SRL	1984	Uni. Karlsruhe, W. Germany	Independent	Independent	---	-------	Blume and Jacob 1984

Table 3.2 Background information on robot languages.

Language	Language level	Base lang.	Comm. availability	Applications	Sensors	Vision	Data types	Parallel execution	Lang. type
WAVE	Assembly Language	--------	no	Assembly operations	yes	yes	Frames, vectors, loop counters	--------	Interpretive
AL	High level	Conc. Pascal	yes	Assembly operations	yes	yes	Scalar, Arrays, Vector, Frames	yes	Interpretive
HELP	---	Mixture of HLL	yes	Assembly operations	yes	no	same as ALGOL	yes	Interpretive
LAMA-S	Assembly level	LAMA (MIT)	no	handicapped people	yes	no	Frames	yes	compiled
SIGLA	Machine Lang.	----	no	--------	no	no	counter	yes	Interpretive
RAPT	High	APT	no	Assembly operations	no	no	Frames, Vectors, Against, Fit etc	no	------
VAL	Assembly level	new	yes	process control	yes	no	same as WAVE, VALii Semaphore	yes	Interpretive
MAL	--	BASIC like Syntax	no	Mechanical Assembly	yes	no	Arrays, Variables, Constants , Frames	yes	Interpretive

AML	High	PL1	Yes	Plant floor Automation	yes	yes	INT, Real, Strings	yes	Interpretive and Compiler
AUTOPASS	Very High level	PL1	no	Mechanical Assembly	yes	yes	PL1, Frames, Vectors	------	------
LM	High	Pascal	yes	Assembly operations	yes	yes	Pascal, Frames	---	----
MCL	High	APT	yes	Assembly operations	yes	yes	Frames	no	Compiled
LRS	High	Pascal	no	General	yes	yes	Pascal and Frames	----	Interpretive
ROBEX	Low level	APT	no	FMS	yes	no	Variables, Frames, Arrays	no	----
PASRO	High	Pascal	no	-----	-----	------	Int, Real, Boolean, VEct, Rot, Frame	no	-------
FA-BASIC	High	BASIC	no	FMS	no	yes	basic	---	Interpretive
RAIL	High	Pascal	yes	Welding, Man., Vision	no	yes	Pascal and Frames	-------	Interpretive
AR-BASIC	High	Basic	yes	Man. systems, FMS	yws	yes	Basic and Position	---	Interpretive
SRL	High	AL and Pascal	no	Assembly operations	yes	no	INT, Real, String, Vect, Rot, Frame, Interrupt, Sqrt, Exp	yes	-------

Table 3.3 comparing main features of robot programming languages

3.5.1 OTHER LANGUAGES AND SOFTWARE SYSTEMS

In addition to the languages described above there are other language systems have been reported in the literature. These include MHI, MINI, TEACH, ML, EMILY and MAPLE [LOZANO PEREZ 1983], LMAC, LPR, portable AL, VML [GINI et al. 1979].

ANORAD, FUNKY, PAL, RCL, RPL, T3 [BONNER 1982], LMAC [HAURAT and THOMAS 1983], voice communication with robots [LEVAS and SELFRIDGE 1983], HANDEY [LOZANO PEREZ et al. 1987]. Other programming and language systems are summarised in table 3.4 below :

AUTHOR	COMMENTS
PRICE 19 84	Off-line programming system known as ROPL for ASEA robots.
AZZAM and UNUVAR 1985	Graphical off-line robotic planning and programming system (GRIPPS)
DERBY 1984	Off-line programming of two industrial robots using a general arm simulation program (GRASP)
GOMMA and LAWRENCE 1985	Interface for robot programming
CARAYANNIS et al. 1989	Integrated programming environment for a robotic workcell (SAGE)
BURCKHARD and MARCHINDO 1984	Multi-robot system for programming system for programming the assembly robot known as LERNA.
WILL 1979	Software issues for using robots in assembly applications.
ASADA and IZUMI 1987	A methodology for the automatic generation of robot programs for hybrid position/force control.
PERTIN-TROCCAZ 1987	Accessibility analysis for automatic grasping.
RAZ 1989	Use of graphical software for simulating robot motion.
STOBART 1987	Use of geometric modelling system for the off-line programming.
BLUME 1984	Use of a robot Data Base (RODABAS) for implicit level of programming.
LIM and LOY 1984	Use of robot programming and teaching using Graphic Tablet.
FAVERJON 1986	A system for object level programming of industrial robots.
HAURAT and THOMAS 1983	Software tools known as LMAC for programming of robot system.
TROVATO and SCHREINER 1987	Language system known as LABICS for dedicated real time control of robot system.
KOSSMAN and MALOWANY 1987	A system known as RCCL which is modular, extensible and portable.

JETLEY 1984	Development of a Rhino operating language (ROL) for Rhino robot.
ADORNI et al 1984	A system known as D-LISP for vision processing.
SOROKA 1987	A language known as CONC for analysing concurrent robot programs.
NAYLOR et al. 1987	A graphical off-line robot programming system (PROGRESS).
KURAU 1979	PEARL for Programming computer controlled manipulator.
LAUGIER 1984	Robot programming using LM and CAD facilities.
CHAN and VOELCKER 1986	A new language for machine processing/programming language (MPL).
TRONCIE et al. 1988	Graphical interactive programming.
WLOKA 1986	Development of a robot simulation system known as ROBSIM.
SATO and HIRAI 1987	Language-aided Robotic Tele-operation system (LARTS).
SHENG and DAVIES 1987	A high level approach to programming a robotic FMS.

Table 3.4 robot programming systems and languages.

3.6 CLASSIFICATION OF ROBOT PROGRAMMING LANGUAGES

Robot programming languages described in the previous section may be classified on the basis of their development as a robot programming language. The earliest robots during the 1960's were used only in the teach-repeat mode for simple operations. This mode of operation was sufficient for applications such as spray paintings and spot welding where a single task was endlessly repeated. Rapid developments in the application of robots in industry coupled with the availability of inexpensive and powerful computers has changed this trend towards programming robots via programs written in programming languages. Usually these languages have special built in features which apply to the problems of programming robots (manipulators) and hence are called Robot Programming Languages. These languages can be grouped into 3 main categories [CRAIG 1986] :

3.6.1 SPECIALISED MANIPULATION LANGUAGES

These newly developed languages have been devised for the sole purpose of controlling robots and contain many robot specific features and unique routines. They may or may not be considered true general purpose computer programming languages. One such language VAL, developed in 1975 to control the industrial robots is based on WAVE, for use as a manipulator control language. As a general purpose computer language it is quite weak. For example, it does not support floating point arithmetic and character strings. Even the sub-routines cannot pass arguments. A more recent version of VAL called VALII [SHIMANO et al. 1984] was introduced which overcomes some of these shortcomings. Another language of this type is AL which is a high level programming language developed at the Stanford AI laboratory and has ALGOL like block structure and syntax.

3.6.2 EXTENDED EXISTING COMPUTER LANGUAGES

These robot programming languages have been developed by adding a library of robotic specific sub-routines to existing popular computer languages such as PASCAL and BASIC. The user writes the program in the original computer language by making calls to the specialised robot specific routines. AR-BASIC and FA-BASIC are both special versions of BASIC. PASRO, Pascal for Robots which has been developed by the German company Biomatic, is based on the Pascal with additional data types and procedures devised to perform robot specific tasks which are stored in a library and are callable by the Pascal compiler . The procedures used are provided to drive the arm either in point to point or along a continuous path. JARS [CRAIG 1986] developed by NASA's Jet propulsion Laboratory, is also based on PASCAL.

3.6.3 NEW GENERAL PURPOSE LANGUAGES

These robot programming languages are developed by first creating a new general purpose language and then adding a library of predefined robot-specific sub-routines. Examples of such languages are LM, HELP, LAMA-S, MAL, SRL, ROBEX, LMAC AL , RAPT, SIGLA, and AUTOPASS. Since a great deal of time and effort is involved in developing and testing new languages, it is much more convenient to develop extensions to existing general purpose languages.

3.7 LEVEL OF ROBOT PROGRAMMING

The level of programming depends on the level of detail in which robot operations are to be expressed. BONNER and KANG [1982] have described five levels of programming: microcomputer, point-to-point, primitive motion, structured and task oriented level of programming. According to GINI [1987] there are four levels of robot programming: joint level, manipulator level , object level and task level, which are summarised in figure 3.3 below:

Number	Level of Programming	Language/s
5	Human Intelligence	
4	Task Level Programming	AUTOPASS, LAMA

3	Object Level Programming	RAPT, AUTOPASS
2	Manipulator Level Programming	AL, AML, MCL, ROBEX, SRL, VALii
1	Joint Level Programming	VAL, SIGLA, HELP, MCL

Figure 3.3 Levels of robot programming

3.7.1 JOINT LEVEL PROGRAMMING

This is the simplest and lowest level of robot programming. The task to be performed by the manipulator is expressed simply in terms of the control commands required to drive the individual motors and actuators of each joint so that the manipulator end-effector is aligned at a particular position and prescribed orientation. The user then instructs the individual joints of the manipulator to execute the task. Typical instructions may include :

move joint 1 by 5 steps.

This method of programming robot is best suited to the programming of cartesian robots where robot joints slide in straight lines. The main disadvantage of this level of programming is that the user has to know the complex geometry and mechanism for accessing each actuator of the robot. There is some confusion over this level of programming. VAN AKEN and VAN BRUSSEL [1988] describe this level of programming as teach or guided level. This point of view is arguable since from programming point of view teach by guiding is not true programming since the user merely guides the robot through its work space by aids such as teach pendants and merely saves the desired points in computer memory without the need to write any textual program. BONNER and KANG [1982] describe joint level programming as primitive level programming as it is first step towards writing a true textual program using a programming language. The robotic languages which supports this level of programming include SIGLA, HELP, VAL and AML.

3.7.2 MANIPULATOR LEVEL PROGRAMMING

At this level of programming the operator controls the position and movement of the manipulator in Cartesian space quite independently of the arm joints. This method is marked improvement over the previous level as the user does not need to be familiar with robot geometry. The motion of the robot involves the specification of successive locations in the form of a frame which is related to the robot end-effector. Some of these languages support sensors such as force torque, touch, proximity and vision system which may be used to verify the location of objects or to modify robot motions. This type of programming is more commonly known as explicit level as the user must specify explicitly each step the robot has to take, such as, specification of position and orientation of the object to be manipulated, robot path, speed and destination location etc. Most languages which support this level of programming use structured control constructs and provide extensive use of co-ordinate transformations, complex data structures, sensor integration and parallel processing facilities. A typical instruction may include, move arm to grasp bolt, which itself may be present and oriented in a specified position. Languages which offer this level of programming include, VALII, AML, ROBEX, SRL, AL and MCL. VALII allows the user to express the position and orientation of the manipulator end-effector in space in terms of transformations. In ROBEX and MCL, which are extensions of NC machine tool programming techniques, the points and matrices can be defined to describe positions and orientations of the manipulator. Some authors [BONNER and KANG 1982] describe this level as structured level programming

3.7.3 OBJECT LEVEL PROGRAMMING

Object-level languages allow the user to define operations in terms of objects being manipulated and hence simplify the programming task. It allows the human operator to describe in a natural way an assembly task the robot is to perform. This level of programming requires some knowledge of the objects which are present in the manipulator's working environment. Additionally it might make use of external sensory information. A typical example might be to instruct the manipulator to pick up an object, decide its shape, position and orientation, grasp it and deliver it to the required destination. The languages which have such features include AL, SRL, RAPT which represent objects with frames. For example a typical command in RAPT may look like this :

move/A , perto, (TOP of B),-3 i.e. it will move object A towards the top of B by 3 units of distances. Languages such as AL, LAMA and AUTOPASS also provide some features which support this level of programming. But LAMA and AUTOPASS have not been yet implemented commercially.

3.7.4 TASK LEVEL PROGRAMMING

This is a completely new approach in robot programming, is the highest and most complex level of programming and is mainly used by AI researchers. This level of programming conceals low level aids like sensors and co-ordinate transformation from the user. These languages allow the user to command the desired sub-goals of the task directly, rather than specify the details of the every action the robot has to take. This level of programming is sometimes referred to as implicit level programming as opposed to explicit level programming. In the latter type the user is required to explicitly express in a computer program all the sequences of manipulator motions required to accomplish the task which necessitates the user being familiar with all of the basic operations of a manipulator in order to write the required program.

A task-level programming system must have the ability to perform many tasks automatically. It allows the user to describe the task in a high level language (tasks specification), a task planner will then consult a data base (word model) and transform the task specification in to robot level programming (robot program synthesis) which will accomplish the task. For example if an instruction to

"screw the interlock and the bracket together"

is issued, the system must have an accurate and most up to date account of its environment, shape, position and orientation of the objects. It must be able to choose a good grasping location of the object, grasp it and plan a collision free trajectory. Thus a task level programming system must have the ability to perform many tasks automatically. Whilst this might appear to be an ideal way of programming, no such system as yet exist and is an active topic of research. The world model for such a system must be able to cope with unexpected obstacles, error recovery, poor or failed grasping of objects, degree of accuracy of manipulator and integration with external sensors.

AUTOPASS devised by IBM is claimed to meet the criteria of a true task level programming. It uses instructions such as might be given to a human assembly worker. Although this type of programming of robots appears to offer an ideal approach the actual program becomes so complex which along with numerous uncertainities which arise in the real world domain of the robot that real task level programming is still in its infancy. Consequently much more research is needed in this area. Another language which provides some features for this level of programming is LAMA.

3.7.5 LEVEL OF PROGRAMMING BY NATO's REPORT

A NATO report [NATO 1986] on robot programming languages has identified the following 3 levels of robot programming :

(a) joint level

(b) manipulator level

(c) task level

The report has recommended the inclusion of two new levels of programming: DEVICE level and FEATURE level. The device level is a level where low level interface to the sensors and joints of the robot takes place. The feature level lies between manipulator and task levels and is the level at which high level sensor data processing occurs. The report has further emphasized that a language system should provide a complete working environment for the development of robot software so as to support world modelling, sensors , user interface etc. as shown in figure 3.4 below:

Figure 3.4 Levels of programming NATO [1986]

CHAPTER--4 REQUIREMENTS ANALYSIS OF ROBOT PROGRAMMING LANGUAGES

4.1 INTRODUCTION

Just like other computer applications such as database, process control etc, which need specific programming facilities related to that area, robot programming also requires certain unique features in the programming languages. In this section an attempt will be made to highlight some of the essential components of a programming language used to develop software for robotic applications. Unfortunately there is no agreement on the robot programming requirements amongst robot software developers. This in the main, may be ascribed to two factors; firstly since the field of robot programming is currently an active area of research, therefore new robot programming languages with supposedly better robot specific features are appearing quite rapidly and secondly the diversity and application range of robots is increasing continuously with the development of more accurate, faster, and sensor based sophisticated robots. Hence it is difficult to generalise on the requirement at present state. However it is possible to discern, in the field of robot programming some general and special requirements as follows in the next section.

4.2 GENERAL REQUIREMENTS

This section covers the general desirable features of those HLL's which are useful in writing software for computer programming as well as robotic applications. However, it must be born in mind that some of these requirements affect each other and must not be considered in isolation as they are desirable features for providing a suitable environment for the development and maintenance of software for particular applications. For example, maintainability of a program depends upon readability, writeability and modularity etc.

The desirable features of a robot programming language include :

1 EASY TO LEARN

2 EFFICIENCY AND SPEED

3 MODULARITY

4 PORTABILITY

5 DATA TYPES AND ABSTRACTIONS

6 READABILITY

7 MAINTAINABILITY

8 COMMERCIAL AVAILABILITY

9 PROGRAMMING SUPPORT/ENVIRONMENT

10 WRITEABILITY

11 FLOW OF CONTROL

12 INTERACTIVITY

13 EXTENSIBILITY

14 CONCURRENT PROGRAMMING

4.2.1 EASY TO LEARN

Every programming language has its own unique syntax, structures and programming style . As the fundamentals of some languages are easier to learn than others, some of which may require an extended learning period before serious use can be made of the features offered by that language. RAMBIN and TAYLOR [1986]suggest that if the language is easy to learn it will considerably shorten the software development time thereby reducing the overall cost of the system. Clearly, a language which is easy to learn is much more likely to be accepted by the users and this factor could play a crucial role in acceptance of the software and hence its future marketability. This fact is even more important for a robot programming language since most users will be non-professional programmers or might have some knowledge of machine tool programming[WECK et al.1984]. Thus a good high level language with sophisticated syntax may find difficulty in being accepted on the shop floor. Industry may be reluctant to accept such sophisticated program systems which require qualified computer science professionals in terms of an extra cost involved in employing them.

4.2.2 EFFICIENCY AND SPEED

Efficiency of a programming language can be regarded as speed of execution and utilisation of memory space. The purpose of the optimisation is to reduce the program size, increase execution efficiency and reduce data storage requirements. A language should therefore provide transformations which allows efficient use of a code. With considerable decrease in hardware cost efficiency may not appear to be important in terms of processing power and storage capacity, but it is still relevant under real time systems such as robotics where the response time may often lie within certain critical constraints. Since general real time programs and robots using real time features such as complex vision systems and mathematical models requiring many iterative calculations can be highly computationally intensive so that the language must be fast (i.e capable of efficient implementation). However, overloads caused by using high level languages for clarity , simplicity, structured programming techniques, control structures etc do outweigh the loss of speed rather than using machine codes or assemblers.

4.2.3 MODULARITY

Modularity is the capability of dividing a program in to some smaller units, known as modules which may be developed independently of each other. Each module may by itself perform a simple task. At subsequent stage, these modules can be integrated into one larger module. This is quite a desirable feature of any programming language because software may be developed and tested as smaller units. Additionally, a complex program can be developed as free standing modules by different programmers and subsequently combined together to create a hierarchical system.

Modularity leads to structured programming which leads to enhanced flexibility, so that a programmer can at will interchange modules, adding to and modifying them, or removing them during program development or maintenance time, at will. It also adds to the reuseability of the code.

A language should provide facilities for the definition (creation) of modules and also for information hiding. Such a language should also provide facilities for interfacing a sequence of modules into a higher level more complex hierarchical modules.

4.2.4 PORTABILITY

Portability is the independence of a program from the underlying hardware. This is very important feature of software for computer applications including robotics. Ideally the software written for an application should run on other hardware without modification. Unfortunately, complete portability is almost impossible to achieve in robotic applications. For a robot software to be portable it should be independent of:

(a) Computer hardware

(b) Type of robot

(a) COMPUTER HARDWARE

Large industrial manufacturing systems often employ a variety of computers ranging from main frame to microprocessors. They also use a variety of operating systems. Hence a programming language must be compatable with as many of these computers as possible.

WECK et al. [1984] recommends that Computer portability requires the use of standard computer system and a well accepted high level programming language such as Pascal or Fortran for this purpose.

(b) TYPE OF ROBOT

Robot programming suffers from program portability as different robots have different dimensions, geometry, sensors and employ different mathematical models to solve kinematic equations. If a manufacturing company uses robots supplied by different companies , all using different programming languages, then maintaining a complete integrated system may turn out to be very complex, which might even require professional programmers who are familiar with different systems. Even interfacing one system with other system may prove very tedious. Hence industry will gain significant benefit from a general portable language which can be independent of type of robot used. Unfortunately, at present, the majority of current robot languages and systems are highly dependent on the type of robot used.

Although there are obvious difficulties in achieving a complete portability it would be much more desirable to have some degree of portability, where most of the language constructs are independent of the robot configuration, types of sensors and processors and I/O devices used. Thus in order to transfer software from one robot type to another it will always be necessary to make certain changes, but the fewer the changes required the better it is from software portability point of view. Perhaps a better approach would be to design the software in modular form so that only certain modules would need to be rewritten or modified.

Languages like Forth do support modular programming and most of the modules are independent of robot type and I/O devices. Thus transportability of software in such languages is much easier to achieve. In such a case, for instance, the user only need to change the contents of certain variables and constants used in configuration and lengths of robot joints.

4.2.5 DATA TYPES AND ABSTRACTIONS

All high level languages do provide some basic data structure available to the programmer such as variables, constants, arrays, files etc. These data structures are defined for programmer who explicitily manipulates them in program statements. Each of these data structures can be used to create new data objects according to predefined data types such as integer, character, real and string etc. supplied by the language.

Each language comprises a set of primitive data types that are built in to the language. A good language should provide facilities which allow the programmer to define new data types. These features are not included in old languages like Fortran and Cobol, whereas modern languages such as Pascal, Ada and Forth do allow the user to declare new data types.

A programming language should provide facilities that assist the programmer in constructing his own abstractions. Information hiding is the term used for the design of programmer-defined abstract data types so that each component should hide as much information as possible from the user of the component. Any information hidden in an abstract data type is referred to as encapsulated when the language prohibits the user access to the information hidden threin. Information hiding is desirable feature in any program and can be implemented using any programming language, but encapsulation does depend on the design of the language chosen.

SIMULA 67, FORTH, concurrent Pascal, CLU, Modula-2, Smalltalk and ADA, allow a programmer, to a varying degrees, to define an abstract data type by providing a special language construct for encapsulation [GHEZZI and JAZAYERI 1987]. A comparative study of data structures in Pascal, ADA, C, ALGOL 68 and FORTRAN has been described by FEUR and GEHANI [1984] and MALONE [1984]. The importance of data structures, types and abstractions in the improvement of system clarity, reliability and modularity in programming has been discussed by SHANKAR [1980]

Robot programming involves the manipulation of different objects in its work space. This requires the programming language to declare appropriate data structures suitable for each application. The data structure may be in the form of variables, constants, arrays etc. For efficient programming the names of these variables should be task orientated e.g name of joints variables should if possible be given as waist, shoulder, elbow, gripper etc for ease of understanding. Similarly variables which store co-ordinates positions should have names like Xlen, Ylen, Zlen, Yaw, Pitch, Roll etc. The most common way of describing positions and orientations is by means of frames. The base of the robot and its end-effector can also be defined by a suitable frame. The relation between these two frames depends upon the geometry of the robot, number and types of joints. The solution of inverse kinematic problem of a robot depends upon solving these frames. A programming system must contain appropriate support modules to solve these transformations. In the same way, each object, along with properties present in the working environment of the robot have a characterstic frame assigned to them. Actual robot programming is thus assigning and relating frames to each other. According to VAN AKEN and VAN BRUSSEL [1988] defining a frame comprises two elements :-- (i) the assignment of numerical values to the variables defining positions and orientations of the frame. (ii) the selection of a suitable reference for the frame.

A language must provide facilities for the declaration of these data types which is not only useful in the development of program but also at latter stages when it may be necessary to extend or modify the existing program. Thus data structures which allow the user to define abstract data types is an essential requirement of a robot programming language. Languages such as Forth contain inbuilt features of this type. One of the major advantages of using a high level language for robot programming is that it can offer data structure to the user which hides away minor details of an instruction and only permits the use of an abstract command e.g P1 P2 move. The move command is easy to understand by operators who do not need to know the information hidden therein. Thus data abstraction is a very desirable feature in robot programming.

4.2.6 READABILITY

Readability is one of the desirable features of any programming language and of robot programming in particular. If commands convey clearly the action they are intended to take it is much easier to use these commands especially if the user happens to be a non-programmer, which is quite likely to be the case in robotic applications. In addition it is much easier to modify the section of the code if it is readable and its action is well and clearly understood. This could be of great importance and could save considerable time in program maintenance since the person making the modifications is not likely to be the original designer.

Readability helps in understanding the logic of the program and is of great assistance in error identification merely by examining the program. It also decreases learning time. Readability is however largely a matter of style and taste yet considerably enhances program understanding by examining the code. For example, the commands like GOTO or GOSUB makes it very difficult to read and follow the flow of the program. The program cannot be followed from one end to another instead one has to jump around within the program in order to understand it. Readability in robot programming can be considerably increased by assigning suitable application orientated names to variables and constants such as waist, shoulder, arm, move etc. For example, longer the name of the variable, easier it is from the user point of view to understand its meaning and purpose. Thus a language should ideally allow sufficiently long names for variables and constants.

4.2.7 MAINTAINABILITY

Maintainability relates the maintenance of software after it is released to the user. All software in its life time will require some maintenance, which imposes certain requirements on the programming language. The program must be readable but readily modifiable. Maintenance is generally the most expensive item within the whole software cycle. It may be as much as 80% of the overall cost of the software product [Dyer 1985]. Summervile [1982] classifies actions taken under the heading of maintenance of software falls into three categories:

(a) PERFECTIVE

(b) ADAPTIVE

(c) CORRECTIVE

A survey by LIENTZ and SWANSON [1980] suggests that approximately 65% maintenance was perfective, 18% adaptive and 17% corrective. In robot programming the perfective and adaptive maintenance may play major roles, as robot applications may require occasional modifications in tool size, shape etc. Thus software maintainability is quite important in robotic applications.

4.2.8 COMMERCIAL AVAILIBILITY

It is quite important that a language be readily available on commercial scale for various hardware configurations with suitable extensions for various specialised applications at a reasonable cost. This may result in robot manufacturers selecting the language for the system development. This ready availability may result in the following;

(i) greater awareness by the people.

(ii) more literature being available to assist in learning the language.

(iii) possibility of training the staff in learning the language.

(iv) easier to accept the language for adaptation to application.

4.2.9 PROGRAMMING SUPPORT/ENVIRONMENT

Robot program development, like other computer application programs require programming support such as editors, debugger etc. A good programming language must provide a programming environment in order to support the programmer. Robot programs tend to be complex and can be difficult to debug and, like other programs some time require external data, ask the user to input the data or even may require corrective actions. It is easier to develop and test individual small sections of code one at a time instead of writing a complete program. Thus an interactive programming language is much more advantageous than the compile type which may involve many laborious edit-compile-run type of cycles. Therefore by using interpretive languages, the small modules can be interactively tested and debugged.

The robot tasks may involve complex motions requiring long execution time, hence a program failure must have a facility to restart the program at an error stage rather than resetting and starting all over again so that robot programming systems should have the facility to modify the program on-line and restart at any time. The debugging system should have the facility to record sensor output so that during debugging, the interaction between robot via the sensors and its environment can be used in the debugging process if required. Since the sensory information and real time interactions are not usually reproducible. A simulation facility which can allow the program to be tested without actually using the robot during the development of the program can save considerable amount of time. Simulation can be useful since complex trajectories and motions are difficult to visualise without actually seeing their actions so that simulation can be very useful tool in program development. Since robot systems are rarely used as stand alone machines they need to communicate with other robots and other devices such as belts, moving parts, vision etc, so that programming languages should address such problems like communication, interaction, synchronisation with such devices which form part of the complex robotic system such as in the case of FMS type of applications.

4.2.10 WRITEABILITY

Writability of a programming language refers to the possibility of expressing a program in such a way so that it appears natural to the application [GHEZI and JAZAYERI 1987]. For example, a robot program may contain a command :

" move waist 5 steps"   which appears natural to this application.

Thus the program should appear as a problem solving tool without having to worry about language detail. High level languages generally are far more writeable than low level languages. Since latter has to include addressing mechanisms and actions of the contents of certain registers which makes them very difficult to understand.

4.2.11 FLOW OF CONTROL

Flow of control refers to the method by which programmer can specify the flow of execution within the program. Most high level programming languages provide the constructs to control the sequence in which individual statements are executed. A language providing good control structure can considerably enhance the readability and the maintainability of the program. There are three main types of control structures provided by most high level languages, namely sequencing, selection and repetition.

(i) SEQUENCING

Sequencing mechanism specify that execution of a statement must follow the execution of another statement. For example,

```
BEGIN ---------------- END
```

(ii) SELECTION

selection allows the programmer to specify a choice to be made among a certain number of possible alternative statements. For example,

```
IF-------THEN-----ENDIF

IF -------THEN-----ELSE----ENDIF
```

(iii) REPETITION

Repetition allows the programmer to repeat a certain code of a program over a specified number of times or until a condition is satisfied. For example,

```
   DO------------------------LOOP

DO------------WHILE---------LOOP

BEGIN ------------WHILE------END
```

For non-sensor robot programming the program follows a fixed sequence of an operation, but a robot using signals from various sensors may have to take certain decisions on the basis of information received from them, so that, for example, output from such sensors may require the robot program to take certain actions or branch to different sections within the program. For instance, a robot using a vision system may wish to ascertain whether a part to be picked from a specified location has arrived or not. If not arrived clearly there is no point in trying to pick it up. The robot may have to consult some inner model and branch accordingly under these unexpected circumstances. Similarly error detection and correction may require decisions from sensors e.g if an object to be manipulated is in the correct position and orientation then move it, otherwise send an error message.

Most robot actions are performed under conditions of great uncertainty due to the real world situation e.g an object may not be present at specified location, it may have wrong size or shape etc. To deal with such uncertainties the programming language must provide a rich control structure. With increasing use of sensors in conjunction with complexities of robot programming it is essential to have as many additional control structures as possible including those found in fourth generation languages such as

REPEAT ------UNTIL,

BEGIN-------------UNTIL etc.

In addition to these control facilities, robot program may require to receive control signals from other devices working as a part of a complex systems such as FMS, multirobots or other machines such as feeders, belts and NC machines etc.

4.2.12 INTERACTIVITY

Interactive languages allow the user to develop and test the software immediately and save considerable time during the development stage of the software as the small sections of the code can be interactively tested for the desired action. If necessary, they can be quickly modified and retested. This supports the modular design of software, since each module can be independently developed and tested and considerable time can be saved in debugging the program as well.

In the case of compiled languages the user has to write the complete program, compile it and then run it. All the errors in such languages are reported at compile or run time and the program debugging becomes complex. Even for a slight modification in the program, the programmer must go through edit compile and run cycle. Thus the programmer loses valuable time during software development stage.

Interpretive languages have many advantages over compiled ones in robotic applications as well, since interpreter execute the codes as it encounters them. Hence a program is quicker to change as minor modifications in program do not require the whole program to be recompiled. Interpretive languages help in rapid debugging, another ideal feature for developing a large and complex software such as robotics. Although interactive languages tend to be rather slow but they far outweigh the advantages in complex applications such as robotics.

4.2.13 EXTENSIBILITY

Extensibility is an extremely important feature of a robot programming language. Since every programming language will require some extensions in order to make it suitable for new application as no language can possibly provide facilities for every conceivable types of applications. This is more important in robotics as new and diverse applications are emerging all the time. Language extensibility can be a major factor in the selection of a language for robot programming [RAMBIN and TAYLOR 1986]. An extensible language should provide facilities to the programmer to add new data structures and program features (constructs) to fulfill new application requirements. ADA, AML [MANDUTIANU 1988], C and Forth are examples of extensible languages.

4.2.14 CONCURRENT PROGRAMMING

A concurrent program consists of parts which can run parallel to each other. Concurrency arises naturally in real time control systems including robotics. On a single processor concurrency is achieved by performing one section of a code of a subprogram at a time, then jumping to another and finally returning to original one. Whilst this feature is important, since it increases the speed of a system, this kind of facility is not found in many general purpose high level languages. Modern languages which do provide concurrency include concurrent Pascal, Modula, Modula-2, Occam, concurrent C and ADA. Some of the constructs provided by these languages to achieve concurrency are SEMAPHORES and RENDEZVOUS.

Concurrency is important in robot programming since robot programs tend to be complex and involve the moving of joints, monitoring sensors, performing complex calculations, preventing collisions, interfacing with the operator and communicating with other real time devices. Common areas which are suitable for concurrent programming, comprises the performance of mathematical calculations such as multiplications of rows by columns in solving homogeneous transformations and simultaneous movements of robot joints in order to align the robot end-effector to a prespecified location. Such sensor based robots may have to wait for signals from sensors in order to synchronise with external events. Multirobot systems will have to communicate with each other and other devices. Hence, concurrent programming can play significant role in such applications.

Parallel processing is becoming more significant with the development of autonomous and adaptive robotic systems. COX and GEHANI [1989] have discussed the advantages of concurrent programming in robotics and recommend that concurrent C is an ideal language for robot programming. The importance of concurrency in robot programming is presently being realised and even the use of transputers in robot programming has been reported. BROEK and BOER [1989] have described a parallel transputer system for automatically loading different sized parcels into a container. STAVENUITER et al. [1989] have described the use of transputers for the control of a flexible robot arm. Similarly, GEFFIN and FURHT [1989] have discussed the use of parallel processing using transputers and OCCAM to control the motion of a robot arm.

4.3 SPECIAL ROBOT REQUIREMENTS

In addition to general desirable features of a robot programming language discussed in the previous section (4.2), the field of robotics demands some additional features from a programming language. Some of these special requirements are discussed by WECK et al. [1984], LOZANO-PEREZ [1983], RAMBIN and TAYLOR [1986], HAYNES [1985], BONNER [1982] and VAN AKEN and VAN BRUSSEL [1988]. An ideal robot programming language should also fulfill the following robot specific requirements :

1 METHODS OF DEFINING POINTS IN SPACE

2 MOTION SPECIFICATION

3 SENSOR INTERACTION

4 DECISION MAKING

5 COLLISION CHECKING

6 HUMAN ROBOT INTERACTION

7 MATHEMATICAL LIBRARY

8 WORLD MODELLING

9 ON-LINE PROGRAMMING FACILITY

4.3.1 METHODS OF DEFINING POINTS IN SPACE

A Robot programming language must provide some means of specifying the position and orientation of the end-effector in three dimensions. This may be in Cartesian co-ordinates (x,y,z) system for cartesian robots or in cylindrical co-ordinates (x,y,θ) both of which can be defined relative to fixed reference systems. This reference system may coincide with the base of a robot or some other convenient location. This method of co-ordinate specification is known as world co-ordinate system. In the case of other types of geometrical configurations such as polar or jointed arm robots, an alternative suitable co-ordinate system may be necessary which may ultimately require conversion into cartesian co-ordinate system.

4.3.2 MOTION SPECIFICATION

In those applications where a robot end-effector must follow a specific path, it may be necessary to define a set of points the locus of which define the trajectory through which robot arm must pass. The specification of initial and final position may not be adequate in those circumstances because some applications might require the robot to approach the grasping position from a certain direction in order to avoid hitting or colliding with other objects in the work space.

In the same way a specific path must be followed by the robot for certain applications such as assembling some parts together. There may be occasions when it may be necessary for the robot to follow a specific path in order to avoid collisions with the obstacles on its way to its final destination. This can be provided by specifying via points. The via points are those points through which robot must pass while moving from source to its destination. Clearly, robots do not have to stop at these points. A typical sequence may be like this:

" move P1 P2 via P3 "

The continuous path between the points may be in the form of straight line, along a circular path or other more complex route which will depend upon the specific application.

4.3.3 SENSOR INTERACTION

For robots to be able to interact with their surroundings or environment they must be able to obtain feedback from the sensors attached to them. Without these sensors robots are of very limited use. In non-sensor based robots the parts to be manipulated by the robot must be delivered consistently to an accurate pre-specified location and at a precise orientation. The destination must be clearly stated, so must be the path of the robot. It is the responsibility of the operator that these criteria be met otherwise robot may miss the object, collide with an obstacle on its way or even deliver the object at the wrong place. This may also require the building of special tools to place objects at the correct position and orientation. In non-sensor based robots there can be no provision for grasp failure or ability to cope with uncertainties. Slight variations in size, shape and position of the tool may require additional modification of other tools which could add considerable costs.

The use of sensors can overcome lots of such problems. LOZANO-PEREZ [1983] and VAN AKEN and VAN BRUSSEL [1988] have suggested the following advantages of using sensors:

(i) the identification of objects

(ii) checks for the presence of parts to be manipulated.

(iii) the determination of the relation between the robot and other objects.

(iv) the determination of the exact location of the part.

(v) checks for the presence of a part at a specified location.

(vi) the initiation and termination of robot actions.

(vii) to take appropriate branching decisions on the basis of sensory information.

(viii) the compliance with external constraints.

The programming language must provide general input/output mechanism for obtaining sensory data and controlling their operation. Additionally , languages should provide programming flow control facilities so that suitable alternative algorithm may be followed depending upon the type of sensory data.

4.3.4 DECISION MAKING

A robot program must be able to receive data from its sensors, analyse it and make appropriate decisions on the basis of this received data. For example, feedback from a sensor may inform the robot controller that an object has arrived so that it can proceed to pick it up. On the other hand if it failed to arrive at the expected destination there is no point in trying to pick it up. Hence a programming system must provide such fail-safe decision making facilities.

4.3.5 COLLISION CHECKING

Robot programming languages should supply features such as collision checking and collision avoidance. A sensor may inform the robot that there is an unexpected object on its way to its destination then provision must be provided to take appropriate preventative measures to avoid such collisions so that robot must be able to react to varying conditions within its working area. This is very complex since it requires the robot to have complete up to date knowledge of all the objects, their numbers, size and shapes in order to ensure that by transferring from location A to B it will not collide with other objects in the proposed trajectory. Such a model will require input of clearance/tolerance limits in the program for avoiding collision. For checking collisions, it will be necessary to obtain input from sensors. Therefore these features are more relevant to the task level of programming. Current robot programming languages do not provide as yet, this type of facility.

4.3.6 HUMAN ROBOT INTERACTION

Any robot programming system would almost certainly involve some kind of interaction with the user. Commands supplied by the language must be easy to learn by the operator and easy to use. In the earlier robots, programming using on-line techniques involved considerable human interaction. For example, teaching it a job, saving and retrieving a trajectory when required. This feature is equally important in off-line programming. The vocabulary of commands provided by the language must be easy to type and understand. Unfortunately, most of the currently available robot programming languages are not easy to understand from the user point of view, since they are designed by and for the people who are familiar with the computers.

MORRISSEY [1985] has discussed the importance of user friendly robot programming language with reference to a language known as SAVVY. This author emphasises that majority of the robot users are not familiar with any computer programming language and are certainly not likely to be familiar with hexadecimal addresses, baud rates, arrays etc. The user would normally expect to be able to write a useful program in a very short time. For this reason the programming languages must be very easy to learn, avoid any complicated commands and the declaration of arrays or variables etc. be as self explanatory as possible. The language must have straight English syntax and must provide meaningful error messages when necessary. The interactive languages also improve human robot interaction as the user can type a command and be able to test immediately whether it works or not and whether it performs its designated action.

4.3.7 MATHEMATICAL LIBRARY

Since off-line programming systems involves performing complex mathematical calculations the language must provide a complete library of functions such as square, square root and trigonometric functions, otherwise a considerable amount of time may be wasted in implementing them by the user. These functions should, if needed, be available as an extension to the language for special applications. Robot programming involves the solving of complex equations using these functions. Thus the implementation of these functions must be efficient to improve the speed of the system.

4.3.8 WORLD MODELLING

World modelling involves the description of geometric and physical properties of the objects and their interelations in the working environment of the robot. This kind of information is essential for robots which require to sense their environment so that they can, independently plan the sequence of their operations upon receipt of one command from their operator. This type of programming is more commonly known as task level programming which involves three components viz (i) world modelling which is a data base representing robot and objects in its environment (ii) the task specification which is a command used to describe the task in a high level language (iii) robot program synthesis which is a process of transformation of task into robot level program . World modelling is an essential component of task level programming, but is not required where robot actions have been explicitly specified by the operator.

The task level programming is used by AI researchers, since the programming requires data which specifies the type, size, location and numerous other physical characteristics such as appearance, type of surfaces etc. This requires the setting up of a data base for the robot so that when it receives information about its surroundings from the sensors, it can consult this data base (model) and infer the physical properties of the object, its interrelation with other objects.

4.3.9 ON-LINE PROGRAMMING FACILITY

There is no doubt that programming a robot off-line has many advantages, however in an industrial environment circumstances may arise when it may be desirable to program them on-line. This may be due to the fact that a particular task is very difficult to model mathematically, the textual program is complex or the user need to transfer his skill by moving robot arm very skilfully through a particular path perhaps at a speed which can be controlled by intuition only. Hence a programming language must provide language constructs so as to permit robot use in order to teach a job manually. It should also offer facilities for saving such tasks perhaps in secondary memory and retrieving it when desired.

CHAPTER --5 FORTH

5.1 HISTORICAL BACKGROUND

Forth, a high level computer language, was invented by MOORE [1974], for real time programming, while working at The National Radio Astronomy Observatory, Virginia, U.S.A. He was looking for a language, suitable for use on minicomputers, which is fast, requires less memory and is flexible so as to control a radio telescope. The word FORTH, originated from FOURTH, since Moore wanted to call it a Fourth generation language, but the computer IBM-311, permitted only 5 characters, hence the truncated name FORTH. The first professional application of Forth was in 1971 on a Honeywell 316 minicomputer and its purpose was to collect Astronomical data at National Radio Astronomy Observatory (NRAO). In 1973, Moore founded his company for the development of Forth Programs known as Forth Inc. Since its introduction, Forth has steadily gained in popularity, particularly for use on micros. There is a Forth-Interest-Group (FIG), which is essentially a users' group and its aim is to increase the knowledge of Forth. There is a Forth standards team, the latest is Forth-83, which has superseded Forth-79 standard. There are different versions of Forth, which are suitable for many and diverse applications such as Poly-Forth, FIG-Forth, MMS-Forth etc.

5.2 FORTH APPLICATIONS

Forth is lately gaining in popularity as more and more programmers are realising its power and advantages over conventional languages. Forth can be implemented on most of the popular micros which are commercially available such as Intel, Motorola, Zilog, Texas, Commadore Pet, IBM PC etc. Atari [SALMAN et al. 1984] has developed a special version of Forth called Game-Forth for TV games. The film industry has used Forth to control cameras for producing films like "Star Wars", "Battle Beyond the Stars" etc. The use of Forth in controlling and monitoring the Hopkins Ultra-Violet Telescope, in space shuttle experiments is discussed by BALLARD [1984]. VAN BREDA and PARKER [1983] have described the use of Forth for the collection of data at the Royal Greenwich Observatory using Motorola 6800 and 6809 microprocessors. HARPER [1983] has described the role of Forth in the development of a multichannel analyser for low illumination level image detection in astronomy and space research.

Forth has also been used in multi-tasking applications [BUTTERFIELD 1984] in which a microcomputer performs more than one task under programmed conditions. WEISS [1984] has described applications of Forth in special areas which include Artificial-Intelligence, Diesel Electric Locomotive Trouble Shooting Air (Delta) expert systems, data-base systems like SAVVY (which is a natural language system for generating application programs) and SIMPLEX (an integrated data-base system with a Pascal-like interpretive language), word processing, Macintosh-like windows and graphics. TULSKIE and DIMEO [1983] have given a brief summary of a special version of Forth known as FORC, for Robotic Control. These authors claim that FORC offers all the convenience of a high level language but retains the speed of assembly language. LAGERGREN [1983] has described the use of Forth in monitoring the speed of tow boats, in order to determine the fuel consumption. LOTSPIECH and RUEHLE [1984], described the generalised interrupt handler for Intel 8085 microprocessor. BRODIE [1981], in the introduction of his excellent book, refers to applications in such diverse fields as cardiac monitoring, automotive ignition analysis, the measurement of moisture content of grains and baggage handling by a major U.S. airline. MACINTYRE (a,b) [1985], described the use of Forth in the automation of a laboratory using MMS-Forth.

5.3 FORTH AND ROBOTICS

Forth provides a complete programming environment for software development. Forth has the following important features which make it suitable language for a real time programming including robotics :

(i) Interactivity

(ii) Extensibility

(iii) Compilation

(iv) Editor

(v) Assembler

(vi) Secondary memory

(vii) Operating System

(viii) Transportability

(ix) Execution Speed

(x) Compactness

(xi) Direct memory access

5.3.1 INTERACTIVITY

Forth is an interactive type of language[HEDLEY 1981] which means that the commands, known as words in Forth, are executed as soon as they are entered at the keyboard. For example by entering $ 41 Emit, the output will be (ASCII character) letter A. This feature of Forth renders it much easier to debug since each instruction can be tested immediately and reflects the modular approach of the language.

5.3.2 EXTENSIBILITY

Forth has a vocabulary of certain core words in its dictionary. In addition, it allows the user to write application oriented words which can be added to the dictionary or kept separately in a secondary memory such as disc or tape, for loading into Forth when required.

5.3.3 COMPILATION

Forth has a built-in compiler and every newly defined word is complied into dictionary. Each newly complied word, is defined in terms of the previously defined words. For example, Forth words * (multiply), and . (print) can be combined to form a new word which can be defined as follows:

```
: MLTPLY-PRNT  (n1,n2 --------)

*  .  ;
```

Once this word is compiled, it will perform the function of these two words on execution.

5.3.4 EDITOR

Forth provides an editor, which allows the user to edit and change a program as and when necessary. The editor itself is a vocabulary of Forth words, which are usually stored on a disc and can be loaded into memory.

5.3.5 ASSEMBLER

Forth often has a built-in assembler, which allows the user to define assembler mnemonics in machine code which can be used in the same way as other Forth words. There are two special words in Forth for this purpose known as CODE and NEXT, the former being equivalent to a Colon (:) and later to a semi-colon (;), (as defined in section 5.4.11). For example, SEI, is a mnemonic in 6809 processor, to set the interrupt mask bit in its condition code register and its machine code is $ 0F. The Forth assembler code will be:

```
HEX CODE SEI 0F NEXT.
```

This facility is very useful in handling peripheral devices. The programmer can build an assembler vocabulary of suitable words for a particular application. This Forth code runs nearly as fast as machine code, thus increasing the execution speed of the language.

5.3.6 SECONDARY MEMORY

Forth provides words which allow blocks of memory to be loaded from mass storage devices such as disc . It provides a small area of memory called Buffers, which in Forth used in this research is of 2K bytes. Forth organises its mass storage memory into screens of 1024 characters so that there are 16 lines of 64 characters per line on each screen and each screen is numbered. The Forth word LOAD, loads the contents of a specified screen (block) into memory. These words will now become part of the Forth dictionary during current session. This feature of Forth, which makes mass storage memory to appear as part of Forth dictionary, is known as virtual memory. If the user wishes to load more screens than the number of block buffers available on the system, the block which was loaded first of all will be stored back into secondary memory before bringing in the new block.

5.3.7 OPERATING SYSTEM

Forth provides a complete operating system environment, since it provides the user with facilities like editing, assembling, interpreting, compiling, addressing mass memory storage (DOS) and handling peripherals. All these facilities are available in a comparatively very small memory size which is smaller than a typical operating system.

For example, the Forth operating system requires significantly less than 8k bytes of memory.

5.3.8 TRANSPORTABILITY

Forth has been implemented on almost all the popular micros and mini computers. Changing from one micro to another for instance, only requires the assembly section of the vocabulary to be rewritten.

5.3.9 EXECUTION SPEED

The execution speed of Forth is much faster than other high level languages (HLLs) and is 20-75% slower than equivalent assembly language programs [MANNONI 1980]. However, Forth does allow time critical sections of the program to be written in assembler form, which will run much faster .

5.3.10 COMPACTNESS

Forth is very compact language. For example, the memory occupied by core Forth words and complete operating system is usually less than 8K bytes.

5.3.11 DIRECT MEMORY ACCESS

Forth provides words like @ (pronounced as fetch), ! (pronounced as store) and DUMP etc which provide direct access to memory locations. This is an important feature of the language which is only provided by a few high level languages such as MODULA and C .

5.4 FORTH LANGUAGE

Forth language has five key elements [RATHER and MOORE 1979] : words, stack/stack operations, arithmetic/logical operations, extending the compiler and dictionary.

5.4.1 WORDS

A word (contrary to general computer terminology of 2-bytes) in Forth represents an instruction and is named as any combination of characters (1-31) and numbers even including punctuation marks. Each Forth word is separated by at least one space (delimiter). For example, the following are permissible Forth words:

+, *, ?, ., DUP, APPLE, PORT, ASIDE, IRQ-?, OVER, LIST etc.

In reality, Forth language is simply a vocabulary of such words.

5.4.2 STACK / STACK OPERATIONS

Forth usually uses a 16-bit push-down stack. The stack [HILLBURN and JULICH 1976] is a set of memory locations, which works on a last-in, first-out basis, abbreviated as LIFO. When a datum is placed on the stack, all the previous data items are moved down by one location. This is known as push operation. When a data is removed from the stack, all items move one location upwards, this is known as pop operation. Another point to note is that Forth uses post-fix (Reverse Polish Notation - RPN) notation instead of infix notation to which most users are more accustomed. The figure 5.1 shows the difference between infix and post-fix notation.

Infix notation	Postfix notation
6 * 3	6 3 *
3 + 2 + 7	3 2 7 + +
6 * (3 + 2)	6 3 2 + *

Fig.5.1 Infix/Post-Fix Notations.

Forth is a stack oriented language and has two types of stacks: the data stack and return stack. The data stack deals with data handling and the return stack is more often used by the Forth system to store some indices or pointers while executing Forth words. The typical use of return stack is in executing loops when indices are placed on the return stack. The user is however, allowed limited access to the return stack in order to move some parameters there. The essential condition is that any number placed on the return stack during the execution of a word must be removed before the end of definition otherwise Forth will crash. The return stack words are summarised in Appendix B (table B5).

Forth 83 provides certain stack manipulation words for the numbers placed on the stack so that they can be removed, duplicated or moved around to change their relative positions. These words include DUP, SWAP, ROT, OVER, PICK, ROLL etc. Some of these stack manipulation words are summarised in Appendix B (table B1).

It is customary to show stack effect before and after an operation by enclosure within parentheses. For example, the word DUP expects a 16 bit number n on the stack before execution and leaves two identical numbers (n n) on the stack at the end of its operation i.e

DUP (n -------- n, n)

5.4.3 NUMBERS AND ARITHMETIC OPERATORS

Forth allows the user to work in different number bases by storing an appropriate base number in a variable called BASE. For example, to change to base hexadecimal one can type:

```
16  BASE  !
```

The Forth word ! (store) saves 16 into variable called BASE. To change this base in to binary, one can type:

```
2  BASE  !
```

Thus it is much more easier in Forth to work in different number bases. It is also possible to convert a number in one base to another. For example, let us assume that the current base is 16 and by typing the following will leave 26 on the stack.

```
1A  DECIMAL    --------->26
```

Most versions of Forth provide the word HEX or HEXADECIMAL, which sets the current base to 16. In the same way the word BINARY sets the base to 2.

Forth provides arithmetic operator for the following types of numbers:

(a) 16 bit single precision integers (n).

(b) 16 bit unsigned single number (u).

(c) 32 bit double length integers (d)

(d) 32 bit unsigned double length numbers (ud).

Forth provides arithmetic operators such as +, -, *, / etc. for performing mathematical operations. These are summarised in appendix B (table B2).

A suitable combination of arithmetic and stack operators may allow solution of various algebraic expressions. For example, to calculate n1(n1 + n2), the value of n1 and n2 are needed on the stack. The word needed to evaluate this equation may be defined as follows:

```
: EQUATION  ( n1, n2 ----------n )

OVER                ( move second item to top )

+                   ( add n1 and n2 )

*                   ( multiply n1 by sum of n1 and n2 )

;                   ( end of definitions )
```

This could be shown by the following stack diagram:

operation	**stack**
	n1 n2
over	n1 n2 n1
+	n1 (n1+n2)
*	n1 * (n1+n2)

The above operators work only on single length numbers ranging between + 32767 to -32768 This means with 16 bits one can represent numbers from 0 to 65535. Some applications might require the handling of numbers which are outside this range.

For these circumstances Forth supplies double precision arithmetic and stack operators such as 2SWAP, 2DUP, D+ etc.These words are summarised in Appendix B (table B3).

5.4.4 MIXED LENGTH OPERATORS

Forth supplies some mixed length operators such as M+, M/, M* etc., of which the most common ones are summarised in Appendix B (table B4). Forth also supplies words like U. and U* which works on unsigned single length numbers.

5.4.5 FLOATING POINT VS FIXED POINT ARITHMETIC

Most computers allow the use of floating point arithmetic. However, Forth advocates the idea of fixed point arithmetic i.e by treating all numbers as integers, leaving up to the programmer to know the exact position of the decimal point. However, both types of number representations have advantages and disadvantages. A Math co-processor can give considerable advantage when using floating point numbers.

5.4.6 INPUT OUTPUT (I/O) WORDS

Forth provides output words for printing characters. The most important character output word is EMIT, which sends the character represented by a number on the stack to the VDU. For example,

```
65 EMIT ----------> A

48 EMIT ----------> 1
```

For the control of input from the keyboard, Forth provides words such as KEY, ?KEY, EXPECT, WORD etc. The word KEY allows the user to put ASCII characters on the stack from the keyboard. For example, by typing word KEY followed by return, nothing appears to happen as Forth is waiting for the user to press a key from the keyboard. On receiving a key, it will leave on the stack its equivalent ASCII value. The main use of this word is to wait for the user to press a key to proceed so that the ASCII value entered by the user can be used for example, to branch off within the next part of the program. A related word to KEY is ?KEY, available on some systems which allows execution of program to continue until a key is depressed. The word EXPECT, allows a specified a number of characters to be received from the keyboard and also expects the starting address where these characters are to be stored. Another useful input word is WORD, which reads a number from the input, using the character (commonly blank, ASCII 32) as a delimiter. It leaves the address of the string, containing the character count in the first byte on the stack.

5.4.7 COMPARISON OPERATORS AND BRANCHING

Forth provides comparison operators such as >, <, = etc. for decision making in the algorithm. These operators carry out comparison with numbers on top of the stack and leaves true (1) or false (0) flag on the stack. The value of this flag may be used to jump to different program sequences as desired. The common comparison operators are summarised in Appendix B (table B6) .

In order to branch the sequence of an operation, on the basis of information (flag) provided by the comparison operators, Forth provides the following two types of branch instructions:

IF ---------- THEN

IF ---------- ELSE ------------ THEN

These instructions can only be used in colon definitions. Forth word IF expects a flag on the stack . In case of IF --- THEN, a positive number or value 1 will allow the execution of Forth words followed by the IF and false flag will make it to jump to word/s after THEN. The use of IF ---- ELSE ------- THEN allows additional branching choice. The true flag preceeding IF allows the code to be executed placed between IF and ELSE and then jump to code after THEN. However a zero flag, will allow it to skip execution after ELSE and on completion of which it will continue with the code placed after THEN.

Logical operators like AND , OR etc. can be used to combine the effect of more than one comparison operator to prepare a combined argument for the IF statement. For example, a word to calculate the area of a rectangle can be defined in the following way :

```
: AREA    ( L W-------------- area or error )
DUP              ( duplicate width )
 >0             ( check if width is  positive )
OVER             ( copy length to the top of stack )
>0              ( check if length is positive )
AND              ( combine 2 flags with AND )
IF              ( start branching )
*               ( calculate area )
ELSE             ( if false flag )
2DROP             ( clear stack )
." error "         ( print error message )
THEN
```

```
;                 ( end of definition )
```

Forth provides words like EXIT, QUIT, ABORT to prevent Forth to continue execution of words and return the control to keyboard. This is more commonly used along with IF THEN type of words. The word EXIT terminates the execution of the current word and proceeds to branch off to execute next word in the sequence. The word QUIT completely quits i.e it ignores the rest of the words and returns control to the keyboard. ABORT, on the other hand does the same job as the QUIT, but clears the stack as well. These words are useful in error checking such as preventing division by zero and branches programs flow to other words or completely quitting the Forth.

5.4.8 LOOPS

Loops provides a facility for the programmer to repeat a certain section of the code by a prespecified number of times. For ordinary repetitive actions, Forth provides two types of words of constructs known as (a) definite and (b) indefinite loops.

(a) DEFINITE LOOPS

The definite looping is achieved by :

DO-------------------------LOOP

The word DO expects two parameters on the stack i.e the limit value and the index value as shown below:

```
n1   n2    DO   action    LOOP
```

where n1 and n2 are limit and index values respectively

This word must be used inside a colon definition (as defined in section 5.4.11) and its use can be illustrated by the following word:

```
: LOOPING  5  0  DO  I . LOOP ;
```

This word on execution will print numbers 0 through 4 but not 5.

In the above example, the loop index has been increasing by 1 every time the action within DO------LOOP is repeated. This can be changed by using +LOOP, as shown in the following examples:

```
:  HEXAJUMP  24  0  DO  I .  6  +LOOP ;
```

This word will print numbers 0, 6, 12, 18.

(b) INDEFINITE LOOPS

Indefinite loops allow the programmer to repeat a certain code indefinitely until a specified condition is met. The Forth words to achieve this include:

BEGIN------------ UNTIL

BEGIN------------- WHILE-----UNTIL

These words must be enclosed within a colon definition. The operation of the BEGIN----UNTIL loop is illustrated below:

```
BEGIN    ACTION    f UNTIL
```

The word BEGIN will start the execution of action which must leave a flag (f) for UNTIL. If this flag is false (0), the loop will continue, whereas, a true flag (1) will terminate the loop.

Another indefinite loop word is:

BEGIN -------------- WHILE --------- REPEAT

In this type of loop, the test occurs before WHILE. If the flag is true, the execution will continue to the end of the loop and transfers the control to BEGIN in order to start the loop again otherwise the loop will terminate.

5.4.9 FORTH DICTIONARY AND VOCABULARIES

All the words defined in Forth are saved in a linear list of words known as dictionary in the order in which they are compiled. During search time, this dictionary is searched backward starting with the latest word. It grows from low to high address.

Vocabulary is a collection of related words which is a subset of the whole language in the dictionary. The main vocabulary is named FORTH. Some systems also provide additional vocabularies such as EDITOR and ASSEMBLER. In fact, first word in the dictionary is Forth which is actually the name of Forth vocabulary. It has the effect of calling the basic vocabulary of Forth and is in effect executed on start up. Forth allows the user to create as many vocabularies as necessary. A new vocabulary can be created by a defining word known as vocabulary as follows:

```
VOCABULARY    ROBOT
```

This will create new dictionary known as ROBOT. In order to enter new words in to this dictionary, it has to be made current which is achieved by typing:

```
ROBOT    DEFINITIONS
```

There are two user variables, CONTEXT and CURRENT which are associated with vocabularies. The CONTEXT allows a particular vocabulary to be searched first. For example, by typing ROBOT will allow this vocabulary to be searched first. If a word is not found in this vocabulary, then the interpreter will search vocabulary Forth which is a default vocabulary. CURRENT is another user variable which points to the vocabulary in to which new words are currently being added. The word DEFINITIONS may be defined :

```
:   DEFINITIONS   CONTEXT @   CURRENT  !   ;
```

It is important to realise that whilst the words are added in to dictionary as they are compiled, during search time, the only words which will be searched will be those present in the CONTEXT vocabulary. That is one reason, why one can have more than one word with same name in different vocabularies. It is also possible to define vocabularies within a vocabulary as illustrated below:

```
VOCABULARY    ROBOT          ( parent vocabulary )

ROBOT DEFINITIONS

VOCABULARY    SIMULATOR         ( sub-vocabulary of ROBOT )
```

5.4.10 FORTH DICTIONARY STRUCTURE

As described above, Forth saves all the words in a dictionary. Each word consists of two parts; the header which enables the word to be identified and the body which defines its contents.

(a) HEADER

The header of a word consists of two parts, the name field address (NFA) and the link field address (LFA). The NFA of a word contains character count and actual characters (1-31) which constitute it. The first byte of NFA also contains some additional information required by the interpreter during execution of a Forth word. Bits 1-5 contain word length (1-31 characters). Bit 6 is known as smudge bit which is used to indicate whether a Forth word is compiled without an error. Bit 7 is known as precedence bit which informs the interpreter whether this word is to be executed immediately. Bit 8 is used during dictionary search to locate the start of NFA of a word.

The LFA contains the address of the NFA of the previous word in the dictionary, which is used by Forth interpreter during a dictionary search. Thus LFA of each word is linked to the word defined before it. This type of arrangement is known as threaded-coding, which renders dictionary search faster. At search time, the interpreter starts with most recently defined word and follows the "chain" backwards, using the address in each link field to locate the next definition in reverse order. Fig. 5.2 shows dictionary entry of a Forth word ROT.

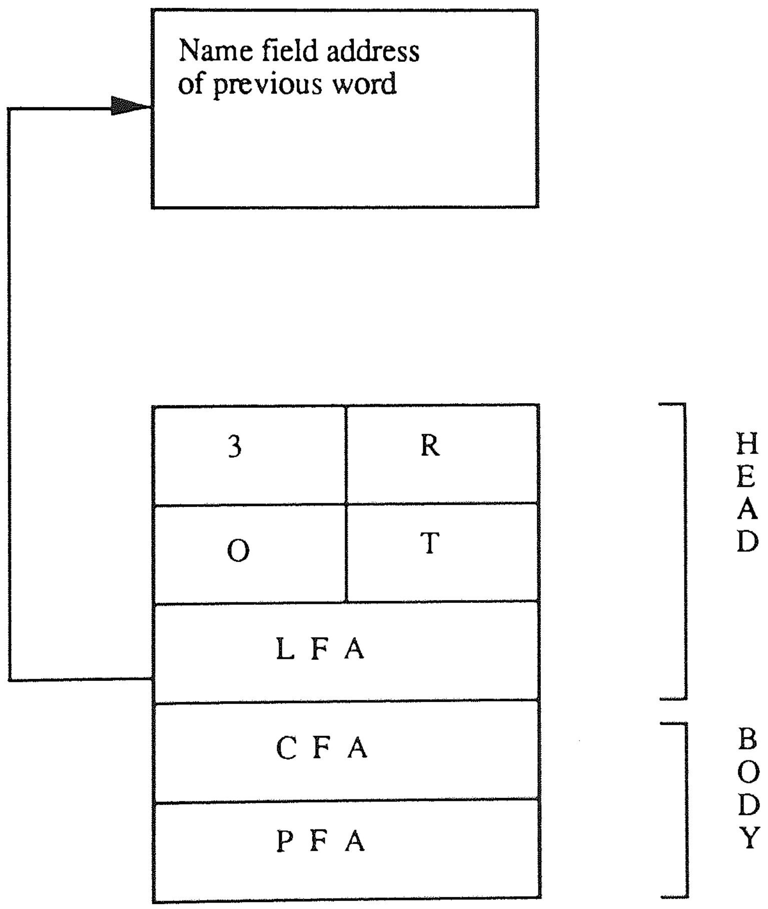

Fig.5.2 Dictionary entry of Word ROT.

(b) BODY

The body of a Forth word also consists of two parts, code field address (CFA) and parameter field address (PFA). The CFA of a word, distinguishes between a variable, a constant or a colon definition. This is the beginning address of an instruction (word) that is loaded into the instruction pointer when the word is to be executed. For example, in the case of a variable, the CFA points to the code that places the address of the variable on the stack whilst in case of constant, it points to the code which pushes the value of the constant on to the stack. In the case of a colon definition, CFA points to the code that executes the rest of the definition.

The PFA of a word contains the beginning address of the definition of the word. It also contains the PFA's of other words which constitute this word.

Forth provides certain words to get addresses of the words compiled in dictionary. The word ' (pronounced as tick) finds PFA of the word in the dictionary. The word FIND returns CFA of a word in the dictionary. Forth 83 provides certain words such as >BODY, >NAME, >LINK etc. which converts addresses of the Forth words and some of them are summarised in Appendix B (table B7).

5.4.11 EXTENDING THE DICTIONARY

One of the most striking features of Forth is that the programmer can define new words according to a particular application. There are five ways of extending the Forth dictionary:

(a) COLON

(b) CODE

(c) VARIABLES/CONSTANTS

(d) VOCABULARY

(e) DEFINING WORDS (CREATE and DOES>)

(a) COLON (:)

The definition of a new word begins with a colon (:), followed by its heading (name), then its contents and ends with a semi-colon (;). Consider the following example of a word known as DVD-PRNT:

:	DVD-PRNT	(n1,n2 ---- n)	/ .	;
Beginning of compilation	name	stack notation	contents	End of compilation

This word will divide second number by the top number on the stack and print the result.

(b) CODE

As described in section 4.3.5, the word CODE is used to define assembler mnemonic words in its terms of their machine codes. For example, the word CLI in 6809 system can be defined in Forth like this:

```
HEX   CODE   CLI   0E   NEXT
```

Now CLI becomes a part of Forth vocabulary and can be used like other words. The word CODE is equivalent to colon and NEXT is equivalent to ; in Forth.

(c) VARIABLES/CONSTANTS

(i) Variables

Variables are used to create new words whose contents can be changed when required. A typical example of the use of variable is a Forth word BASE, which contains the number base that is currently being used. In order to change from one base to another, the number base stored in this variable is changed. For example:

```
: Decimal      10 Base ! ;

: HEX          16 Base ! ;
: Binary        2 Base ! ;
```

Variables produce a similar dictionary entry to other words and allot two bytes to store its value. When word variable is invoked, it places its address on the stack.

(ii) CONSTANTS

Constants are similar to variables, except that formers are used to store the values that will not change. For example, a constant known as limit whose value is 500 can be defined as follows:

```
500 Constant Limit
```

(d) VOCABULARY

As described above Forth allows the grouping of similar words into a vocabulary. In a current vocabulary the compiler can only search for certain words and allows the user to branch off to different vocabularies. The main advantage of this facility is to reduce considerably the search time by looking for a limited number of words.

(e) DEFINING WORDS (CREATE and DOES>)

One of the unique features of Forth is its ability to define a word which can be used to define a series of related words. The newly defined words are compiled like other Forth words in the dictionary. A defining word on execution creates a dictionary header for the created word along with other necessary information which is needed for the new word to be executed. This is a very powerful feature of Forth. The basic Forth word to create a defining word is CREATE. For example:

```
CREATE  FRED
```

This will create a header for the word FRED in the dictionary and when this word is executed its address is placed on the stack. In fact the words like variable and constant are also defined using CREATE. For example:

```
:  VARIABLE  CREATE  0  , ;
```

Keying in VARIABLE TEMPERATURE creates a new dictionary entry called TEMPERATURE and will initialise to 0, its first two bytes. The user can define any number of variables using this defining word. Another word which is used in conjunction with CREATE is DOES>. This word specifies the execution behaviour (run time) of the newly defined word. The word CONSTANT may be defined :

```
: CONSTANT  CREATE  ,  DOES>          @     ;

      compile time behaviour             run time behaviour
```

At compile time, the CONSTANT expects the name of new word and its contents which are saved by the word compile (,) into the address of new word. For example: 1000 CONSTANT 1K

This will create a constant 1K and compile 1000 in its address. After compilation when word 1K is executed (i.e at run time) its address is put on the stack automatically, but word @ after DOES> will read the value saved at this address and will place it on the stack.

The use of defining words can be illustrated for defining arrays. A defining word to create one dimensional arrays can be defined as follows:

```
: 1D-ARRAY   ( n--------)

CREATE                 ( create dictionary entry )

2 *                 ( calculate byte offset)
```

```
ALLOT          ( move dictionary pointer)

DOES>          ( run time behaviour )

SWAP           ( move element number to top )

2 *            ( calculate byte offset )

+              ( address + offset )

;              ( end of definition )
```

This word is known as the defining word since it can be used to define different one dimensional arrays. The new arrays can be defined this way:

```
10  1D-ARRAY  TEMP

15  1D-ARRAY  PRESSURE

40  1D-ARRAY  VALUES
```

At execution, each array expects the element number on the stack which is multiplied by 2 to make it a byte offset and is then added to the start address of the parameter field. This returns corresponding address of the element in the array. To save some values in the array one can type:

```
5    0    TEMP    !      ( will store 5 in the first element )

20   1    TEMP    !      ( will store 20 in to second element )
```

Later, to recover these values, one can type:

```
1  TEMP @      ------->  20
```

The array index must be on the top of the stack before executing TEMP. It must also be within the range 0 to 9 in this example, since the program contains no checks upon the range of the index. However, range checking can be integrated in to the definition of 1D-ARRAY, if necessary.

5.4.12 FORTH EDITOR

Editor is used to enter and alter programs. For this purpose, Forth supplies an EDITOR which is a vocabulary of related Forth words made available to the programmer. In order to conserve memory, the Forth editor is kept on a disc and can be loaded when necessary by keying in:

n LOAD ------ where n is the start block number of the editor.

This will compile all the editor words in to a vocabulary known as EDITOR. Forth, unlike most common high level languages does not save information in files. Instead, Forth words are stored in units of 1024 characters (1K) known as blocks. The user can list a specified block on the screen, and load its contents into the computer memory with suitable words. For example:

```
75  LIST
```

This will list the contents of block number 75 onto the screen. The Forth word LIST stores the block number in a variable called SCR and then transfers its contents to a buffer. Another command word with block is LOAD which is used like this:

75 LOAD This word will send block 75 to the text interpreter via the input stream which will compile all definitions into the dictionary.

The Forth editor is not standard and different versions of Forth use different editors, but they all include facilities to add, delete, insert words, clear screen, copy screens etc. A block of screen is made available to the user with line numbers ranging from 0-16 and each line can have up to 64 characters. In addition to this, 64 bytes of memory is made available above PAD and is referred to as scratch pad memory. This is used to store a line which is being deleted or added so that it can at later stage be re-inserted or replaced into a block buffer. At the end of editing in order to transfer this change back to disc, Forth provides a word called FLUSH or SAVE-BUFFERS which are synonymous.

Forth editor uses buffers for editing. Most common versions provide two buffers of 1024 characters each. The word BLOCK moves a specified block number into one of these buffers. For example:

```
20 BLOCK
```

This word will transfer the contents of BLOCK number 20 into buffer which is least recently used and hence leaves its starting address on the stack. The contents of a block can be examined by typing:

```
20 BLOCK 1024 TYPE
```

The availability of such buffers helps to treat secondary memory as a virtual memory. Since the contents of the block most recently used was saved in block buffers, the system does not have to read from the disc each time. This speeds up reading and writing to disc, particularly if same data is being used frequently. Any changes made to the contents of a disc can be made permanent by a word known as UPDATE which marks the buffers to be stored on the disc before they are used again by setting a flag. The only difficulty with UPDATE, is that, if the buffers are not used again before the power is turned off, the modified contents will not be saved on the disc. To ensure that, user must type SAVE-BUFFERS or FLUSH before the end of the session as a precaution. Some versions of Forth, provide another word called EMPTY-BUFFERS, which unassigns the marked buffers which are not be saved on the disc. In some versions it fills the buffer with nulls or blanks (ASCII 32). However, this word is not required by Forth 83 standard.

5.4.13 INTERPRETATION, COMPILATION AND EXECUTION

Forth allows the user to enter a set of characters or numbers from the keyboard. All the input occurs via the input stream, a flow of numbers and words separated by spaces/blanks (ASCII 32). As soon as enter key is depressed the Forth acts upon them by the interpreter. Input from the keyboard or discs is interpreted in the same way. To receive terminal input, Forth is in an indefinite loop which is a part of the word QUIT. This word clears return stack, sets execute mode and expects terminal input until the input stream is not expanded and there is no error. If there is an error, QUIT passes control to ABORT which resets the system and starts again and comes out of the loop with a suitable message. If no error is detected, QUIT attempts to interpret the input stream by breaking it into separate words and numbers, which may be compiled, executed or placed on to the stack. After the interpretation of the input stream, the control is returned to the keyboard to receive new input. If the input is coming from terminal, then variable BLK contains 0, otherwise it is from the disc in which case BLK contains the block number of the disc which is sending the input. Each word or number must be separated by a null character [ASCII 32]. Forth then searches for it in the dictionary to see if it is a Forth word. It then checks contents of a variable called STATE. If the contents of this variable is 0, (i.e Forth is in compile mode) and word is in the dictionary, it is executed. However, if the contents of the variable STATE is 1, then, if the word is immediate, it is executed otherwise its CFA is compiled in to dictionary given by Forth word HERE, which points to the next available location in memory where new words are to be added. If the interpreter cannot locate it as a word, then it tries to convert it into a number in the current base. If the number is a valid number (i.e all digits are 1 less than the number in the current BASE), then this number is either compiled (if STATE = 1 i.e Forth is in compile mode) or placed on the stack (STATE = 0), otherwise an error message is returned and control is passed to ABORT again. If no errors are detected, during interpretation of the input or doing execution, a usual Forth message " O.K " is displayed on the screen.

Compilation is the process of entering new words, numbers or strings to the dictionary. As discussed previously, Forth provides words like colon :, CREATE, VARIABLE, CONSTANTS etc. to define new words. The : is the most common Forth word used to compile new words. Its action can be illustrated by the following word:

```
:  NEW        WORD1  WORD2   WORD3   ;
```

The : (colon) creates a dictionary entry and saves the value at HERE as its name field address (NFA). If this word is already in dictionary, it will display a suitable error or warning message. Then it finds NFA of the previous word in the dictionary and stores it in the link field address (LFA) of the word NEW, in order to allow a dictionary search at subsequent stage. This is called header of the new word. Then the address of the machine code needed to execute colon definitions is placed in the code field address (CFA) of the word NEW. Then CFA of the subsequent words in the definition i.e of WORD1, WORD2, WORD3 are saved in the parameter field address (PFA) of this word. The ; (semicolon) marks the end of the definition by placing the address of EXIT in the last position of the body of word NEW and also returning Forth to its execute mode by saving 0 in the variable STATE. The semicolon also checks to ensure that there are no compilation errors. If no error is detected, it sets the smudge bit so that the new word can be detected in dictionary search. However, in case of error, the dictionary pointers backs to where it was originally. Some Forth systems, leave the header in the dictionary and simply prevent its access by not setting smudge bit.

Forth allows a word to be executed during compilation by making it immediate. For this purpose, there is word known as IMMEDIATE. Consider the following example:

```
:   TEST1    ." compiling " ;  IMMEDIATE

:   TEST2   TEST1  ." compiled  "  ;
```

When TEST2 is being compiled, TEST1 being immediate word will be executed and will print the message " compiling " on the screen. However, when the word TEST2 is executed, the message will only be " compiled "

The word IMMEDIATE informs Forth to mark the most recently defined word to be executed immediately when it is encountered, even though Forth is in compile mode. This is achieved by setting the precedence bit in the name field of the word made as immediate. However, an immediate word can be forced to compile during compilation by using the word [COMPILE], pronounced as " bracket-compile ", as shown in the following example:

```
:  TEST3        [ COMPILE ]  TEST1   ." COMPILED " ;
```

During compilation of TEST3, no message will be displayed. However, on execution the following messages will be displayed:

```
compiling     compiled
```

The word [COMPILE], simply ignores the precedence bit. Another word which can cause execution during compilation is the use of [, pronounced as left bracket and], pronounced as right bracket which can be used as shown below:

```
:  TEST4   [ ." checking for compilation " ]  ."  I am  compiled  ";
```

This word during compilation will print the first message. After compilation, on execution of TEST4, will only print second message. The word [, simply changes Forth from compile mode to execution mode and], from execution to compile mode by changing the content of variable STATE.

Execution of a compile word in Forth involves using pointers such as the instruction (interpreter) pointer, word pointer, jump register and return stack. The first stage in executing a word is to load the CFA of the word in to the instruction pointer. If it is a colon definition, the contents of its CFA are saved on return stack and the instruction pointer loads the PFA of the first word which is a part of original word's definition. It increments the pointer to the next address, saves its contents on return stack and goes on to execute the next word. The process is repeated until each word is executed and is terminated by encountering EXIT.

5.4.14 FLEX FORTH IMPLEMENTATION

Forth used in this project was originally implemented by CADGE [1986]. Forth, as described previously, is a stack oriented language supporting two types of stack. MOTOROLA 6809 architecture which provides two hardware stack pointers is suitable for the implementation of Forth, and hence eliminates the need for software pointers. Forth was therefore written on 6809 assembler and uses Flex Operating System sub-routines for both terminal and disk input / output. The original implementation was according to Forth 79 standard whose limitations were :

(a) It only provided 16 bit arithmetic.

(b) No floating point was available.

(c) It only provided line editor.

As a result, it was updated by PARKES [1987] in an attempt to overcome these drawbacks and at the same time incorporate some features of Forth-83. This newer, updated version provided the following additional facilities:

(a) Double number words.

(b) Forth editor.

(c) Copy disk utility.

(d) Mixed length arithmetic operators.

(e) Extended precision.

(f) Floating point arithmetic.

(g) Graphic routines including a turtle graphic compiler.

The implementation detail for this latest implementation of Forth is given by PARKES [1987]. Forth memory map using FLEX operating system is given in Appendix C.

5.5 FORTH SOFTWARE DEVELOPMENT

From the software engineering point of view, program design in Forth is top-down, structured and interactive in nature. At the design stage, the programmer splits his problem into smaller units using a top-down approach, until small, easily definable independent modules known as words are arrived at, each of which can be written and tested individually. Program development and testing involves writing these low level words using a bottom-up approach. Subsequently these words are carefully combined together to form high level word/s. The process of combining existing words into higher level word/s is repeated, like building a pyramid, until the programmer is left with one or two very high level words to be executed for a particular application. Such a very high level abstract word on execution will perform the action of all the underlying words. The software design methodology is illustrated in figure 5.3 below:

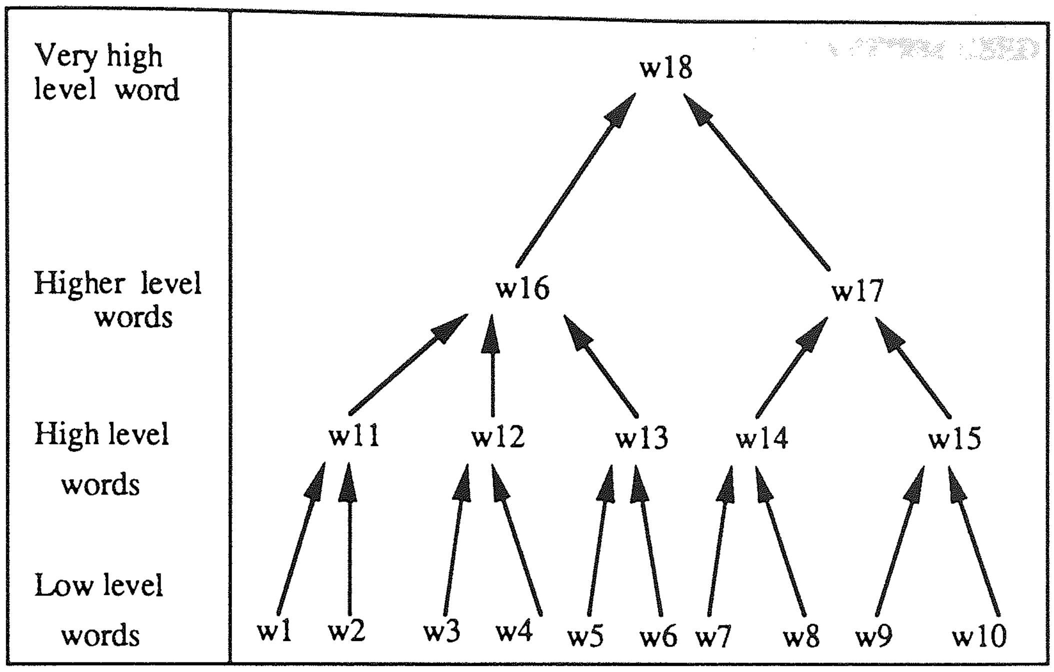

Fig. 5.3 Forth program development.

CHAPTER---6 DESCRIPTION OF THE DEVELOPMENT SYSTEM USED

6.1 ROBOT DEVELOPMENT SYSTEM

The robot supplied by System Control was equipped with an interface which did not work satisfactorily. Thus in order to drive the robot by the microcomputer it was necessary to build a replacement hardware along with other hardware. The robot system is shown in block diagram format in figures 6.1 and 6.2.

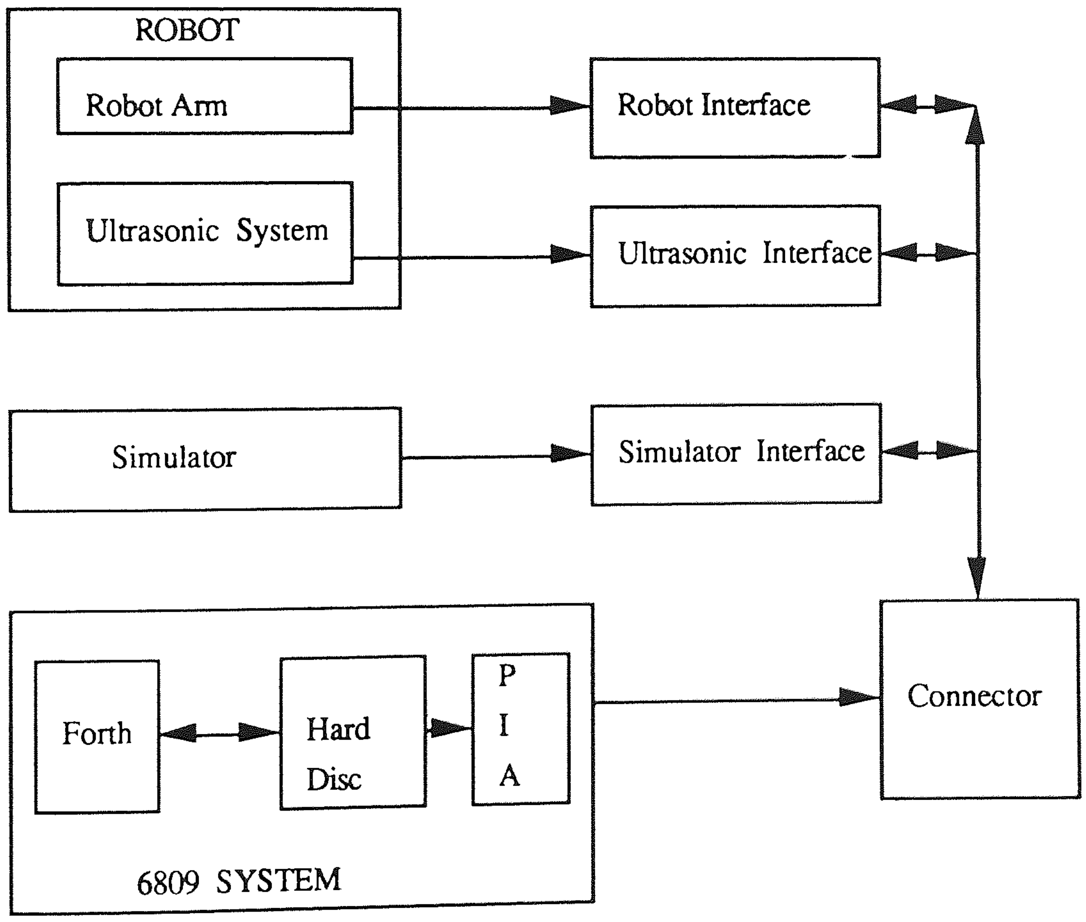

Figure 6.1 Robot development system.

This system includes the robotic arm, the ultrasonic system and a simulator each of which is interfaced to 6809 microcomputer through appropriate interfaces. The purpose of these interfaces is to allow the transfer of data between the microcomputer and the robotic arm with its associated units. These interface units convert information coming from the robot to the microcomputer into a compatible form for the computer and, during reverse transfer, to convert information from the computer to the robot system in an appropriate format.

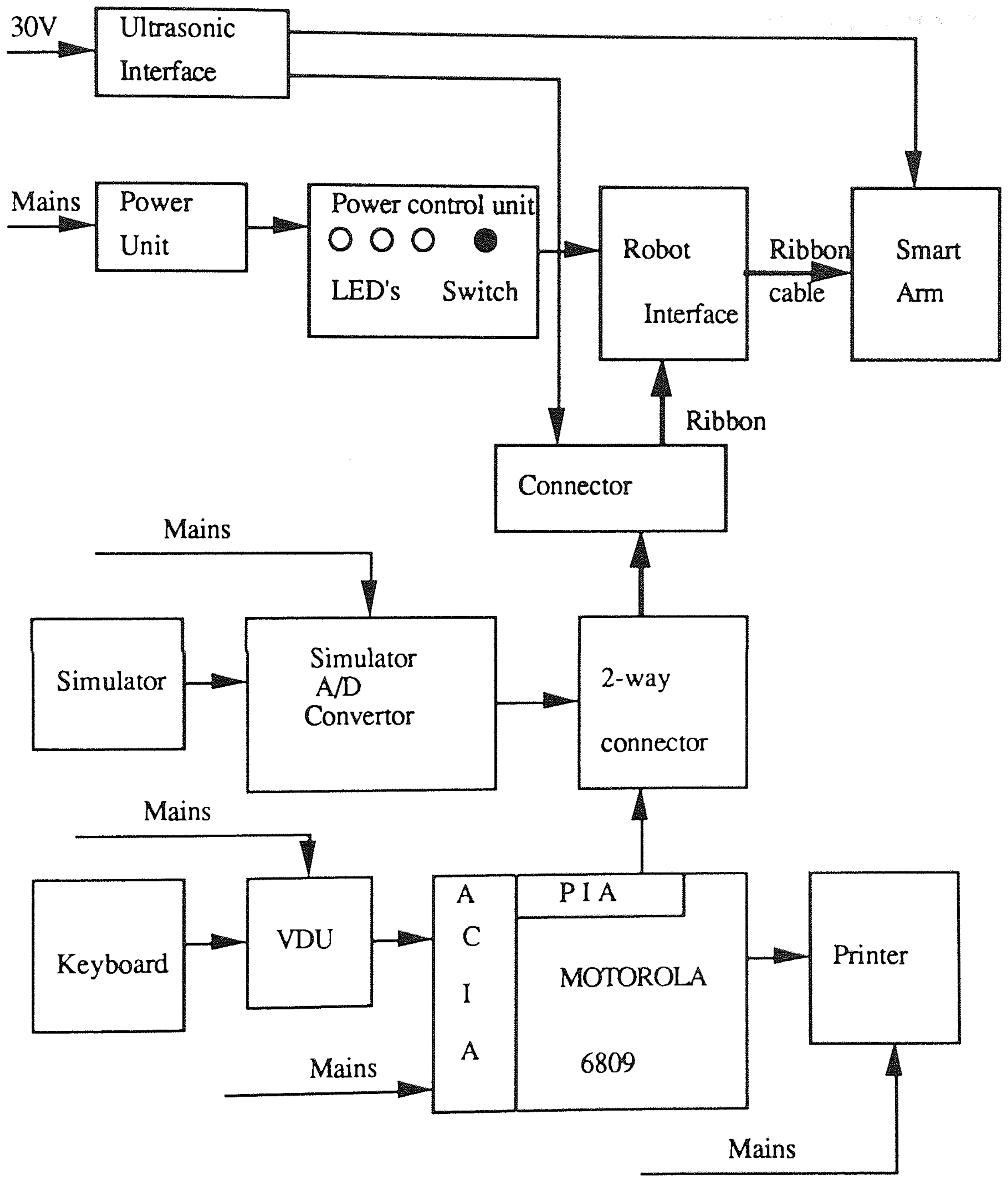

Figure 6.2 Robotic development system system showing connections.

6.2 6809 MICROPROCESSOR (MPU)

The MC6809 is an 8-bit microprocessor developed by Motorola. Even though it is an 8-bit processor, it contains many facilities that would only be expected to be found on 16-bit CPU'S. It is the successor to MC6800 and interfaces with all 6800 peripherals. The processor has many advanced features compared with its predecessor [MOTOROLA 1979] such as additional registers, instructions and addressing modes. Associated with the MPU are Random Access Memory (RAM), Read Only Memory (ROM) and programmable interfaces like Peripheral Interface Adapter (PIA), Asynchronous Communication Interface Adapter (ACIA). All of these devices use a single 5-V power supply. Each of the 6809 family components can be connected directly to the microprocessor address and data buses.

6.3 THE REGISTERS

MC6809 MPU has five 16 bit registers : X and Y index registers, user stack pointer (U), hardware stack pointer (S), and program counter (PC). Additionally, it has four 8 bit registers : direct page register (DP), condition code register (CC), accumulators A and B (A,B) as illustrated in figure 6.3 below:

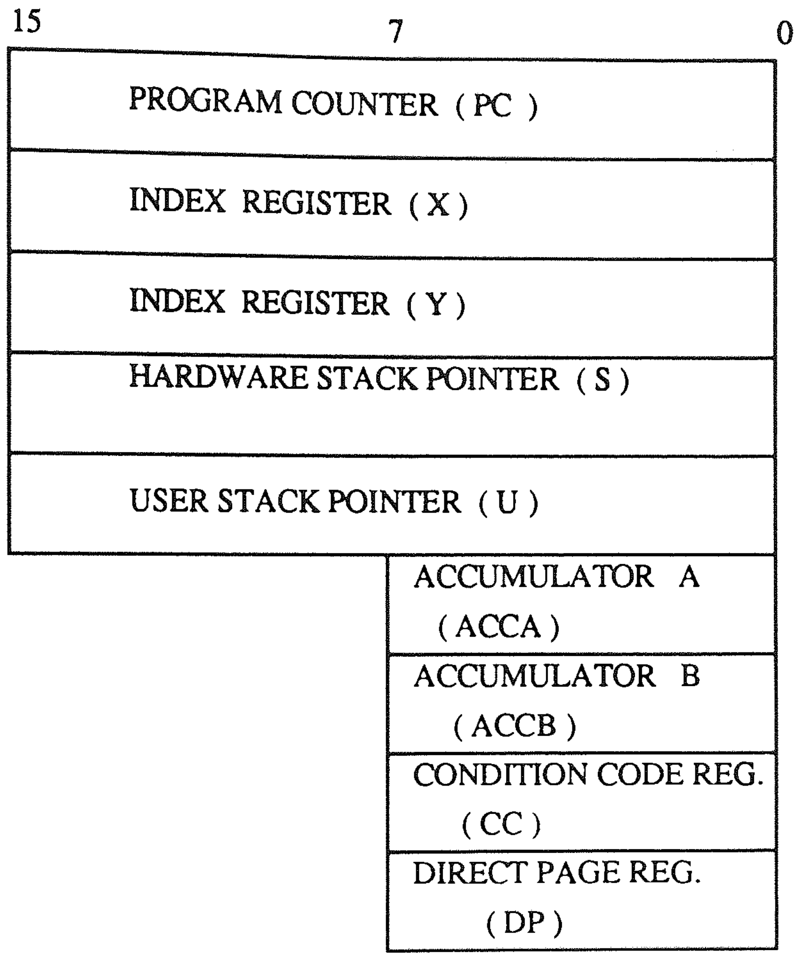

Fig.6.3 Showing Registers of the 6809.

6.3.1 ACCUMULATORS (A,B)

Registers A and B are 8 bit general purpose accumulators which are used for arithmetic calculations and data manipulation. The two accumulators can be concatenated to form a 16-bit register known as D register in which case the A register forms the most significant byte.

6.3.2 DIRECT PAGE REGISTER (DP)

The direct page register of the MC6809 is provided to enhance the Direct Addressing Mode. The content of this register appear at the higher address output (A8-A15) during direct addressing instruction execution. The direct page is concatenated with the byte following the direct mode op code to form a 16-bit effective address. This allows the direct mode to be used at any place in memory, under program control. In order to allow compatibility with MC6800, all the bits are cleared by RESET.

6.3.3 INDEX REGISTERS (X,Y)

MC6809 provides two identical 16-bit (two-byte) registers (X,Y) which are primarily used to modify addresses. These registers may be incremented, decremented, loaded, stored or compared by means of appropriate instructions. These registers are used for indexed mode of addressing. They provide a 16-bit address to be added to or subtracted from an optional offset for indexed instructions to generate an effective address of the instruction. This address may be used to point to data directly. As indexed mode of operations, provide automatic pre-increment and post-decrement options, these registers may be used to implement software stacks, queues and buffers.

6.3.4 PROGRAM COUNTER

PC is a 2-byte register that holds the address of the next location whose contents are to be fetched. As soon as the contents of a memory location are fetched, the Program Counter Contents are incremented and points to the next program location. Relative addressing is provided so that program counter can be used as an index register.

6.3.5 STACK POINTERS (U,S)

The stack pointer contains an address of a location where the status of the MPU registers may be stored under certain conditions, when it has to perform other functions such as during an interrupt or a branch to subroutine. The address which is in the SP is the starting address of sequential memory locations in Random access memory (RAM), where the contents of microprocessors registers will be stored. MC6809 provides two stack pointers: hardware stack pointer (S) and user stack pointer (U) both of which are 16-bit registers. They contain the address that points to the top of a push-down/ pop-up stack. The hardware stack pointer (S) is used by the hardware during sub-routine calls and interrupts to save sub-set or entire machine status. The user stack pointer (U) is exclusively controlled by the programmer to allow arguments to be passed to and from sub-routines. Both pointers allow data and machine state to be pushed on to the stack or pulled from the stack in a last-in first-out (LIFO) manner. The main difference between MC6800 and MC6809 stack pointer is whereas the former points to the next free location on the stack, the latter points to the top of the stack i.e the last byte placed on the stack. Both pointers support push and pull instructions. The push instruction decrements the stack pointer before the data is stored while the pull instruction increment the stack pointer after the data is recovered. Both the U and S registers have the same indexed-mode addressing capabilities as the X and Y index registers. This allow the MC6809 to be used efficiently as a stack processor, greatly enhancing its ability to support higher level languages and modular programming.

6.3.6 CONDITION CODE (STATUS) REGISTER (CC)

Condition Code is a 8-bit register which is used to test the results of certain instructions. The results are mainly used by the MPU for branch instructions. Such branches will occur according to the status of specific bits in this register. Figure 6.4 shows the diagram of the 6809 status register.

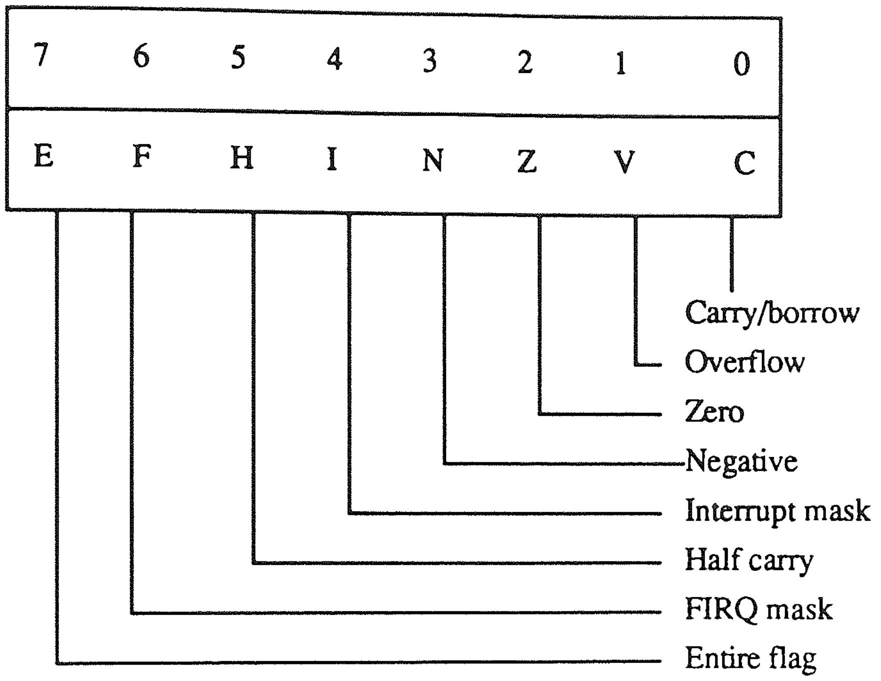

Fig.6.4 The Condition Code (Status) Register.

6.4 INTERFACING AND PERIPHERAL DEVICES

Microcomputers communicate with the outside world through Inputs/Outputs (I/O) devices or peripherals. The most common peripherals include visual display unit (VDU), paper printer, magnetic tape cassette, disc memories, keyboard, analog-to-digital (A/D) and digital-to-analog (D/A) converters etc. The process of connection of one or more of these devices to a computer is known as interfacing, as depicted in fig. 6.5. This requires a special hardware circuitry, the purpose of which is to allow [FALK 1974] the transfer of information between the microprocessor and a particular device such as VDU, keyboard and printer etc. Interface hardware converts the information coming from the device to the computer into a compatible format for the computer and, during a reverse transfer, to convert information from the computer to the device into the appropriate format. In addition, the interface must reconcile any timing difference between the MPU and the device.

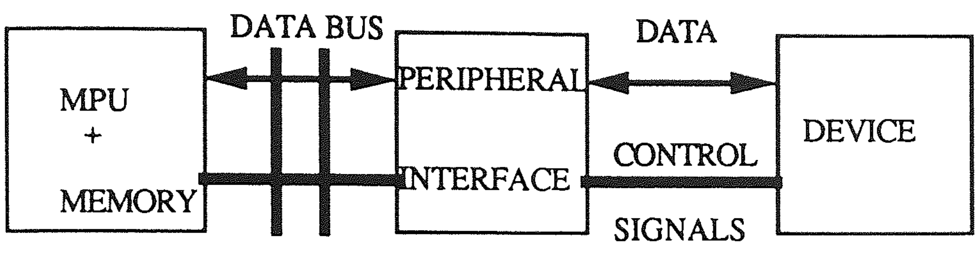

Fig.6.5 Block Diagram of a Typical Interface.

The 6809 microprocessor has mainly two programmable interface adaptors known as Peripheral Interface Adaptor (PIA) and Asynchronous Communications Interface Adaptor (ACIA). The former is used for parallel transfer of data and latter for serial transfer of data to or from the device. Both devices are flexible as their function can be changed by programming.

The 6809 microprocessor has memory-mapped I/O, which means that it treats the I/O devices in the same way as memory as shown in Fig.6.6. It is worth noting that PIA, ACIA, RAM and ROM all share the same data bus and address bus with the MPU.

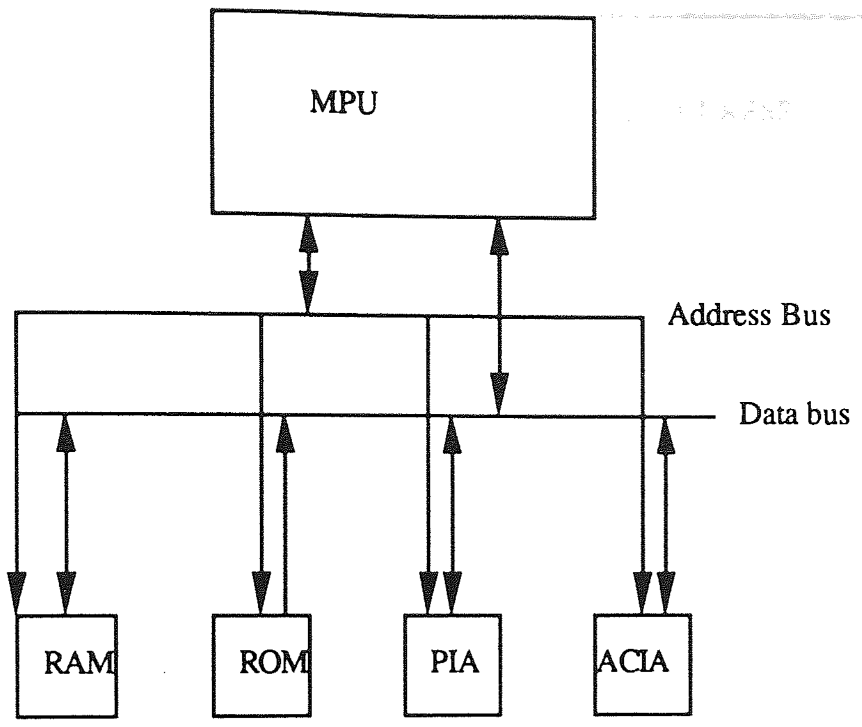

Fig.6.6 Memory-Mapped I/O System.

6.5 PORT ADDRESS

The 6809 system has 4 I/O ports for interfacing the MPU to its peripherals. Ports 0 and 1 are for general interfacing and ports 2 and 3 are dedicated for controlling the disc drive and the printer respectively. Figure 6.7 below summarises the port addresses:-

PORT NUMBER	MEMORY ADDRESS (Hex)
0	E040-E043
1	E020-E023
2	E068-E06B
3	E06C-E06F

Figure 6.7 showing port addresses in memory.

6.6 PERIPHERAL INTERFACE ADAPTOR (PIA)

6.6.1 INTRODUCTION

PIA [MOTOROLA 1979] is a special piece of hardware which allows peripherals devices to be interfaced to the MPU. It is programmable, and is used for the parallel transfer of data to and from a device and can be used in interrupt driven or non-interrupt driven modes. The PIA consists of two ports A and B, which are independent of each other and are almost identical in their mode of operation. It communicates with the MPU via an 8-bit bidirectional databus, three chip select lines, two interrupt request lines, a read/write line, an enable line and a reset line (fig.6.8).

The PIA has 16 peripheral data lines (PA0-PA7 and PB0- PB7) for transfer of data between PIA and the peripheral devices. These lines are organised into two separate sets of 8, referred to as set A and B, corresponding to the ports A and B, of the PIA. These lines can be programmed to be all input or all output or any desired combination of both. Each port of the PIA, consists of three registers known as, Peripheral Data Register (DR), Data Direction Register (DDR) and Control Register(CR) , as shown in Block diagram of the PIA in Fig.6.8 below:

Fig.6.8 Block Diagram of MC6821 PIA Chip (Motorola's Manual).

6.6.2 PERIPHERAL DATA REGISTER (DR)

This register, also known as the Data Register (DR), is an 8-bit register in the PIA which holds the current data to be input to or output from the PIA. In addition, attached to this register on each port are two peripheral control lines CA1/CB1 and CA2/CB2. The CA1/CB1 is interrupt input line only, active high to low or low to high transitions on this line set the interrupt flag bit-7 of the control register. The control line CA2/CB2 on DR may be programmed to act as interrupt input line or output line by setting the appropriate bits of the control register.

6.6.3 DATA DIRECTION REGISTER (DDR)

This is also a 8-bit register, the purpose of which is to decide whether each bit in data register is to be used as an input or output. For example, storing 1 or 0 in a particular bit of DDR will make the corresponding line of the DR as an output or an input respectively.

6.6.4 CONTROL REGISTER (CR)

The two identical control registers A and B, one on each side of the PIA are also 8-bits wide. These control registers allow the MPU to control the operation of the four peripheral control lines CA1, CA2, CB1 and CB2 and also the direction of the flow of data from the peripheral. The function of the individual bits of the CR as shown in Fig.6.9 below:

7	6	5	4	3	2	1	0
IRQ1	IRQ2	CA2/CB2			DR/ DDR	CA1/CB1	

Fig.6.9 Control Register (CRA).

6.6.5 ADDRESSING THE PIA's REGISTERS

Each PIA has 3 registers on each side. But only four addresses are allocated for them. Each control register has its own address. The data register (DR) and data direction register (DDR) share one address as shown in fig. 6.10 below:

SIDE	REGISTER	ADDRESS
A	DRA/DDRA	E040
	CRA	E041
B	DRB/DDRB	E042
	CRB	E043

Fig.6.10 Addresses of PIA Registers (Port 0).

The initial configuration of PIA involves writing 0 into bit-2 of CR. This selects DDR. By writing a suitable number into the DDR, the direction of data lines is selected. The configuration is completed by storing 1 in bit-2 of the CR, which selects the data register.

6.7 FLEX DISC OPERATING SYSTEM

The FLEX Operating System was released by Technical systems Consultants Inc. (TSC) in 1978. The Operating System is comprised of three main parts, the File Management System (FMS), the Disk Operating System (DOS) and Utility Command Set (UCS). The FMS handles the allocation and removal of files on disks. All of the file space is allocated dynamically, and the space used by a file can be reused immediately on deletion of that file.

The UCS contains many utility programs which reside on disk and are only loaded as and when they are required. This, beside saving on memory, allows easy expansion of FLEX by just creating a new command file either with the new utility or by an application contained within.

The DOS performs functions such as terminal input/output, file specification parsing, command argument parsing and error reporting. It provides the communications link between the user and the FMS.

Programs developed in 6809 machine code on the FLEX system can make use of calls available to the FLEX operating system to handle such things as disk and keyboard input/output. The standard FLEX editor is the TSC text editing system which is a line based editor. It supports such commands as those to insert, delete, replace and find, single or groups of lines and characters. The FLEX operating system also supports the TSC 6809 Mnemonic Assembler which will accept all the standard 6809 Mnemonics. It will assemble a file with source code of any length so long as the memory can accommodate the symbol table. The assembler is of the two pass type and produces output in the form of a binary disk file. Further information about the FLEX operating system and accompanying editor and assembler can be found in the FLEX users Manual [1979] and FLEX programmer's Manual [1979].

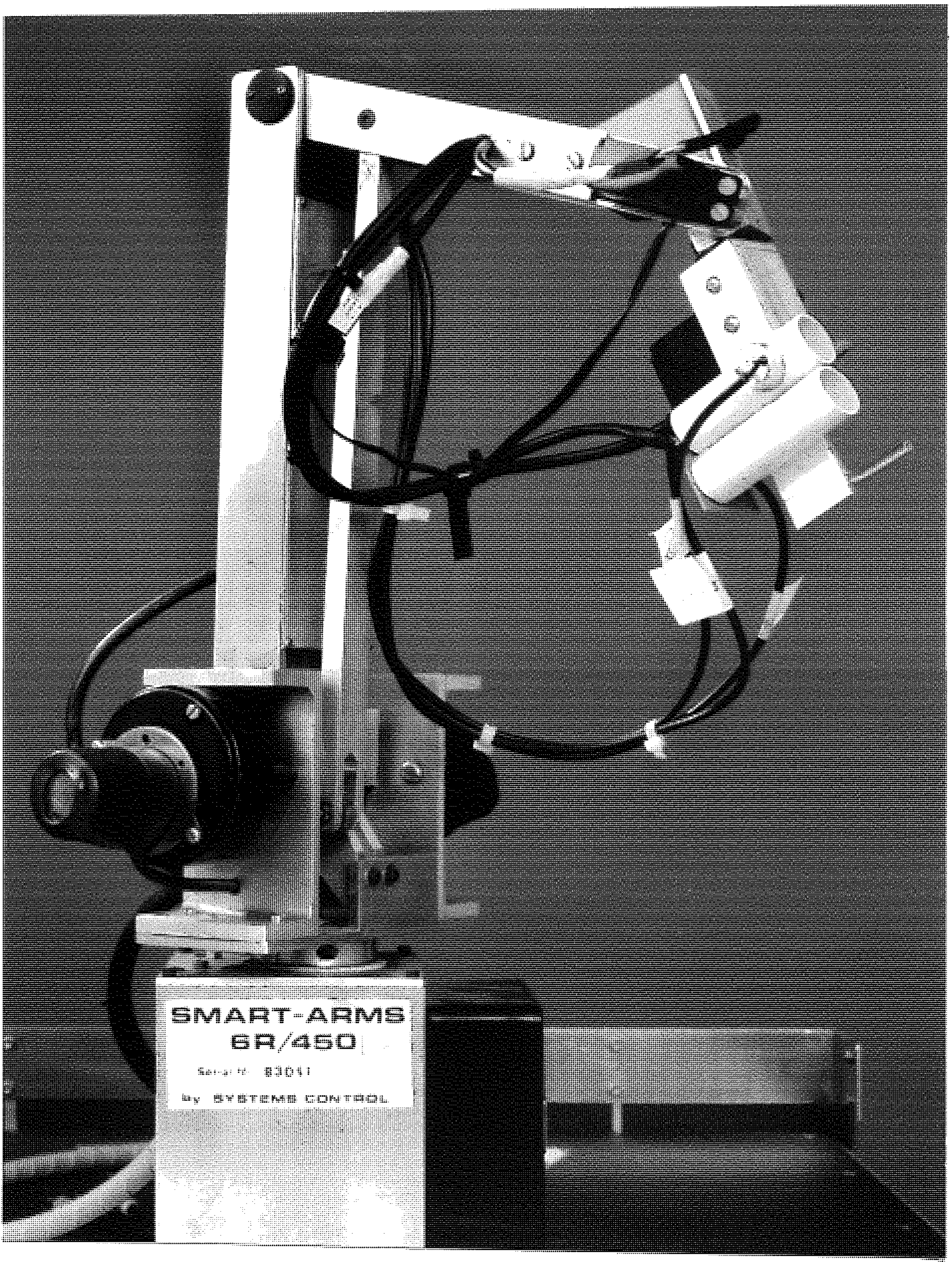

Plate 1

6.8 SMART-ARM

Smart Arm 6R/450, serial number 83041 was supplied by Systems Control of Cleveland U.K who produce a range of arms designed for research purposes. The robot arm is shown in plate 1 (facing this page) and its schematic diagram is given in figure 6.11. The arm was rigidly fixed to a stable base by screwing the base plate by fixing screws through 4 drilled holes. This plate has a square base of 140 mm length.

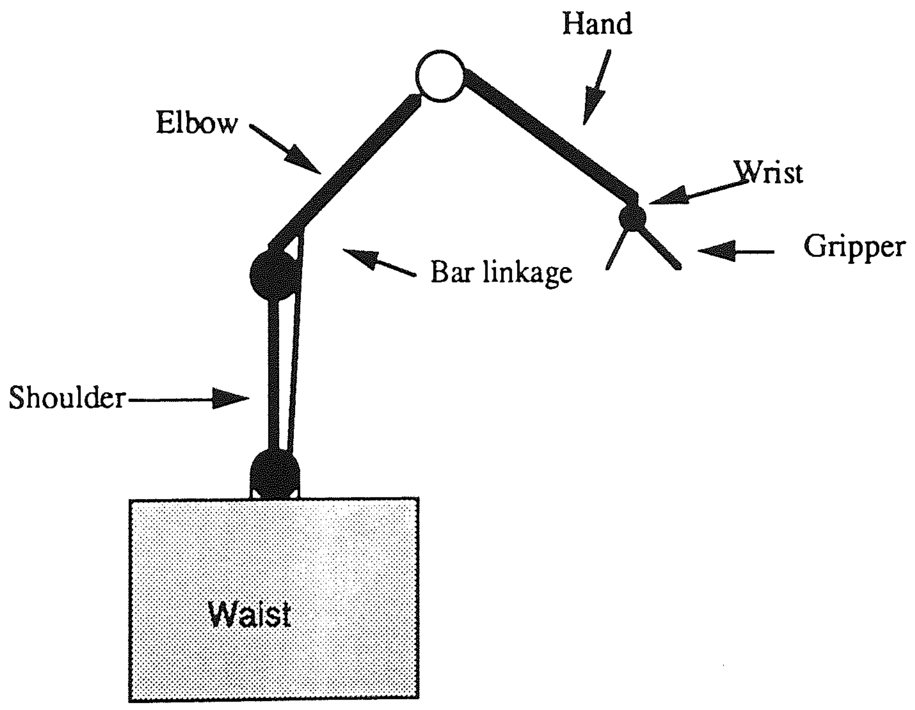

Figure 6.11 showing schematic diagram of the Smart arm

The arm is fairly rigid and made of mainly Aluminium for lightweight. It has 6 revolute joints commonly known as waist (rotate), shoulder, elbow, wrist, hand and gripper. Whilst each joint can rotate in one plane only, the arm has a maximum of 5 degrees of freedom, since gripper only opens or closes for holding the object and hence does not affect the arm co-ordinates. Each of three larger joints, namely waist, shoulder and elbow are driven by servo motors which operate on 0-12 volts range. The remaining three joints are driven by servo motors which operate on 5 volts.

6.8.1 WAIST

The base of the robot which houses the waist motor is cylindrical in shape with square base 100 mm wide and 147 mm in height. The waist rotates in a horizontal plane through approximately 120 ^{o} of arc.

6.8.2 SHOULDER

Above the base of the arm is a set of plates to which a motor is attached on each side to activate shoulder and elbow joints respectively. The shoulder joint, which is cylindrical is attached to a shaft. As the motor rotates the shaft, the shoulder joint moves up and down. It has a maximum span of 180 ^{o} , but for manipulating objects on the ground in front of the working area of the arm, it was decided to use angular span of 0-90 ^{o}. The shoulder is 27 cm long, 6.3 cm wide and 0.5 cm thick.

6.8.3 ELBOW

Elbow is linked to the end of the shoulder unit. The movement of the elbow is achieved by means of a horizontal bar which is attached to the base shaft which in turn is linked to the motor. This avoids having to attach a motor at the junction of shoulder and elbow. The horizontal bar is attached to the base shaft by a linking mechanism such that as the shaft moves, it moves the linking mechanism, which subsequently moves the bar, hence the elbow joint. As the pivoted joint is moved inwards or outwards it pushes the elbow joint via the horizontal bar downward or upward respectively, so that the angle which elbow joint makes with the shoulder arm plate also varies. Hence in order to keep this angle constant the movement of the shoulder also results in the movement of the elbow by about 90% of its value. When the elbow moves alone, it does not affect the angle made by the shoulder joint with the horizontal plane. The purpose of this mechanism is that when shoulder moves, the angle made by the elbow with respect to horizontal plane remains constant. The maximum angular span of the elbow is also 180 ^{o}. Due to this bar linkage mechanism it is only possible to operate up to 120 ^{o} in order to avoid hitting this horizontal bar against the shoulder. This joint is 18cm long and 5.1cm wide.

6.8.4 HAND

Hand is attached to the rear end of the elbow joint by cutting a hole in its plate. The servo mechanism controlling it is also attached here. It also moves by a revolute joint which allows the hand to move up and down. It has an angular span of 120 ^{o}. The hand extends to the rear end of the gripper plate.

6.8.5 WRIST

Wrist is directly linked to the hand and has no length of its own, since it only rotates that section of the hand which is connected to the gripper, both clockwise and anti-clockwise. It is also controlled by a servo attached to itself. Its maximum rotation is about 270 ^{o}. It allows the flat end-plate of the gripper to orient in different directions in order to find the best position to grasp the object.

6.8.6 GRIPPER

The gripper also forms part of the hand. It is operated by a servo to which it is directly attached. It has a flat end-plate which is 4.5cm long and 2.5cm wide on one end and a rod which can be moved in and out by the servo to open or close the gripper. When fully open the distance between the fixed flat plate and the movable rod is about 4.0 cm in length.

6.9 ROBOT INTERFACE

The purpose of this hardware unit is to select a specific joint and send appropriate digital data to move the selected joint by a specified amount from the 6809 microcomputer. It receives input from the PIA of 6809, power pack unit and sends output to the servo units/motors of the arm via a ribbon lead. The digital data received from the computer is converted into the corresponding analogue signal which is further converted into a pulse width of 1-2ms. This pulse is compared with the pulse produced by the robot servo mechanism which depends on the current position of a joint by means of a comparator. Then appropriate signal is sent to the motor to move a joint to the required position. The main components of the interface are shown in figure 6.12 below:

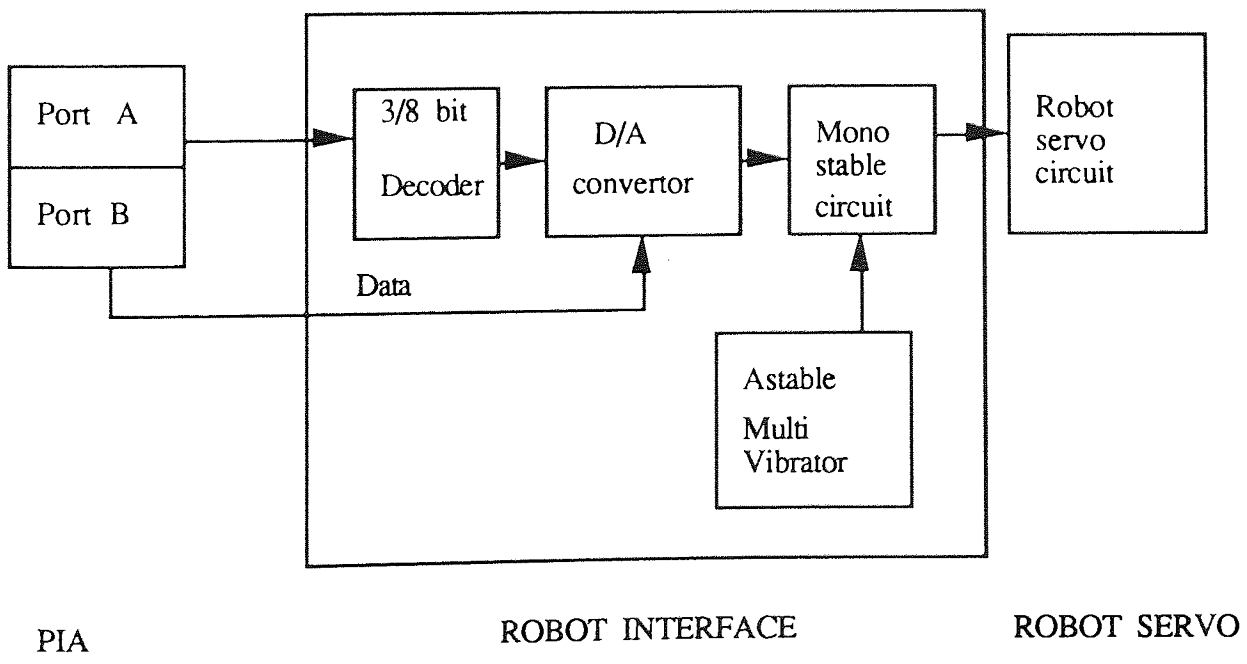

Figure 6.12 Showing robot interface (block diagram)

Since the interface is designed using standard integrated circuits (IC's) supplied by the RS components, only brief information of each component is given in this section.

Robot Interface consists of a 3/8 bit decoder, 6 D/A converters, 6 buffers (operational amplifiers), 6 transistors, 6 monostable circuits, 6 capacitors and resistors (10 nF), an astable circuit with a capacitor as shown in figure 6.13 below:

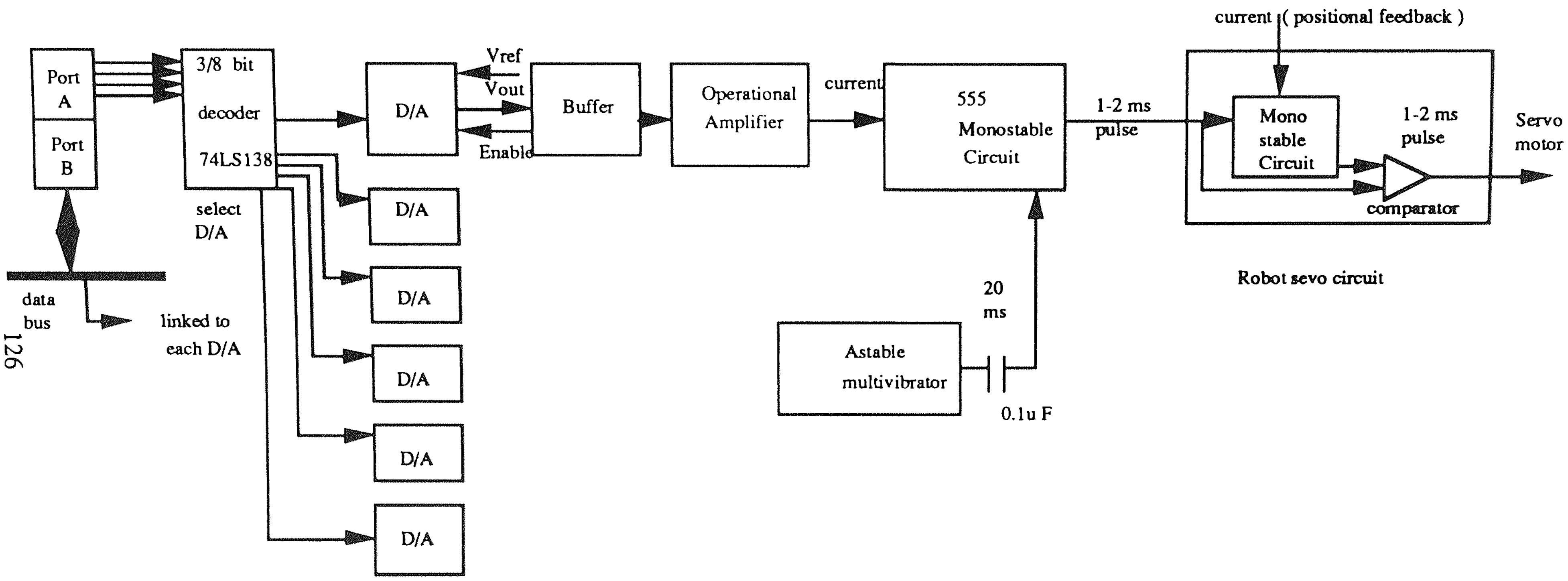

Figure 6.13 showing smart arm interface

6.9.1 3/8 BIT DECODER

This IC is known as 74LS138, receives inputs A0, A1, A2 from port A of PIA and also has one enable line (A3) which acts as a chip select and is active high. It has eight output lines. The output from each of the 6 lines is connected to each of the 6 D/A convertors which permits the selection of the required D/A convertor.

6.9.2 D/A CONVERTOR

There are 6 D/A convertors one for each joint of the arm each of which receives an output from the 3/8 bit decoder. Each IC is unipolar 8 bit D/A convertor (Z2848). It is supplied with a reference voltage (V_{ref}) of 2.5v and has an enable line which is active low. The data in the converter is held even if it is disabled i.e goes high. It produces an output voltage known as (V_{out}).

When the chip is enabled it latches 8 bit data from the data bus and converts it into an output voltage which is proportional to the digital data received by the converter as shown in figure 6.14 below :

DATA	VOLTAGE
0	0
128	1.25
255	2.50

Figure 6.14 showing relationship between data and voltage

Accuracy = 2.5/255 = 0.0098V approximately 0.01V/bit(10mv)

It was found that, by supplying voltage directly in intervals of 0.01V to the arm's servos, the movement of the arm was very small indeed. So it was decided that using more accurate D/A converters like 12/16 bits will not significantly improve the precision of the arm and cost would be much higher. Assuming a joint has a maximum of 180 ^{o} angular movement, then it can be programmed in one of the following ways as shown in figure 6.15 below :

% movement	Angular movement	Input data	Output voltage
0	0	0	0
50	90	128	1.25
100	180	255	2.5

Figure 6.15 showing relationship between data and movement

6.9.3 BUFFER (operational Amplifier)

It receives an output (V_{out}) from the D/A converter and its main function is to keep this voltage stable and pass it to the transistor circuit.

6.9.4 TRANSISTOR

It receives V_{out} from buffer and converts in to proportional current.

6.9.5 MONOSTABLE CIRCUIT

This IC known as 555 monostable circuit for timing and produces a pulse width modular output depending upon the input voltage. It is connected to the transistor via a resistor and a capacitor (0.01μf). It works on the principle that it generates a pulse at certain time interval (in this case 20ms) equal to the time taken for the voltage to attain 2/3 of its V_{CC}. The pulse width generated by this circuit depends on the data received by the D/A converter from data bus, and hence V_{out} produced by it. For a data value of 255 a pulse of 1ms width is generated and for 0 data a pulse of 2ms is generated. The pulse width produced this way is fed into the servos of the robot arm.

6.9.6 ASTABLE MULTIBRATOR

This is another standard IC which produces a pulse at 20ms interval. It generates a +ve as well as -ve pulses at 20ms intervals but 0.01μf capacitor ignores the +ve pulse. This circuit generates a pulse for each of the 555 monostable circuits and acts as a relay so that the signals coming from D/A converters are activated (seen) by it and sent to the Servo circuit of the robot arm.

6.10 ROBOT SERVO CIRCUIT

Each joint of the arm is operated via a servo circuit. The output pulse of 1-2ms, which is proportional to the data sent to a particular joint from each of the monostable circuits is fed in to the appropriate servo circuit. Each servo circuit has a built in monostable circuit similar to the 555 monostable in the robot interface. Each motor of the arm moves the joint by moving a potentiometer to which it is attached. The current position of the potentiometer is read into this monostable circuit to which it is linked via a capacitor . The output of this circuit will also generate a pulse between 1-2 millisecond width, depending upon the current position of the joint. These two pulses, one coming from the joint position and the other from the interface are compared by a circuit known as comparator. The output of this comparator produces a resultant pulse. If the two pulses are of the same width and phase, no movement of joint is necessary. On the other hand if the resultant pulse is positive then the joint rotates in the appropriate direction.

6.11 POWER PACK/UNIT

This is connected to mains supply and supplies power to robot's motors. It can send voltage up to 12V, but it was found that values in the range 8-10V kept the arm very steady. The higher voltages result in a shaky arm. It also shows a current drawn by the circuit for generating pulse width at 50ms interval. The output from this unit is fed to power control unit.

6.12 POWER CONTROL (VOLTAGE REGULATOR) UNIT

This unit is connected to power pack. It is used to supply power of 5V to robot interface and send voltage first to the three large motors i.e Waist, Shoulder and elbow followed by the 3 small servo's i.e hand, wrist and gripper. This was done in order to avoid sending data from the programmer first to smaller Servo's which may drag the arm along the ground and eventually cause damage to the Servos or the IC's in the interface during the power on procedure and other movements of the arm.

When the power, from the power unit is applied, the red LED comes on. At this stage PIA is configured and the required data (usually 128) is sent to each of the joints.The data is actually sent to the robot's arm via the robot interface only after the push button switch (red) has been pressed. When the yellow LED is lit indicating that the data has been sent to the large motors, after which data is sent to the remaining 3 joints (hand,

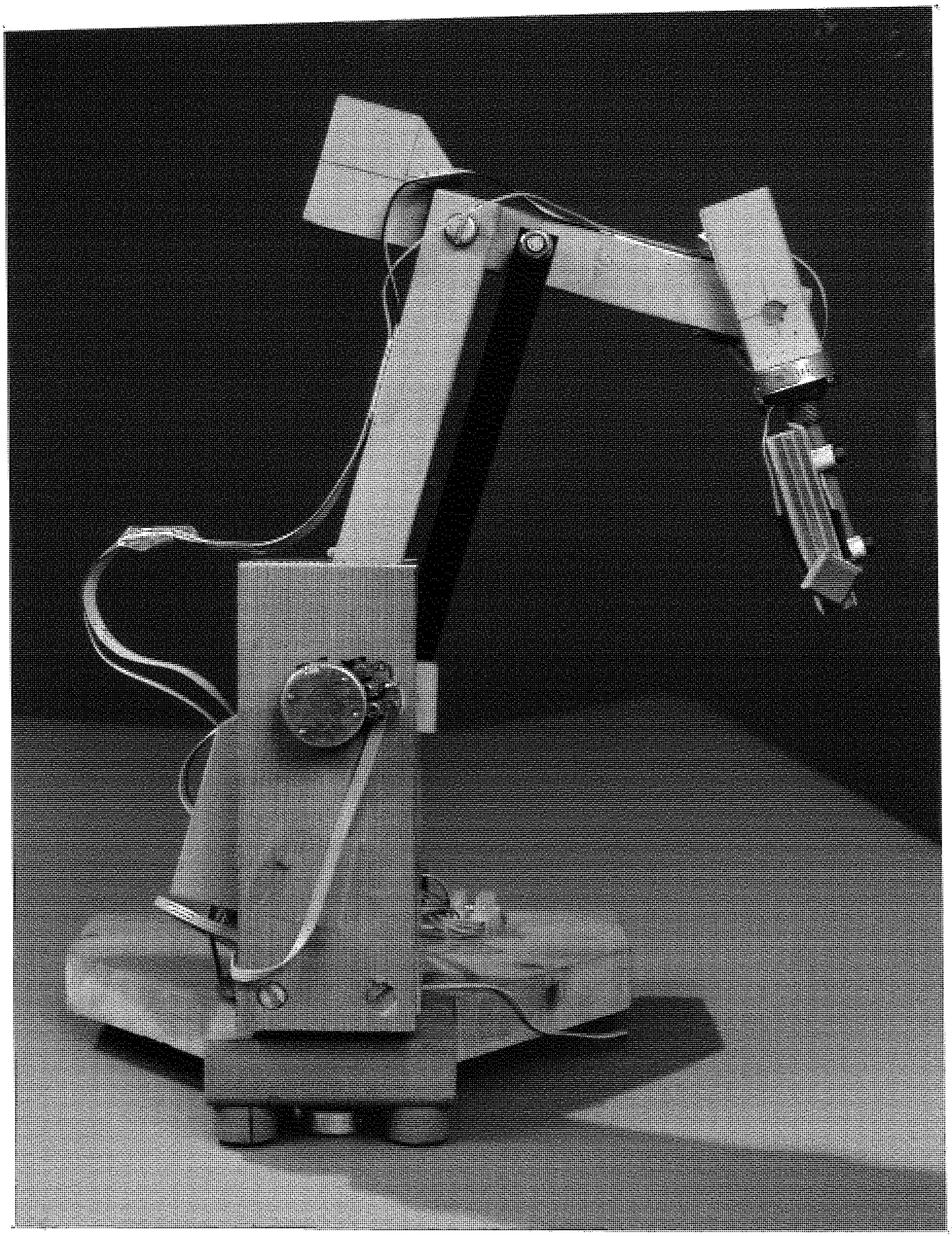

Plate 2

wrist and gripper) resulting in a green light. For the best results, it is advised that IC's should be supplied with power for about 5 minutes before sending data to the arm for warm up. Before this unit was built, it was found that by careless use and handling such as hitting the joints by obstructions etc. used to upset the joints calibration considerably. This unit has overcome these problems. Also the arm used to be shaky if the large motors were sent voltage above 7, but now the arm is quite stable even up to 10V, which has increased the speed of the joints.

6.13 SIMULATOR

This unit (see plate 2 facing this page) is designed to be look alike of robot-arm. The purpose of this unit is to allow the user to teach robot a task by guiding through the various steps involved in the execution of a task. Although programming a robot to perform a task is advantageous in many ways, there are still occasions when the writing of a program may involve tedious programming and complex mathematical calculations even for relatively simple tasks, such as welding along a seam and remote controlled applications. Thus the user needs to have this facility available when it is more convenient to teach a task by saving various steps involved in memory and be able to repeat this task when necessary. Manufacturers of robots tend to recognise this need and usually provide a teach pendant, which is hand held device and allows the user to move the arm in steps and save them. The simulator has advantages over teach pendants, since the user is holding it in his/her hand and can use a combination of intuition and skill to direct the arm for every minute movement in the desired direction. The user does not need to be close to the actual arm, which can be of great advantage if robot is performing a task in a hazardous environment such as in nuclear plant, handling corrosive substances etc. from a remote and safe distance.

The simulator was constructed of wood as shown in figure 6.16 for low cost and light weight. It has a scale of 1:2 (simulator : robot) for each joint, so that its appearance is very similar to the arm itself.

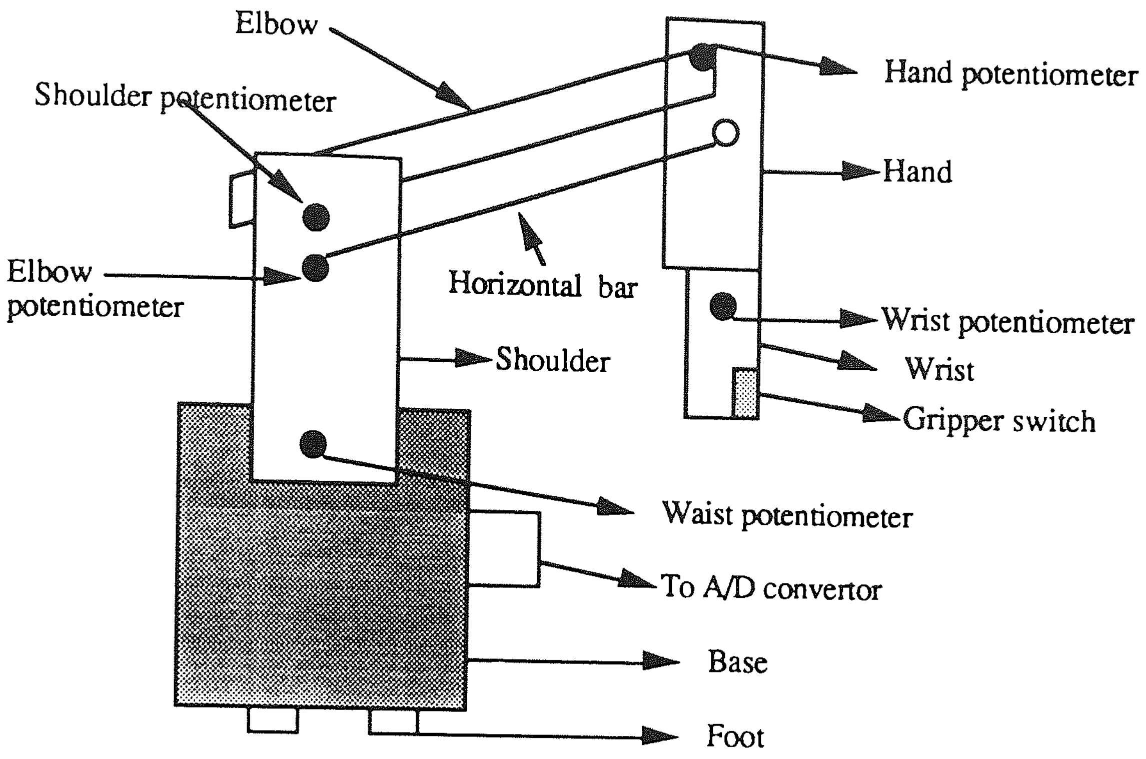

Fig. 6.16 showing schematic diagram of the simulator.

The movement of each joint of the simulator is achieved by placing a potentiometer between the two joints. As the joint is moved, the attached potentiometer produces a proportional voltage which is fed to the A/D converter. Altogether there are 5 potentiometers fitted to the simulator, one for each of the following joints: waist, shoulder, elbow, hand and wrist. Since the gripper is only used for opening and closing the arm's gripper, it was decided to use a push button switch so that when the switch is fully open, it sends a 0 data to the arm's gripper, when it is pressed it sends a data of 255 which closes it. The voltage input from each of these joints is fed in to the simulator interface via a switch.

6.14 SIMULATOR INTERFACE

This mainly consists of a 8 channel, 8 bit A/D converter I.C (7581 JN) supplied by RS components. The device contains an 8-bit successive approximation analogue to digital converter, an 8 channel input multiplexer, 8*8 dual port RAM, three state data drivers, address latches and microprocessor compatible control logic. It can accept up to 8 analogue inputs and subsequently converts each into an eight-bit binary word using the successive approximation technique. In this case only 6 analogue channels were used, one from each of the simulator's six joints. The converted results are stored in the internal 8*8 dual-port RAM. Conversion from each channel takes 80 clock periods with a complete scan through all 8 channels taking 640 input clock periods. When a channel conversion is complete, the successive approximation register contents are loaded in to the appropriate channel location of the 8*8 dual port RAM. When the successive channels are completed the converted data, in digital form for each of the 6 joints, is saved in a memory slot, which is ready and can be read in to the port B of PIA via the data bus. The converted data for each joint lies between 0-255 and is saved into an array, from which it can be sent to the robot arm in any desired order.

6.15 ULTRASONIC TRANSDUCER

This unit was devised to attach a transducer to the arm, so that it can obtain information regarding its environment. Robots with transducers are becoming more important, so that the arm if necessary and under programmed conditions, can interact with its surroundings and take necessary actions.

The ultrasonic transducer was selected for its low cost, readily availability, well established technique with which most technologists are familiar. It has wide applications in industry, medicine and other areas of technology and consequently has attracted considerable interest in robotic research as well.

The ultrasonic transducer supplied by RS components (stock no. 307-351 and 367) operates at 40KHZ. It consists of a transmitter unit and a receiver unit and is attached to the gripper of the robot arm. The range measurements are made by measuring the time taken for an ultrasound pulse to be reflected back from an obstacle and received by the receiver unit. This timing process is implemented in the hardware in its interface. The timing delay can be used in calculating the range of the object (obstacle) which can be used for obstacle avoidance when robot is programmed to move from one point to another via a specified trajectory. Another application of this transducer is object location. Ultrasonic transducers are also widely used in mobile robots.The ultrasonic transmitter is capable of emmitting 106 dB (0 dB=$2*10^{-4}$ µbar) and the receiver has a sensitivity of 0-65 dB (0dB=1µbar/vm). The transducer contains piezoelectric crystal as shown in figure 6.17 below:

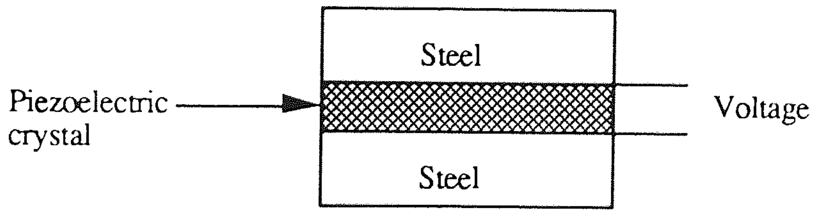

Fig. 6.17 showing ultrasonic transducer.

6.16 ULTRASONIC INTERFACE

This consists of a transmitter (307-351) (figure 6.18) which is capable of emitting 106dB, a receiver (307-367) has a sensitivity of -65dB and a timer to count pulse delay times.

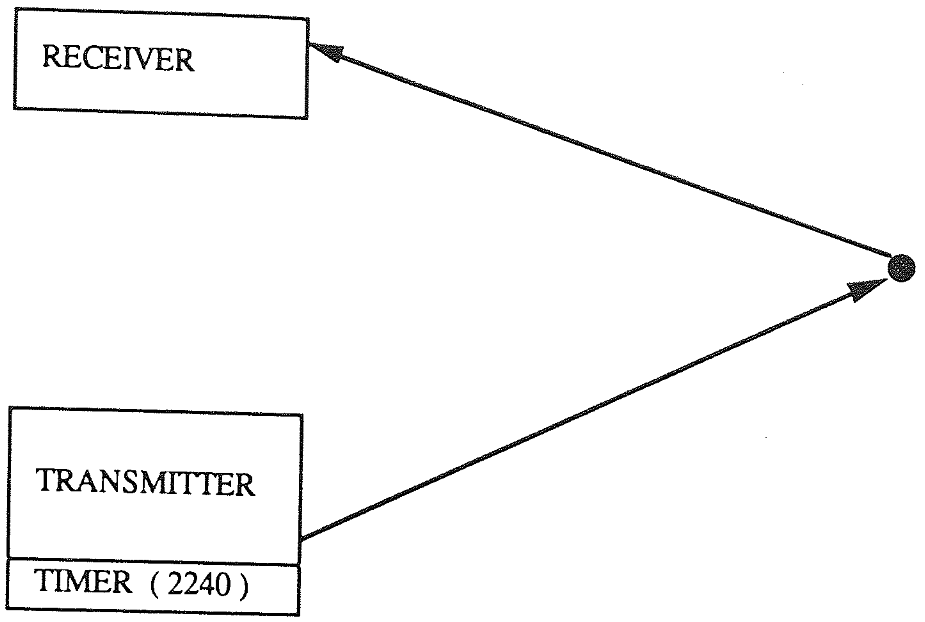

Fig. 6.18 Delay in firing and receiving ultrasonic wave.

After firing the ultrasonic pulse from the transmitter, the time counter unit 2240 is started and terminated when the echo is received by the receiver. In order to eliminate any delay propagated within a timing circuitary, a feedback transducer is used to ensure that timing begins at exactly the moment the ultrasound pulse leaves the transmitter. To use the transducer for range measurement, it has timer which is to be resetted and initialised by the software. The contents of the timer is calibrated, by placing an obstacle at a known distance, firing the transducer and reading the contents of the timer. The procedure is repeated by placing the object/ obstacle at a different distance. These delays can be plotted against measured distances to determine a relation between these two quantities. The contents of the delay timer can be read by sending them to the port B of the PIA. The actual distance of the object can be calculated by the equation developed from calibration graph as shown in chapter 7.3.

CHAPTER ---7 RESULTS AND EVALUATION

7.1 INTRODUCTION

In order to test the effectiveness of Forth as a robot programming language, the development of software and its suitability is described in this chapter. Software design philosophy of Forth is illustrated with an example in section 7.6

A method for the calibration of robot joints, the solution to inverse kinematic problem (IKP) for the Smart arm and the calibration of ultrasonic transducer and necessary extensions to Forth are described. The required extensions to Forth may be sub-divided into the following sections:

(a) Additional mathematical operations words.

(b) Input/Output (I/O) words.

(c) Joint movement words.

(d) Point-to-point movement words.

(e) Simulator words.

(f) Trajectory generation words.

(g) Ultrasonic sensor words.

(h) Block movement words.

In section 7.7 the usefulness of Forth in Robotics is described with examples using the developed software. Section 7.8 contain brief summary of the chapter. Since description of each and every word will render this chapter very lengthy, it was decided to describe a selection of high level abstract words in each section. The low level words which are necessary to develop these high level words are listed along with their actions in appropriate sections of appendix D.

7.2 CALIBRATION OF JOINT MOVEMENT

The Smart arm has revolute joints, which move in an angular form in a fixed plane. Since the digital data needed to cause these joints to articulate is sent from the computer via the robot interface which converts the data into proportional voltage and moves the joints in an angular fashion, it was necessary in the first instance to establish a relation between the data and the resultant angular movement. For this purpose the following approaches were considered:

(a) potentiometrical method

(b) Direct angular measurement

(c) Trigonometrical method

(a) POTENTIOMETRICAL METHOD

This involves using an angular potentiometer with a supplied voltage. The potentiometer can be rotated and minimum and maximum voltages produced are recorded. The potentiometer can also be linked to a circular scale by a needle which allows angular readings to be read from the scale. Thus by rotating the potentiometer, the angle described and voltage output can be recorded.

This potentiometer can now be inserted at the junction of two joints and the output from it can be linked to the voltameter. As the joint is slowly moved by sending data in short intervals, the output voltage is recorded. Thus a set of readings can be obtained for data value and voltage output produced. Since the relation between voltage and angular movement of potentiometer is already known, it is possible to relate data to the angle between the joints.

The method appears to offer a fairly accurate way of relating joint angle to the input data. The main difficulty encountered was that for most joints, it was not possible to place a potentiometer at the junction of these joints due to the construction of the arm so this approach could not be used.

(b) DIRECT ANGULAR MEASUREMENT

In this method, each joint in turn was moved by sending data in small intervals and angle between the two joints was measured by placing a protractor at their junction. It was possible to measure angle for joints such as hand and elbow, but again due to geometry of the arm, it was difficult to measure waist, shoulder and gripper angles. Although the method is simple, does not require any mathematical conversions, it was found to be not very accurate and cannot be used for all the joints, so that it was also not pursued.

(c) TRIGONOMETRIC METHOD

This method relies on the fact that when data is sent to a joint, the link moves in an angular fashion with respect to the other link to which it is attached and hence form a fixed angle. Since the links are rigid solids, they can be treated as straight lines of fixed length hence a trigonometric function can be derived. Thus, by varying the data to a particular joint, and measuring the corresponding height of the moveable joint from the ground, the calculation of the joint angle for each data value is possible. This method of calibration being simple, accurate and applicable to all joints was adopted and will be described in the following sections.

7.2.1 WAIST CALIBRATION

Waist or rotate joint rotates in a horizontal plane. This was calibrated by mapping the working area of the arm as shown in figure 7.1 below.

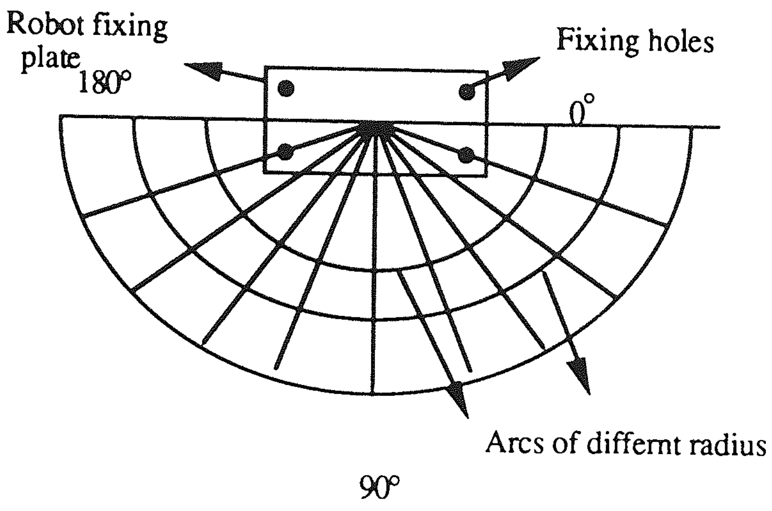

Figure 7.1 showing mapped area covered by waist

For this purpose white paper was placed on the floor to which the arm is fixed. The arm was removed and the centre of the base plate, was determined by drawing straight lines through the centre of the holes and their intersection gave the centre of the arm's base.

By using a large protractor, the working area was divided in to sectors from 0-180^{o} at interval of 5^{o}. Using a large compass the area was also divided into arcs of varying radius referenced to the centre of the base plate. The radius ranges from 0-42cm.

For calibration purposes the waist was removed so that the centre of the gripper's plate was just above a line corresponding to one of the angles on the mapped area, the data value for this was noted. The results obtained are summarised in table 7.1 below:

Waist angle	40	50	60	70	80	90	100	110	120	130	140	150	160
Data	35	70	94	114	134	153	170	186	202	216	230	242	254

Table 7.1 showing relationship between angles and waist data.

This procedure was repeated to cover most of the mapped area. The graph, depicted in figure 7.2 shows the relation between input data values and angles measured in degrees.

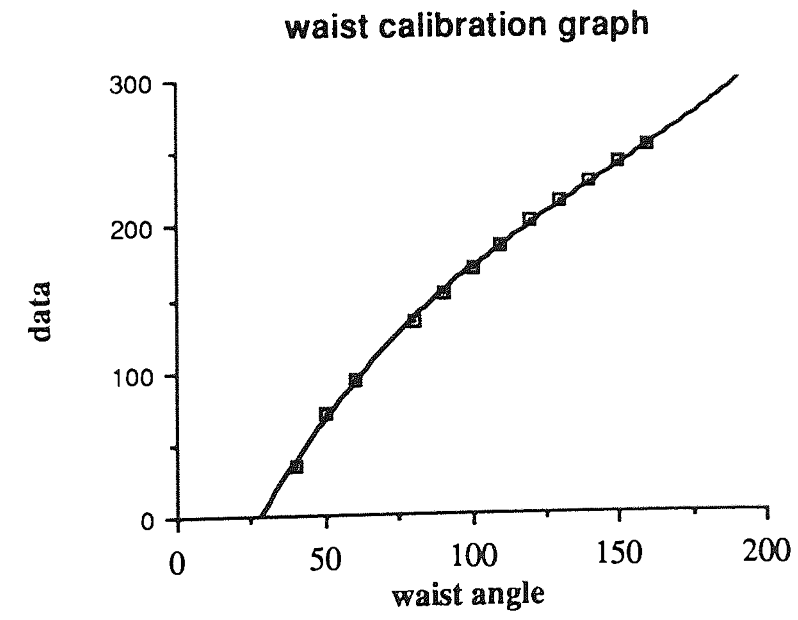

Figure 7.2 showing graph between waist angle and data

Since the graph was not linear, it was divided in to smaller sections of linear segments and the slope for each segment was calculated. Suitable equations were developed for each segment which are summarised in table 7.2 below:

Angle range (°) θ_1	Equation
0-60	$2.75 * \theta_1 - 69.5$
60-100	$2.00 * \theta_1 - 27.0$
100+	$1.41 * \theta_1 + 30.5$

Table 7.2 Angular range and appropriate equations for waist.

During the actual conversion of an angle to a specified data, the software selects the appropriate equation and carries out relevant conversion. The angle which waist makes with respect to the horizontal plate is referred to as θ_1.

7.2.2 SHOULDER CALIBRATION

For the calibration of the shoulder the relation was derived by reference to figure 7.3 below:

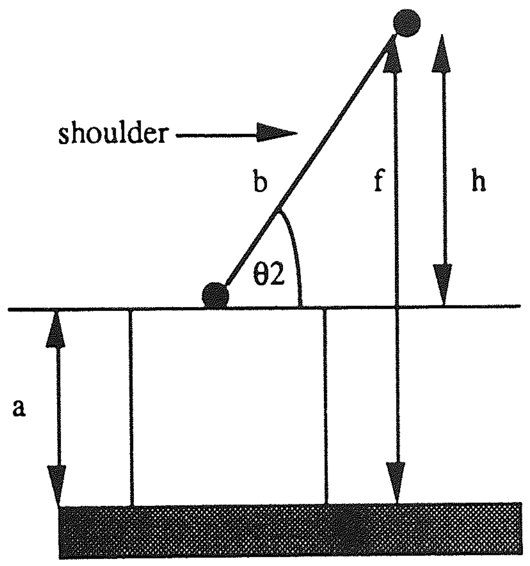

Figure 7.3 showing angle made by the shoulder

θ_2-------Angle made by shoulder at the moveable shaft, measured anti-clockwise in the horizontal plane.

a--------Fixed height of robot base from ground to shaft (220mm)

b--------Fixed length of shoulder (270mm)

f--------Height of tip of shoulder from the ground, when it makes an angle θ_2.

f-a------Distance from top of shoulder tip to the horizontal surface.

From Trigonometry:

$\sin \theta_2 = (f-a)/b$

therefore $\theta_2 = \sin^{-1} (f-a)/b$

Since a and b are constants, θ_2 is only function of f, the total height. The calibration is performed by changing input data and measuring the corresponding height of shoulder above the ground (f). The results are summarised in table 7.3 below:

Data value	0	20	40	60	80	100	120	140	150	160	170	174
Calculated angle θ_2	90	83	77	68	58	48	38	24	17	10	3	0

Table 7.3 showing relationship between measured angles and shoulder data.

The graph as shown in figure 7.4 below shows the relation between the data values and the angle θ_2.

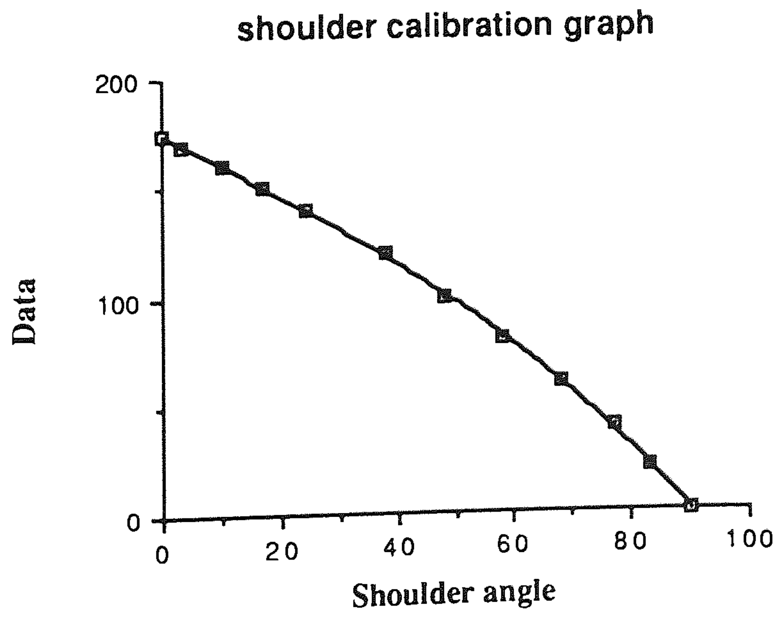

Figure 7.4 showing graph between shoulder angle and data

The graph was not linear hence it was divided in to different linear sections and the relevant equations developed are summarised in table 7.4 below:

Angle range θ_2 (°)	Equation
0-38	$-1.43 * \theta_2 + 174$
38-80	$-2.09 * \theta_2 + 200.2$
80 +	$-2.8 * \theta_2 + 257.4$

Table 7.4 Angular range and appropriate equations for shoulder.

Whilst shoulder is capable of movements up to 180^o, but the best range was considered to be from 0-90^o since movement of shoulder away from the user is of little use for manipulating objects.

7.2.3 ELBOW CALIBRATION

The elbow joint was calibrated in the same way as shoulder. The elbow was moved by sending data to it and the height from the centre of the washer which connects end of elbow to the hand was measured. The relation between the angle and the height was derived using figure 7.5 below:

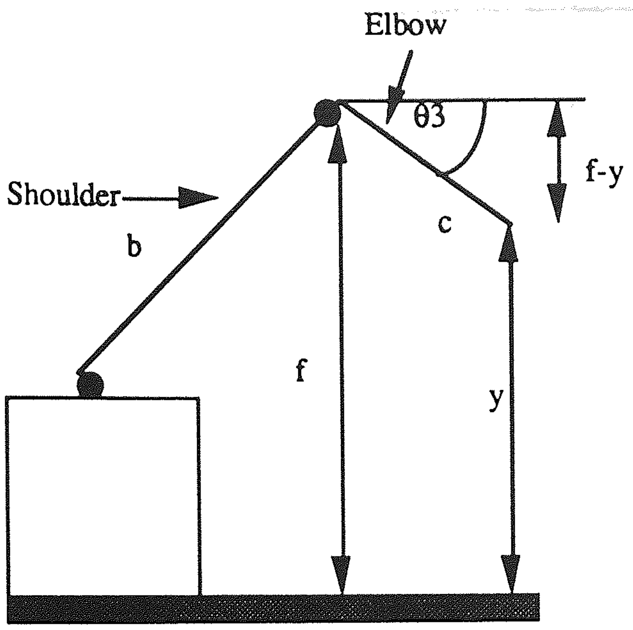

Figure 7.5 showing relationship between shoulder and elbow

θ_3 - Angle made by the elbow joint with horizontal plane.

c - Fixed length of elbow joint.

f - Height of the end of shoulder attached to elbow from the ground.

y - Distance of free end of elbow from the ground.

From Trigonometry: $\sin \theta_3 = (f-y)/c$

$\theta_3 = \sin^{-1}(f-y)/c$

Since c is constant, f has to be measured only once, changing data will only affect y. Since the motion of the elbow is restricted by the position of shoulder, the angular span considered was between 0-90^o. The results obtained are summarised in table 7.5 below:

Data value	150	137	125	112	98	82	66	44	20	0
Calculated angle θ_B	0	10	20	30	40	50	60	70	80	90

Table 7.5 showing relationship between measured angles and elbow data.

The relationship between input data values and measured angles is shown in figure 7.6 below.

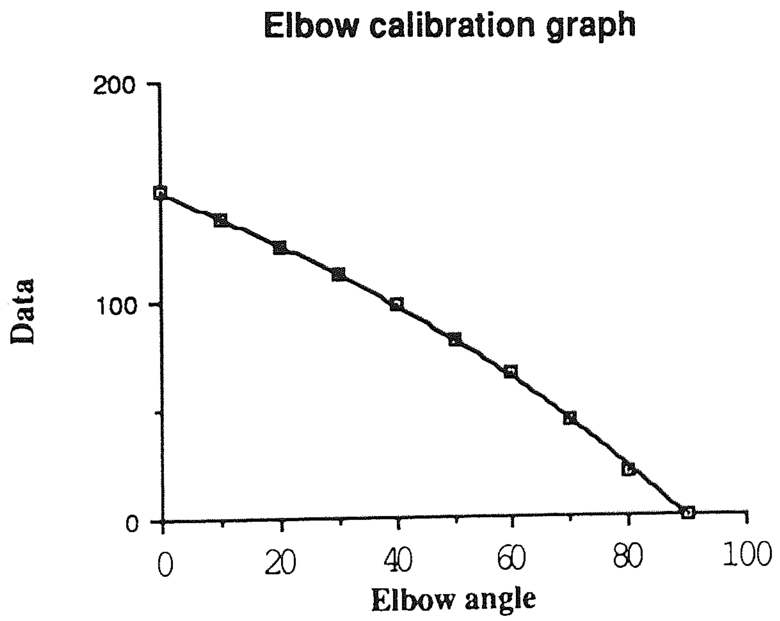

Figure 7.6 showing graph between elbow angle and data

From this graph suitable equations were developed which are summarised in table 7.6 below:

Angle range θ_3 (o)	Equation
0-32	$-1.25 * \theta_3 + 150$
32-58	$-1.53 * \theta_3 + 158.5$
58 +	$-2.2 * \theta_3 + 198$

Table 7.6 Angular range and appropriate equations for elbow.

7.2.4 HAND CALIBRATION

The hand was calibrated using the trigonometric relation derived by setting the arm as shown in figure 7.7 below:

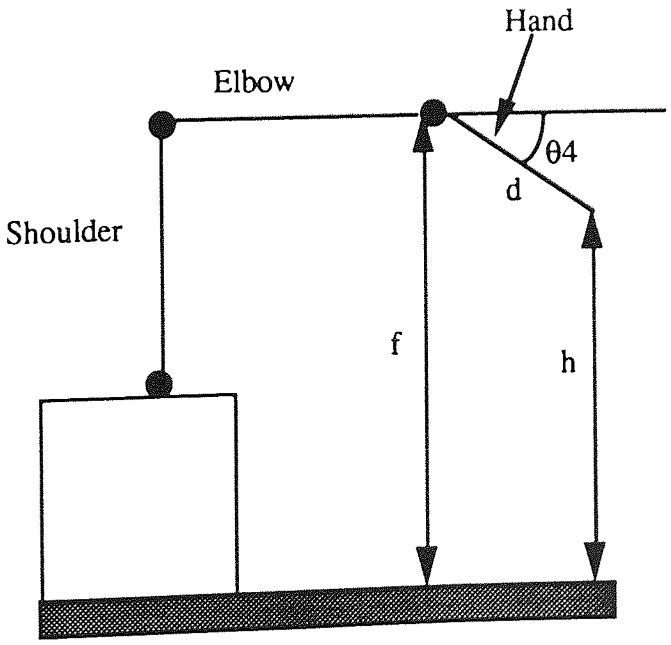

Figure 7.7 showing relationship between hand and elbow

θ_4--Angle made by hand with elbow measured clockwise.

h---variable height, tip of hand from the ground.

d---fixed length of hand.

f---height from ground to the joint of elbow and hand ,

measured from the centre of the washer.

$\sin \theta_4 = (f-h)/d$

$\theta_4 = \sin^{-1} (f-h)/d$

The shoulder is set to the vertical by sending zero data.The elbow joint is maintained perpendicular to the shoulder so that it is always parallel to the ground and makes a zero angle with horizontal plane. By sending different data to the hand, the corresponding height (h) was measured and results obtained are summarised in table 7.7 below:

Measured angle	90	70	50	30	10	0	-10	-30	-50
Data	230	212	194	170	148	135	124	94	70

Table 7.7 showing measured angles and data values for hand.

A graph correlating input data values with resultant measured angles is depicted in figure 7.8 below.

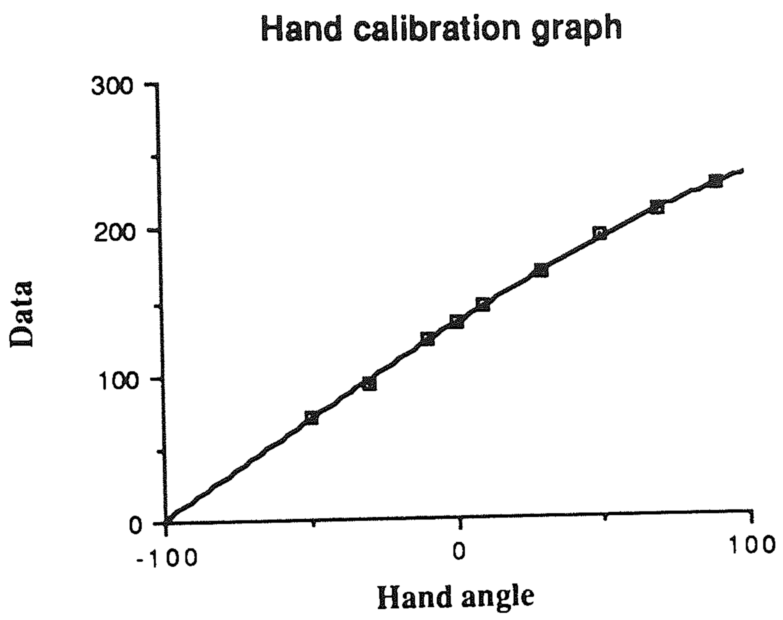

Figure 7.8 showing graph between hand angle and data

The equations developed from this graph are summarised in table 7.8 below:

Angle range θ_4 (0)	Equation
40-90	$-0.9 * \theta_4 + 230$
8-40	$-1.25 * \theta_4 + 247$
< 8	$-1.43 * \theta_4 + 265.91$

Table 7.8 Angular range and appropriate equations for hand.

7.2.5 WRIST CALIBRATION

Since the wrist rotates in a plane which is perpendicular to the working area of the robot, the distance of the end effector (gripper) from the base remains constant, so that the method of calibration used for shoulder, elbow and hand joints cannot be applied here. The wrist calibration was carried out by moving the arm in such a way, that when the waist angle is 90^o, the hand points vertically down such that the tip of the gripper is about 5mm from the ground surface. Wrist data was recorded with the flat plate of the robot arm facing towards the observer which was chosen as the reference. The wrist was then rotated through 90^o first in clockwise and then in an anti-clockwise direction. The required data was recorded each time. It was assumed that the wrist rotates linearly for the intermediate values. Although this does not appear to be a very accurate method, but since wrist is mainly used to orient the hand gripper, the most suitable orientations of the gripper are 0^o, and 90^o. A more accurate method may be developed to measure the angle made by wrist by placing the arm gripper a few cm above the ground which is mapped by protractor and sending different data values to rotate the wrist. The data for the intermediate angles can thus be obtained. The results obtained for the calibration are summarised in table 7.9 below:

Orientation of flat plate of gripper	Left hand of user	Facing user	Right hand of user
Data values	20	156	243
Angles (o)	0	90	180

Table 7.9 Relationship between measured angles and wrist data.

A plot of data values and measured angles is depicted in figure 7.9 below.

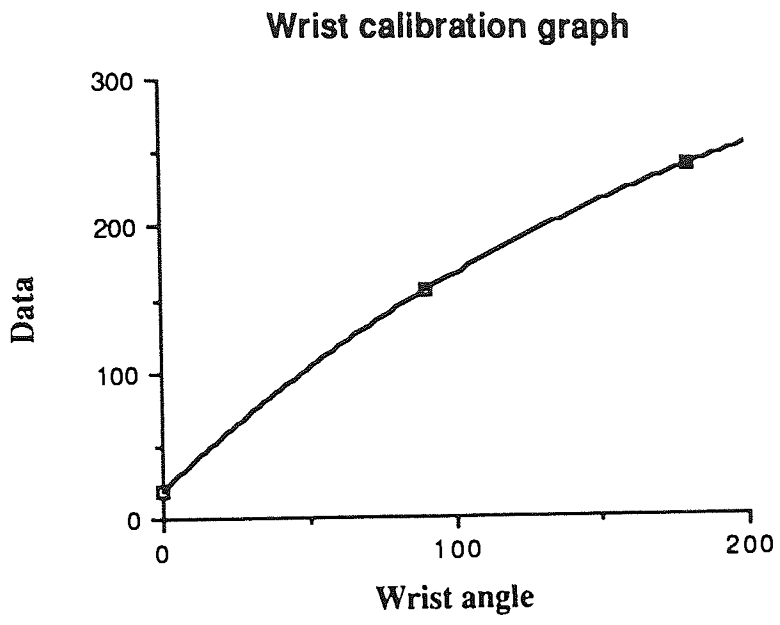

Figure 7.9 showing graph between wrist angle and data

Equations developed from this graph are summarised in table 7.10 below:

Angle range θ_4 (0)	Equation
0-90	$1.53 * \theta_4 + 18$
90-180	$0.98 * \theta_4 + 69.7$

Table 7.10 Angular range and appropriate equations for wrist.

7.2.6 GRIPPER CALIBRATION

The gripper is used for grabbing or measuring the objects to be manipulated. Hence it was considered to use only two positions for it i.e open and closed for which respective data values are 0 and 255 units.

7.3 CALIBRATION OF ULTRASONIC TRANSDUCER

The ultrasonic transducer was fired to an obstacle at a measured distance and the corresponding delay (contents of data register B of PIA) in receiving the reflected ultrasonic pulse was measured. The results obtained are summarised in table 7.11 below:

Distance (cm)	10	15	20	25	30	35	40	45
Delay data	29	56	83	110	137	164	191	218

Table 7.11 showing measured data values against known distance.

A graph of these values as shown in figure 7.10 below is a linear relationship between them.

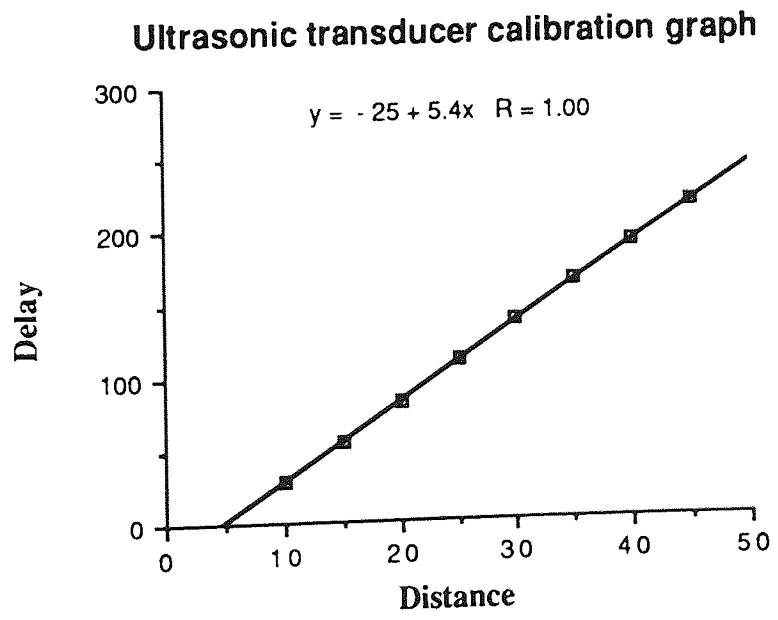

Figure 7.10 showing graph between delay data and distance

From the graph the following equation was developed :

Delay (D) = 5.22 * x - 20.5

where x is the distance in cm of the object/obstacle from the ultrasonic transducer.

7.4 SMART ARM KINEMATICS

Robot arm kinematics deals with the analytical study of the geometry of motion of the arm with respect to a fixed reference co-ordinate system, without regard to the actuators which cause the motion. Industrial robots or manipulators comprise several rigid bodies, called links or joints, connected by revolute (revolving) and prismatic (sliding) joints. The angles between the links joints are called the joint angles. Kinematic correlates the position of these links or joints with the joint angles within the working space of the robot. The relationship between the two main topic areas in robot kinematics are depicted in figure 7.11 below:

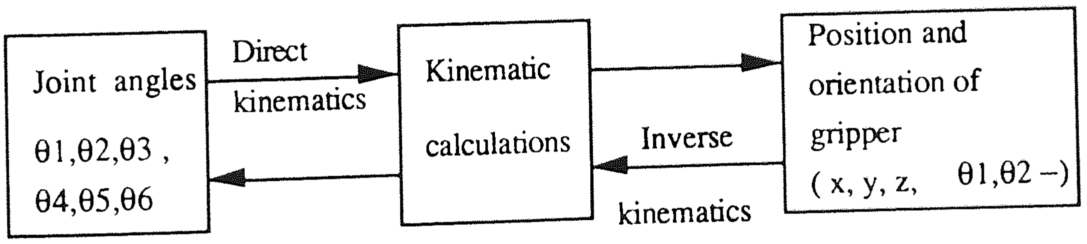

Figure 7.11 Direct and Inverse Kinematic Problem

(a) Direct kinematic problem (DKP)

DKP deals with the calculation of the position and orientation of the end-effector (gripper) with reference to a fixed co-ordinate system, which is usually the base of the arm and hence referred to as the base co-ordinate system, from the input joint angles.

(b) Inverse kinematic problem (IKP)

IKP deals with the calculation of joint angles for each joint, from the given position and orientation of the robot end-effector. This calculation is generally the more useful, since for the manipulation of objects, the user needs to specify the exact position and orientation of the object so that calculations must be performed to work out the necessary joint angles in order to align the arm tip to the required position.

7.4.1 METHOD OF SOLUTION

Generally, there are two approaches taken in solving direct as well as inverse kinematic problems, namely: the mathematical approach and geometric approach.

(a) Mathematical Approach

This method uses the application of vector and matrix algebra to describe and represent the spatial geometry of the links of a robot arm with respect to a fixed reference point. This involves the setting up of a homogeneous transform matrix to describe the spatial relationship between the two adjacent rigid mechanical links and hence to compute relationship between all the links, to achieve a final solution. This approach can be applied both to direct as well as inverse kinematic problems.

Since the geometry of different arms varies there is no universal single method for setting up and hence solving these transformations. The main advantage of this method is that transformations for a given manipulator lead to fairly straightforward method for the computation of the joint angles which can in general be applied to all manipulators.

The main disadvantages of this approach arise when the mathematical derivation may not be straightforward in the case of some manipulators, there is no feel for the physical world and the associated limitations such as singularities and solution to transformations require special trigonometric function such as ATAN.

(b) GEOMETRICAL APPROACH:

In this approach, a close and careful analysis of the robot arm geometry is made so that accurate geometrical relationships between all joints and joint angles is established. Clearly, the geometrical relation must depend on the geometry of a particular robot arm. From the specified position and orientation of the end-effector (gripper), using these geometrical relationships, starting with the target co-ordinates, it is possible to work out the required co-ordinates of each joint one by one. This assigns to each joint a unique co-ordinate for a required location of the end-effector rather than a range of values. In the next section, a geometrical relation for the Smart arm which was used in this project is described.

The main advantages of the geometric approach [SADRE et al. 1984] are that it is a relatively simple method, it avoids solutions which might seem reasonable, but in fact do not work for certain allowable arm configurations and lastly it helps to identify the presence or absence of singularities. Nevertheless this approach does suffer from the fact that there is still no rigorous, well-established method which can be applied to all manipulators. Thus this approach is mainly dependent upon the geometry of a particular manipulator for which a solution is desired and it requires a great deal of geometric visualization capability.

7.4.2 GEOMETRICAL RELATION (IKP) FOR SMART-ARM

For Smart arm which has five degrees of freedom, it was decided that waist, shoulder and elbow joints will be used to describe the position and wrist and hand joints the orientation of the arm. Since the waist rotates parallel to the base plane, it was appropriate to express this angle in degrees of arc. Hence cylindrical co-ordinates were considered to be the best way to describe the position of the arm. It should be noted, however, that the use of cylindrical co-ordinates do restrict the path taken by the robot's end-effector, to arcs of circles which is of some considerable advantage, since the aim is to cover as much volume as possible so that the actual locus followed by the robot is irrelevant. Therefore in order to define the exact position of the end-effector in 3 dimensions x, y, θ_1 co-ordinates are required. The wrist and hand angles were used to describe the orientation of the object to be manipulated. The origin of the co-ordinates was taken at the centre of the base of the robot which is actually 22cm below the base of shoulder joint. The co-ordinate positions and angular conventions for all the joints are summarised in figure 7.12 below:

NAME	EXPLANATION
θ_1-----	defines the angle made by waist measured in clockwise direction.
θ_2-------	defines the angle made by the shoulder with the respect to the base measured from the plane parallel to the ground surface in anti-clockwise direction.
θ_3-------	the angle made by the elbow with shoulder joint measured clockwise from the plane parallel to the ground surface.
θ_4-------	the angle made by the wrist measured in anti-clockwise direction. When flat plate of the gripper faces user it is equal to 90^o.
θ_5------	the angle made by the hand with elbow measured clockwise, parallel to the ground surface.
θ_6-----	the angle made by the gripper usually zero when closed and 120 when opened.
x------	the distance of the object to be manipulated from the centre of the base of the arm towards the user (radius of the arc).
y------	the height of the object to be manipulated from the ground surface or working area.

Figure 7.12 Explanation of positions and angles used.

Since the value of θ_1, is independent of the values of x, y, it is possible to define position in terms of x, and y and θ 1 will define a particular point on the arc of radius x cm as shown in figure 7.1. Hence the co-ordinate system is reduced to two dimensional system namely x and y. Thus the working area of the arm can be viewed as arcs of different radius and location of each section is given by angle θ_1, and the height by y.

Consider as an example, that a user is required to align the tip of the robot's arm for the co-ordinates shown in figure 7.13 below:

Position	Orientation ('0)
x = 32 cm	θ4 = 90
y = 15 cm	θ5 = 90
θ1 = 90'0	

Figure 7.13 Required position and orientation of gripper.

Starting with the known position of the gripper (point P1), the stepwise calculation of other joint co-ordinates is performed by using figure 7.14 shown below:

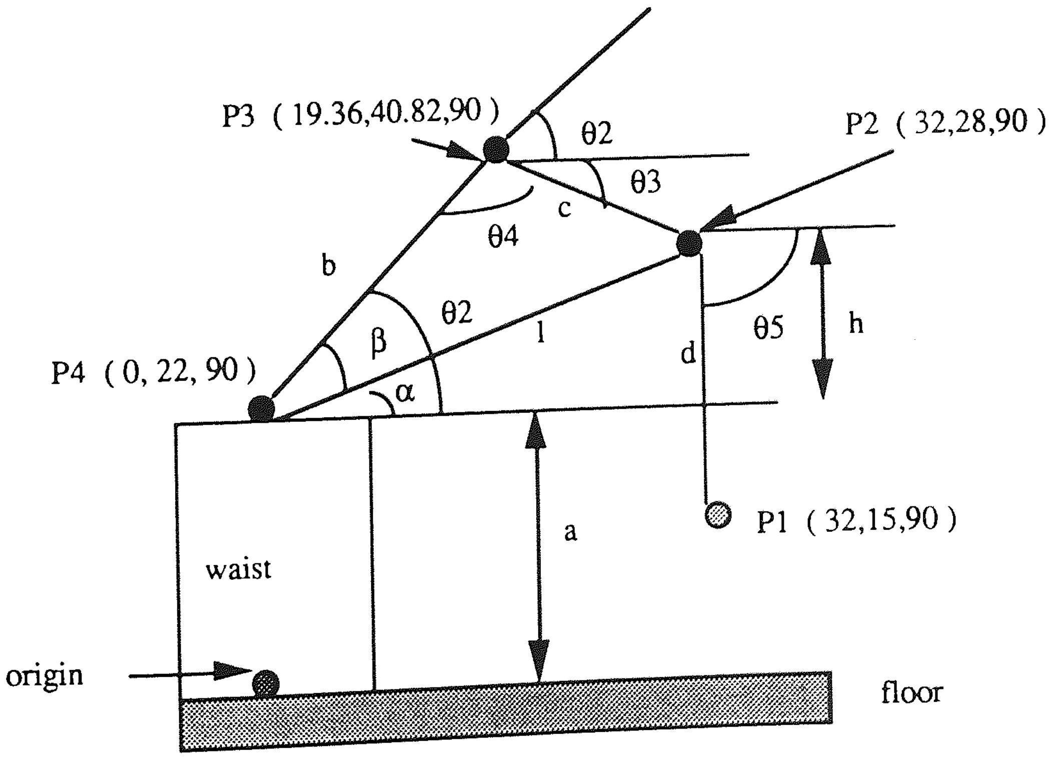

Fig. 7.14 showing calculation of joint co-ordinates.

Since θ_1 only affects the position of the end-effector on an arc and θ_4 the orientation of the flat plate of the gripper they are not needed for the calculation of the joint angles involving the shoulder, elbow and hand joints.

CALCULATION OF CO-ORDINATES OF POINT P2 in fig. 7.14 :---

Firstly θ_5, the angle made by the hand with respect to a horizontal surface must be considered. It is convenient to consider vertical and horizontal components separately.

horizontal component i.e.along x-axis = d Cosθ_5

(d-- length of hand in cm)

= d* cos 90

= 0 cm

vertical component i.e along y-axis = d Sin θ_5

= d Sin (90)

= 1*d = 1*13 = 13 cm.

Therefore, the co-ordinates of point P2 becomes:

x = 32-0 = 32cm

y = 15+13 = 28cm.

$\theta_1 = 90^{o}$

CALCULATION OF CO-ORDINATES OF POINT P3 :--

This requires the calculations of angles θ_2 and θ_3 in fig. 7.14

CALCULATION OF THE ANGLE θ_2

From figure 7.14 $h = 28-22 = 6$cm.

where h is the value of y-axis (height) from the level of shoulder movement.

since $\theta_2 = \alpha + \beta$ (fig. 7.14)

and $h/l = \mathrm{Sin}\ \alpha$

therefore $\alpha = \mathrm{Sin}^{-1}(h/l)$

$= \mathrm{Sin}^{-1}(6/32.56)$

$\alpha = 10.62$

Angle β can be computed by considering triangle with sides b,c and l in fig. 7.14 . Application of the cosine rule gives :-

$\cos\beta = (l^2 + b^2 - c^2)/(2*b*l)$

which on substitution yields

$\beta = 33.57^{o}$

Therefore $\theta_2 = \alpha + \beta = 44.19^{o}$

CALCULATION OF θ_3

Let $\theta_3 + \theta_2 = \gamma$ (fig. 7.14)

$\theta_4 = 180 - \gamma$

and $\mathrm{Cos}\ \theta_4 = -\cos\gamma$

using cosine rule:- $\cos\gamma = (l^2-b^2-c^2)/(2*b*c)$

so that $\gamma = \cos^{-1}\ (l^2 - b^2 - c^2)/(2*b*c)$

which on substitution yields

$\gamma = \cos^{-1}\ (1060-729-324)/(2*27*18)$

$= \cos^{-1}\ (7/972) = 89.59\ ^o$

since $\gamma = \theta_2 + \theta_3$

$\theta 3 = \gamma - \theta 2 = 89.59 - 44.19 = 45.40\ ^o$

since θ_4 only affects the orientation of the gripper face plate, its value does not effect the calculations of positions of various joints.

Now if desired, it is possible to calculate the co-ordinates of point P3 :

Sin θ_2 = D/b (D is vertical height from P3 to shoulder level in fig. 7.14)

or D = b*sin θ_2

= 27*sin (44.19)

=18.82

thus the y component of the point p3 is:-

= 18.82 + 22.0 = 40.82 cm (a = 22 cm)

similarly, the x-component for point P3 can be calculated as follows:

since cos θ_2 = x3/b (x3 is the distance at the shoulder height along x-axis)

x3 = 27*cos θ_2

= 27* cos(44.19)

= 19.36 cm

Therefore co-ordinates of point P3 using cylindrical co-ordinates may be written in the form :-

(19.36,40.82,90 ^{o})

In order to align the tip of the gripper to the point P1 as shown in figure 7.13 (i.e 32, 15 90 ^{o}) only the calculations of angles θ_2 and θ_3 are required. Using appropriate equations obtained from the calibration graphs (section 7.2), these angles values may be converted into the corresponding data values required for each joint and when these values are sent to the arm, the gripper will align itself at the required position and orientation.

7.5 EXTENSIONS TO FORTH

In order to move robot under different programmed conditions necessary extensions to Forth are described. These words are kept separately on a disc between screens 41 and 243 and can be loaded when required by a command such as:

```
41 LOAD
```

These extension words are described below:

7.5.1 ADDITIONAL MATHEMATICAL OPERATIONS WORDS

Forth does not support specialised arithmetic operations such as square roots, cube roots, powers, trigonometric functions and floating point operations etc. In order to perform co-ordinate transformations i.e DKP and IKP in robot kinematics, it is necessary mainly to implement trigonometric functions. The implementation of such functions can be achieved either by the use of polynomials or by means of look up tables. Unfortunately, polynomial solutions like Taylor's series, are very time consuming since they involve large number of iterations. Thus in the interest of speed it was decided to go for a look-up table for each angle.

(a) SINE LOOK-UP TABLE

Screen (41) contains the sine values of the angles from 0^o to 90^o in the interval of 1^o. These values are saved in a constant known as SIN at the next available address in dictionary given by Forth word HERE. Because Forth does not support floating point decimal, although this implementation has some especially written words for it, therefore it was decided to save the sine values of the angles multiplied by 10,000. The Forth word SINE is defined as follows :

```
: SINE   ( n ----------- n )

ABS                     ( returns absolute value )

360  MOD                ( scale to within 360 )

180 /MOD                ( divide by 180, leaves remainder and quotient )

SWAP                    ( move remainder to the top of stack . )

90 /MOD                 ( divide by 90  )

IF                      ( if quotient is positive )

90 swap -               ( subtracts from 90 )

THEN 2 *                ( calculate offset )

SIN +                   ( add to the start address of SIN )

@                       ( read angle value )
```

```
SWAP                 ( move sign flag to the top of stack )

IF NEGATE            ( change sign if flag is positive )

THEN

;                    ( end of compilation )
```

For example, Forth word

35 SINE

on execution will leave the sine value of angle 35^o multiplied by 10,000 on the stack.

Upon input of any angle, it is first converted in to the $0-360^o$ range then it is further examined if the angle lies between 180-360, in which case the sign of the angle value will be negative. Finally it is reduced to $0-90^o$ range and the corresponding sine value is read from the table. Some of angle values obtained this manner are listed in table 7.12 below:

Sine value * 10,000	5,000	-5,000	-10,000	-8,660	0	10,000	-10,000
Angle	30	210	270	300	360	450	630

Table 7.12 showing sine values

(b) INVERSE SINE

In order to calculate an angle of an inverse sine value from an input number which is multiplied by 10,000, it was decided to search the sine look-up table at specified intervals, until the nearest integral angle value is found. Standard interpolation techniques such as Newton's Forward or Backward Difference Formula (NFF/NBF) were examined. However, in the interest of speed and level of accuracy required a simple linear interpolation method of calculating the actual angle was found to be sufficiently accurate. For this purpose first some low level words were written, which include :

ARC-SINE-RANGE? (n------n, f)

This word check range of the input value and check that it lies within the range of 0 to 10,000 and returns true flag if it is out of these limits.

ARC-SINE-SGN (n---------[n]

This word saves the sign of input number in a variable and returns the absolute value on the stack.

The overall high level word is ARC-SINE which carries out limits checks, saves sign, performs the angle search and interpolation and returns the appropriate angle on the stack. This word is defined as follows :

```
: ARC-SINE        ( n ------------ Angle )

ARC-SINE-RANGE?              ( check range )

IF DROP ." Out Of Range "    ( error message )

ELSE ARC-SINE-SGN            ( check and save sign )

ARC-SINE-LIMITS?             ( Equal to 0 or 90 ? )

IF 1 DROP                    ( Clear stack )

ELSE  SEARCH                 ( Search and interpolate )

THEN ARC-SINE-ANGLE          ( get angle )

THEN SGN C@                  ( get sign value )
```

```
IF   180 SWAP -                    ( If -ve , take away from 180 )

THEN

;                                  ( end of compilation )
```

The angles obtained by inputting ARC-SINE values are shown in table 7.13 below :

Arc-sine * 10,000	5,000	5,736	7,071	8,192	9,063	9,659	9,962	10,000
Angle	30	35	45	55	65	75	85	90

Table 7.13 showing arc-sine values.

(c) COSINE TABLE

Since sine and cosine of an angle are related to each other by the relation:

cosx = sin(90-x)

Thus to find cosine of an input angle, first 90^o is added to it then its sine value is found using sine look-up table. Therefore no separate implementation of this function was considered necessary.

The word COS is defined like this :

```
: COS              ( n ---------- n )

90  +                         ( adds 90 to input angle )

SINE                          ( look up sine look-up table )

;
```

(d) ARC-COSINE

ARC-COSINE, like ARC-SINE checks range, saves the sign and calculate angle from Sine table and then converts it in to corresponding cosine. The word ARC-COS is defined as follows :

```
: ARC-COS          ( n --------- angle )

( expect no. * 10,000 on the stack )

ARC-SINE           ( get ARC-SINE angle from SINE LOOK-UP Table )

SGN C@             ( check if sign is negative )

IF 180 SWAP -      ( if so, subtracts from 180 )

ELSE 90 SWAP -     ( otherwise subtracts 90 )

THEN

;
```

(e) TANGENT TABLE

It was decided to implement this table from $0\text{-}90^{o}$ in the interval of 1^{o} in the same way as sine look-up table except double precision numbers and operators were used. The difficulty arises when angle is closer to 90^{o}, since the curve rapidly increases towards infinity. The word tangent returns the tangent value as a multiple of 10,000 times its actual value and is defined as follows :

```
: TAN          ( n ------------ d )

4 *                      ( multiply n by 4 to calculate offset address )
```

```
TANGENT                    ( supply start address of look up table )

+                          ( calculate actual address )

2@                         ( read contents of double length address )

;                          ( end of definition )
```

(f) SQUARE   (n ---------- n)

This word returns the square of an input number.

(g) SQUARE-ROOT   (n ---------- n)

This word returns the square-root of an input number.

7.5.2 INPUT/OUTPUT (I/O) WORDS

The words used to configure robot to the microcomputer, select particular joint and send the appropriate data to move it were developed and saved in a vocabulary known as a ROBOT. This vocabulary contains the following words:

(a) PIA CONFIGURATION WORDS

Forth words to configure PIA as well as to select appropriate bits to be stored in its control register for different purposes were developed.

The word PORT, which allows the user to select the appropriate PIA is defined as follows :

```
HEX         ( change number base to 16 )

: PORT                      ( n ------------- Addr )
```

```
20 *                          ( multiply port no. by 20 )

E040                          ( place this address on the stack )

SWAP  -                       ( subtracts from E040 )

;
```

For example, 1 PORT will generate address $EO20, the commencement address of the PIA at that port.

Since each side of the PIA has three registers, but only two addresses are allocated in memory for them as described in section 6.6.5 the data register and data direction register share one address between them and control register has its own address. This requires the special mechanism for addressing these registers as described previously. The three registers on each side of PIA are declared as variables with their obvious abbreviations as shown in figure 7.15 below :

Port	Name of register	Abbreviation
A	Data register	DREGA
	Data direction register	DDREGA
	Control register	CREGA
B	Data register	DREGB
	Data direction register	DDREGB
	Control register	CREGB

Figure 7.15 showing variables names for the PIA Registers.

Appropriate words were defined and tested to select each side of peripheral data lines as all inputs or all outputs or any desired combination as may be required. There are words to read data from data registers, clear flag bits (bits-6, 7) of control register, write into the control register, or select the PIA to act in handshake mode, or pulse mode and choice of enabling or disabling interrupt etc.

For example, if it is desired to configure the A side of the PIA on port 0, so that all peripheral data lines are inputs to the MPU, and to select the data register, it is necessary to use the following combination of Forth words in order to achieve this:

```
: APORT-INPUT     ( -------------- )

APORT                          ( configure Port A of PIA )

ALL-INPUTS                     ( leaves 0 on the stack for all lines input)

 DATADIRSA                     ( write 0 in DDR for all data lines input )

SLCTDR                         ( leaves 4 on the stack )

WRITECRA                       ( complete configuration )



;
```

In a similar way the words like APORT-OUTPUT, BPORT-INPUT and BPORT-OUTPUT were defined. In addition to this, words which programme PIA for handling interrupts, hand shake modes, reading the contents and writing in to the data register were also developed and tested.

(b) ROBOT CONFIGURATION WORDS

To configure the robot to the PIA and select its appropriate joint, suitable Forth words were written. For example, consider the word :

```
: CONFIG-ROBOT  ( ---------- )

APORT-OUTPUT

BPORT-OUTPUT ;
```

This word on execution will configure both ports of the PIA as outputs in order to enable robot. In order to improve the readability of words, it was decided to use the names of the arm joints, as constants, which are listed along their values in table 7.14 below :

Joint name	Waist	Shoulder	Elbow	Wrist	Hand	Gripper
Value stored	1	2	3	4	5	6

Table 7.14 showing values of joint constants.

Each constant name upon execution leaves a corresponding value on the stack.

7.5.3 JOINT-MOVEMENT WORDS

The movement of each joint is achieved by selecting an appropriate joint via port A and sending the specified data via port B of the PIA. In order to select a joint, the joint number and 8 is written in to port A . This number sets up bit 3 of the control register which is an enable bit for the robot. The movement of the joint is accomplished as follows :

```
WAIST  128 MOVE
```

This will select joint waist and send data value of 128 to it. The word MOVE is a high level word which is defined like this :

```
:  MOVE            ( n1, n2-------------- )

                                         ( expect joint no. n1 and data value n2 on the stack )

OVER DUP                                 ( move second item to the top and duplicates it )

WRITEDATA                                ( Save joint no. in data reg. A )

DATA +  C@                               ( get previously sent data to this joint )

SWAP DUP ROT =                           ( compare new data with old data )

IF                                       ( if it is true )

DUP DUP CR ." same data, no movement is required "

ELSE                                     ( otherwise )

DUP WRITEDATB                            (  write new joint value in to data reg. )

OVER DATA  +  C!                         ( write new data in to array DATA )

DUP 8 +                                  ( add 8 to joint no. )

WRITEDATA                                ( enable robot joint )

DELAY                                    ( allow joint to reach its destination )

WRITEDATA                                ( disable robot joint )

THEN                                     ( end if )

;
```

The word MOVE expects both joint number and a data value on the stack. All data values sent to the arm are saved in an array called DATA. When new data is sent to a joint, it is compared with the previous contents of the array DATA. If both are same, a message is returned to the user indicating that since the new data is exactly same as that previously sent no movement of the specified joint is required.

In order to read the latest data sent to a joint the following type of words were written WAI-DAT ----- which leaves on the stack, the latest data value sent to the joint waist .In the same way words like SHL-DAT, ELB-DAT, WRI-DAT, HAN-DAT and GRI-DAT were written for the other joints.

In order to allow movement of a specified joint from an existing place (data value) to another location the word called STEP was defined. It allows the movement of the selected joint to its left or right; up or down; clockwise or anti-clockwise appropriate to the joint movement, by a certain number of steps. The word STEP is defined as follows :

```
: STEP                 (n1 n2 ----n1 n3 n4 )

3 *                         ( multiply step no. by 3 )

OVER DATA C@                ( get previously sent data )

SWAP ;

: LEFT                 ( n1 n2 ----n  )

 -                          ( decrement previous data by a certain value)

;
```

A typical use of this word would be :

WAIST 3 STEP LEFT MOVE

This word will move joint WAIST by 3 steps leftwards from its present position.

7.5.4 POINT-TO-POINT MOVEMENT WORDS

Point-to-point movement of the arm is achieved by the user pre-specifying position and orientation, the end-effector must adopt within the working volume of the robot arm. The word POSTN expects 3 numbers x, y, θ on the stack and saves them in the variables x2, y2 and array theta respectively. The word ORIENT saves wrist angle (θ_4) in array theta + 3 and hand angle in theta + 4. The word ARM-MOVE using IKP as described earlier, calculates appropriate joint angles, selects the relevant equations from calibration of joint graphs, calculates the data for each joint and then sends them to the robot in a prespecified way. All the joint data values are saved in an array called JOINTS, the contents of which can be examined by a word known as JOINTS?. Intermediate calculated angles are saved in an array called ANGLES whose contents may be examined by use of the word ANGLES?. Some illustrative examples of the application of these words are :-

```
32 12 90 POSTN

90 90   ORIENT

ARM-MOVE
```

These words will allign the tip of the arm gripper at 32 cm. away from the centre of the arm base, 12 cm. above the ground surface at a WAIST angle of 90^o which is along the horizontal axis. The gripper's flat plate will be oriented towards the user and hand of the arm will point vertically downwards so as to be poised to grab an object from the ground.

The word ARM-MOVE is defined using another high level Forth word CALC-SAVE which is defined as follows:

```
: CALC-SAVE      ( ---------- )

CALC-XY                   ( calculate co-ordinate at the end of elbow)

CALC-L2                   ( calc. X1'2 and Y1'2 and save in var.L2 )

CALC-THEETA2              ( calc. angle theta 2 and save in var. )

CALC-THEETA3              ( calc. angle theta 3 and saves in var. )

CALC-J1                   ( convert theta 1 in to joint data )

CALC-J2                   (  "      2     "      )

CALC-J3                   (  "      3     "      )

CALC-J4                   (  "      4     "      )

CALC-J5                   (  "      5     "      )

;
```

The word ARM-MOVE is defined as follows :

```
: ARM-MOVE   ( ----------- )

CALC-SAVE                      ( convert co-ordinate in to joint values )

2 JOINTS 1 + C@ MOVE DELAY     ( move joint no. 2 )

3 JOINTS 2 + C@ MOVE           ( move joint no. 3 )

4 JOINTS 3 + C@ MOVE DELAY     ( move joint no. 4 )
```

```
5 JOINTS 4 + C@ MOVE DELAY          ( move joint no. 5 )

1 JOINTS     C@ MOVE DELAY          ( move joint no. 1 )

;
```

A combination of Forth words POSTN, ORIENT and ARM-MOVE were used to measure the positional accuracy of robot arm. For this purpose, the arm was allowed to move to a specified position and orientation. In order to measure accuracy along an axis, all the co-ordinate values apart from the one being investigated are kept same. After moving the arm, the actual position of the gripper was measured in cm using a ruler. The average results obtained for x-axis are summarised in table 7.15 below:

Entered X co-ordinate	20	22	24	26	28	30	32	34	36	38	40	42	44
Measured X co-ordinate	25	24	26	28	29	30	32	33.5	35.8	38	39.6	42	43

Table 7.15 showing entered and measured X co-ordinate value.

The results reveals that for the most accurate range along the x-axis values lies within 28-42 cm when the measured error is less than 1 cm. In the same way the values for y co-ordinate were changed while others were maintained constant. The results obtained are summarised in a similar manner in table 7.16 below:

Entered Y co-ordinate	2	4	6	8	10	12	14	16
Measured Y co-ordinte	2	4	5.9	7.8	10	12	14.1	16

Table 7.16 showing entered and measured Y co-ordinate values

It was found that the measured values along y-axis are more accurate than in the case along the x-axis. It was also found that during repeated use of the arm the positional accuracy was affected particularly if the gripper happened to collide with other objects or ground. Thus it was found necessary to repeat the joint calibration process periodically. This is a time consuming process, but nevertheless it is recommended that this should be carried out from time to time to achieve reasonable positional accuracy. A comparison of the input values with experimentally measured values indicates that the error level lies within the range 0-1 cm.

Some experiments were conducted to show the overall effect of positional error by changing more than one joint axis at a time. The results are summarised in table 7.17 below:

number	entered co-ordinates			measured error		
	x (cm)	y (cm)	θ (o)	x (cm)	y (cm)	θ (o)
1	26	6	90	-1.0	0.5	-2.0
2	28	8	60	-1.0	0.2	1.0
3	30	12	100	-2.0	0.2	1.0
4	30	10	90	-1.5	0.3	1.0
5	32	18	120	-1.0	0.1	-2.0
6	32	10	90	-0.25	0.0	-2.0
7	34	6	80	-1.0	0.0	1.0
8	36	10	90	-1.0	0.0	-1.0
9	38	12	110	-1.0	-1.0	-2.0
10	40	6	70	-1.5	-0.3	2.0
11	42	16	60	1.0	-0.4	-1.0
12	44	14	120	0.5	0.2	1.0

Table 7.17 showing overall positional error

These results show that overall positional error lies between 1.0 cm and -2.0 cm.

7.5.5 SIMULATOR WORDS

In order to teach a task using simulator, save it in secondary memory for future use if desired, retrieve it and repeat its action at subsequent stage a considerable number of Forth words were developed and tested.

To enable the simulator to the microcomputer, the PIA should be configured as :

APORT-OUTPUT BPORT-INPUT

The microcomputer receives digital data after conversion with A/D converter and saves it in an array called SIMU. This data is later on scaled by a word called CONV-SIMU-DATA and saved in an array called ARM-DAT. The simulator is then disabled and the scaled data is sent to the robot arm. The purpose of scaling is to convert data coming from simulator so that when the scaled data is sent to arm, for the convenience of the user, each joint appears to be more or less in the same position and orientation as the simulator's joints.

Often at the end of the learning session with the simulator, the user would have liked to save few trajectories, for future use. One way of achieving this is to use Forth editor and save the data values on a given screen or block and reload them when needed in the future. This method is rather cumbersome, since a large amount of data may have to be manually typed onto the screen using screen editor. Instead a better approach was taken to use some dedicated blocks/screens for saving data and transfer it automatically via buffers in to the main memory.

Typical use :

<Trajectory name> BLK-SAVE

During a learning session, the user will be asked if he wishes to save this trajectory on a dedicated screen. During a learning session with the simulator, the user cannot be sure before-hand, of number of points required to complete a task and whether a taught trajectory needs to be saved on the disc for future use. For this purpose, it was decided to compile the total number of points and joint data for each point in to the PFA of trajectory or task being taught. At the end of teaching session the user is asked if he/she wishes to save the taught task permanently on the disc. In order to achieve this, it was decided to create two arrays by using a defining word known as BLOCK-ARRAY which is used like this :

```
347 2 BLOCK-ARRAY  INDEX-BLOCKS

350 4 BLOCK-ARRAY  DATA-BLOCKS
```

The newly defined word INDEX-BLOCKS will reserve 2 blocks for saving trajectory index (i.e block number, block-offset, trajectory name). In the same way, DATA-BLOCKS will reserve 4 blocks starting with block number 350 for saving trajectory data (i.e block number, block offset, number of trajectories). The data to and from these arrays can be moved to the disc and vice versa via buffers as illustrated in figure 7.16 below :

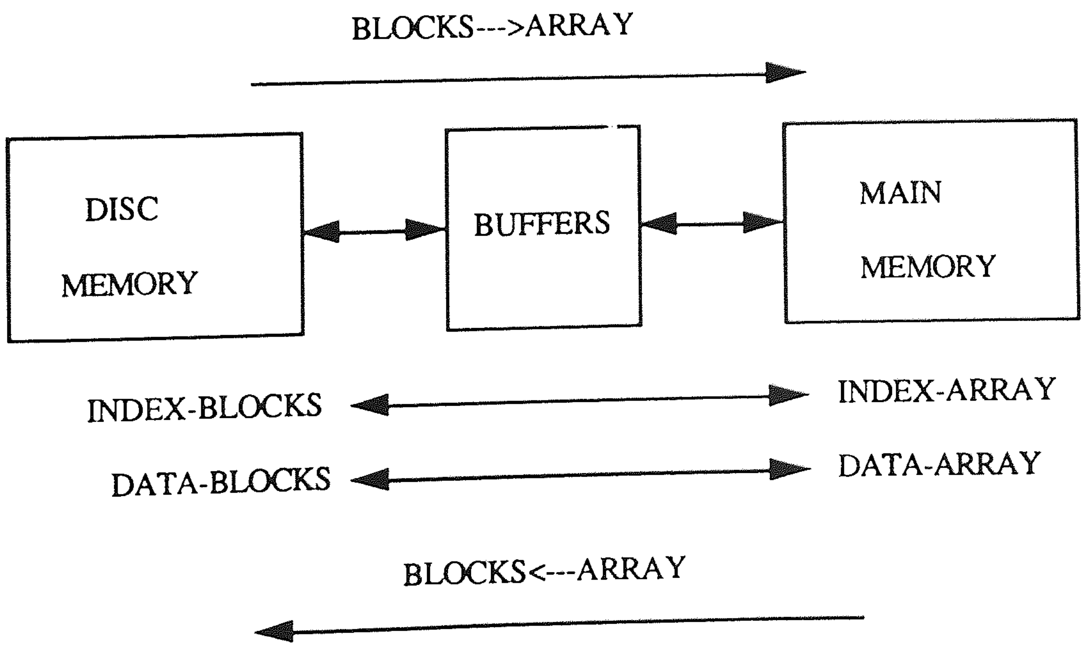

Figure 7.16 showing movement of trajectory data.

These words are defined as follows :

```
: BLOCKS-->ARRAY    ( -------- )

TRJ-IND-ARRAY                    ( move index blocks to array )

TRJ-DAT-ARRAY                    ( move data blocks to array )

;

: ARRAY-->BLOCKS        ( ----------- )

ARRAY-TRJ-IND                    ( transfer contents of index array to blocks )

ARRAY-TRJ-DAT                    (    ---------------- data  -------------- )

;
```

The words CLR-IND-BLKS and CLR-DAT-BLKS are used to initialise index and data blocks respectively. Another high level word, BLK-SAVE, moves the contents of data and index blocks to corresponding arrays and asks the user if a trajectory has ever been saved on these blocks in which case the previous block number, block-offset and number of trajectories is written in to appropriate variables otherwise the arrays are initialised and the trajectory data is saved in to array called DATA-BLOCKS. The word UPDATE-INDEX saves the block number, block-offset and first 3 characters of trajectory name in the array called INDEX-BLOCKS. The overall very high level word is LEARN which was defined by a defining word CREATE---DOES>. It is a defining word which at the time of execution will expect the name of a trajectory to be created. During the execution it will ask the operator a choice if he/she wishes to save a particular point by pressing the key 'S' from the keyboard or terminate by pressing key 'T'. The data will be saved in the named trajectory. The word LEARN is defined as follows:

```
: LEARN          ( -------------- )          ( expect name of trajectory at run time )

CREATE

HERE                                         ( get PFA of new word )

DUP 0 ,                                      ( Initialise PFA of new word )

START-KEY BEGIN                              ( ask for an input key )

ENABLE-SIMULATOR WAIT-KEY  CR CR  ( wait for an input key )

DUP 83 =                                     ( is S key depressed ? )

IF COMPILE-DATA ."save the point"  CR  THEN

 DUP 84 = IF                                 (is key T depressed )

TERMINATE                                    ( disable simulator)

ELSE DROP 0 THEN                             ( remove key value and leave 0 for loop)

UNTIL

CR CR ." do you wish to save data on disc now? "

CR SELECT-KEY                                ( wait for an input Y/N )

IF BLK-SAVE UPDATE-INDEX

ELSE DROP THEN                               ( remove address )

SMUDGE DOES>                                 ( leave PFA of new word on execution )

;
```

In order to search and retrieve the saved trajectory some words such as REPLAY, TRAJ-ENTER, SEARCH-NAME, COMBINE-FLAGS, BLK-NUM-OFF, CALC-BUFF-ADDR etc. were developed and tested. The word TRAJ-NAME-LOCATE which was defined by the combination of some of these words allows the user to enter name of the trajectory to be searched, compare it with the names of saved trajectories and returns its address otherwise it warns the user and aborts. The high level word is TRAJ-DATA-LOCATE which is defined as follows :

```
: TRAJ-DATA-LOCATE        ( ---------- Addr )

TRAJ-NAME-LOCATE                 ( start addr. of traj. name in index array )

BLK-NUM-OFF

( get blk. no. and offset of trajectory )

CALC-BUFF-ADDR                   ( get addr. of data for traj. in data array)

;
```

The use of this word can be illustrated like this :

```
TRAJ-DATA-LOCATE  REPLAY   -------- from disc

<Trajectory name > REPLAY   ------- from dictionary
```

A dedicated screen 347 will save name, block no. and offset of a particular trajectory. So when an operator types the name of a trajectory, a search is carried out in the trajectory index. If a trajectory of that name already exists then its block number and block-offset where its corresponding data is kept is returned on the stack. In order to terminate the search, a dummy 999 is saved in place of trajectory name after the last trajectory in the index. Therefore if a trajectory is not found i.e. 999 is encountered, an appropriate error message is returned to the user.

A taught trajectory after a while may become redundant, hence it may be necessary to remove it from the disc. Some Forth words were written to implement this change. The overall word to delete a trajectory is called TRAJ-DELETE. This word transfers the contents of disc blocks to arrays, asks the user to enter the name of trajectory, delete it and then readjust the contents of index and data arrays by moving the appropriate number of bytes.

The word TRJ-DIRECTORY was written for the user to search array INDEX-BLOCKS and displays the trajectory name, block-offset and number of points for each trajectory.

In order to test the developed words, new trajectories were created using word LEARN. These trajectories were saved on the dedicated blocks for future use. Every time a new trajectory was saved, the word TRAJ-DIRECTORY was used to see if the contents of trajectory index were updated. The contents of array containing trajectory data were also examined to check if corresponding data for this trajectory has been added to it. At later stage, one of the saved trajectory was deleted using Forth word TRAJ-DELETE and again contents of trajectory index and data array were examined to check if they were updated. All the words related to simulator were tested and found to work satisfactorily and each of the saved trajectories accurately repeat the stored action.

From user point of view, these simulator words offer great advantage as all the house keeping including creating, saving, deleting and updating trajectories is done automatically. However, it is quite complex and involved writing and testing of a considerable number of Forth words as shown in Appendix D.

The use of a Simulator offers greater advantage than hand held teach pendent, as it looks like the arm, human touch is involved, so that during teaching session the operator can use their skills and select which point is worth saving. It is also ideal for remote control applications such as encountered in nuclear reactors and hazardous environment.

7.5.6 TRAJECTORY GENERATION WORDS

A trajectory can be defined by specifying the series of intermediate points needed to execute a desired task. For this purpose, some Forth words were developed and tested to define individual points which can subsequently be combined to define a new trajectory. The low level words developed for this purpose include POSXY, INPUT-NUM, CLR-SCR, SPATIAL-MESSAGE, ENTER-POINT-DATA etc. The high level word DEFINE-POINT expects the name of the point and allows the user to enter data for the location of the point with respect to a fixed co-ordinate system. The word INPUT-POINT allows the user to enter a predefined point name and returns its PFA. The READ-POINT-DATA expects the PFA of the point and returns the data values for the co-ordinates. To define a new trajectory, the high level word is DEFINE-TRAJ. It is a defining word and expects the name of new trajectory at the run time. Other high level words include MOVE-POINT and MOVE-TRAJ which expect the name of previously defined point or trajectory respectively before execution and send the appropriate data to the arm. The typical use of these words can be illustrated as below:

```
<point name> MOVE-POINT

<Trajectory name> MOVE-TRAJ
```

If the user wishes to save any of the defined points or trajectories, the words created for saving simulator trajectories can also be used. However, it must be borne in mind that only first three characters of the name will be saved on the disc. All trajectories saved in this way can be subsequently retrieved using words created for simulator trajectories.

7.5.7 ULTRASONIC SENSOR WORDS

Some words were written and tested for range measurements. The word CONFIG-ULT, configures ultrasonic transducer to the microcomputer and is defined as follows:

```
: CONFIG-ULT        ( ---------- )

APORT-OUTPUT                    ( select port A as output )

BPORT-INPUT                     ( select port B as input )

0 WRITEDATA                     ( clear data register A )

;
```

After the ultrasonic transducer is fired, it starts the clock and later, terminates it when the pulse is returned. This delay in time is written into data register B of PIA. This time delay is later on converted into a measurement of the distance between object and transducer by the word CALC-DISTANCE. The calculations are based on the equations developed (section 7.3) by measuring the time delay for known distances of the objects. The high level word is ENABLE-ULT, which configures, resets, fires the transducer and returns the distance of the object. It is defined as follows:

```
: ENABLE-ULT        ( -------- )

CONFIG-ULT                        ( configure ultrasonic )

RESET-ULT                         ( reset the clock )

DELAY  FIRE-ULT                   ( fire ultrasonic and start clock )

DELAY  READATB                    ( read delay between firing and receiving )

DUP CR."the time delay read from DREGB is " CR .
```

```
CALC-DISTANCE                  ( convert delay in to distance )

0 WRITEDATA                    ( disable ultrasonic by clearing DREGA )

;
```

7.5.8 BLOCK MOVEMENT WORDS

Present Forth implementation does not have any words needed to move words contained in one screen to another screen. This facility is very desirable since during software development it may be necessary to move blocks around, insert new screens in between existing ones etc. Towards overcoming this deficiency, it was decided to write suitable words to provide this type of facility. The word CLR-BLK, initialises a given screen. BLK-COPY, copies the contents of a source block (screen) to a given block. A similar word BLK-MOVE transfers the contents of a block to a specified block. In order to move multiple blocks, the word BLKS-MOVE was defined as follows:

```
:  BLKS- MOVE         ( source no., destination no., no. of  screens)

0 DO                           ( start loop )

2DUP                           ( copy top two items )

I +                            ( add loop index to no. of screens )

SWAP I +                       ( add loop index to destination block no. )

SWAP BLK-MOVE                  ( move the blocks )

LOOP                           ( end loop )

2DROP                          ( clear stack )
```

```
;
```

The movement of a block is achieved by first copying the content of a block to buffer and then transfering it to its destination block. The definition of BLK-COPY is as follows:

```
: BLK-COPY      ( n1, n2 ------ )

                              ( n2 is destination block and n1 is source block )

SWAP BLOCK                    ( move source screen to buffer )

SWAP BUFFER                   ( assign buffer to destination screen )

1024                          ( no. of bytes to move )

CMOVE                         ( transfers bytes )

UPDATE                        ( set buffer flags for updating )

SAVE-BUFFERS                  ( transfer contents to disc )

;
```

In the same way, BLKS-COPY was defined to copy a certain number of blocks from a source to destination block.

7.6 FORTH SOFTWARE DESIGN

As described earlier in chapter 5.5, Forth software development involves top-down design and bottom-up developing and testing. In order to illustrate this design philosophy, the development of a Forth word ARM-MOVE will be described. This word as previously defined (section 7.5.4) is used to move the arm to a specified position and orientation. It expects the desired co-ordinate values in appropriate variables. First of all this abstract high level word is sub-divided into new words MOVE and CALC-SAVE. This process is repeated until fairly low level words which can be easily defined are obtained. Implementation starts by defining these low level words including declaration of variables and constants. In this particular example, this includes declaring variables and constants such as alpha, beeta, joints and theta etc and defining associated words such as sine, square, port etc. These low level words are progressively assembled in to a hierarchy which culminates in the high level abstract word ARM-MOVE, the execution of which in turn causes the execution of all the underlying words. Words developed at each level are tested for their desired action. The word ARM-MOVE is easy to use as it conveys the meaning of its action. Whilst the user is normally, only expected to use this high level word , however the related low level words can also be used directly in the development of new word/s or on their own. For example, the word move in this example is also used in the definition of words such as TRAJ-MOVE, POINT-MOVE etc thus allowing reusability of code. The hierarchical development of word ARM-MOVE is shown in figure 7.17 below:

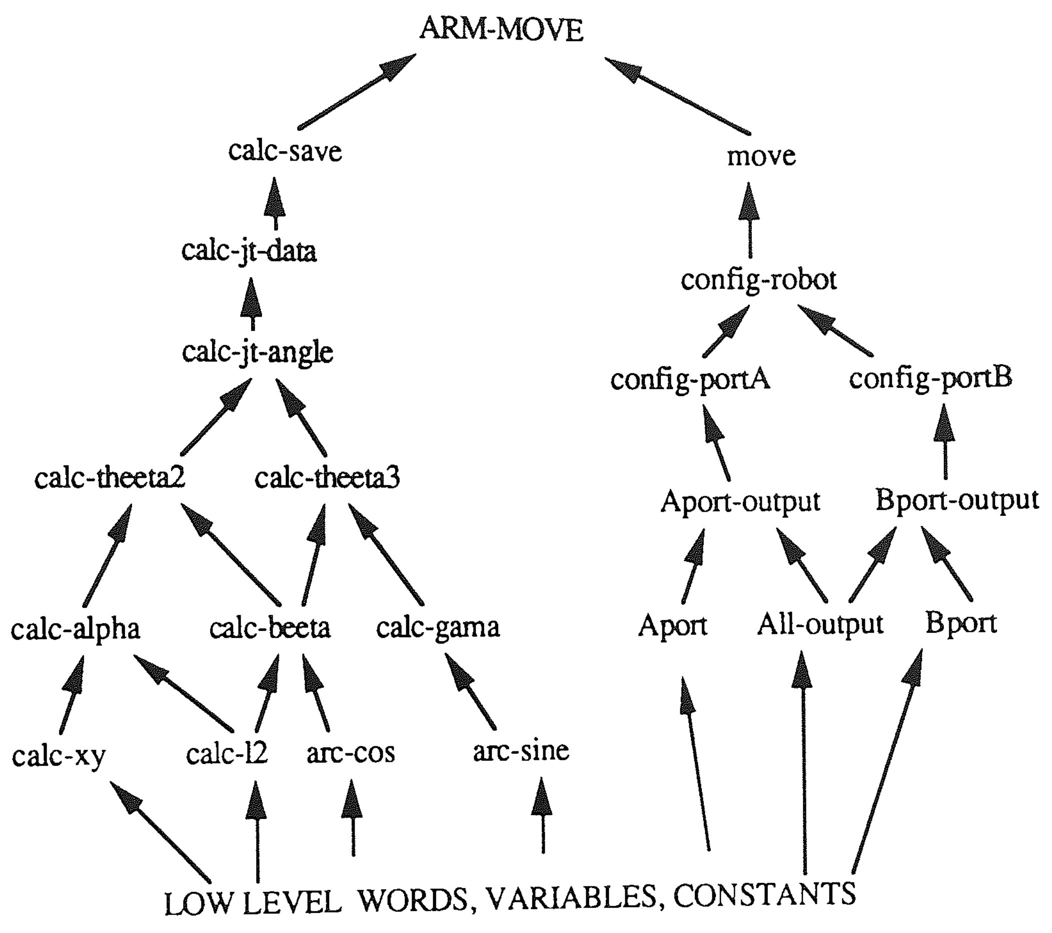

Figure 7.17 showing the development of Forth words

In a similar way, hierarchical relationship between other Forth words developed in section 7.5 is shown in appendix D (fig. D1-D6).

7.7 FORTH SOFTWARE AND ROBOT PROGRAMMING

Forth appears to be an ideal language for robot programming since it contains the following features which are particularly suitable for software development in robotics, viz:

(a) interactiveness

(b) modular design

(c) extensibility

(d) extended data structure

(e) user-friendliness

(f) reusability of code

(a) INTERACTIVENESS

Since Forth is interactive in nature, the word defined in Forth can be tested immediately in order to ascertain whether it performs the expected action. This is sharp contrast to the compiled languages such as Pascal where it is necessary to edit, compile and run in order to check whether the code is working. This feature of Forth saves considerable time in software development. To illustrate this, consider the Forth word PORT. This word on execution, expects a port number on the stack and returns the appropriate address. This can be tested by typing the following commands which will return the following addresses:

```
0 PORT--------> $E040

1 PORT--------> $E020
```

(b) MODULAR DESIGN

Forth allows software to develop in small modules by combining simple Forth words which already exist in the dictionary. Each module can be tested interactively. Different modules can then be combined together to produce an another more complex module which itself, can be tested interactively. For Example :

module 1

```
:  PORT      ( n -- addr )           ( provide port address)
```

```
20 * E040 SWAP - ;
```

module 2

```
: ASIDE  ( addr ---- )   ( saves address in appropriate variables )

DUP DUP DREGA DDREGA 1 + CREGA ;
```

After developing and testing these two modules, they can be combined to define a new module as follows:

module 3

```
: APORT ( ---- )    ( perform the action of modules 1 and 2 )

0 PORT ASIDE ;
```

It is noteworthy that each of these smaller modules can be combined together to form progressively larger and more complex modules. This feature allows the user to develop abstract high level word which on execution will perform the action of all the underlying words. In fact all the high level words described in section 7.5 are developed and tested in a modular way.

(c) Extensibility

Another advantage of Forth is the ease with which the language can be extended which allows the development of application words such as waist, shoulder, port, all-inputs, all-outputs, move, move-arm, simulate, learn and replay etc. All the extension words described in section 7.5 represent extensions to the Forth kernel. Once these words have been compiled, they become part of Forth dictionary and can be used along with other core words.

(d) EXTENDED DATA STRUCTURE

Forth allows the user to define words, which at the time of execution can be used to create a set of new words. For this purpose the words defined are CREATE and DOES>. The word Create defines the compile time behaviour of the defining word and creates a dictionary header, while DOES> defines the run time behaviour of the defining word and returns the PFA of the word at execution time. In addition extra code can also be suffixed after DOES> to be executed at run time.

An example of CREATE and DOES> as a defining word is LEARN, which names trajectories at teaching sessions with the simulator. This word expects the name of new trajectory as input from the keyboard ,e.g

```
LEARN   TRAJECTORY1

LEARN   TRAJECTORY2
```

Thus user can define as many trajectories he wishes and name them according to choice. Each of these trajectories named by LEARN will be just like other FORTH words which can be executed according to the code written after the word DOES> in LEARN. The trajectory named this way will not only create a dictionary header like other FORTH words but also save the data in it. The main advantage of Reverse Polish Notation (RPN) and stack operating languages is that actual data is not required at the time of defining new data structure. For example, the word LEARN is defined to save data in the PFA of the word defined by it so that at the time of defining LEARN the user is not concerned about the extent of data to be saved which may vary from one to many thousands of points. At later stage when data is saved in a new trajectory it can be retrieved to repeat the action of a taught trajectory or be transfered by suitable words to be saved on the disc for future use. To repeat the trajectory learned in this way there is another FORTH word known as REPLAY which can be used as follows:

<trajectory name> REPLAY

e.g. TRAJECTORY1 REPLAY

which will replay the data saved in the trajectory. Additionally the programme will ask the user at the end of teaching session whether the data in the trajectory is to be saved in secondary memory (i.e DISK) for future use. This transfer will be done automatically rather than going through the EDITOR which would be very laborious and time consuming especially for trajectories containing large number of points.

(e) USER-FRIENDLINESS

This is another useful feature of Forth, that the words written more or less resemble the action they are taking, so the syntax of the command is very much like ordinary English, so that user does not need to have any detailed knowledge of the language. However, the user might will have to get used to the fact that the commands are entered using postfix notation e.g waist 100 move, rather than move waist 100.

LEARN <TRAJECTORY> --- to teach a trajectory by the simulator and to replay it.

<TRAJECTORY> REPLAY etc.

(f) REUSABILITY OF CODES

The words written for one module can also be used in developing other modules as well, e.g the word MOVE can be useful just to move a joint by a special data, or it can be put in other words like MOVE_ARM, JT_MOVE, REPLAY etc. In the same way word REPLAY was developed to replay a taught trajectory. However, this word can also be used to repeat the operation of a trajectory defined by entering co-ordinate values.

7.8 SUMMARY

In this chapter methods of calibration of robot joints and ultrasonic transducer are described.

Methods of solution to inverse kinematic problem (IKP) for robots are discussed and a geometrical method of solution to IKP for Smart arm used in this project is described with an example.

A number of extensions to Forth necessary to control robot under various programmed conditions are described with suitable examples. In order to keep the chapter as brief as possible only a selection of high level abstract words are described. Forth software design philosophy is illustrated with an example in section 7.6. The usefulness of Forth as a robot programming language is described using developed software. It appears that Forth is an ideal language for programming robots.

In order to determine whether this version of Forth on Motorola 6809 was fast enough to perform the required trigonometric calculations involving slution to IKP, a timing comparison of Forth software used in performing these calculations with actual speed of the robot motors was undertaken. It is described in Appendix E and the results obtained are summarised in table E1. The results indicate that for this robot the time taken to perform these calculation is on the averge 4-5 times faster for each degree of joint movement. However, for a fast robot, the computation time may become significant proportion of the equivalent joint movement, in which case techniques involving faster calculations should be examined.

CHAPTER---8 CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

8.1 FORTH AS A ROBOT PROGRAMMING LANGUAGE

Forth appears to be an ideal language for robot programming. The software developed is designed for the user who is non-programmer. The commands are easy to understand, having ordinary English-like syntax and convey the meanings of the actions to be performed by the robot so that a user can commence to program the robot with a minimum of training. For programming, it is only necessary for the user to be provided with a vocabulary of robot specific commands at appropriate levels from very low level to very high abstract level. The only initial difficulty the new user might possibly experience arises from the fact that Forth uses postfix notation, and hence commands must be entered in reverse order. For example, in order to move a shoulder joint by a specified data, Forth command will be shoulder 100 move instead of move shoulder 100.

Forth allows the programmer to develop data structures in the form of defining words suitable for specific application. The idea of a defining word is very useful in Forth to produce a number of generically related words. For example, a defining word DEFINE-TRAJ can be used to define as many trajectories as required.

Forth supports the development of software in layer fashion by combining multiples of low level words to produce high level abstract word which allows the programmer to develop a hierarchical software. The user then has the option to choose the level of abstraction required in order to execute a task. Since Forth is a stack based language which uses postfix notation, it allows the programmer to define data structures without needing actual data. This feature of Forth was found to be very beneficial in robot programming.

Forth can also be used as an object oriented programming language (OOPL) [POUNTAIN 1987]. This feature of Forth is worth exploring as object oriented programming is drawing considerable attention even in robot programming.

Forth is relatively a simple language and provides low level commands to the programmer who is expected to use them to develop high level commands. This may on occasions be frustrating as sufficient programming power may not be immediately available to the programmer.

Standard implementations of Forth do not provide mathematical operators such as square, square root and trigonometric functions which are frequently needed in robot programming. Implementation of these functions is in consequence rather time consuming. However, some vendors are now in a position to supply these and other specialised extensions to the language. Forth does not supply any debugging aid to the programmer. However, this is not a major drawback, since the language is interactive and allows software to be developed and tested interactively.

8.2 FORTH AND REQUIREMENT ANALYSIS OF ROBOT PROGRAMMING

In chapter 4, a requirement analysis of robot programming was conducted. Figures 8.1 and 8.2 below summarise how Forth fulfils these requirements:

GENERAL REQUIREMENTS	COMMENTS
Easy to learn	fairly easy
Efficiency/speed	fast and compact
Modularity	supports modular programming
Portability	portable, may require minor changes
Data types and abstractions	supports abstraction and encapsulation
Readability	good, programmer dependent
Maintainability	good
Commercial availability	implementation available on most mini and micro-computers
Programming support/ environment	very good, provide complete operating system
Writability	being extensible, very good
Flow of control	provide very good control structure
Interactivity	yes
Extensibility	yes
Conc. programming	some versions provide multi-tasking facilities

Fig. 8.1 Forth and general requirements of robot programming

SPECIAL REQUIREMENTS	COMMENTS
Methods of defining points in space	extensible hence possible
Motion specification	possible
Sensor interaction	easy to integrate I/O devices including sensors
Decision making/collision checking	possible not exploited, require sensors
Human robot interaction	very good, as no programming experience is required by the user
Mathematical library	standard versions do not support floating point and standard mathematical operators. However, extensions are available or easy to implement.
World modelling	beyond the scope of this work, possible to implement.
On-line programming	easy to implement facility

Fig. 8.2 Forth and special requirements of robot programming

8.3 FORTH AND LEVELS OF ROBOT PROGRAMMING

As described in chapter 3.7, the robot programming languages can also be classified according to the level of programming they offer. This level varies from simplest point to point to complex task level programming. Most of the existing robot programming languages operate at one or two of these levels. This approach is very restrictive, because, if for example, an application needs to be changed or modified, the level of programming may also change as a result. In such circumstances, either a new language is required or the whole software will have to be rewritten. However, since Forth software is, characteristically developed in layer fashion so that, one level of programming is built on another level which allows the user considerable flexibility in selecting the desired level of programming according to the application. Forth is, therefore, a superior language in this respect as it allows user access from lowest machine level to complex abstract level. Therefore, because of this characteristic, Forth being a better language does not fit into this arbitrary kind of classification.

8.4 FORTH AND TASK ORIENTED PROGRAMMING

Task oriented programming uses sensors, world modelling and AI techniques. TRACY [1987] has reported that Forth can be used as an AI language since it is easier to extend Forth with data structures than it is to add performance to an existing language. However, it is to be seen how well Forth behaves as an AI language. Common AI languages like LISP and PROLOG are not very suitable for real time programming [ROCK 1989]. Thus Forth indeed, may emerge as an ideal candidate for sensor based task oriented robot programming language and further investigation in this direction is recommended.

8.5 FORTH IMPLEMENTATION

Forth used in this project was initially implemented by CADGE [1986] and subsequently updated by PARKES [1987]. This implementation seems to be working satisfactorily. Although most of the words conform to the Forth 1983 standard, few additional words need to be added to the Forth kernel. It was also noticed that every word defined by using CREATE and DOES> requires a code which is about 18 bytes long in order to specify that it is a defining word. This is a poor approach and waste of time and memory, which may be avoided by pointing a code to different pointers when the instruction DOES> is encountered in the same way as variables, constants and colon words are distinguished from each other.

Some difficulty was experienced with the word LOAD. For example, by defining the Forth word LOADS as follows:

```
: LOADS  18 LOAD  100  LOAD 200 LOAD ;
```

Upon execution, LOADS only loads blocks starting with block number 200 and does not load previous blocks.

Forth control structure works as follows:

```
IF --------- THEN

IF --- ELSE -- THEN
```

The word THEN at the end is misleading and a better word, ENDIF, can be defined as follows:

```
: ENDIF  [COMPILE] THEN  ; IMMEDIATE
```

8.6 SOFTWARE PORTABILITY

As discussed in chapter 4.1 the robot programming suffers from program portability because robots manufactured by different manufacturers are different. This can present considerable difficulties to industry using different robots and programming languages. Most of the current robot programming languages supplied by robot manufacturers are tailored to be used on specific robot type and hence are not easily portable. Software portability is, in consequence a very desirable feature. One way of overcoming this type of problem is to devise a standard interface between programming system and robot controller which would take into account the size, shape, type and number of joints of each robot.

Forth software is moderately portable since most Forth software is independent of robot type and underlying hardware. In order to transfer software in Forth, programmer only has to change system dependent low level, I/O and robot specific words. Therefore, it would be an interesting and useful exercise to investigate how much modification is necessary when using the developed software to control robots designed by a range of manufacturers.

8.7 SMART ARM ACCURACY

The robot arm used in this project is not suitable for very accurate positioning of the end-effector. As stated in chapter 7.5.4 some experiments were conducted to show the net effect of positional error by changing more than one joint axis at a time. Results show that the combined positional error lies between 1.0 cm and -2.0 cm. These results are not conclusive but show the possibility of error cumulation or cancellation. However, robot positional accuracy is a separate field in its own right and further investigation in this field is recommended. It is to be seen that how much inaccuracy is related to the limitation of the arm and to what extent can it be improved by strategies of movements in software such as deliberately slowing down the robot before reaching the target. For example, rather than sending the arm to its final destination in one go it may be possible to move it closer to the target and complete the final path in small increments. A second difficulty arose with the arm from accidental hitting of the arm against floor and or other objects, which caused further errors in its positional accuracy so that the robot joints had to be recalibrated fairly frequently which is time consuming and affects the robot's efficiency.

8.8 ROBOT SENSORS

The use of sensors in robot programming is a separate field in its own right. It was neither intended nor possible to integrate many sensors in the present study. However, in order to demonstrate that Forth can readily integrate sensors, an ultrasonic transducer was attached to the arm and suitable Forth words, to measure distance of an object from the hand were developed and tested. The use of ultrasonic in robot programming is certainly worth investigating.

It is possible to use ultrasonic in automatic object location and grasping. This can be achieved by allowing the robot to scan its work zone by systematically moving it in small intervals. For this purpose, the robot hand should be horizontal and ultrasonic transducer should point downwards. The robot should be allowed to move in small steps and each time an ultrasonic transducer enabled to measure the distance from the floor. The measured distance can be saved in an array. When the distance between two successive locations is appreciable it is due to the presence of an object on the floor. The co-ordinates of the arm saved previously in an array can be used to identify the location of the object.

Other areas where ultrasonic can be useful are in collision avoidance, object recognition and collision free trajectory planning. Since velocity of sound varies with temperature and humidity etc. it is desirable to calibrate the transducer from time to time or work in a consistent and carefully controlled environment.

REFERENCES

ADORNI, G., GAGLIO, S., MASSONE, L., ZACCARIA, R., 1984 "A Distributed System for Symbolic Computation in Robotic vision." Advanced Software in Robotics. A. Danthine and M. Geraldin (eds.) Elsvier Publishers B.V. (North Holland) pp. 111-121

AMBLER, A.P., 1982, "RAPT: An object level robot programming language." Colloquim of Languages for Industrial Robots, Institute of Electrical Engineers, Savoy Place, London, 8th February 1982 pp. 1-5.

ASADA, H., and IZUMI, H., 1987, " Direct teaching and automatic program generation for the hybrid control of robot manipulators." 1987 IEEE Int. Conf. on Robotics and Automation, vol. 3, March 31-April 3, 1987, North Carolina, Cat. no. 87 CH 2413-3, pp. 1401-1407

AZZAM, S.J., UNUVAR, M.U., 1985, "Off-line Robot Programming with GRIPPS" Robots 9 -- Conf. Proc.2-6 June, 1985, Detroit, Michigan, Vol.2 pp. 18-23-33.

BALLARD, B., 1984, "Forth Direct Execution Processors in the Hopkins Ultra-Violet Telescope." The Journal of Forth Applications and Research, Vol. 2 No. 1 1984, pp. 33-47.

BANZANO, T. BURONZO, A., 1979, "SIGLA-OLIVETTI Robot Programming Language." International Seminar on Language Methods Programming Industrial Robots. June 1979 pp. 117-124.

BINFORD, T.,1979, "The AL Language for Intelligent Robot." International Seminar Programming Methods for Industrial Robots, pp. 73-87.

BIOMATIK, 1983, "PASRO-PASCAL for Robots," Biomatik Co. Freiburg, West Germany .

BLUME, C., 1984, " Implicit Robot Programming Based on a High-Level Explicit system and using the robot data base RODABAS." Proc. of first robotics Europ Conf. Brussells. 27-28 June,1984. Pub. Springer-verlag, IFS (publication) Ltd., U.K. in 1985, pp. 156-171

BLUME, C., JAKOB, W. , 1984, "Design of the Structured Robot Language (SRL)." Advanced Software in Robotics. A. Danthine and M. Geradin (eds.) Elsevier Science Publishers B.V. (North-Holland), 1984 pp. 127-143.

BOLLES, R., and PAUL, R.P., 1973 , " The use of sensory feedback in a programmable assembly system," The Stanford Artificial Laboratory, Stanford university, AIM-220 (Oct.)

BONNER, S. SHIN, K.G., 1982, "A Comparative Study of Robot Languages." Computer, Dec. 1982, pp. 82-96.

BORROWMAN, G.L. , 1979, " Remote manipulator systems, Spaceflight, 21(12), pp. 495-496

BRODIE, L., 1981, Starting Forth, Prentice-Hall, Inc., Englewood Cliffs, N. J.

BROEK, VAN DEN W., BOER DE H., 1989, "Adaptation of a Robotics Algorithm for a Distributed Implementaion using Transputers." Microprocessors and Microsystems April 1989 pp. 195-202.

BURCKHARDT, C.W., MARCHIANDO, C., 1984, "A Multi-Robot High Level Programming system for Assembly." Advanced Software in Robotics. A. DANTHINE and M. GERADIN (eds.) Elsevier Science Publications B.V. (North-Holland) pp. 301-309.

BUTTERFIELD, K., 1984, "Multi-tasking in Forth." Micro, no. 75, Sept 1984, pp. 18-23.

CADGE, B., 1986, "6809 Flex Forth." University of Aston, Dept. of Computer Science, Final Year Project , Aston Triangle, Birmingham.

CARAYANNIS, G., FREEDMAN, P., MALOWANY, A., 1989, "An Integrated Programming Environment for a Generic Robotic Workcell." J. of Robotic Systems 1989 pp. 149-173.

CHAN, S.C., and VOELCKER, H.B., 1986, "An Introduction to MPL-A new Machine Process/Programming Language." 1986 Conference Proc. Robotics and Automation, April 7-10 , 1986, San Francisco, CA, Vol. 1, pp. 333-344.

COX, I.J., GEHANI, N.H., 1989, "Concurrent Programming and Robotics." The International J. of Robotics Research Vol. 8, no. 2, April 1989 pp. 3-16.

CRAIG, J.J., 1986, Introduction to robotics. Addison-Wesley , publishing co., Inc., U.S.A. ISBN 0-201-10326-5

DARIAO, P., DOMENICI, C., BARDELLI, R., DE ROSSI, D., PINOTTI, P. C., 1983, "Piezoelectric Polymers: New Sensor Materials for Robotic Applications." 13th ISIR April 1983.

DERBY, S., 1984, "Off-line Programming of Two Industrial Robots." Robots 8 -- Conf. Proc. June 4-6, 1984, Detroit, Michigan. Vol.2, pp. 20-65-76.

DONATO, G., CAMERA, A., 1980, "A High Level Programming Language for a New Multiarm Assembly Robot." Proceding First International Conf. on Automated Assembly pp. 67-76. 1980.

DYER, K.D.,1985, The use of HLL in Robot Programming. M.Sc. Thesis, University of Hull, U.K.

EJIRI, M., UNO, T., and DA, H.Y., 1972, " A prototype intelligent robot that assembles objects from plane drawings, IEEE Trans. comp., C-21 (2) , pp. 199-207 (Feb.)

ERNST, H.A., 1962, " MH-1, a computer -oriented mechanical hand," Proc. 1962 spring , joint computer conf. , San Francisco, California, pp. 39-51

FALEK, D., PARENT, M., 1979, "LAMA-S An Evolutive Language for an Intelligent Robot." International Seminar on Language Methods for Industrial Robots, Grenoble, France, 27-29 june 1979, pp. 157-168.

FALK, H., 1974, " Linking microprocessor to the real world" IEEE Spectrum, Sept. 1974, pp. 59-67

FAVERJON, B., 1986, "Object Level Programming of Industrial Robots." Proc. 1986 IEEE Int. Conf. on Robotic and Automation, April 7-10,1986, San Francisco, Vol. 3 pp. 1406-12.

FAVERTO, M., 1978, " Polar 6000 - a new general purpose robot particularly suited for spot-welding applications," Poc. of the 4'th Int. Conf. on industrial robot technology, Stuttgart, Germany, Mau 30-June 1, pp. 67-77

FEUR, A.R., GEHANI, N. eds., 1984, Comparing and Assessing Programming Languages ADA, C, PASCAL. Prentice-Hall Inc. Englewood Cliffs, N.J., U.S.A .

FINKEL, R., TAYLOR, R., BOLLES, R., PAUL, R., FELDMAN, J., 1975, "An Overview of AL, A Programming System for Automation." Tutorial on Robotics by C.S. Lee, R.C. Gonzalez, and K.S. Fu. pp. 372-379 .

FLEX 1979, User's manual. Technical Systems Consultants, Inc., Chapel Hill, North Carolina. 27514

FLEX 1979, Programmer's manual. Technical Consultants, Inc., Chapel Hill, North Carolina. 27514

FRANKLIN, J.W. and VANDERBURG, G.J., 1982, " Programming Vision and Robotic Systems with RAIL". SME Robots 6 Conf., March 1982, pp. 392-406

FU, K., GONZALEZ, R.C., LEE, C.S., 1987, Robotics-- control, sensing,vision, and intelligence, Mc Graw-Hill International edition, New York.

GEFFIN,S., FURHT, B., 1989, " Transputer Based Data Flow Multiprocessor for Robot Arm Control." Microprocessors and Microsystems vol.13,no. 3 , April 1989. pp. 219-226

GHEZZI, C., and JAZAYERI, M., 1987, Programming Language Concepts 2/E, John wily and sons , New York

GILBERT, A., PELTON, G., WANG, R., MOTIWALA, S., 1984, "AR-BASIC An Advanced User-Friendly Programming System for robots." Robots 8 Conf. Proc. June 4-6, 1984, Detroit , Michigan, vol.2, pp. 20-47-63.

GINI, M., 1987, The Future of Robot Programming. Robotica (1987), Vol. 5 pp. 235-246.

GINI, G. and GINI, M.,1984, "Robot Languages in the eighties." Proc. of first Robotic European Conf. Brussel, June 27-29 1984 pp. 126-138.

GINI, G., GINI, M., GINI, R., and GIUSE, D., 1979, "MAL-A Multi-Task System for Mechanical Assembly." Proc. Programming Methods and Languages for Industrial Robots, IRIA, France 1979 pp. 145-156.

GOETZ, R.C., 1963, " manipulator used for handling radioactive materials , in E.M. Benett (ed.), Human Factors in Technology, chapter 27, Mc Graw-Hill, New York.

GOMAA, H., LAWRENCE, J., 1985, "Menu Programming - A User Friendly Interface for Robot Programming." Robots 9 -- Conf Proc. June 2-6, 1985, Detroit, Michigan. Vol. 2 pp. 18-34-51.

GROOVES, M., 1988, Industrial Robotics Prentice Hall, Englewood Cliffs, N.J.

GROSSMAN, D., 1985, "AML As A Plant Floor Language." Robotics and Computer-Integrated Manufacturing. Vol. 2, no.3/4, pp. 215-217

GROSSMAN, D., SHORT, W., 1985 , "AML - Much More Than a Robot Language." Robots 9, Con. Proc., current issues, future concerns, vol. 2, Detroit, Michigan, June 2-6, 1985, pp. 18-14-22.

GRUVER, W.A., SOROKA, B.I., CRAIG, J.J., TURNER, T.L., 1983, " Evaluation of commercially available robot programming languages." 13th Int. Symp. on industrial robots and robots 7 , Chicago, Illinois, April 17-21, 1983, vol. 2, pp. 12-58-68

HARPER, R., 1983, "Microprocessor Based Multichannel Analyser Developed Using PolyForth." Microprocessors and Microsystems, Vol.7, no. 5, June 1983 pp. 217-222.

HAYNES, L. S., 1985, "Opportunities and Issues in Robot Standards." Robotics and Computer-Integrated Manufacturing, Vol. 2, no. 3/4, pp 161-164

HAURAT, A., THOMAS, M. C., 1983, "LMAC: A Language Generator System for the Command of Industrial Robots." Conf. Proc. 13 th Int. Symp. on industrial robots and Robots 7, April 1983, Chicago, vol. 2, pp 12-69-78.

HEDLEY, 1981, "Programming Forthwith. Conference on Microprocessor systems, Brisbane, 17-19 Nov. 1981, pp. 40-44.

HENDY, B., BRALEY, C., 1986, "High-Level Textual Robot Programming." AUTOMACH, Australia 1986. Pub. Soc. Man. Eng. Deurbor, MI, U.S.A. pp. 4-41-55.

HILBURN, J. L., JULICH, P. M., 1976, Microcomputers, Microprocessors: Hardware, Software and Applications. Prentice Hall, Englewood Cliffs, N.J. , U.S.A.

HILL, J. W. and SWORD, A. J., 1973, "Manipulation Based on Sensor Directed Control: An Integrated End effector and Touch Sensing system." Proc. of the 17th Annual human Factor Society Convention, Washington D.C., Oct. 1973.

JETLEY, S. K., 1984, "ROL: A Rhino Operating Language for the XR-2." Robot 8 -- Conf. Proc. June 4-6, 1984, Detroit, Machigan, Vol.2, pp. 13-45-55.

KOSSMAN, D., and MALOWANY, A., 1987, " A Multiprocessor Robot Control system for RCCL under IRMX" Proc. 1987 IEEE conf. on Robotic and Automation, March 31-April 3, 1987, Raleigh, North Carolina, Vol.3 pp. 1298-1306

KURAU, A., 1979, "Why The Programming Language PEARL could be useful tool for Programming in computer Control Manipulator." International Seminar on Programming Methods of Industrial Robots, 27-29 June,1979 , Grenoble, France, pp. 107-116.

LAGERGREN, P. J., 1983, Forth : Cheaper than Hardware. Forth Dimensions, Vol. 5, no. 2 pp. 13,14,32.

LAUGIER, C. , 1984, "Robot Programming Using a High Level Language and CAD Facilities." Proc. First European Robotics conf. Brussel June 1984, pp. 139-155.

LAUGIER, C., PERTON-TROCCAZ, J., 1985, "SHARP: A System For Automatic Programming Of Manipulation Robots." Third Int. Symp. Robotic Research, France 1985 pp. 125-132.

LEATHAM-JONES, B., 1987, Elements of Industrial Robotics, Pitman Publishing , London

LEVAS, A., SELFRIDGE, M., 1983, "Voice Communications with Robots." Conf. Proc. 13 th Int. Symp. on industrial robots and Robots 7, April 1983, Chicago, vol.2, pp. 12-79-83.

LHOTE, F., KAUFFMANN, J-M., ANDRE, P., TAILLARD, J-P., 1987, Robot Components and Systems, Robot Technology Vol. 4, Kogan Page, London

LIEBERMAN, L. I., WESLEY, M.A., 1987, "AUTOPASS:An Automatic Programming system For Computer Control Mechanical Assembly." IBM Journal of Research and Development Vol. 21 no. 4, July 1977. pp. 321-333.

LIENTZ, B.P., and SWANSON, E.B., 1980, Software Maintenace Management, Addison-Wesley, Reading (Mass.) , USA

LIM, K.B., and LOY, W.W., 1984, " Robot teaching and programming using graphic tablet." Robots 8 -- Conf. proc. June 4-6, 1984 , Detroit, Michigan, vol. 2, pp.20-20-35

LOTSPIECH, J. B., RUEHLE, T. M., 1984, :Generalised Interrupt Handler For A Forth Machine." IBM Technical Disclosure Bulletin, Vol. 27, no. 7A, Dec. 1984. pp. 3839-3845.

LOZANO-PEREZ, T., 1983, "Robot Programming." Proc. of the IEEE, Vol. 71, no. 7, July 1983 pp. 821-841.

LOZANO-PEREZ, T., JONES, J.L., MAZER, E., O'DONNELL, P.A., GRIMSON, W.E.L., 1987, " HANDEY : A robot sustem that recognizes, plans and manipulates." Proc. 1987 IEEE Int. Conf. on Robotics and Automation , Raleigh, North Carolina, March 30-April 3, 1987, vol.2 , pp. 843-849

MACINTYRE, F. (a) , 1985, "FORTH: The Optimum Angles Language for Microcomputers, Part One: Language; " International Laboratory, March 1985, pp. 12-18.

MACINTYRE, F. (b), 1985, "FORTH: The Optimum Language for Microcomputers, Part Two; Applications in the Laboratory. " International Laboratory, April 1985, pp. 14-22.

MAIR, G. M., 1987, Industrial Robotics, Prentice Hall , New York.

MALCOLM JR., D. R., 1988, Robotics -- An Introduction. 2nd Addition. PWS-KENT Publishing Co. Boston, U.S.A

MALONE, J. R., 1984, Comparative Languages. Pub. Chartwell-Bratt, U.K

MANDUTIANU, D., 1988, "Current Issues on High Level Robot Programming." Computers and Artificial Intelligence, Vol. 7, no.3, 1988. pp 203-217.

MANNONI, M., 1980, "Forth- An Extensive Path to Efficient Programs. Electronic Design." July 19, 1980, pp. 1-8.

MAZER, E. LM-GEO, 1984, "Geometric Programming of Assembly Robots." Advanced Software in Robotics. A. Danthine and M. Geradin (eds.) Elsevier Science Publishers, B. V. (North Holland), pp. 99-110.

Mc CARTHY, J., 1968 , " A computer with hands, eyes, and ears, " 1968 fall, joint computer conf., AFIPS proc. pp. 329-338

MC LELLAN, E., 1981, "Robot Languages -- The Current Position." Robot Technology by A. Pugh, IEE Control Series 23, pp. 112-124.

MILLER, R., 1988, "Fundamentals of Industrial Robots and Robotics" PWS - Kent Publishibg Company, Boston, U.S.A

MOHRI, S., TAKEDA, K., HATA, S., MATSUZAKI, K., HYODO, Y., 1985, "Robot Language from the stand point of FA system development -- an outline of FA-BASIC." Robotics and Computer-Integrated Manufacturing, Vol. 2, no. 3/4, pp. 279-292

MOORE, C.H., 1974, "FORTH : A new way to program a mini-computer, Astron. Astrophys. Suppl. 15, pp. 497-511.

MORRISSEY, W.J., 1985, "The Programming Language SAVVY for Robot Control" IPRC 85 pp. 170-174.

MOTOROLA, 1979 " M6809 Microprocessor Programming Manual" MOTOROLA, Inc., Semi-conductor Products Division Phoenix. U.S.A.

NAYLOR, A., SHAO, L., VOLZ, R., JUNGELAS, R., BIXEL, P. and LLOYD, K., 1987, "PROGRESS -- A Graphical Robot Programming System." Proc. 1987 IEEE Int. Conf. on Robotics and Automation, March 31-April 3, 1987,Vol. 3 , pp. 1282-91.

NATO, 1986, "Report of the working group -- Robot programming languages" Languages for sensor-based control in robotics , eds. U. Rembold and K. Horman, held at Inst. of comp. Sc.iii, Robotics Research Group, univ. of Karlsruhe, Fed. Rep. of Germany. Pub. Springer-Verlag, pp. 601-611.

PARKES, R., 1987, "FORTH Final year project 1987." Dept. Computer Science, University of Aston, Aston Triangle Birmingham, U.K.

PAUL (a) R. P., 1976, "WAVE : A Model -Bases Language for Manipulator control."

Technical paper, MR 76-615, Soc. of Man. Engineers, Dearborr, Machigan, 1976.

PAUL(b), R. P., 1983, "The early stages of Robotics." IFAC Real Time Digital Control Applications, Guadalajara, Mexico 1983 pp. 19-32.

PERTIN-TROCCAZ, J., 1987, " On-line automatic robot programming : A case study in grasping." 1987 IEEE Int. Conf. on Robotics and Automation, vol. 3 , March 31-April 3, North Carolina, Cat. no. 87CH2413-3, pp. 1292-1297

POUNTAIN, D., 1987, Object-oriented Forth -- Implementation of data structures. Academic Press Inc., (London) Ltd.

PRICE, R. P., 1984, "Off-line Programming with a Microcomputer." Robots 8 - Conf. Proc. June 4-6, 1984, Detroit, Michigan, vol. 2, pp. 20-95-102.

PUENTE, E. A., BALLAGUER, C., BARRIENTOS, A., 1986, "LRS : A High Level Explicit Programming Language for Sensor Based SCARA type Robots." Languages for sensor-based control in robotics, eds. U. Rembold and K. Hormann, Inst. of comp. sc. iii , Robotics Research group, Univ. of Karlsruhe, Fed. Rep. of Germany. Pub. Sprnger-verlag , pp. 25-43.

RAMBIN, R., and TAYLOR, E., 1986, " Robotic oriented computer languages." Wescon 86 Conf.record, Anaheim, CA, U.S.A. 18-20 Nov. 1986, Los Angeles, CA. pp. 1-9

RATHER, E. D., MOORE, C. H., 1979, "Forth High-Level Programming Technique on microprocessors, Forth Inc., pp. 1-8.

RAZ, T., 1989, " Graphics Robot Simulator for Teaching Introductory Robotics." IEEE Transactions on Education, vol. 32, no. 2, May 1989 pp.153-159

REBMAN, J., TRULL, N., 1983, "A Robot Tactile Sensor for Robot Applications." ASME Conference on Computers in Engineering, Chicago, August 1983.

ROCK, S.T., 1989, "developing robot programming languages using an existing language as a base --- a viewpoint " Robotica, vol.7, 1989, pp.71-77

RUOCCO, S. R., 1987, Robot Sensors and Transducers. Open University Press, Milton Keynes , U.K.

SADRE, A., SMITH, R., CARTWRIGHT, W., 1984, Co-ordinate transformations for two industrial robots . 1984 IEEE Int. Conf. on Robotics. Cat. no. CH2008-1/84 pp. 45-61

SALMON, W. P., TISSERAND, O. and TOULOUT, B., 1984, Forth, McMillan Computer Science series

SATO, T., HIRAY , S., 1987, "LAnguage- Aided Robotic Teleoperation System (LARTS) for Advanced Teleoperation." IEEE Journal of Robotics and Automation, Vol. RA-3, no. 5, Oct. 1987, pp. 476-481.

SHANKAR, K. S., 1980, "Tutorial: Data Structures, types and Abstractions." IEEE Computer April 1980. pp. 67-77.

SHENG, W. N., DAVIES, B. J., 1987, "A high Level approach to Programming a robot." Int. J. Mach. tools Manufact. Vol. 27, no. 1, pp. 57-63

SHIMANO, B., 1979 "VAL : An Industrial Robot Programming and Control system." International Seminar on Languages Methods for Industrial robots, pp. 47-59.

SHIMANO, B., GESCHKE, C., SPALDING III C., SMITH, P., 1984, "A Robot Programming System Incorporating Real Time and Supervisory Control : VAL II" Robots 8, Conf. Proc. vol. 2 , June 4-7, 1984 Detroit, Michigan, pp. 20-103-119.

SNYDER, W. E., 1985, Industrial Robots : Computer Interfacing and Control. Prentice Hall, Inc., Englewood Cliffs, New Jersey.

SOROKA (a), B. I., 1983, "What Can't Robot Languages do?" 13th Int. Sympossium on Industrial Robots and Robots 7 Vol. 2, April 1983, Chicago, pp. 12-1-5.

SOROKA (b), B. I., 1987, "CONC : A Program For Analyzing Concurrent Robot Programs without Loops." Proc. 1987 IEEE Conf. on Robotics and Automation, March 31-April 3, 1987, Raleigh, North Carolina, Vol. 3, pp. 1450-1454.

STAUGAARD, Jr., A.C., 1987, Robotics and Artficial Intelligence: An introduction to applied machine intelligence. Prentice Hall, INC, Englewood Cliffs, N.J. 07632

STAVENUITER, A. C. J., TER REEHORST, G., BAKKERS, A. W. P., 1989, "Transputer Control of a Flexible Robot Link." Microprocessors and Microsystems. Vol. 13, no. 3, April 1989, pp. 227-232.

St.CLAIR, J. and SNYDER, W. E., 1978, "Conductive Elastomers for Industrial Parts Handling Equipment." IEEE Transactions on Instrumentation and Measurement, IM27, March 1978, pp. 94-99.

STOBART, R.K., 1987, " Geometric Tools for the Off-line programming of robots ." Robotica (1987) vol.5, pp.273-280

SUMMERVILE, I., 1982, Software Enginnering, Addison-Wesley, London.

TODD, D. J., 1986, Fundamentals of Robot Technology : An Introduction to Industrial Robots, Teleoperators and Robot Vehicles. Kogan Page Ltd., London

TOMORIC, A., and BONI, A., 1962, " Development of a hand with pressure sensors," M.I.T. Lincoln laboratory, Res. report no. 192

TRACY, M.J., 1987 , " Forth in artificial intelligence " Proc. of third annual conf. on artificial intelligence and advanced computer technology, Long Beach, CA, 22-24 April, 1987. vol. 1 pp.118-121

TRONCIE, A., MARTINEZ,S., BABA, E., and HUGUES, C., 1988, " Modular Robots -- Graphical Interactive Programming." Proc. 1988, IEEE Int. Conf. on Robotics and Automation, April 24-29, 1988, Philadelphia, Pennysylvania, vol. 3, pp.1739-1743

TROVATO, K. S., SCHREINER, W., 1987, "LABICS : A Language for a Distributed Hierarchical and Dynamically Reconfigurable Control System." 1987 IEEE Workshop on Languages for Automation, Vienna ,pp. 43-47.

TULSKIE, W. A., DIMEO, D. R., 1983, "FORC : Forth Oriented Robot Control," Conf. Proc. of the Robotic Intelligence and Productivity Conf, Deteroit, MI, U.S.A. 18-19 Nov. 1983, pp. 51.

VAN AKEN, L., VAN BRUSSEL, H., 1988, Robot Programming Languages: The Statement of a Problem. Robotica (1988) Vol. 6, pp. 141-148.

VAN BREDA, I. G., PARKER, N. M., 1983, "Forth and Microprocessor Applications at the Royal Greenwich Observatory." Microprocessors and Microsystems, Vol. 7, no. 5, June 1983, pp. 203-211.

WECK, M., WIRTCH, D., EVERSHEIM, W., NIEHAUS, T., ZUHLKE, D., KALDE, M., 1984, "Requirements for Robot Off-line Programming Shown at the example ROBEX." Adv. Software in Robotics. A. Danthine and M. Geradin (eds.). Elsvier Sc. Pub. B. V. (North Holland) 1984, pp. 321-330.

WEISS, R., 1984, "Forth Forsakes its Image, Moving from Laboratory to Software Engineering. " Electronic Design, Dec. 27th, 1984, pp. 51-54.

WILL, P., 1979, Very High Level Languages for Robots. Int. Seminar on Language Methods for Industrial Robots, pp. 23-46.

WILL,P., and GROSSMAN, D., 1975, " An experimental system for computer controlled mechanical assembly," IEEE Trans. comput. , vol c-24, no. 9, pp. 879-888

WILL, P., 1979, " very high level languages for robots." Int. seminar on language methods de programming ind. robots, Grenoble, France, pp. 23-46

WLOKA, D.W., 1986, " Robsim -- Arobot simulation system." Proc. 1986 IEEE Int. Conf. on Robotic and Automation, April 7-10 , 1986, San Francisco, CA, Vol. 3, pp. 1859-1864

WOOD, B. O., FUGELSO, M. A., 1983, "MCL, The Manufacturing Control Language." 13 th Int. Symp. on Industrial Robots and Robots 7, April 1983 Chicago, vol. 2 , pp. 12-84-96.

APPENDIX A - FEATURES OF ROBOT PROGRAMMING LANGUAGES

1 WAVE

WAVE was one of the first robot programming languages developed at Stanford artificial intelligence laboratory in 1975 [PAUL a,1976], [PAUL b 1983]. It was designed to control Stanford arm and runs on the PDP-10 and PDP-6 computers. The language is interpretative and resemble assembly language programming. It was mainly developed for assembly applications such as assembling a water pump, brush calligraphy, crank turning, assembly of a pencil sharpener and mounting a door hinge. The language allows declaration of transformations to convert cartesian co-ordinates to joint co-ordinates. The position and orientation of objects is described by 4*4 homogeneous transformation matrices. The position is described in cartesian co-ordinates and orientation in Euler's angles. A transform variable T1 is used to combine position and orientation like this:

```
TRANS T1 20,30,1,0,90,0
```

where 20,30,1 are distances along x,y,z axis of the robot hand with respect to a fixed reference and 0,90,0 are Euler angles. The other data types provided by the language are VECTORS and LOOP COUNTERS. It also supports sensors such as force, touch and vision. The language supports the definition of MACROS. It is reported that the speed of operation of WAVE for assembly operations is between one third and quarter of that of a human operator. The main reason for this is that manipulator comes to rest at the end of each operation. Also the form of the program is very much like assembly language and lacks structure.

2 AL (Arm language)

AL, also was developed at Stanford artificial Intelligence Laboratory, is a high level programming language system and was developed as an extension to WAVE. It has

ALGOL like control and block structure and syntax. The AL system consists of three components [Mc LLELLAN 1981]

(i) The compiler

(ii) The interactive source code interpreter

(iii) Run time system.

The components (i) and (ii) run on a PDP-10 under the WAITS operating system and (iii) runs on a PDP-11. The language is written in PDP-11 assembler. It is based on concurrent PASCAL. It can be used to control up to four robot arms (2 PUMA 600 and 2 Stanford arms) simultaneously. The original implementation required a large main frame computer but stand alone computer systems were made available at later stage. The language uses co-ordinate systems which can be related to each other. It provides facilities for looping, branching and synchronising parallel processes such as performing calculations and simultaneous motion of arm. The language also has provision for describing the motion of robots and use of sensory data. It also provide data types for scalar (e.g arithmetic operations) as well as vector operations such as rotation and translation. [FINKEL et al.1975] have discussed language features with programming examples. A new interactive version of AL was written in Pascal OMSI to run on a PDP-11/45. This version which is particularly useful for controlling PUMA robots and can be obtained commercially. [BINFORD 1979] has pointed out the following features of the language:

(i) High level language with ALGOL like control structure.

(ii) Language is written in high level language (SAIL).

(iii) Language designed for two levels i.e motion level and planning level of assembly operations.

(iv) Control structures consists of FOR, WHILE, DO, IF CASE and MACROS.

(v) Data structures include scalars, arrays and variables.

(vi) Supports data base/world model.

(vii) Supports concurrency, synchronisation and exception handling.

(viii) Support force sensor control and vision.

3 HELP

HELP is an interactive high level language developed by General Electric company (Digital Electronic Automation, D.E.A) for their PRAGMA A3000 and ALLEGRO assembly robot [DONATO and CAMERA 1980]. It runs on DEC PDP-11 computer. It has ALGOL like block structure and syntax like PASCAL. The locations are specified in cartesian co-ordinates. The language is easy to learn and supports structured programming. It provides facilities for branching, subroutines, and parallel processing. The language can control up to four arms having up to 12 degrees of freedom for all arms. Sensors can also be integrated in to the system. The main application areas include automotive industry, electronic assembly and precision mechanisms.

4 LAMA-S

LAMA-S was developed by SPARTACUS project, France [FALEK and PARENT 1979]. It is based on a language LAMA developed at Massachusettes Institute of Technology (MIT), Artificial intelligence(AI) laboratory in 1976. It was meant for controlling AI robots. The purpose of the language was to develop robots for handicapped people in everyday life such as serving drinks and food. Its syntax is comparable with assembly language programming. But the language is much more flexible and easy to write than other conventional assembly level languages. LAMA-S used APL as implementation language. The user level commands are translated in to an intermediate low level commands language known as PRIMA before execution. The language allows the execution of a program either sequentially or in parallel format. It provides parallel processing, sensor integration, frame instructions, mathematical operations, control instructions and control of I/O devices. According to the authors, APL implementation of the language is not suitable for industrial robots due to the following shortcomings:

(i) Require APL machine to run.

(ii) APL syntax is not convenient to follow.

(iii) The language syntax is not always perfect.

(iv) It is difficult to implement interactive programming.

5 SIGLA (SIGma LAnguage)

SIGLA language was developed in 1975 by Olivetti for their SIGMA robots [BANZANO and BURONZO 1979]. It is a complete software system and incorporate a supervisor to interpret the job, a teaching module to allow teaching by guiding, an execution module, provisions for editing and saving the data. This is mainly used for assembly purposes. The basic system runs on 8K 16bit word length general purpose minicomputers having 4K ROM and 4K RAM with optional RAM available up to a maximum of 32K. The language provides the following features:

(i) Parallel operations

(ii) Simultaneous control of more than one arm.

(iii) Interpretative structure

(iv) Variable instruction set to suit the user's requirements.

The main drawback of the language is that instructions do not convey the meaning of the actions taken by them, hence programs are difficult to write.

6 RAPT (Robot APT)

RAPT is a robot assembly language developed at University of Edinburgh [AMBLER 1982]. The language is based on numerically controlled machine tool language known as APT. It employs world modelling and describe assembly language operations in terms of relative spatial relationship between the objects rather than specifying the actual motion of robot. The program consists of the description of the parts, the robot, workstation and assembly plan. The system keeps a careful record of the relative geometric relation of objects in the assembly after each operation. This type of programming is known as object level programming as description of assembly operations is independent of the robot motion.

RAPT uses contact relations between objects such as AGAINST, FIT, COPLANAR to specify the relationship between object features. These features which can be planar or spherical faces, cylindrical shafts, edges and vertices are defined by means of frames. A typical command may be as follows:

```
PLACE block1 SO THAT (block2-face1 AGAINST block1-face1)
```

This instruction will place block 1 on block 2 with their first face against each other. Another example program example may be like this :

```
move/A,PER TO ,( TOP of B ),-3
```

i.e to move object A towards the top of B by 3 units of distance.

The primary emphasis of the RAPT has been on task specification. However it does not deal with obstacle avoidance, automatic grasping or sensory operations.

7 VAL (Versatile Assembly Language)

VAL is a computer based control system and programming language which was designed specifically to control a PUMA robot of Unimation Inc. [SHIMANO 1977]. It has been designed over a period of several years. The first version of the language was written in 1975 to run on PDP-11/45. The language is based on WAVE which was originally developed at Stanford artificial intelligence laboratory. The original version of the language had only few instructions set such as controlling point to point motion and single segment joint interpolated motions. New facilities such as integer arithmetic, various control structures and increased trajectory control were made available during 1976-78. In 1978 it was commercially made available to control unimation PUMA robot. The purpose of the language is to provide facilities to easily define robot tasks. The language provides an editor which allows the user to create new or modify existing programs. It allows deletion or insertion of new program steps. The language being interactive, the editor can also be used to help program debugging as modifications to program can be made and tested instantly. The language also provide facilities for specification of location (position and orientation) of robot, program branching, subroutines, integer calculations, signalling to external devices and system interrogation (i.e state of currently executing program).

A new version of VAL known as VALII was later introduced. It provides the following facilities [SHIMANO et al. 1984] :

(i) Network communication facilities which allows the system to be interfaced to a supervisory computer.

(ii) Mathematical capabilities equivalent to those found in a structured high level computer programming language.

(iii) Enhanced operator interface.

(iv) Extended sensory interfacing capabilities.

(v) Real time trajectory modification.

(vi) Facilities for performing simultaneous robot and process control activities.

8 MAL (Multipurpose Assembly Language)

MAL [GINI et al. 1979] is a multitask system for mechanical assembly orientated applications. The language was developed at the Milan Polytechnic, Italy to program a specially designed two arm super system robot for assembly operations. The syntax of the language is like basic. The purpose of the language was to design an interactive language which is easy to understand. The main features of the language include parallel execution, sensor support, process synchronisation and various movement instructions.

MAL system is made up of two components; compiling and executing. The compilation part allows the user to create, update and maintain the source program. The execution part is responsible for the execution of the sequences in the program. The important feature of the language is provision of task synchronisation between different activities in the assembly operations for parallel programming.

9 AML (A manufacturing language)

AML was developed by IBM in 1982 initially to control IBM 's robot RS-1 (7565 robot) and 7535 electric robot from SCARA assembly robot which is a cartesian arm with linear hydraulic motors and active force feedback from end effector sensors and is produced in Japan by SANKYO. The language was the result of research carried out by IBM in their laboratory at York Town Heights in the mid 1970's. The aim was to develop a general purpose plant floor automation language rather than just a robot programming language. The language is well structured and highly interactive. It contains subsets which are suitable for the programmers with wide range of experience. Various implementations of the original version were produced to improve the language and overcome its short comings as a general purpose plant floor language between 1982-84.

The main features of the language are summarised below [GROSSMAN 1985], [GROSSMAN and SHORT 1985] :

(i) A powerful language with structured programming capabilities.

(ii) Supports sensor integration.

(iii) Computer independent as it runs on IBM 370, 4360, 5531 and Motorola MC-68000.

(iv) Robot independent as it can control different robots.

(v) Can control NC machines and other mechanical devices such as material transport system, material storage system etc.

(vi) Although language is interactive it also provide compilation facilities in applications which do not involve motion such as vision system and geometric modelling where speed of the system is an essential requirement.

(vii) Offers multitasking facilities, semaphores for process synchronisations and a method for managing the terminal as a shared resources.

(viii) Open system i.e to allow users to link AML to their own software written in C or other languages.

(ix) Other features such as integration with CAD/CAM systems and data types and operator extensibility.

10 AUTOPASS (Automated Parts Assembly System)

AUTOPASS is a very high level programming system for the computer controlled mechanical assembly operation. It is being developed by IBM at T.J. Watson research centre in New York [LIBERMAN and WESLEY 1987]. The language is based on PL1 and make use of the facilities of this language as well as its data structure. It is a task level programming language. The language describes assembly operations rather than robot motions.

The language is based on relationships between objects and assembly operations which allows the user to describe assembly operations to the robot in the same way as to instruct a human operator using ordinary English like statements. A typical assembly instruction given to an operator may be like this:

Screw a nut onto a bolt.

The language involves setting of a data base known as world model and contains information about all the objects. It will have information regarding the shape, size, location, physical properties such as stability of objects and support relationship between the objects. It also contains information regarding the spatial positions and relations amongst objects and assembly or attachment relationship between them. Each object is a modelled as a polyhedron i.e in terms of their vertices, edges and surfaces, by using a geometric design processor and accessed by a pointer at the object vertex. These techniques are widely used in computer aided design (CAD) and allow inference deductions for collision checking during trajectory planning. The world model is updated at the end of each assembly operation as the relationship between the objects may change. The programming system decides on grasp choice and plan robot trajectories.

The language statements can be divided into two classes namely assembly related and miscellaneous statements. The assembly related statements are concerned with specification of assembly operations. The miscellaneous statements are used for specification of control flow, declaration of geometric variables and description of inspection operation. AUTOPASS provides three groups of assembly related statements:

(i) State change statements which describes an assembly operation such as placement and adjustment of parts.

(ii) Tools statements which describes the types of tools to use.

(iii) Fastener statements which describes a fastening operation.

For example, an operation of inserting a bolt and tightening may appear like this in AUTOPASS:

```
PLACE bolt ON beam SUCH THAT bolt-tip IS ALIGNED WITH beam-bore;
```

The compiler transform the assembly instructions in to a program that directs the robot through the necessary motions required to execute the task. The motion commands are generated by consulting a geometric data base which contains up to date information about all the objects and their interrelations. During compilation the user interacts with the compiler to resolve any ambiguities detected by the compiler in the program.

11 LM (Language Manipulation)

LM was developed at the University of Grenoble, France in 1979 for the programming of assembly robots and is implemented commercially by machine dynamics [HENDY and BRALEY 1986]. It was written to control Robitron, Renault and Kremlin robots. The language is based on PASCAL. It has retained many features of Pascal and provides robot specific routines. In addition to the general features of Pascal, it provides special frames for the specification and orientation of the robot. The language allows communication with other devices in the robot work cell including other robots, I/O operations, world modelling, interaction with various sensors including vision system. An extension of the language LM-GEO [MAZER 1984] first developed at the Artificial Intelligence department of Edinburgh university support object level programming by describing geometrical relationships between the objects, a unique feature of RAPT

12 MCL (Manufacturing Control Language)

MCL was developed by McDonald Douglas Corporation of U.S.A. It is a high level language and is an extension of a popular numerical control language known as APT. The philosophy behind the creation of this language was that an off-line programming language must be able to control the modern robotic workcells including robots and associated machinery. The language is aimed at off-line programming of robots and controlling a number of other devices associated with the robots such as conveyor belts, vision system and other machines which form a complete automated workcell. MCL provide the following capabilities [WOOD and FUGELSO 1983]:

(i) To contol any robot by selecting an appropriate routine.

(ii) To control complete manufacturing cells including all the devices in the robot workcell.

(iii) To process real time sensory data and make logical decisions based on this information.

(iv) Vision processing for locating or inspecting components in the workcell.

(v) Program verification to indicate whether a given program can be used on any of the cells described in the data base.

(vi) Allow different frames of reference to define points and objects.

13 LRS (Lenguaje para Robots con Sensors)

LRS is a PASCAL based language to control SCARA type robots with sensors attached to it [PUENTE et al.1986] developed at University of Madrid in 1987. It is claimed that the language is highly portable and can be transferred to other computer hardwares and can control different robots with slight modifications. The language is implemented on PDP-11/73 mini-computer and runs under RSX-11M operating system. The SCARA robot known as DISAM E-65/81 has four degrees of freedom was also developed by the university. It is provided with force/torque, ultrasonic range detector, a mobile camera, a static camera and touch sensors. The language provides basic data structure of PASCAL as well as extended specialised data structure for the definition of locations, specification of motion etc. It also provides two levels of movement instructions namely guarded and compliant motions. In the guarded movements, the system obtains the status of various sensors and verify the command to be executed. The compliant motion require constant feedback from sensors such as force/torque sensors. Since the language is based on PASCAL, it supports structured level programming.

14 ROBEX

ROBEX is an off-line programming system developed at Aachen, West Germany [WECK et al. 1984]. The language is based on popular numerical control machine language known as APT. The aim was to develop a language to control a complete flexible manufacturing system (FMS) controlling robot and other devices in its environment. It allows sensor signal processing and geometric modelling.

15 PASRO (PASCAL for Robots)

PASRO was developed by Biomatic, a German company and is based on Pascal [BIOMATIC 1983]. It has geometric data types similar to SRL. The language is provided with robot specific library of routines to make it suitable for robot programming. It provides standard data types of Pascal such as integer, real, boolean and structured data types like arrays, records and files etc. The language supports arithmetic operations, co-ordinate transformation, trigonometric functions, frames, rotation matrices and world modelling. It also provides move commands by the specification of joint and distances, speed control and sensor integration.

16 FA-BASIC (Factory Automation)

FA-BASIC is a unified language developed as an extension to BASIC [MOHRI et al.1985]. The main purpose of the language was to develop a system which can be used not only to control a robot, but also its associated devices. Normally, these devices are controlled by using different hardware and software systems. This language is meant to control all the devices in a factory workcell. The language consists of the following three sub systems:

(i) FA-BASIC/R ; for controlling robots.

(ii) FA-BASIC/V; for controlling vision systems.

(iii) FA-BASIC/C; for programming programmable logic controllers.

The language provides two types of commands known as common commands and problem oriented commands. The former are used for program declaration, expression and flow control and has the same format as BASIC, whereas the latter are provided for programming devices including robots. The FA-BASIC/R which control robot, uses a teaching system which saves data for taught positions in the form of a table. The language also provides commands for sensor integration and graphical teaching techniques. The software is developed in modular form.

17 RAIL

RAIL was developed by Automatix Inc. as a high level language for computer aided manufacturing [FRANKLIN and VANDENBURG, 1982]. It is designed to control company's Robovision (a system for robot arc welding), Cybervision, (assembly operations) and Autovision (Machine vision) systems and general manufacturing equipment. It is an interpreter language and is based loosely on PASCAL. The system runs on Motorola 68000. The language provides commands for moving robot, welding equipment and offers an easy accessing input or output lines connected to the equipments. Data types includes integers, real numbers, character strings, arrays, points, paths and reference frames. Program control structures are similar to those of PASCAL. The language also supplies a library of mathematical operators such as square roots and trigonometric functions.

18 AR-BASIC (American Robot- BASIC)

AR-BASIC is a trade mark of American robot corporation [Gilbert et al. 1984] and was made commercially available in 1983. The language is based on Basic and provides interpretive programming system. The main aim of the system was to provide a language to control various manufacturing devices in industry including robots. It is proclaimed that the language provides programming environment which increases productivity for designing software for either stand alone robots or complete flexible manufacturing system (FMS) environment. The programming system consists of the following components:

(i) Basic language facilities.

(ii) Provision for position definition and motion control.

(iii) Text file editing.

(iv) Provision for controlling I/O devices.

(v) File and memory system control.

(vi) Vision system programming.

The author claims that the language is highly user friendly and provides a set of English like commands. The language provides interactive debugging facilities and contain commands to locate the position of program breakpoint, robot's current position, speed and frame reference etc. The data structure of the language allows the programmer to define robot position, carry out motion control, control of I/O devices and vision system. It also has provision for the interpolation to achieve straight line, joint co-ordinate or circular motion. It also supports sensor integration via I/O ports.

19 SRL (Structured Robot Language)

SRL was developed in 1984 by [BLUME and JACOB 1984] at the University of Karlsruhe, West Germany. The language is based on experience with the robot programming language AL with PASCAL elements. The language is supposedly easy to learn, independent of hardware used and adaptable to new applications. It provides facilities for parallel processing, sensor integration, world modelling and variety of robot motion commands. The data structure is based on PASCAL with new data types such as SEMAPHORE and SYSFLAG provided for process synchronisation. The SEMAPHORE is used for synchronisation and queuing with programs and SYSFLAG for synchronisation between programs. Some of the data types such as VECTER, ROTATION and FRAME are inherited from AL. The frame is used to describe the position and orientation of the robot gripper. The language supports a planning module for task oriented programming which would consult world model, get sensor feedback and allow the automatic program generation. One part of the program is known as system specification, which allows the language to adopt to different robots, hardware types and sensors. This feature makes the language portable. The language also provides various move commands such as to achieve point-to-point motion, linear interpolation, trajectory calculations and via points etc.

APPENDIX B - FORTH WORDS

STACK MANIPULATION WORDS

Operator	Pronunciation	Stack notation	Action
.	dot	(n-----)	prints top no.
DUP	dupe	(n-----nn)	duplicate top no.
SWAP	swap	(n1 n2---n2 n1)	reverses top two no. on stack
ROT	rote	(n1 n2 n3---n2 n3 n1)	rotate 3rd item to top
OVER	over	(n1 n2 ----n1 n2 n1)	make copy of 3rd no.to top
PICK	pick	(n1 ---n ----n1)	copy nth no. to top of stack
ROLL	roll	(n1---n---n1)	rotate nth no.to the top of stack
?DUP	query-dupe	(n---nn)	dup if n is non-zero
DEPTH	depth	(n1 n2 n3-----n)	return count n, total no. of item

Table B1 showing stack manipulation words.

ARITHMETIC OPERATION WORDS

Operator	Pronunciation	Stack effect	Action
+	plus	(n1 n2---n)	adds n1 and n2
-	minus	(n1 n2--- n)	subtract n2 from n1
*	star	(n1 n2---n)	multiplies n1 and n2
/	slash	(n1 n2---n)	divides n1 by n2 leaves quotient
MODE	mod	(u1 u2----urem)	returns remainder only
/MODE	slash-mode	(u1 u2---urem,uquot)	returns remainder and quotient after division

Table B2 showing arithmetic operators.

DOUBLE PRECISION NUMBERS AND OPERATORS

Word	Pronunciation	Stack effect	Action
2SWAP	two-swap	(d1 d2---d2 d1)	swap top two numbers
2DUP	two-dup	(d----d d)	duplicate top number
2DROP	two-drop	(d---)	remove the top number
2OVER	two-over	(d1 d2---d1 d2 d1)	copy second number to top
D+	d-plus	(d1 d2---d)	adds two 32 bit numbers
D-	d-minus	(d1 d2---d)	subtracts d2 from d1
DNEGATE	d-negate	(d-- -d)	changes sign of 32 bit no.
D.R	d-dot-r	(d n-----)	prints signed 32 bit number right justified

Table B3 double length stack and arithmetic operators.

MIXED LENGTH OPERATORS

Word	Pronunciation	Stack effect	Action
M+	m-plus	(d n------d)	adds a 32 bit no. to a 16 bit no., returns 32 bit number
M/	m-slash	(d n-----n)	divide 32 bit no. by 16 bit no.returns 16 bit quotient
M*	m-star	(n d-----d)	multiply two 16 bit nos, returns 32 bit result
M*/	m-star-slash	(d n n----d)	multiply 32 bit no. by 16 bit no., divides resultby 16 bit, return 32 bit result

Table B4 showing mixed length operators

RETURN STACK WORDS

Word	Pronunciation	Stack effect	Action
>R	to-R	(n----)	move n to return stack
R>	r-from	(-------n)	transfer to parameter stack
I	i	(---------n)	copies top of return stack no. to parameter stack

Table B5 return stack words

COMPARISON OPERATORS WORDS

Operator	Stack effect	Action
=	(n1 n2-----f)	true flag if equal
>	(n1 n2 ----f)	true flag if n1 > n2
<	(n1 n2 ----f)	true flag if n1 < n2
0=	(n---------f)	true flag if n is 0
0<	(n --------f)	true flag if n is -ve
0>	(n----------f)	true flag if n is +ve
NOT	(f1---------f2)	reverses the sign of flag

Table B6 showing comparison operators.

FORTH 83 ADDRESS CONVERSION WORDS

Word	Pronunciation	Stack e
ffect		
>BODY	to-body	(CFA----PFA)
>NAME	to-name	(CFA-------NFA)
>LINK	to-link	(CFA--------LFA)
BODY>	from-body	(PFA--------CFA)
NAME>	from-name	(NFA--------CFA)
LINK>	from-link	(LFA--------CFA)
N>LINK	Name-to-link	(NFA--------LFA)
L>NAME	Link-to-name	(LFA -------NFA)

Table B7 showing some Forth 83 address conversion words.

APPENDIX C - FORTH MEMORY MAP

Figure 1 below shows the memory map for the implementation of the Forth in this research project. Under the Flex operating system the RAM from $0000 to $BFFF is free for application programs to use. The RAM from $C000 to $FFFF is reserved for Flex. The area of memory between $0000 to $05FF is free for use by the Flex print spooler, which enable Forth users to access this function while using Forth. The number of block buffers are implementation dependent. However, there is a trade off between memory used and speed of operation. The more buffers that are available, the faster disk access will be but less RAM will be available to the user. As a compromise two block buffers were used in this implementaion, which is also, in most of the Forth implementations. The system variables are placed immediately after the predefined dictionary and before the user dictionary.

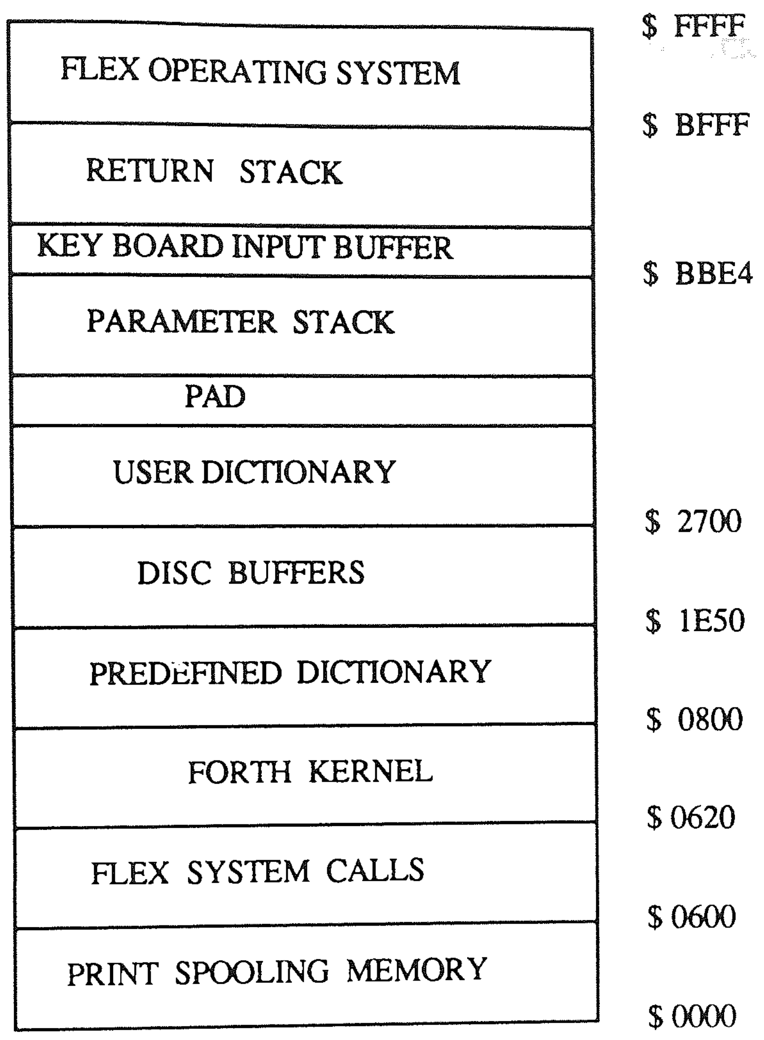

Figure C1 showing Forth memory map.

APPENDIX D - FORTH EXTENSION WORDS AND THEIR HIERARCHICAL DIAGRAMS

The following tables include Forth extension words in alphabetical order with brief explanation and stack manipulations using standard Forth notations. The symbols to the left of dashes indicate the order in which item are placed on the data stack and those on the right-hand indicate the items left on the stack after the execution of a Forth word. The hierarchical relationship between these words is shown in figures D1-D6.

(a) ADDITIONAL MATHEMATICAL OPERATIONS WORDS

word	stack manipulation	explanation
ARC-COS	n ----- angle	given input arc-cos value, returns corresponding angle.
ARC-SINE	n ----- angle	given input arc-sine value, returns corresponding angle.
ARC-SINE-ANGLE	n ----- n	return arc-sine angle
ARC_SINE-RANGE?	n ---- n f	carries out range check on input angle, returns angle and flag.
ARC-SINE-SGN	n ---- [n]	save sign of an angle in variable SGN and returns absolute value.
COS	n --- n	return cosine value multiplied by 10,000 of an input angle.
EQL-FLG	--------	variable contains 0 or 1 if two angles are same or different.
INTERPOLATE	n,n ---- angle	carries out interpolation
INTERVAL	n --- 2 or 10	decide on search interval of 2 or 10.

SEARCH	n --- angle	carries out search, interpolate and returns angle.
SEARCH1	n ---- n	search look up table at 30^{o} interval
SEARCH2	n ---- n, interval	perform second search at 2 or 10^{o} interval.
SGN	---------	variable to save sign of an input angle.
SIN	-----	constant to save sine values multiplied by 10,000 for angles from 1-90^{o}
SINE	n --- n	return sine value multiplied by 10,000 of an input angle.
SN	-------	varible to hold intermediate sign flag
TAN	n ---- angle	return tangent value multiplied by 10,000 of an input angle
SQUARE	n ------ n	provide square of an input number
SQRT	n -----n	calculate square root of n
TANGENT	-------	constant which contain tangent values times 10,000 for angles between 1-90^{o}
UPDATE-SN	n ------	update contents of variable SN
?=<	n1,n2 ---n1,f	compare n1 and n2, returns n1 and true flag if n1 is equal to or less than n2.

Table D1 Additional mathematical operations words

(b) INPUT/ OUTPUT WORDS

word	stack manipulation	explanation
ALL-INPUTS	------ 0	leaves 0 on the stack

ALL-OUTPUTS	-------- FF	leaves 255 on the stack
APORT	---------	provide address for port A
APORT-INPUT	-------	configure port A as data input
APORT-OUTPUT	------	configutes port A as data output
ASIDE	addr -----	write port A addresses in to variables.
BPORT	---------	provide address for port B
BPORT-INPUT	------	configure port B as data input
BPORT-OUTPUT	------	configure port B as data output
BSIDE	addr -------	write port B addresses in to variables
CLINE1	n1,n2 ----- f	perform logical OR on n1 and n2
CLINE2	n1,n2,n3 --- f	perform logical OR twice
CLRFLGSA	-------	read data register A to clear control reg. A
CLRFLGSB	------	read data register B to clear control reg. B
CONFIG-ROBOT	------	configure robot to computer
CREGA	------	variable to hold the address of control register A
CREGB	------	variable to hold the address of control register B
DATADIRSA	n -----	configure port A as input or output
DATADIRSB	n -------	configure port B as input or output
DDREGA	-------	variable to hold the address of data direction register A
DDREGB	------	variable to hold the address of data direction reister B
DREGA	-------	variable to hold the address of data register A
DREGB	----	variable to hold the address of data register B

ELBOW	----- 3	constant leaves 3 on stack
GRIPPER	----- 6	constant leaves 6 on stack
HAND	----- 5	constant leaves 5 on stack
H>L	---- 0	detect high to low transition
HNDSHK	----- 0,0	leave 0,0 on the stack for PIA to act in handshake mode
INPUT	------- 0	leave 0 on the stack for selecting port as input
IRQ	----- 1	used to select PIA for interrupt mode.
IRQCA1?	--------	check if bit 7 of the control register A is set
IRQCA2?	------	check if bit 6 of the control register A is set
IRQCB1?	------	check if bit 7 of the control register B is set
IRQCB2?	----	check if bit 6 of the control register B is set
L>H	----2	detect low to high transitions
NOIRQ	-----0	configure PIA for no interrupt mode
OUTPUT	------ 4	leave 4 on the stack for selecting port as output
PPORT	n --- addr	return address of 0 port on the stack
PULSMD	---- 1,0	configure PIA for pulse mode
READATA	------n	read contents of data register A
READATB	-----n	read contents of data register B
READCRA	----- n	read contents of control register A
READCRB	------ n	read contents of control register B
SET-ROBOT	--------	move robot to set position
SHOULDER	------ 2	constant leaves 2 on stack
SLCTDR	----- 4	leave 4 on the stack
TOGDN	-------2, 0	configure PIA for toggle down mode

TOGUP	---- 2, 1	configure PIA for toggle up mode
WAIST	------ 1	constant leaves 1 on stack
WRIST	------ 4	constant leaves 4 on stack
WRITECRA	n ----	write n into control register A
WRITECRB	n ------	write n into control register B
WRITEDATA	n -------	write n into control register A
WRITEDATB	n ----	write n into control register B
>CREGA	n1, n2 ----	write into control register A
>CREGB	n1, n2 ---	write into control register B

Table D2 Input/Output words

(c) JOINT MOVEMENT WORDS

word	**stack manipulation**	**explanation**
ACLK	n1,n2 ---- n	used with STEP to move hand anticlockwise
BCK	n1, n2 ---- n	used with STEP to move shoulder backward
CLKW	n1, n2 ---- n	used with STEP to move wrist clockwise
DATA	-----------	array to store data sent to joints

DELAY	---------	cause delay of approx. 1 second
DELAYS	n ------	cause specified delay
DISABLE-JT	-------	disable robot joint from computer
DN	n1,n2 ---- n	used with STEP to move elbow downward
ELB-DAT	------- n	provide latest elbow data
ENABLE-JT	---------	enable a robot joint to computer
FRW	n1, n2 ------ n	used with STEP to move shoulder forward
GRIP-CLOSE	--------	closes gripper
GRIP-DAT	------- n	provide latest gripper data
GRIP-OPEN	------	open gripper
HAN-DAT	------ n	provide latest hand data
INIT-DATA	---------	initialises array called DATA
JT-ADDR	------	vriable to hold address of a joint
LEFT	n1, n2 ---n	used with STEP to move waist to left
MOVE	n1, n2 ----	move a specified joint (n1) by data n2
READ-JT	------- n	read contents of a variable JT-ADDR
RIGHT	n1, n2 --- n	used with step to move waist right
SHL-DAT	------ n	provide latest data sent to shoulder
STEP	n1,n2 --- n1,n3,n4	used to move a joint (n1) by n2 steps from current position
UP	n1,n2 ---n	used with step to move elbow joint up
UPDAT-JT	n ------	update contents of variable JT-ADDR
WAI-DAT	--- n	provide latest data sent to waist
WRI-DAT	----- n	provide latest data sent to wrist

Table D3 Joint movement words

(d) Point to point movement words

word	stack manipulation	explanation
A1	-----------	constant to hold fixed height of shoulder
ALPHA	-------	vriable to hold intermediate angle α
ANGLES?	----- n1, n2 ---- n6	display all joint angles
ARM-MOVE	--------	move arm to specified position and orientation
B1	-------	constant to hold length of shoulder
BEETA	---------	variable to hold intermediate angle
C1	--------	constant to hold fixed length of elbow
CALC-ALPHA	-------- α	calculate intermediate angle α
CALC-BEETA	---------- β	calculate intermediate angle β
CALC-GAMA	----------γ	calculate intermediate angle γ
CALC-J1	--------	calculate data for joint 1 and saves in array JOINTS
CALC-J2	---	calculate data for joint 2 and save in array JOINTS
CALC-J3	------------	calculate data for joint 3 and save in array JOINTS
CALC-J4	---------	calculate data for joint 4 and save in array JOINTS
CALC-J5	-----	calculate data value for joint 5 and save in array JOINTS
CALC-j6	-----	calculate data for joint 6 and save in array JOINTS
CALC-L2	----------	calculate sum of x^2 and y^2 and save in variable L2
CALC-SAVE	-----------	calculate data for all joints and save in array JOINTS

CALC-THEETA2	------	calculate angle θ_2 and save in array THEETA
CALC-THEETA3	---------	calculate angle θ_3 and save in array THEETA
CALC-XY	-------	resolve hand angle into x, y components and save them in variables X1 and Y1 respectively
D1	-----------	constant to hold fixed length of hand
INIT-JOINTS	-------------	initialises array JOINTS
INIT-THEETA	----------	initialises variable theeta
JOINTS	-------	array to hold data for all joints
JOINTS?	------ n1, n2, n3 --- n6	display data for all joints
ORIENT	---------	write wrist and hand angles into array THEETA
POSTN	x,y,θ ———— –	write co-ordinate values into variables X2, Y2 and array THEETA respectively
THEETA	--------	array to hold joint angles
X1	------	variable to hold intermediate value of x co-ordinate
X2	-----------	variable to hold input value of x co-ordinate
Y1	----	variable to hold intermediate value of y co-ordinate
Y2	-------	variable to hold input value of y co-ordinate
=<	n1, n2---- n1,f	check if n1 is equal to or less than n2
=>	n1,n2 ---- n1, f	check if n1 is equal to or greater than n2

Table D4 point to point movement words

(e) simulator words

ARM-DAT	--------	array to save arm data
?ARM-DAT	----- n6,n5--- n1	leave on stack data sent to arm
ARRAY-BLK	addr, n2,n1 -----	move n1 blocks starting with block n1 from array whose starting address is addr
ARRAY-->BLOCKS	-------	transfer contents of index and data arrays to disc
ARRAY-TRJ-BLKS	--------	move arrays to disc blocks
ARRAY-TRJ-DAT	-------	transfer trajectory data from array to blocks
ARRAY-TRAJ-IND	--------	move trajectory index from array to blocks
BLK-ARRAY	addr,n2,n1 -----	move contents of n1 blocks starting with n2 to array whose address is addr
BLK-NO.	-----	variable to hold block number on which trajectory is saved
BLK-NUM-OFFSET	addr ----- n1,n2	given address (pfa) addr of a trajectory, returns its block number n2 and block offset n1
BLK-OFFSET	-------	variable to store block offset of a trajectory in a block
BLK-SAVE	addr,n1,n2 ---	given start adress (addr) of a trajectory name, save it on a block n2 and at an offset of n1
BLOCK-ARRAY	n1,n2 ---	a defining word, expect start block no. (n1), and no. of blocks (n2) to be used for defining an array whose name should follow this word
BLOCKS-->ARRAY	----	transfer contents of index and data blocks from disc to array
C,	b -------	allow 8 bit value to be compiled at address given by HERE

CALC-BUFF-ADDR	n1,n2 ---- addr	given block no. n2, block offset n1 give starting address of a trajectory
CALC-BYTE-MOVE	n1,n2,--- n1,n2,n3	calculate no. of bytes of trajectory data to be removed
CALC-BYTE-REMOVE	n1 ----- n1,n2	calculate no. of bytes to be removed from trajectory index
CALC-LINE-NO.	addr --- addr,n	calculate line number to be removed from index array given trajectory address (addr) , leave line no. n on stack
CHECK-END	addr---- addr,f	given pfa (addr) of a trajectory, check if end of trajectory index is reached
CLR-DAT-BLKS	-------	initialise disc blocks used for saving trajectory data
CLR-IND-BLKS	-------	initialise disc blocks used for saving trajectory index
CLR-TRJ-BLKS	----	initialise index and data blocks on disc
COMBINE-FLAGS	addr,f,f,f --- addr	combine flags during index search
COMPILE-DATA	addr,b ---- addr,b	compile data at address (addr) for b no. of points
CONV-SIMU-DATA	------	scale simulator data before sending to arm
DATA-BLOCKS	----------	word defined by block-array to be used for storing data of trajectories to be created using simulator
DATA-BYTES-PULL	n1,n2,n3 -----	pull n3 bytes in array data given block-offset n1 and and no. of bytes n2 in a trajectory
DELAY	------	causes delay of approx. 1 second
ENABLE-SIMU	----	configure simulator
ENABLE-SIMULATOR	--------	enable simulator to the arm
GET-BLKS	addr -----n2,n1	given starting address of an array return no. of blocks used (n1) and start block no. n2
INDEX-BLOCKS	----	word defined by index-block to save index of trajectories to be saved using simulator

IND-BYTES-PULL	addr,n ---	pull index bytes using addr as starting address of trajectory name and n its line no.
INIT-SIMU	--------	initialises array simu
JT-MOVE	addr ------	send simulator data to arm joints
LEARN	---------	expect trajectory name on stack at run time for creating a new trajectory using simulator
LINE-TYPE	addr1,addr2,n - ----	display a specified (n) trajectory index line
LINES-TYPE	addr1,addr2,n - -----	display n no. of trajectory index lines
READ-ARRAY	addr--- addr,n1,n2	read contents of array, return start address addr, no. of bytes used n1 and no. of start block
REPLAY	addr --------	given address of a trajectory , pfa replay its action
SAVE-SIMU-SCALE	-------	save scale up values in array simu-scale
SEARCH-NAME	addr---- f,f,f,addr	search the name of an entered trajectory in the index, given its pfa
SELECT-KEY	---------f	wait for a y or n key to be pressed
SIM	----	enable simulator on inputting a key
SIMU	-----	array to save simulator data
?SIMU	-----n1,n2---n6	read data from simulator
SIMU-ARRAY	n -------	defining word which expect size n and name of an array at compile time
SIMU-DATA	------	array created to save simulator data using defining word simu-array
SIMU-RPT	----	allow simulator to enable again
SIMU-SAVE-DATA	---------	enable simulator, saves its data in an array called simu-data
SIMU-SCALE	------	array to hold scale up values for simulator
START-KEY	------	allow simulation to start after pressing a key
TERMINATE	addr,b ------	ask for an input key to be pressed to terminate simulation

TRAJ-DELETE	-------	ask to input a saved trajectory and delete it
TRAJ-ENTER	------	ask to enter the name of a saved trajectory
TRAJ-INDEX	-	variable to hold start block no. of trajectory index
TRAJ-NAME	----	array to save entered trajectory name
TRAJ-NO.	-------	variable to store no. of trajectories saved
TRJ-DAT-ARRAY	-------	move trajectory data from disc blocks to array
TRJ-DATA-LOCATE	------- addr	ask to enter trajectory name, search index and leave address where data is stored
TRJ-DIRECTORY	------	display all the trajectories saved in index
TRJ-IND-ARRAY	------	move trajectory index from block (disc) to array
TRJ-NAME-LOCATE	----addr	ask to enter trajectory name, search index and leave its start address or an error message
UPDATE-BLK-OFFSET	n1---- ---	update contents of variable block-offset
UPDATE-DATA-BLKS	n1,n2 -----	updata data blocks after removing n2 bytes with block offset n1
UPDATE-IND-BLKS	n1,addr ------	updat index block, n1 is no. of bytes in trajectory and addr is address (pfa) of trajectory name
UPDATE-IND-OFF	n1,n2 -----	update index offset when n1 is no. of bytes in trajectory removed and n2 its line number
UPDATE-INDEX	n1,n2 -------	update the contents of an index on a disc block n2 and at block offset n1
WAIT-KEY	----n	wait for a to input

Table D5 simulator words

(f) trajectory generation words

word	stack manipulation	explanation
DEFINE-POINT	---	defining word to create new points, expect point name to be entered
DEFINE-TRAJ	addr1,addr2 --- $addr_n$ ----	defining word which expect the name of trajectory at run time, asks to enter the name of joints to generate a trajectory
DELIVER	x,y,a.b,c -------	deliver an object to a specified location where x,y,a specify position and b,c orientation of destination
ENTER-POINT-DATA	-------	ask to input data to define a point
GRAB	x,y,a,b,c -----	move the arm to pick up an object at these co-ordinate values, see DELIVER
INPUT-NUM	-------[n],f	ask to input a number, returns its absolute valueand true flag if n0. is correctly entered
INPUT-POINT	------- addr	ask to input point name and returns its pfa
MOVE-POINT	addr --------	expect pfa of a defined point and move the arm to that point
MOVE-TRAJ	addr -----	given the start address of a perviously defined trajectory, move the arm to follow this trajectory
PICK-PLACE	------	pick up an object from a prespecified location and deliver it to a known location
READ-POINT-DATA	addr ---- n6,n5 -----n1	enter point name (pfa) andreturns its co-ordinate values
SELECT-POINTS	----- addr1, addr2---addrn	ask to enter names of previously defined points and leave their pfa's
SPATIAL-MESSAGE	-----	display a message on the vdu giving the order in which co-ordinate values are to be entered by the user
VIA-POINT	x,y,a,b,c ------	allow to specify position (x,y,a) and orientation (b,c) of a via point followed by arm

Table D6 trajectory generation words

(g) **ultrasonic sensor words**

words	**stack manipulation**	**explanation**
CALC-DISTANCE	n -------	calculate distance of an obstacle from delay data (n) saved in data reg. B of PIA
CONFIG-ULT	--------	configure ultrasonic transducer
ENABLE-ULT	------	reset ultrasonic transducer, configure and fire it
FIRE-ULT	------	transmit ultrasonic ray
RESET-ULT	-------	reset ultrasonic transducer

Table D7 ultrasonic transducer words

(h) **block movement words**

words	**stack manipulation**	**explanation**
BLK-COPY	n1,n2 -----	copy the contents of block n1 on to block n2
BLKS-COPY	n1,n2,n3 ------	copy n3 blocks starting at block n1 to blocks starting at block n2
BLOCK-LOAD	n-------- addr	copy contents of a block n in to a buffer whose address is addr
BLOCK-SAVE	addr, n -----	save the contents of a buffer to block n
CLR-BLK	n ------	initialise block n
CLR-BLKS	n1,n2 -------	initialise n2 blocks starting at block n1

Table D8 block movement words

HIERARCHICAL DIAGRAMS FOR FORTH WORDS :

(i) additional mathematical operations words

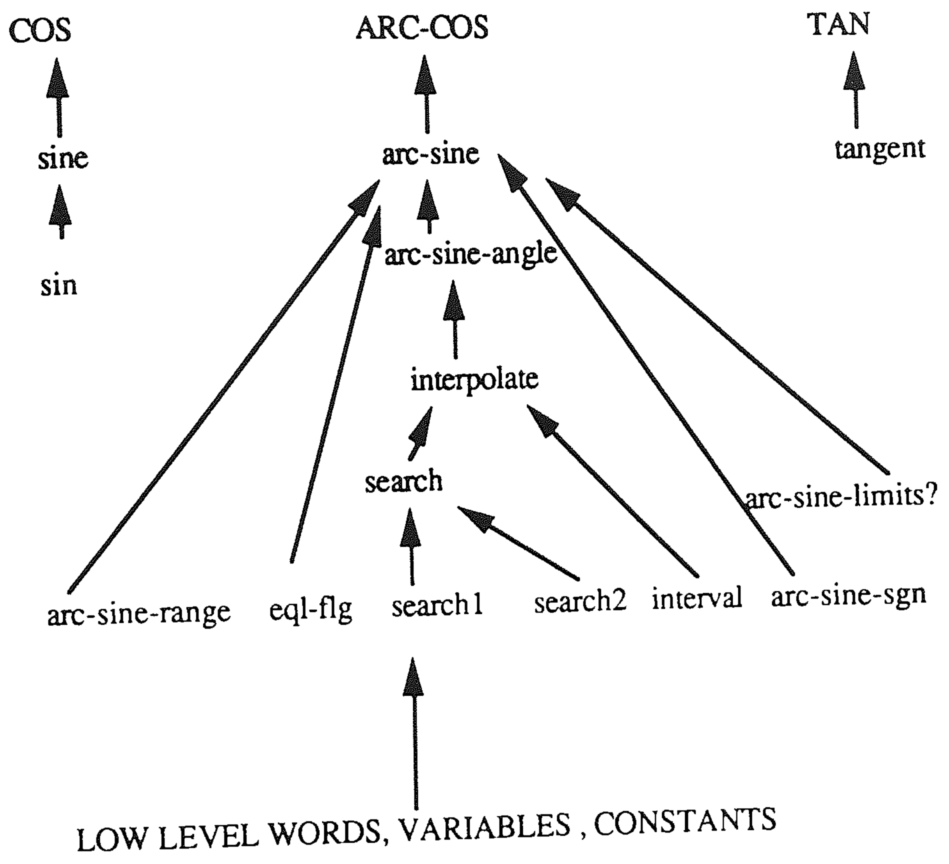

Figure D1 hierarchical relationship between additonal mathematical operations words

(ii) Input/Output (I/O) words

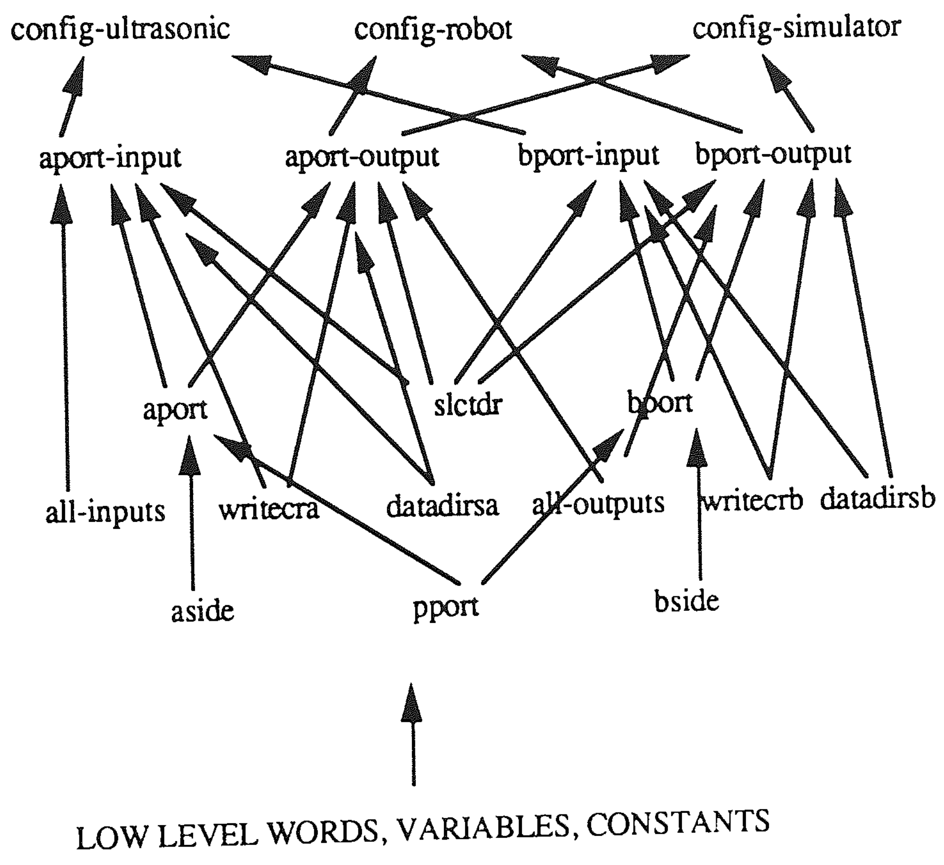

Figure D2 hierarchical relationship between I/O words

(iii) simulator words

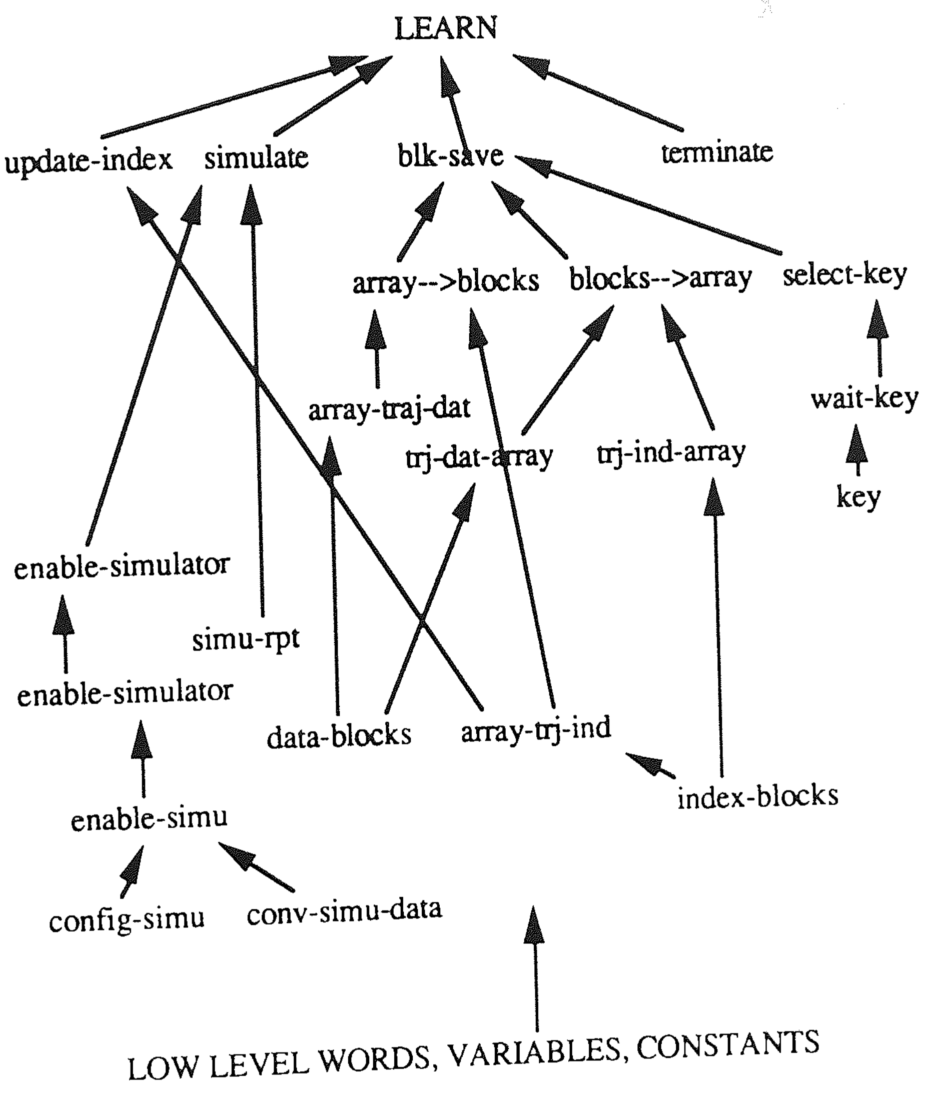

Figure D3a hierarchical relationship between simulator words

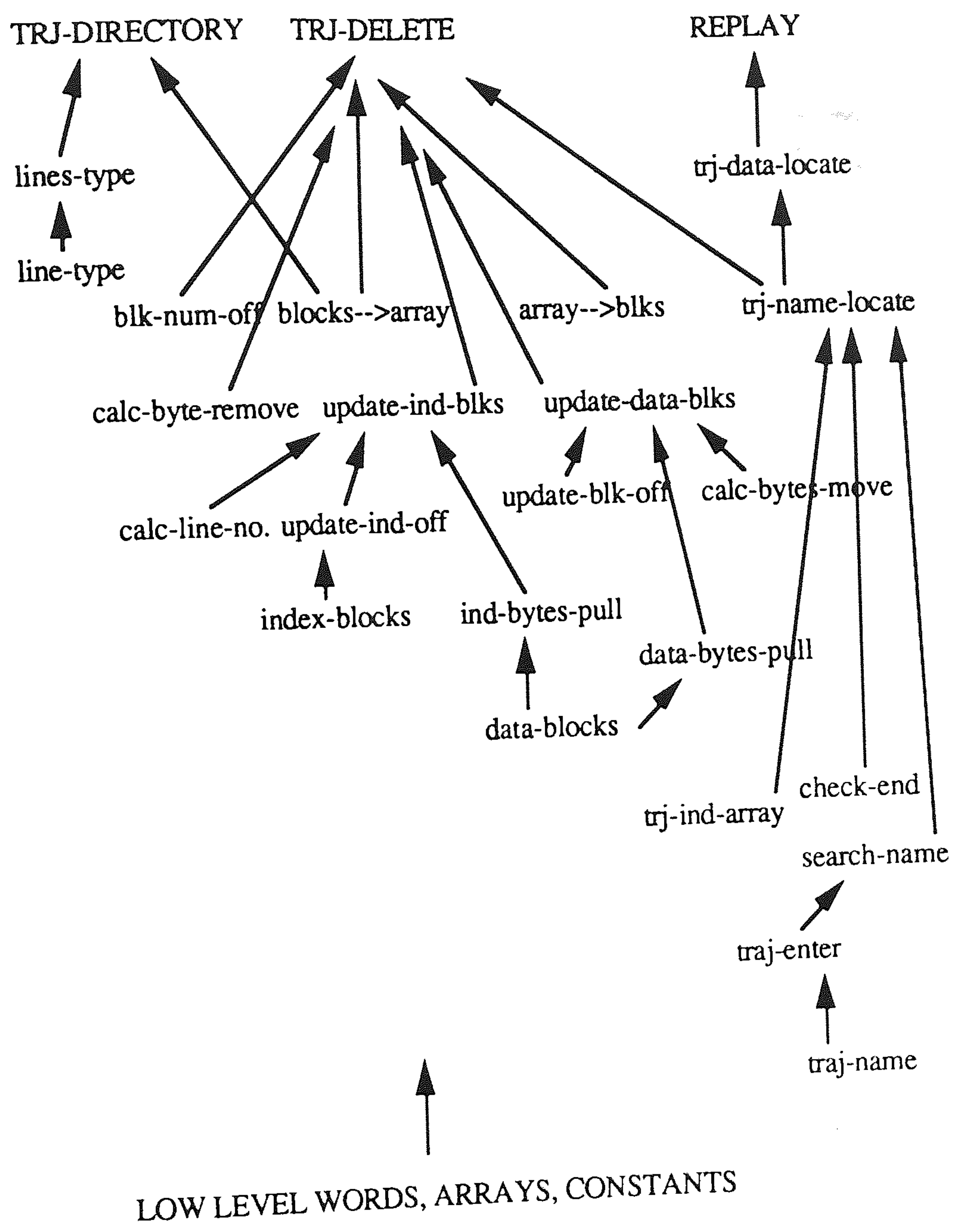

Figure D3b hierarchical relationship between simulator words

(iv) **trajectory generation words**

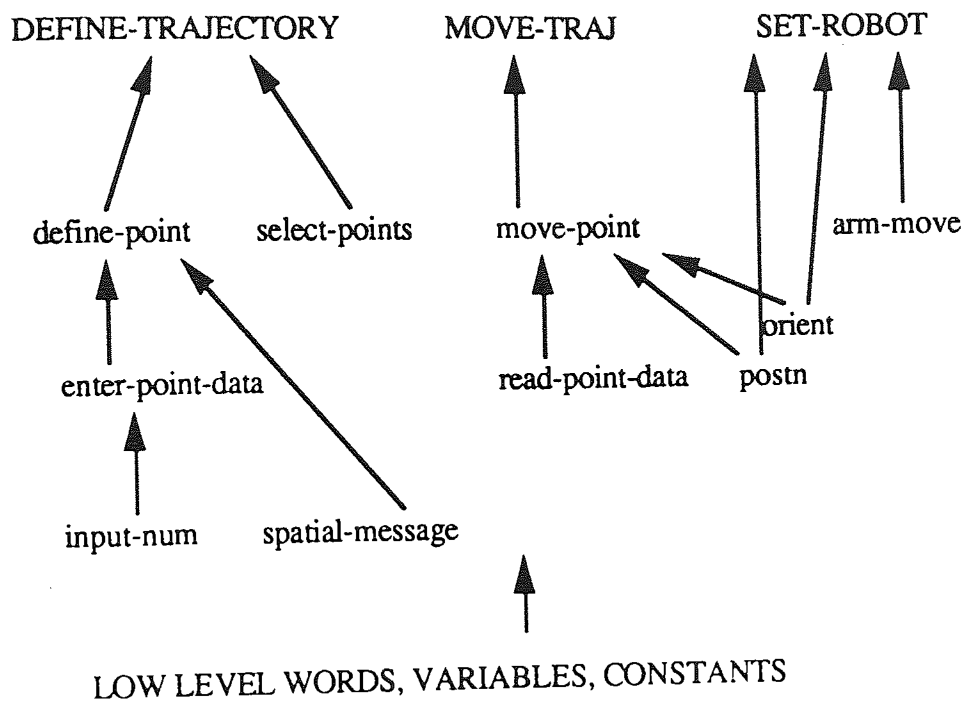

Figure D4 hierarchical relationship between trajectory generation words

(v) ultrasonic transducer words

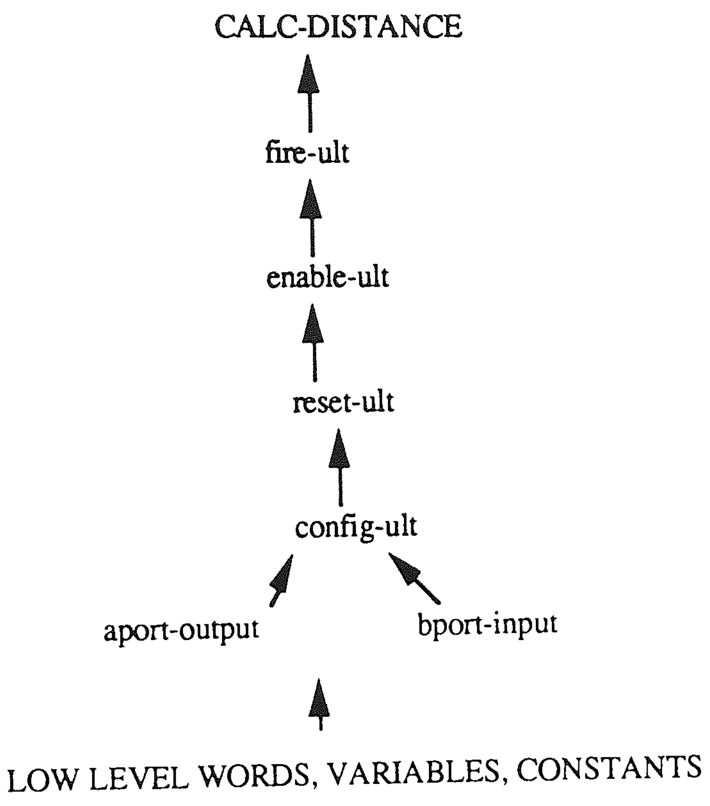

Figure D5 hierarchical relationship between ultrasonic transducer words

(vi) block movement words

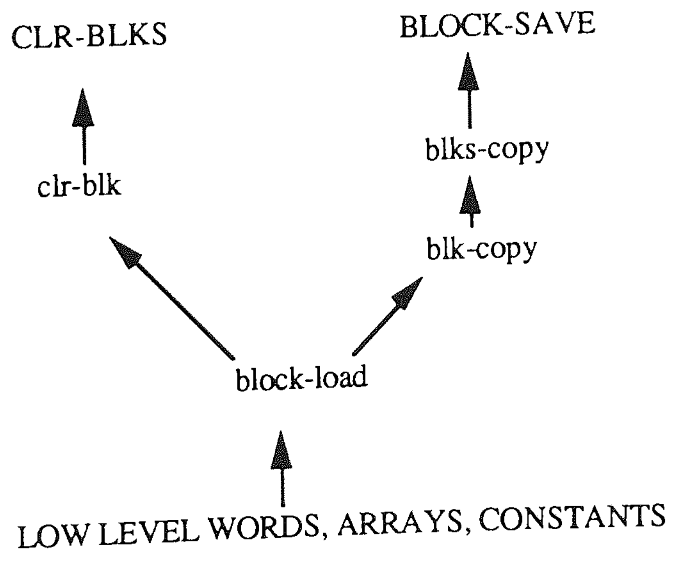

Figure D6 hierarchical relationship between block movement words

APPENDIX E -- TIMING COMPARISON OF FORTH SOFTWARE

In order to align the robot gripper at a specified position and orientation, the solution to the inverse kinematic problem is necessary. This solution involves a considerable number of trigonometric calculations as described in section 7.4.2 in order to compute the data value which is to be sent to each joint of the robot. These calculations are complex and take considerable time. It was decided to establish whether the version of Forth implemented on Motorola 6809 and used in this research project was fast enough to perform these calculations. A comparison between the time taken by Forth to perform these calculations and the time taken by the robot motors to move each joint to the target position was undertaken.

The speed of each motor was determined by recording the time taken by each motor to to rotate it through a known angle. The time taken by Forth words to perform the necessary computations in order to calculate the data value required by each joint to achieve the target position and orientation of the gripper was determined by executing relevant Forth words placed in a definite loop. The results obtained are summarised in table E1 below:

Joint	Time to move a joint one degree (ms/deg)	Computation time for joint movement	
		Time (ms)	equivalent joint angle (deg)
Waist	10	2	0.2
Shoulder	20	2.5	0.125
Elbow	12.5	2.3	0.184
Wrist	10	1.8	0.18
Hand	6	1.7	0.283

Table E1 showing timing comparison.

These results show that although complex trigonometrical calculations take considerable computer time, they are still on the average only equivalent to a quarter degree of joint movement. However, if a fast industrial robot is employed, which when fully extended can move between 1-3 m/s, then the speed of computation will become more relevant, hence much faster computation will be necessary. This may require implementing Forth on a faster processor, or using a maths co-processor which can perform these trigonometrical calculations much faster.