Implementation of a Robot Control Development Environment

John Lloyd

December 1985

Computer Vision and Robotics Laboratory
Department of Electrical Engineering
McGill University

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

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

Abstract

This thesis describes work that was performed in implementing a real-time robot control research environment at the McGill University Computer Vision and Robotics laboratory (CVaRL), based on previous work which was done at Purdue University in 1983. The environment consists of two layers; the lowest layer, called RCI (Robot Control Interface), is a software facility that provides a programmer with an easy way to create real-time robot control procedures in the language C under the operating system UNIX. Built on top of this is the upper layer, called RCCL (Robot Control C Library), which is a collection of C primitives and data structures that provides robot motion control in Cartesian coordinates. Work done in implementing this system consisted of: (1) rewriting the low level (RCI) to provide error handling, a simulator, a compilation interface, increased access to the robot devices, and removal of some system resource restrictions; (2) proposing a methodology for porting this RCI environment to a multiprocessor, multi-robot configuration, (3) deriving the kinematic and inverse kinematic models for both CVaRL robots as required for the RCCL trajectory generator, (4) developing a method to accurately measure the gravity loading and frictional parameters for one of the CVaRL robots as required for static force control, (5) installing the RCCL software and determining the modifications necessary to make it usable in a multi-robot system.

Résumé

La présente thèse décrit le travail accompli pour mettre en œuvre un environnement de recherche en commande temp-réel de robot manipulateurs dans le cadre du Laboratoire de Visionique et de Robotique (Computer Vision and Robotics Laboratory - CVaRL) au Département de Génie Electrique de McGill University Cet environnement est basé sur un travail effectué à Purdue University en 1983 Il s agit d un outil logiciel à deux couches Le niveau inferieur nommé RCI (Robot Control Interface), procure au programmeur la possibilité de créer des procédures temps-réel de commande de robot, en language 'C' et sous le systeme d'exploitation Unix. Sur ce niveau inférieur est bâtie la couche supérieure nommée RCCL (Robot Control C Library) C'est une collection de fonctions primitives et de structures de donnée écrites en "C" qui permettent la commande du robot au niveau effecteur dans l'espace cartésien Les différentes étapes pour la mise en œuvre de ce système furent (1) la reécriture de la couche de bas niveau pour permettre la gestion de erreurs pour fournir un simulateur hors-ligne et un interface de compilation, pour offrir un meilleur accès aux dispositifs robotiques et supprimer certaines restrictions quant aux ressources du système (2) l'établissement d'une méthodologie pour étendre le système RCI au cas multi-processeur, multi-robot, (3) le calcul des modèles géométriques et cinématiques directs et inverses des deux robots du laboratoire ceci pour leur intégration dans le système RCCL (4) le développement d'une méthode de mesure des charges de gravité et des coefficients de friction dont la connaissance est nécessaire à la commande en force, (5) l'installation du logiciel RCCL et la détermination des modifications propres à en faire un système multi-robot

Acknowledgements

I would like to express my thanks and appreciation to Dr. Martin Levine for his support encouragement and advice while I was undertaking the work described here. I would also like to credit the numerous other people who contributed to bringing up the RCCL/RCI system at CVaRL. Mike Parker for his programming support. Woon Thong for his hardware support. Frank Ferrie for convincing us we could do things we thought we couldn't. Roy Kinsella for his many suggestions. Paul Freedman and Gregory Caryannis who built the VAX-robot interface. Rick McConney and David Gauthier who did some work with the joint microprocessors and Don Kossman for his encouragement as he worked on installing RCCL on the Microbo robot. Thanks is also expressed to Vincent Hayward and Frank Ferrie for their comments and many discussions.

Some of the mathematical derivations presented in Chapter 3 were derived with the assistance of MACSYMA, a symbolic manipulation program developed at the MIT Laboratory for Computer Science, and now maintained and marketed by Symbolics, Inc., Cambridge Mass. MACSYMA is a trademark of Symbolics.

Financial assistance for this work was provided by the Natural Sciences and Research Council of Canada and the Department of Education of the Province of Quebec.

Contents

List of Figures

List of Tables

Chapter 1 INTRODUCTION

Investigators interested in robotics research may choose from a wide and multidisciplinary range of topics with the development of robot systems incorporating elements of material science mechanics electronics, control theory, computer architecture communications, software engineering, artificial intelligence, and systems design Such inherent complexity makes strong demands on any environment in which useful research is to be performed Not only is a certain minimum amount of equipment and hardware required (such as a computer, the robot itself, sensory peripherals, and various workspace objects) but tools must be provided which allow the various parts of the system to be interconnected and controlled in a variety of ways. The set of these tools which are manifested in software can be called the *robot programming environment*.

Commercially available robots usually come equipped with a programming environment supplied by the manufacturer, generally in the form of a robot programming language Good examples of these are Unimation's VAL II [Shimano et al 84] and Automatix's RAIL [Automatix 82]. Unfortunately, these languages tend to be geared toward commercial applications and may not alway possess enough generality or flexibility to satisfy the needs of a research group For instance, if research is being done on algorithms for generating robot trajectories, an existing language will not be of much use, unless a "hook" is provided by which a programmer may replace the default trajectory generator of the language with one of his/her own. In response to this, research institutions often develop their own programming environments, tailored to their own needs, as exemplified by Stanford

University's AL [Finkel et al. 75, Goldman 82] which was designed to offer all the features of a conventional programming language in addition to manipulator control constructs. A similar degree of programmability is found in IBM's AML [Taylor et al. 82]. These systems however still do not make it convenient to experiment easily with different levels of the implementation.

Another approach was introduced at Purdue University in 1983 [Hayward and Paul 83]. This involved creating a two level robot software environment embedded in the high level language C. The lower level was called the Real-Time Control (RTC) layer and provided the programmer with primitives by which he/she could write simple control procedures to operate the individual joints of a robot. It served as a substrate for the higher level known as the Robot Control C Library (RCCL) which was a package of routines for specifying the trajectory of a robot in Cartesian coordinates. This layering structure is reminiscent of two well known examples: the ISO protocol for open system communications [Zimmermann 80], and the UNIX* operating system [Kernighan and Pike 84]. In fact the system developed at Purdue University was implemented under a UNIX system running on a VAX minicomputer. At McGill University the RTC layer has been redesigned by the author and is known as RCI (Robot Control Interface); it also serves as a host for a version of RCCL. A somewhat stylized view of the arrangement is shown in Figure 1.1.

1.1 Description of the Work

This thesis is based on the work that was done in implementing both levels of the Purdue environment at the McGill University Computer Vision and Robotics Laboratory (CVaRL). This involved adapting it to the needs of the research facility, and laying the groundwork for future extensions to the system. The work itself ranged from deriving manipulator kinematics to creating extensions to the UNIX operating system. At the time of implementation, the laboratory owned two robots, a Unimation PUMA 260, and a Microbo MR-03 Ecureuil*. The host computer system was similar to the Purdue system consisting of a Digital VAX 11-750 running UNIX. Most of the work described here was done with the PUMA 260, because of time constraints and initial difficulties interfacing

* UNIX is a trademark of Bell Labs.

* The laboratory has since acquired a third robot, an IBM 7565.

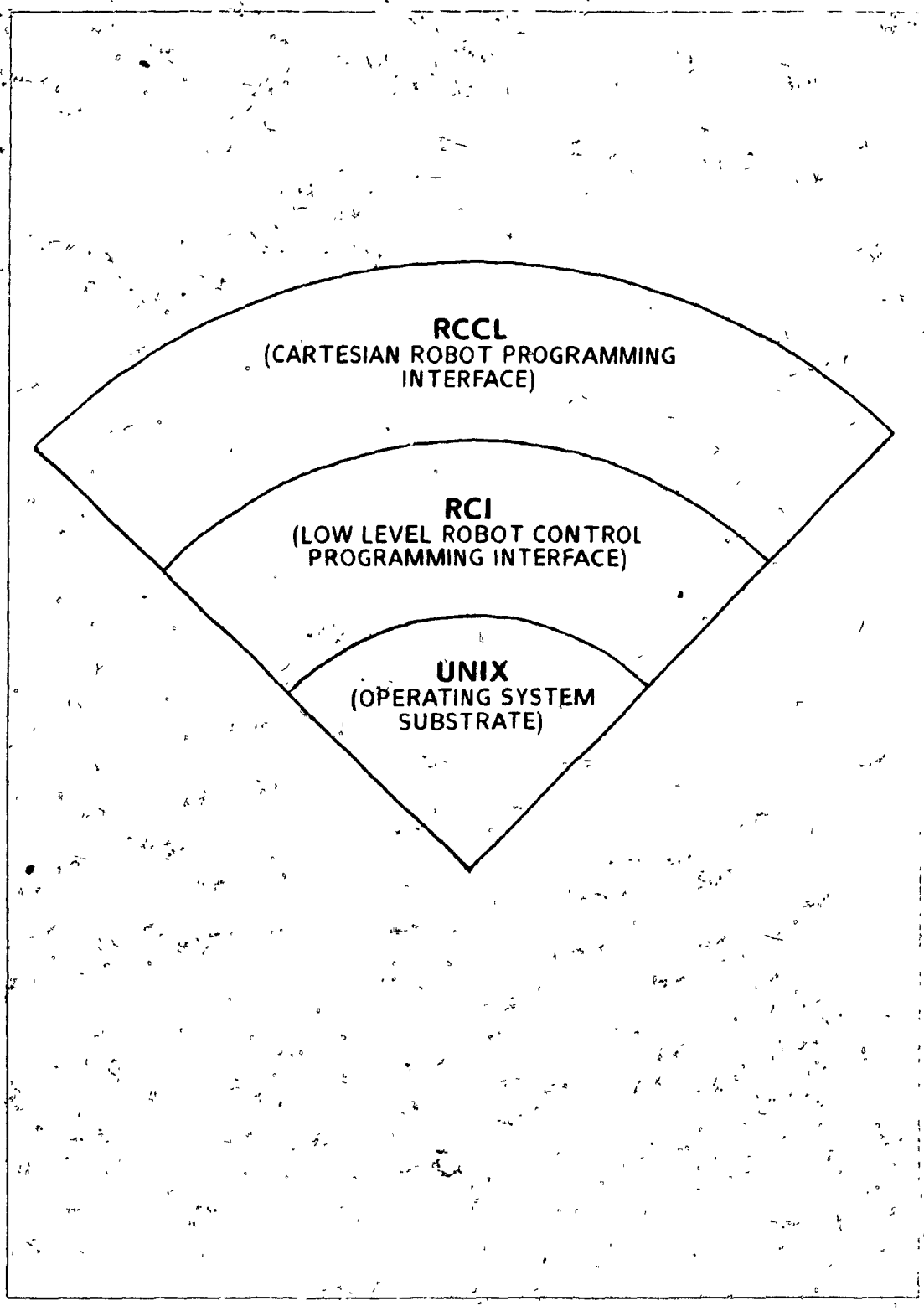


Figure 1.1 Representation of the layering of the RCCL/RCI system implementation

with the Microbo

The first job involved taking the Purdue low level control layer (RTC) and rewriting it in a more general and usable form. To improve its usability, some primitives were added to the UNIX kernel which allowed it to properly handle the real-time software. The new version was renamed RCI (Robot Control Interface). Once operational this tool was used in several different applications, including developing programs to measure the dynamic parameters associated with the robot (Chapter 4 and [Aboussouan 85]) and implementing the Purdue high level control layer (RCCL). A present limitation of RCI is the computational load that it places on a single processor. This led to the consideration of ways to employ multiple processors, while retaining the characteristics of the RCI programming environment which had proven to be useful. A corollary of this goal is to expand the interface to multiple robots. Work is presently under way at CVaRL to extend RCI to the Microbo [Kossman 85].

Bringing up the RCCL software required some additional tasks. Since RCCL works in Cartesian coordinates its internal software must know the kinematics for each manipulator it controls. The kinematic models were obtained for both the PUMA and the Microbo*. The force control features of RCCL also required the measuring of certain robot specific parameters. In the absence of direct force sensors the force control was accomplished (both at CVaRL and at Purdue) by measuring and controlling joint currents. This is possible because of the linear relationship between torque and motor currents in DC servo motors. A technique was developed to use this current – torque relationship to measure the necessary force control parameters (*e.g.* joint friction and gravity loading coefficients). The measurement programs were particularly easy to implement using the RCI software and yielded good results. Measurements were made for the PUMA only, since the mechanical gearing on the Microbo's prismatic joints make force control and measurement using joint torques impractical.

The last task was the actual installation of the RCCL system. After this had been done, and RCCL had been subjected to a certain amount of usage, it became obvious that some improvements would be necessary to augment the usefulness of the system, particularly in any forthcoming multi-robot version. Research is now being done at the laboratory using

* As stated earlier RCCL has to date only been installed on the PUMA.

the RCCL system primitives as building blocks for higher layers of robot control [Gauthier et al 85]

1.2 Outline of the Presentation

The general objectives of this thesis are to

1. Provide a description of the RCI/RCCL programming environment as currently implemented
2. Consider ways in which this environment may be improved and extended.
3. Present the robot kinematic and static force models and describe how they were obtained

The thesis is divided into four main chapters

Chapter 2 discusses the RCI programming interface beginning with a description of the present CVaRL implementation, and a summary of the differences between RCI and the original Purdue RTC system A presentation is then made of the issues involved in generalizing this system to handle multiple robots on a multiprocessor configuration

Chapter 3 presents the derivation of the kinematics and differential kinematics for the PUMA 260 and the Microbo Ecureuil Although the manipulator Jacobian has traditionally been regarded as a rather complex object, it was possible to reduce it to a form that was computationally quite efficient

Chapter 4 defines the static force control of a manipulator, discusses the robot parameters that must be determined in order to perform this control, and introduces a simple method to measure these parameters The results are presented for the PUMA 260

Chapter 5 provides a brief description of the RCCL system itself and suggests extensions necessary to make it useful in a multi-robot environment

The remainder of this chapter contains background material on robot hierarchical control systems, the CVaRL laboratory environment, notation conventions, and some mathematics

1.3 General Background

1.3.1 Hierarchical Robot Control Systems

Hierarchical systems are a popular means of partitioning the robot control problem and there are many discussions of this topic in the literature ([Albus et al 81 Shin and Malin 85 Stephanou and Saridis 76]) We are interested in them here because the RCI/RCCL system is also hierarchical in concept

The general premise of such systems is illustrated in Figure 1.2 Tasks executing at different levels communicate with their respective supervisor and subordinate tasks The nodes at the bottom represent the system actuators and sensory input devices Sensory data is gathered at the lowest level, and then propagates up through the structure being successively interpreted at different levels to yield progressively higher level information Commands originate at the top of the hierarchy, in terms of a high level request and propagate downward resulting in successively lower level requests The sensory data available at each level may be used to execute the commands indigenous to that level

This description is extremely general and makes very few assumptions about the particular system under consideration In terms of computation, the lower levels are usually assumed to perform simpler, more rapid computations while the higher levels perform more complex, slower computations [Albus, et al 81] This principle is rather vividly exemplified by research which is presently being done on developing walking robots [Schwan et al 85] In that particular example, control is effected by a hierarchy of tasks running at different rates, including a human interface (2 Hz), a body motion planner (20 Hz), a leg motion planner (50 Hz), and a leg servo (100 Hz) A rather rough way of distinguishing the lower levels from the higher levels is to say that the lower levels tend to process numerical information, while the higher levels are more symbolically oriented [Harmon 83].

To a certain extent, the hierarchical paradigm is already standard practice in robotic systems, where each robot joint or axis is often controlled by an individual task running on a dedicated processor under the command of some supervisor task, as is the case with the vendor supplied control systems for both robots at CVaRL (Figure 1.3) The individual processors controlling the joints define the *joint level* of the hierarchy The decomposition of the control at this level comes naturally, since once a set of destination joint angles

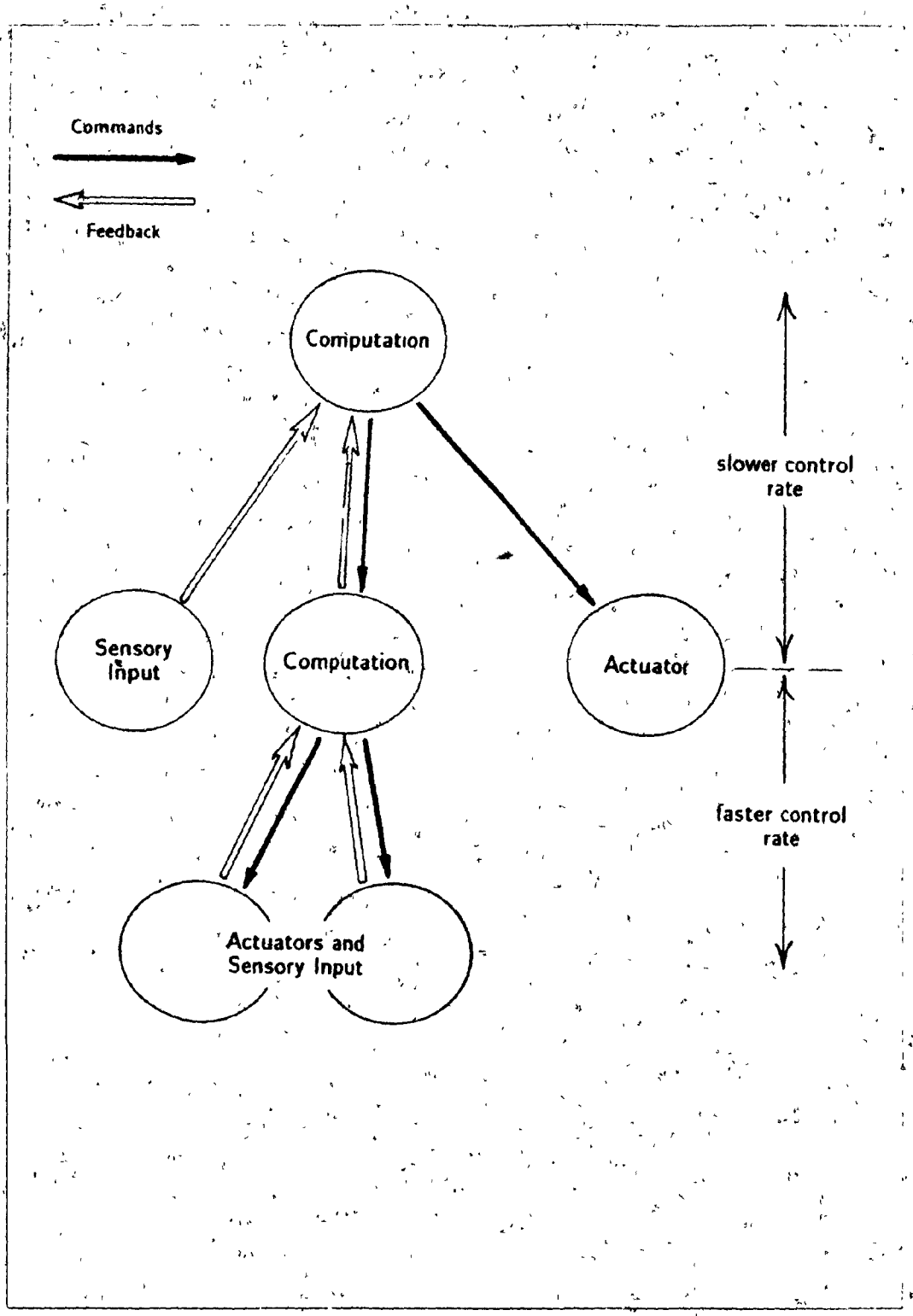


Figure 1.2. A typical hierarchically structured control system.

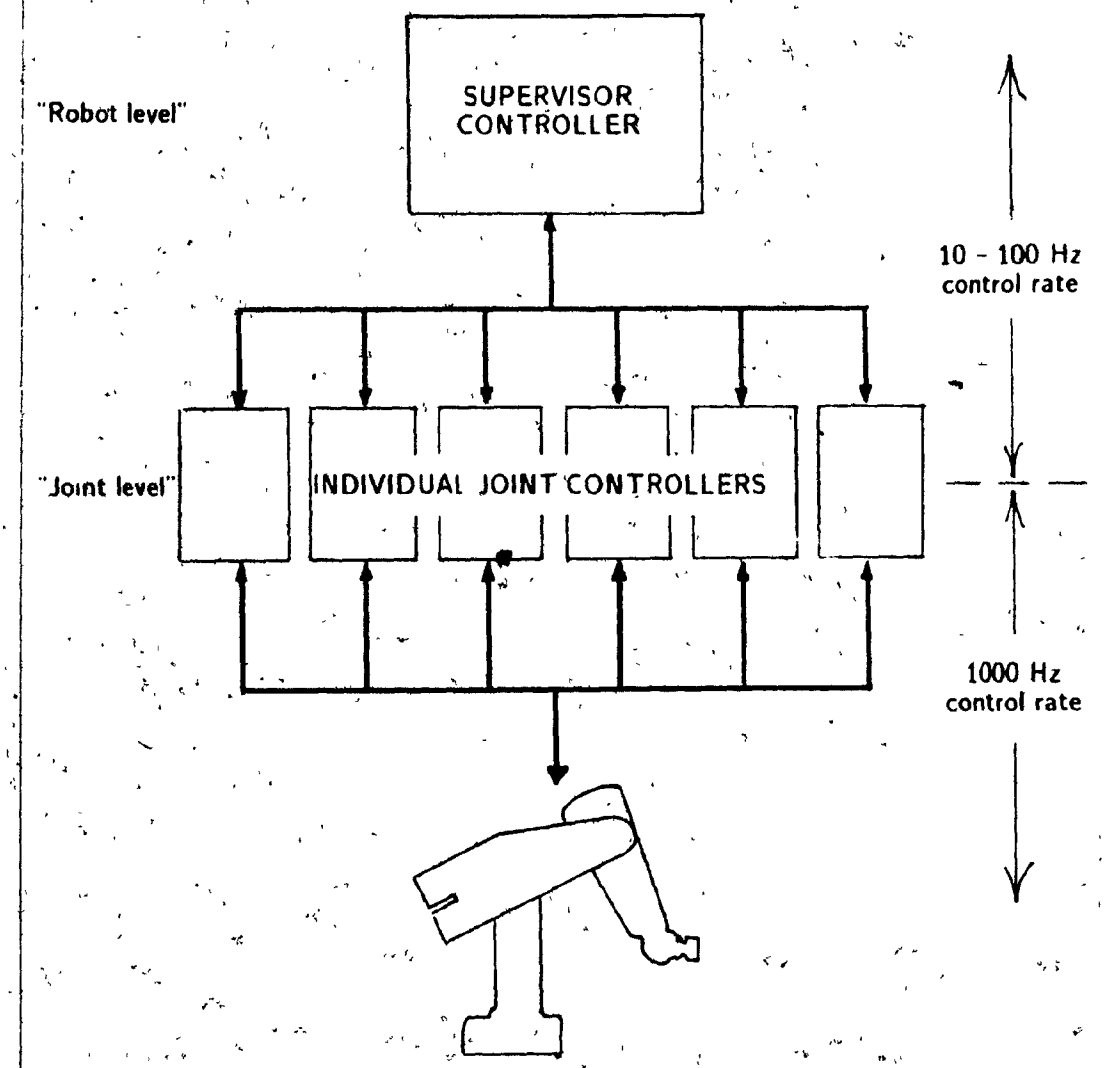


Figure 1.3 A basic robot control mechanism

has been selected, each joint can move to the appropriate location with a minimum of interaction with the other joints* The joint controllers accept very low level commands directing them to move to and hold specific positions Positional information, provided by such devices as optical encoders, is returned from the robot to complete a closed-loop control circuit in each joint controller, and can also be passed up to the higher control levels, which may need to know the position of the robot for carrying out computations such as collision avoidance The joint control level typically operates at very high rates (*e g*, 1000 Hz)

The next level up in the hierarchy (the "supervisor" in Figure 1 3) is often called the *robot level*; it corresponds to the *control* level of the RCI system presented in Chapter 2 In this layer the manipulator is treated as an integrated whole [Shin and Malin 85], it is here that a trajectory specification is mapped into the required sets of joint positions In the case of a single manipulator in an unrestricted workspace, well defined methods exist for planning and executing motion trajectories ([Paul 81, Lee 82]) However, the situation becomes far more complex when it is desired to take into account, at run time, the existence of obstacles in the workspace, or perform a task in close cooperation with another manipulator, or integrate some form of higher level feedback (*e g*., visual information) into the trajectory control process These are open problems. Also, work is still being done with single manipulators to improve trajectory computation methods and investigate ways of performing force control A typical computation performed at the robot level is the so called *inverse kinematic function*, which maps Cartesian coordinates into the joint space of the robot The computation rates tend to lie in the range of 10 to 100 Hz

At levels farther up, the computations performed, and the objects these computations are performed on, become increasingly complex and diverse. The control rate is in general lower and less regular At these levels, the robot ··otion is frequently considered as a set of destination positions in some "world" coordinate system, which it is the responsibility of the lower levels to position the robot at, in accordance with a set of path control parameters (as exemplified by RCCL, described in Chapter 5) Models of the workspace

---

* This assumes that the joint control does not model the arm dynamics Some recent research has been directed at designing special joint level architectures to enable run time computation of the robot dynamics (a highly coupled and computationally expensive problem) and yield a corresponding improvement in robot performance principally at high speeds [Nigam and Lee 85])

can be introduced here alongside computer vision functions which return the locations of objects in the workspace One may also consider at this level (and possibly higher ones) each robot or group of robots as a separate workcell and devise paradigms for the integration of such workcells into a cohesive automated plant ([Faro and Messina 83, Baird et al 84]) These higher levels fit into what the RCI system addresses as the *planning* level

1.3.2 Description of the Laboratory Environment

At the time of this writing the equipment at CVaRL consists of two Digital VAX 750's, running UNIX 4 2bsd, and one VAX 780 running VMS connected together by an Ethernet Link (Figure 1 4) The two 750 s are dedicated chiefly to robotics projects One is directed towards robot vision research and to this machine are attached cameras frame grabbing equipment, and a display monitor The other 750 is used chiefly for robot control applications, and is connected to the two robots A session layer communication protocol has been developed that allows these two VAXs to communicate over the Ethernet for applications involving both vision and robotics [Gauthier, et al 85] The robots are connected to their VAX both with serial lines and high speed parallel links

Research objectives in the lab have focussed on using the robots for inspection and repair applications in the electronics industry Both robots are small, and they are arranged in an overlapping common workspace

1.3.3 Description of the Robots

Each robot is a six link serial manipulator All joints on the PUMA are revolute, arranged in an *anthropomorphic* configuration (Figure 1 5), it is essentially a scaled down version of the larger (and more discussed) PUMA 560/600 robots The actuators are permanent magnet DC servo motors The Microbo is a *cylindrical* high precision manipulator of about the same size, with joints 2 and 3 prismatic (Figure 1 6), and is also driven by DC servo motors

The basic control mechanism for each robot consists of a microcontroller for each joint, under the command of a supervisor unit (Figure 1.3) The controllers themselves are microprocessors which use optical encoder position feedback with a PID control algorithm

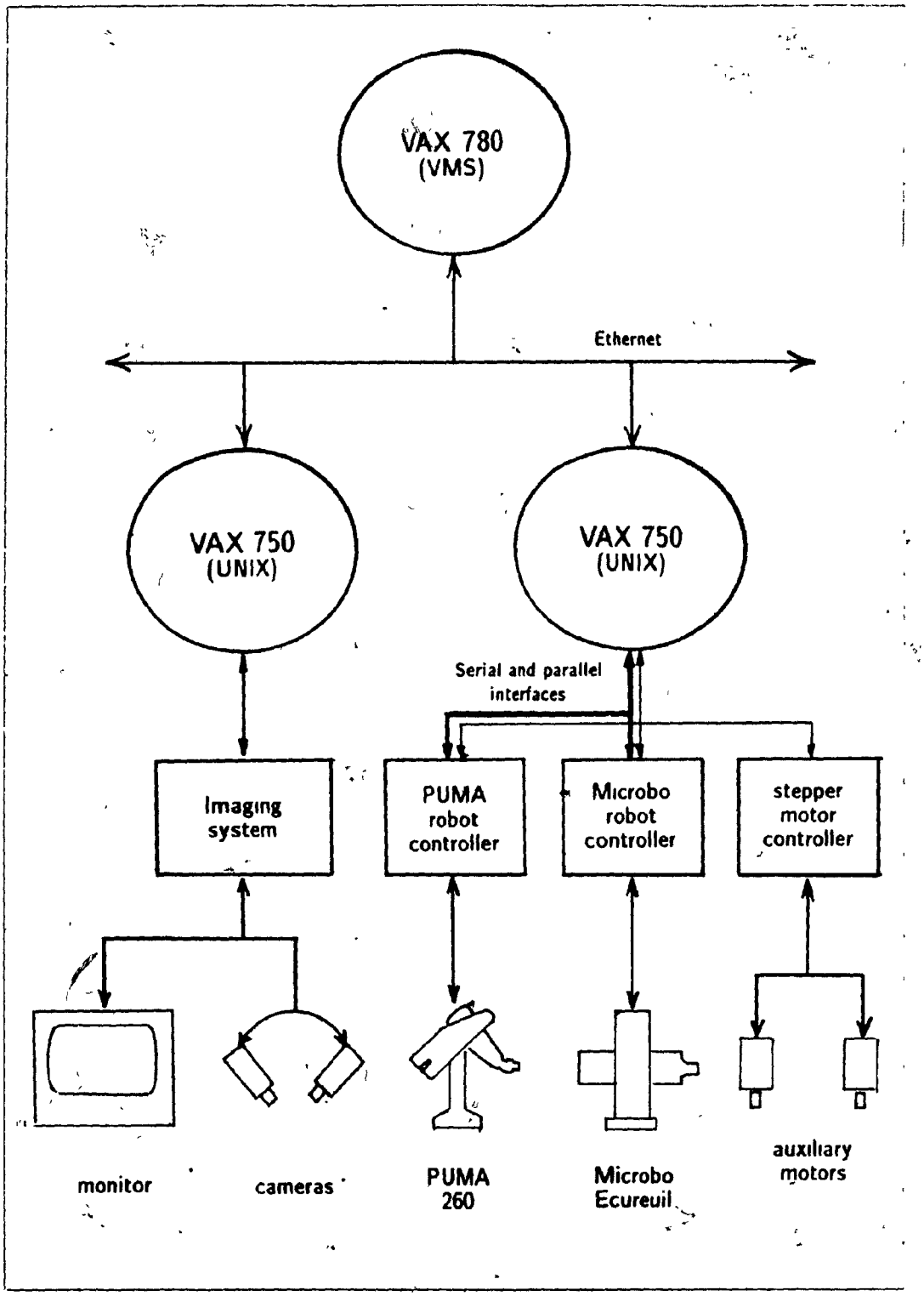


Figure 1.4 The CVaRL laboratory environment

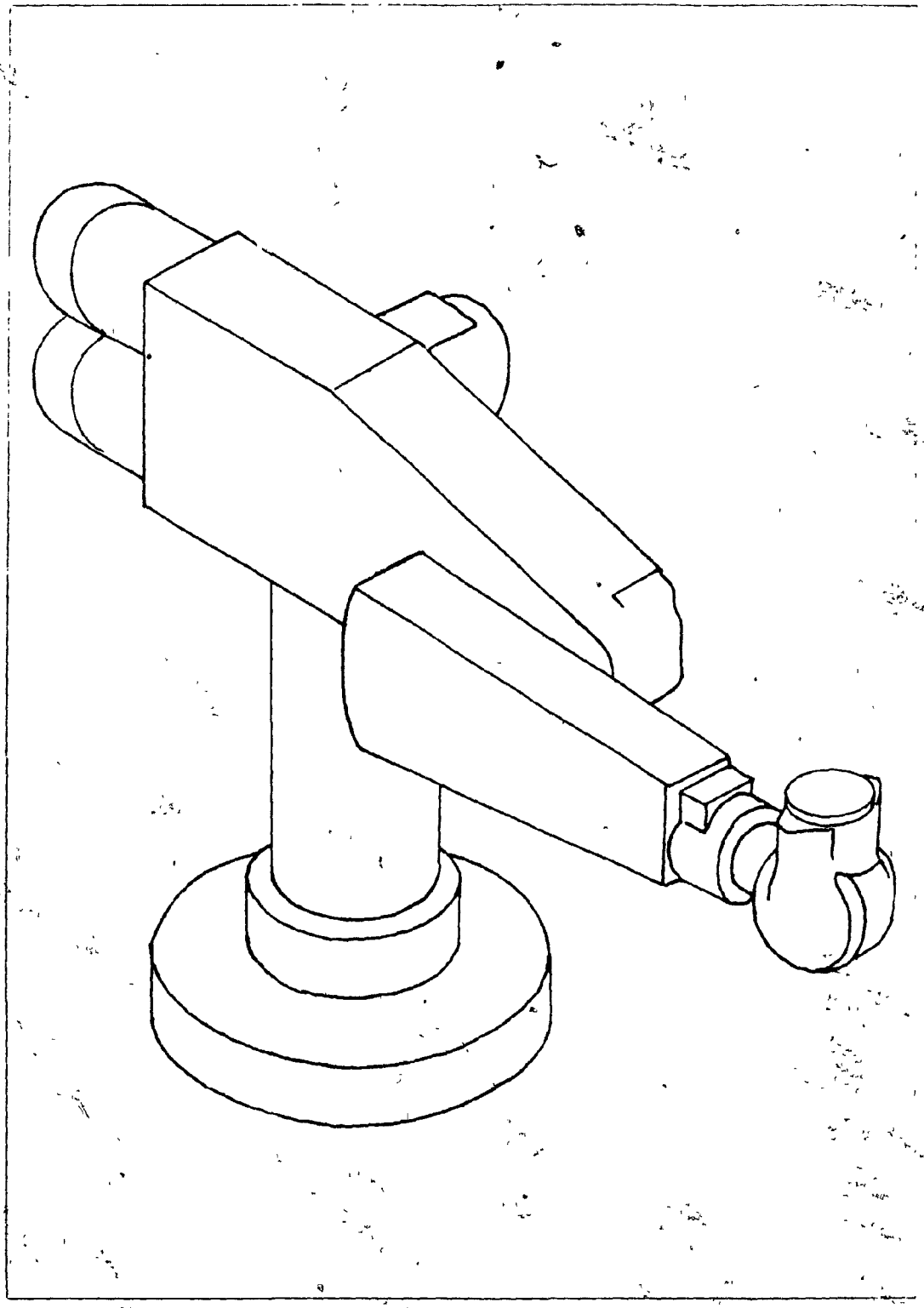


Figure 1.5 The PUMA 260 robot

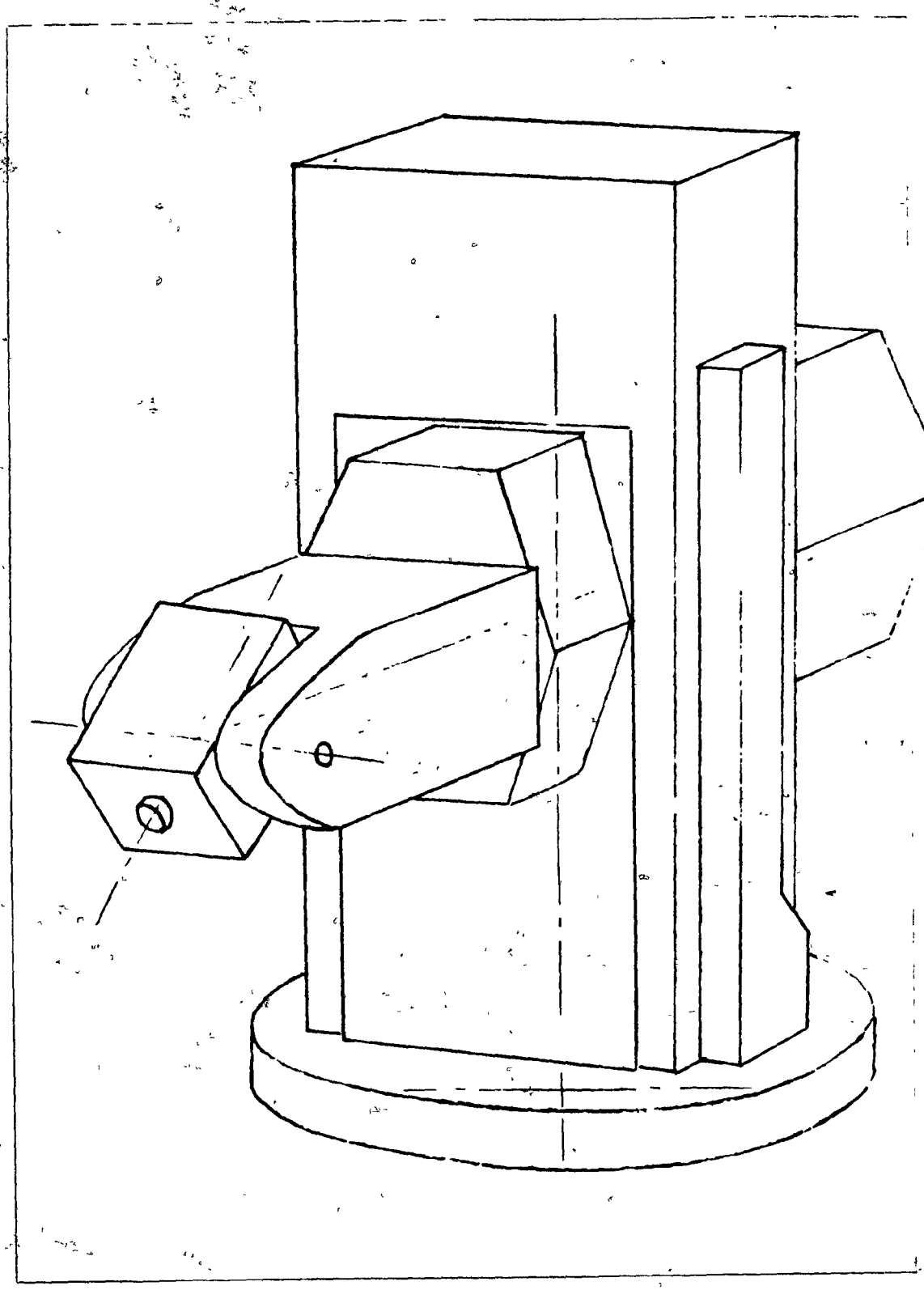


Figure 1.6 The Microbo Ecureuil robot

to determine the current to send to the joint motors. Position control from the supervisor level is hence essentially open loop. The joint angles that correspond to a given Cartesian position are calculated by the supervisor and dispatched to the joint controllers, at a rate in the vicinity of 10 to 100 Hz*, in the case of the PUMA the default rate is 36 Hz. The joint controller PID loops operate at rates close to 1000 Hz.

1.3.4 Notation

Chapters 2 and 5 make numerous references to programming constructs. The actual names of functions, variables, and data types are written in *typewriter* font. In addition, function names are followed by a pair of parentheses, as in `function1()`.

Some use will be made of mathematical vector and matrix notation, for which we shall use the following conventions. Vectors are represented by a bold face, lower case letter, as in $\mathbf{v}$ and $\mathbf{x}$. The only exception to this is the joint variable vector Θ. By default, a vector implies a column vector; a row vector is indicated as the transpose of a column vector, as in $\mathbf{x}^T$. Matrices are indicated by a boldface upper-case letter, as in $\mathbf{A}$ and $\mathbf{X}$.

It is sometimes useful to indicate the coordinate frame with respect to which a vector or matrix is defined. The notation used for this consists of posting the coordinate frame as a preceding superscript. For example, the vector $\mathbf{v}$ defined with respect to coordinate frame A is indicated as ${}^A\mathbf{v}$.

The set of joint variables for a robot are indicated by Θ, the elements of which may represent either distance values for prismatic joints or angular values for rotary ones. Similarly, the vector $\mathbf{t}_j$ is used to describe the joint level forces, and its elements may be either linear forces or torques.

---

* In the Microbo there is no Cartesian control mode supplied by the manufacturer. Because of this, the default supervisor to joint communication is slow and asynchronous. Getting around this problem has been one of the problems involved in providing an RCI/RCCL control interface to the Microbo.

1.4 Mathematical Background

1.4.1 Representation of Coordinate Frames and Positions

The position of a rigid body in space may be described by associating it with a fixed orthonormal coordinate frame. The position of this frame, taken with respect to some other reference frame, then defines the position of the body.

In general, if we consider two frames, O and A, then the location of A with respect to O may be described in terms of a translation vector $\mathbf{p}$ and a 3×3 rotation matrix $\mathbf{R}$ which are applied to the base unit vectors of O. We define the elements of $\mathbf{p}$ and $\mathbf{R}$ as

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{pmatrix} \tag{1.1}$$

Now consider that a point in space may be represented by either the vector ${}^O\mathbf{r}$ or ${}^A\mathbf{r}$, depending on whether it is observed in frame O or frame A. Using the relationships between the two frames described above, we may form the equation

$${}^O\mathbf{r} = \mathbf{R}\ {}^A\mathbf{r} + \mathbf{p} \tag{1.2}$$

If, instead of representing a point in space by the usual three vector $(r_x, r_y, r_z)^T$ we represent it by an extended four vector $(r_x, r_y, r_z, 1)^T$ then equation (1.2) may be rewritten as

$${}^O\mathbf{r} = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} {}^A\mathbf{r} \tag{1.3}$$

This matrix is known as a *homogeneous* transform, and has achieved great popularity in both robotics and computer graphics as a means of describing the relationship between orthogonal coordinate frames. The elements of the matrix in (1.3) are often represented by the four column vectors **n**, **o**, **a**, and **p**.

Numerous other methods exist to describe the rotation characterized by $\mathbf{R}$. These include yaw, pitch, and roll angles, Euler angles [Paul 81], and quaternions [Shoemake 85]. Yaw, pitch, and roll angles are defined as an ordered sequence of rotations about the x, y and z axes, respectively, of the original coordinate frame. These angles are indicated here

by the following Greek letters yaw (ψ) pitch (θ), and roll (ϕ). Well defined mappings exist between the various representations of orientation [Paul 81 Shoemake 85] It will be noted that the n o and a vectors contain nine numbers which is a redundant characterization given that three dimensional rotations involve only three degrees of freedom

Compound transformations may be described by multiplying the appropriate matrices together For instance if **A** describes the transformation from frame O to frame A and **B** describes the transformation from frame A to frame B, then the transformation from frame O to frame B may be computed as **A B**

It is most usual to describe a transformation **A** from frame O to frame A with respect to the frame O i.e. $\mathbf{A} = {}^{O}\mathbf{A}$ However, it should be noted that it is possible to represent a transformation between two frames with respect to a third coordinate frame X using the similarity transform

$$ {}^{X}\mathbf{A} = {}^{O}\mathbf{X}^{-1}\,{}^{O}\mathbf{A}\,{}^{O}\mathbf{X} \qquad (1.4) $$

where ${}^{O}\mathbf{X}$ describes the transformation from frame O to the observation frame X We can rearrange (1.4) as

$$ {}^{O}\mathbf{X}\,{}^{X}\mathbf{A} = {}^{O}\mathbf{A}\,{}^{O}\mathbf{X} \qquad (1.5) $$

From this may be deduced the following rule for transformation matrix multiplication If **X** describes a transformation from O to X, with respect to O, then postmultiplying **X** by a matrix **A** describes a transformation on X with respect to frame X, while premultiplying **X** by **A** describes a transformation with respect to the original frame O

Describing locations using transformations is quite useful, since there is no longer any need to use a fixed coordinate frame to describe all points Locations may instead be defined with respect to any convenient frame for which the transformations linking the two are known

1.4.2 Differential Coordinate Changes

It is frequently necessary to consider infinitesimal changes in Cartesian location These correspond to infinitesimal changes in the appropriate coordinate transformation and may be described using either matrices or vectors.

A vector representation for differential changes is possible since differential rotations commute. We can define a differential change vector dc where

$$\mathrm{dc} = (dx, dy, dz, d\psi, d\rho\ d\phi)^T \tag{1.6}$$

in which the first three elements denote an infinitesimal translation, and the last three elements denote an infinitesimal yaw–pitch–roll about the x, y and z axes.

The matrix notation is now described. For a coordinate transformation defined by $\mathbf{C}$, a differential change in the values of this matrix may be represented by the symbol $\mathrm{d}\mathbf{C}$. This differential change can be decomposed into the product of two matrices

$$\mathrm{d}\mathbf{C} = \mathbf{C}\,\boldsymbol{\Delta} \tag{1.7}$$

where $\boldsymbol{\Delta}$ is defined as the *differential* matrix. Given the above definition of the components of the differential change vector dc, it can be shown that

$$\boldsymbol{\Delta} = \begin{pmatrix} 0 & -d\phi & d\rho & dx \\ d\phi & 0 & d\psi & dy \\ -d\rho & d\psi & 0 & dz \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{1.8}$$

Consider now a differential change ${}^{O}\mathrm{d}\mathbf{C}$ observed in frame O. To describe this change with respect to another frame X, from equation (1.4) we have

$${}^{X}\mathrm{d}\mathbf{C} = {}^{O}\mathbf{X}^{-1}\,{}^{O}\mathrm{d}\mathbf{C}\,{}^{O}\mathbf{X} \tag{1.9}$$

from which may be obtained

$${}^{X}\mathbf{C}\,{}^{X}\boldsymbol{\Delta} = {}^{O}\mathbf{X}^{-1}\,{}^{O}\mathbf{C}\,{}^{O}\boldsymbol{\Delta}\,{}^{O}\mathbf{X} \tag{1.10}$$

Given that

$${}^{X}\mathbf{C} = {}^{O}\mathbf{X}^{-1}\,{}^{O}\mathbf{C}\,{}^{O}\mathbf{X} \tag{1.11}$$

equation (1.10) yields

$${}^{X}\boldsymbol{\Delta} = {}^{O}\mathbf{X}^{-1}\,{}^{O}\boldsymbol{\Delta}\,{}^{O}\mathbf{X} \tag{1.12}$$

Similarly, in vector notation, we may also wish to observe a differential change vector dc in different coordinate frames. This relationship is a linear mapping, and we denote it with the matrix $\mathbf{D}_{O,X}$ which transforms ${}^{O}\mathrm{dc}$ into ${}^{X}\mathrm{dc}$. If $\mathbf{n}$, $\mathbf{o}$, $\mathbf{a}$, and $\mathbf{p}$ are the component

vectors of the matrix ${}^{O}\mathbf{X}$ which transforms from frame O to frame X then (1.12) can be used together with the correspondence between the elements of dc and Δ to show that this mapping has the form described in [Paul 81]

$$\mathbf{D}_{O,X} = \begin{pmatrix} n_x & n_y & n_z & (\mathbf{p}\cdot\mathbf{n})_x & (\mathbf{p}\cdot\mathbf{n})_y & (\mathbf{p}\cdot\mathbf{n})_z \\ o_x & o_y & o_z & (\mathbf{p}\cdot\mathbf{o})_x & (\mathbf{p}\cdot\mathbf{o})_y & (\mathbf{p}\cdot\mathbf{o})_z \\ a_x & a_y & a_z & (\mathbf{p}\cdot\mathbf{a})_x & (\mathbf{p}\cdot\mathbf{a})_y & (\mathbf{p}\cdot\mathbf{a})_z \\ 0 & 0 & 0 & n_x & n_y & n_z \\ 0 & 0 & 0 & o_x & o_y & o_z \\ 0 & 0 & 0 & a_x & a_y & a_z \end{pmatrix} \tag{1.13}$$

1.4.3 Transformation of Generalized Forces Between Coordinate Systems

We now discuss forces, beginning with a fundamental result relating the observed forces acting on a mechanical system as seen in different coordinate systems. This treatment is valid for any coordinate system, not only the orthonormal coordinate frames which have been discussed above. The formulation is based on material in [Goldstein 50], Chapter 1.

Consider a mechanical system to be represented by a set of coordinates $\mathbf{p}_1$ and corresponding forces $\mathbf{q}_1$. Now assume that the same system may be described by a second set of coordinates $\mathbf{p}_2$ and forces $\mathbf{q}_2$, where $\mathbf{p}_1$ and $\mathbf{p}_2$ are related by a differentiable function f

$$\mathbf{p}_2 = f(\mathbf{p}_1) \tag{1.14}$$

If we now consider an infinitesimal displacement in the first coordinate system, acting against the forces $\mathbf{q}_1$, this will produce, by definition, an infinitesimal amount of work dW. Likewise, an infinitesimal displacement in the second coordinate system must produce the same amount of work, since the same mechanical system is being described. This gives us

$$\mathbf{q}_1^T\, d\mathbf{p}_1 = \mathbf{q}_2^T\, d\mathbf{p}_2 = dW \tag{1.15}$$

Differentiating (1.14) yields

$$d\mathbf{p}_2 = \sum_{i,j} \frac{\partial f_i}{\partial p_{1(j)}}\, dp_{1(j)} \tag{1.16}$$

which can be written as

$$d\mathbf{p}_2 = \mathbf{J}_f\, d\mathbf{p}_1 \tag{1.17}$$

where $\mathbf{J}_f$ is the Jacobian of the function f. Substituting into (1.15) yields

$$\mathbf{q}_1^T \, d\mathbf{p}_1 = \mathbf{q}_2^T \, \mathbf{J}_f \, d\mathbf{p}_1 \tag{1.18}$$

from which we obtain

$$\mathbf{q}_1 = \mathbf{J}_f^T \, \mathbf{q}_2 \tag{1.19}$$

It should be remembered that $\mathbf{J}_f$ is, in general, a function of $\mathbf{p}_1$.

This result can be used in specifying the torque transformation matrix for a mechanically coupled system, such as the drive train for a robot, and also justifies using the manipulator Jacobian to transform between joint torques and Cartesian forces acting on the robot.

1.4.4 Forces in Cartesian Space

The static forces acting on a body may treated with a notation similar to that for differential motions. We define a force vector $\mathbf{q}$, where

$$\mathbf{q} = (f_x, f_y, f_z, \tau_x, \tau_y, \tau_z)^T \tag{1.20}$$

with the first three elements indicating translational force, and the last three elements indicating the torques about the x, y, and z axes.

Within a rigid body, it may be desired to observe these forces in different Cartesian frames. Because the body is rigid, differential motions of the body as seen in frames O and X are related by the transformation $\mathbf{D}_{O,X}$ in (1.13), which may assume the role of $\mathbf{J}_f$ in (1.17). Equation (1.19) then yields

$${}^X\mathbf{q} = \mathbf{D}_{O,X}^{-1T} \, {}^O\mathbf{q} \tag{1.21}$$

which defines a linear mapping $\mathbf{F}_{O,X}$ between the different force observations

$$\mathbf{F}_{O,X} = \mathbf{D}_{O,X}^{-1T} \tag{1.22}$$

Given the coordinate transformation $\mathbf{X}$ from O to X, with columns $\mathbf{n}$, $\mathbf{o}$, $\mathbf{a}$, and $\mathbf{p}$, this may be solved to obtain:

$$\mathbf{F}_{O,X} = \begin{pmatrix} n_x & n_y & n_z & 0 & 0 & 0 \\ o_x & o_y & o_z & 0 & 0 & 0 \\ a_x & a_y & a_z & 0 & 0 & 0 \\ (\mathbf{p}\times\mathbf{n})_x & (\mathbf{p}\times\mathbf{n})_y & (\mathbf{p}\times\mathbf{n})_z & n_x & n_y & n_z \\ (\mathbf{p}\times\mathbf{o})_x & (\mathbf{p}\times\mathbf{o})_y & (\mathbf{p}\times\mathbf{o})_z & o_x & o_y & o_z \\ (\mathbf{p}\times\mathbf{a})_x & (\mathbf{p}\times\mathbf{a})_y & (\mathbf{p}\times\mathbf{a})_z & a_x & a_y & a_z \end{pmatrix} \tag{1.23}$$

It should be noted that in the case of translational forces $\mathbf{f}$ only, this transformation reduces to the matrix $\mathbf{X}^T$, if $\mathbf{f}$ is represented by a homogeneous four vector $(f_x, f_y, f_z, 0)^T$.

1.4.5 Petri Nets

Petri nets are a form of graph which may be used to study the flow of events in time. They will be used briefly in this work to model synchronization protocols between different processes. An excellent introduction to Petri nets is given in [Peterson 81], a very cursory description will be given here.

A Petri net consists of a set of *places*, a set of *transitions*, and a set of directed arcs connecting places with transitions and transitions with places. Places from which arcs are directed to a transition are called *input places*, while places which receive arcs from a transition are called *output places*.

Places may contain zero or more tokens. When all of the places connected to a transition contain at least one token, that transition is said to be *enabled*, and may *fire*. Firing a transition consists of taking one token from each of that transition's input places and putting one token in each of that transition's output places (Figure 1.7). The firing occurs instantaneously. In a net where several transitions are enabled, the decision as to which one will fire next is nondeterministic.

The distribution of tokens among the places of a net is known as its *marking*, which for a net with n places can be represented by an n-tuple M. Transition firings redistribute tokens, and result, in general, in a different marking. Given a Petri net and an initial marking, it is possible to describe all the possible firing sequences that may subsequently follow as a tree, the nodes of which are the various intermediate markings of the net. This tree is known as the *reachability tree* for the net. The set of all possible markings is known as the *reachability set*.

In modeling systems, the markings of the net are generally used to indicate different states of the system. The system state is changed by the firing of transitions. An examination of the reachability tree associated with a given initial marking, or state, will give knowledge about all the possible states which may subsequently occur. Since the net makes no assumptions about when enabled transitions fire, time is involved in the analysis only to the extent of determining the order in which possible sequences of states may occur. This can be useful in studying asynchronous systems where it may be difficult or unwise to make assumptions about the time delays associated with the state transitions.

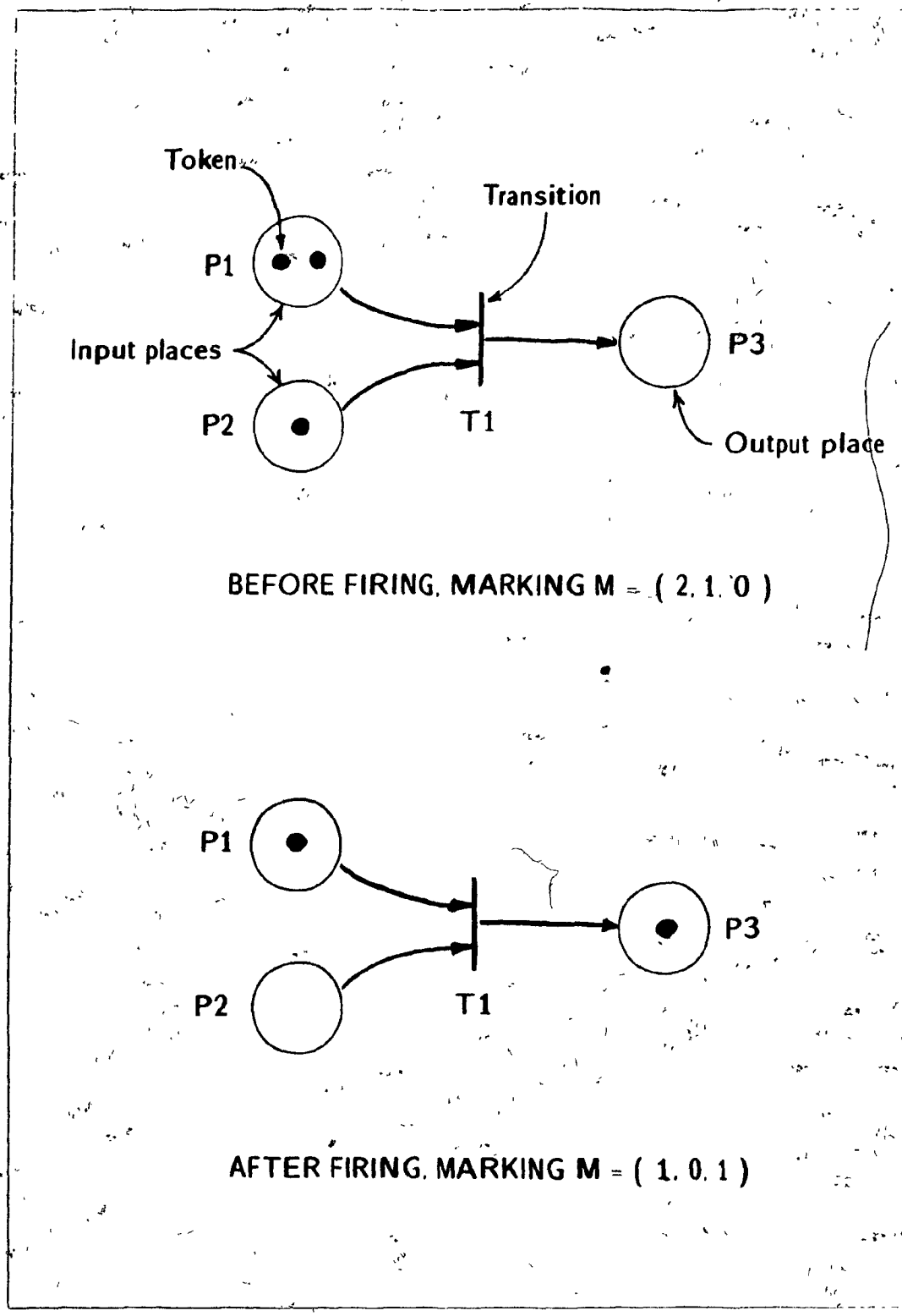


Figure 1.7 A marked Petri net before and after firing

Chapter 2

RCI: A DEVELOPMENT TOOL for REAL-TIME ROBOT CONTROL SOFTWARE

2.1 Overview

This chapter discusses a software system that allows a user to develop and experiment with real-time robot control programs. The system is designed to provide a useful medium for the creation of control procedures that interact with a robot at rates in the vicinity of 10 to 100 Hz. The "user" of such a system is a programmer interested in investigating or building low level robot control software that operates at these frequencies.

Programs written using this system execute as two tasks running in parallel: a *control* task, which executes a robot control algorithm at a periodic sample rate, and a *planning* task, which provides high level directives to the control level. These tasks are arranged as a two level hierarchy. The control level executes at high priority in a non-interruptable context, while the planning level executes in a conventional timesharing context. The control task communicates directly with the joint microcontrollers, taking the place of the supervisor controller in Figure 1.3.

A version of this system, called RCI (Robot Control Interface), has been implemented at CVaRL on a VAX/UNIX host. It is available to the user as a library of command primitives and data structures in the programming language C [Kernighan and Ritchie 78]. RCI is an outgrowth of a previous system, called RTC (Real Time Control), developed at Purdue University in 1983 [Hayward and Paul 83, Hayward 83A]. At present, the CVaRL version

provides an interface to one robot and operates in a uniprocessor configuration. Research is in progress to extend the system to multiple robots and processors.

This chapter is organized as follows. We first present a description of the version of RCI which is currently in operation at CVaRL, followed by a synopsis of its advantages, disadvantages, and the differences between it and the original Purdue system. The last part of the chapter is devoted to a discussion of the issues involved in extending the services offered by RCI to a multiprocessor multi-robot system.

2.2 The Present RCI System

A more detailed description of the various RCI primitives and constructs is given in Appendix A.

2.2.1 Structure

RCI applications are written like conventional C programs. The planning level corresponds to the main part of the program: it interacts with the operator, performs I/O with the file system and other devices, does high level computations, and communicates with the control level. The control level drives a pair of user-written C functions * which implement the desired control algorithm. these functions are "bound" to the control task using special RCI primitives. When in operation, they are called repeatedly, in parallel with the rest of the program, at the sample rate the system is set for†. The planning and control levels communicate through global variables (*i.e.* shared memory). Both levels communicate with the robot through predefined data structures. A data structure, called *how* contains information describing the state of the arm, while a structure called *chg* is used to control the arm‡.

The basic structure of a program written using RCI is illustrated in Figure 2.1. The planning task, which runs in the usual timesharing context, has access to all of the standard resources such as files, devices, and system calls. The control task, in order to meet the

* The reason for having two control functions is explained later.

† The sample rates presently available for the system are 9, 18, 36, and 72 Hz.

‡ The somewhat cryptic names of these two variables are due to historical reasons.

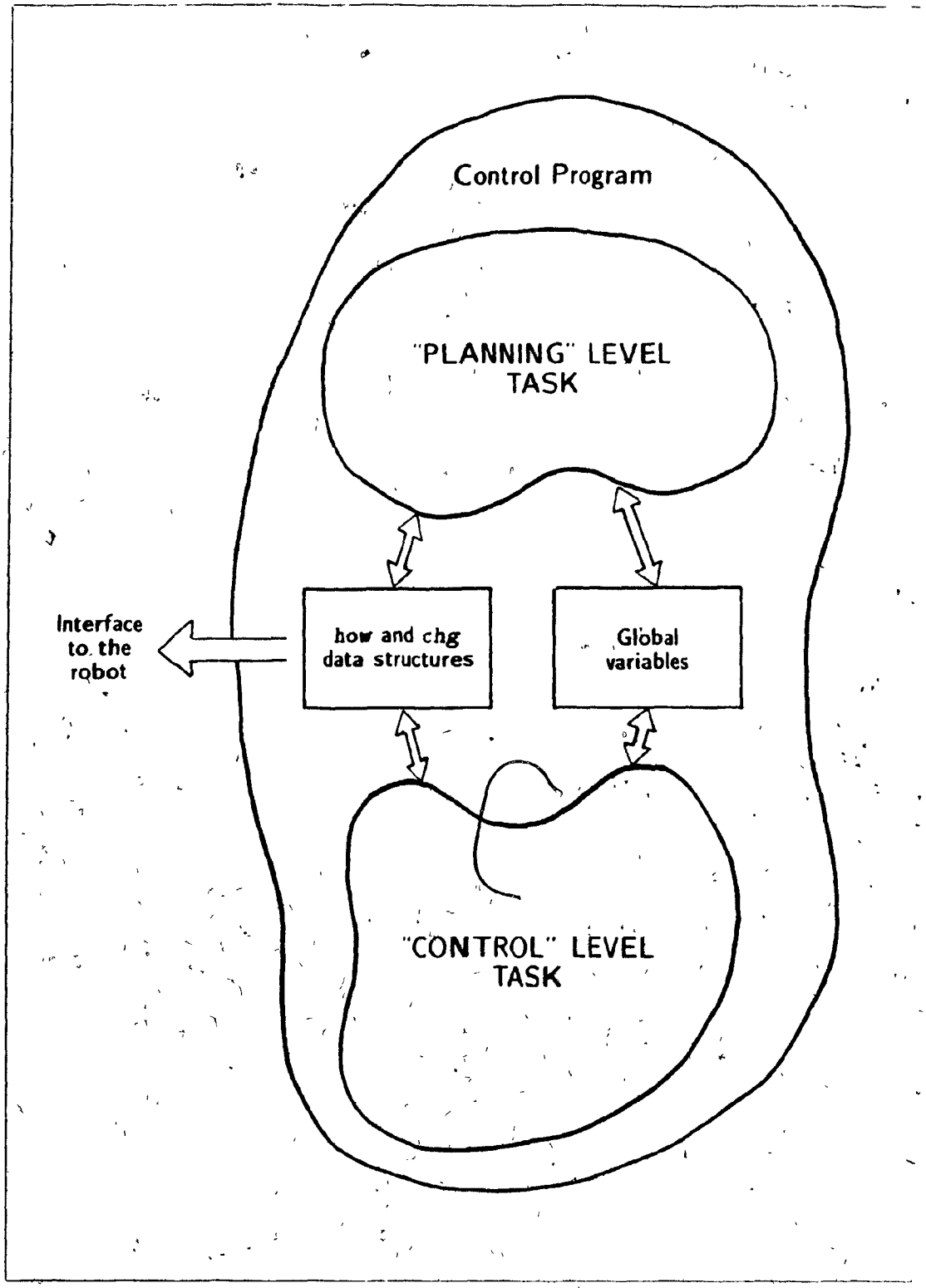


Figure 2.1 An RCI control program

constraints imposed by its sample rate executes at very high priority in system mode and consequently does not have access to the usual system calls or I/O facilities. Since the control functions are invoked asynchronously by the RCI system, they are called with no arguments, all communication with the rest of the program is done through global variables.

The control task operates at the robot level (section 1.3.1), it is responsible for producing commands which are sent directly to the microprocessors controlling the axes of the robot. The user-defined control functions compute these commands on the basis of information contained in the *how* data structure (such as joint positions and observed motor current values), and send them to the robot controller by writing into the appropriate fields of the *chg* structure. Commands are available through this structure to do the following kinds of operations (see Appendix A).

Set a joint position: servo the specified joint to the indicated angle during the next sample interval.

Set a joint current: set the motor current for the specified joint to a certain value (in lieu of position servoing).

The RCI system examines the *chg* structure once every sample period, and relays the requested commands to the robot. Low level sensory input is thus provided by the structure *how* while *chg* provides the actuator output.

We now discuss the operation of the control level in greater detail (Figure 2.2). The user defines two control functions, and binds them to the control task at run time with a special RCI primitive called from the main program (planning level). The control task then begins running. Once every sample period, a buffer of feedback information is transmitted from the joint microcontrollers (located in the robot control unit) to the control task through a high speed link. This information contains such items as the observed joint positions, the joint motor currents, and the robot's status. Some internal checking is first performed on the incoming data, to make sure that the robot system is in "good health". The information is then mapped into the *how* data structure and control is given to the first of the user's functions. After this function returns, the *chg* structure is examined, the indicated joint level commands are sent out to the robot, and the user's second control function is called. When this function returns, the control task waits for the next sample period, and the

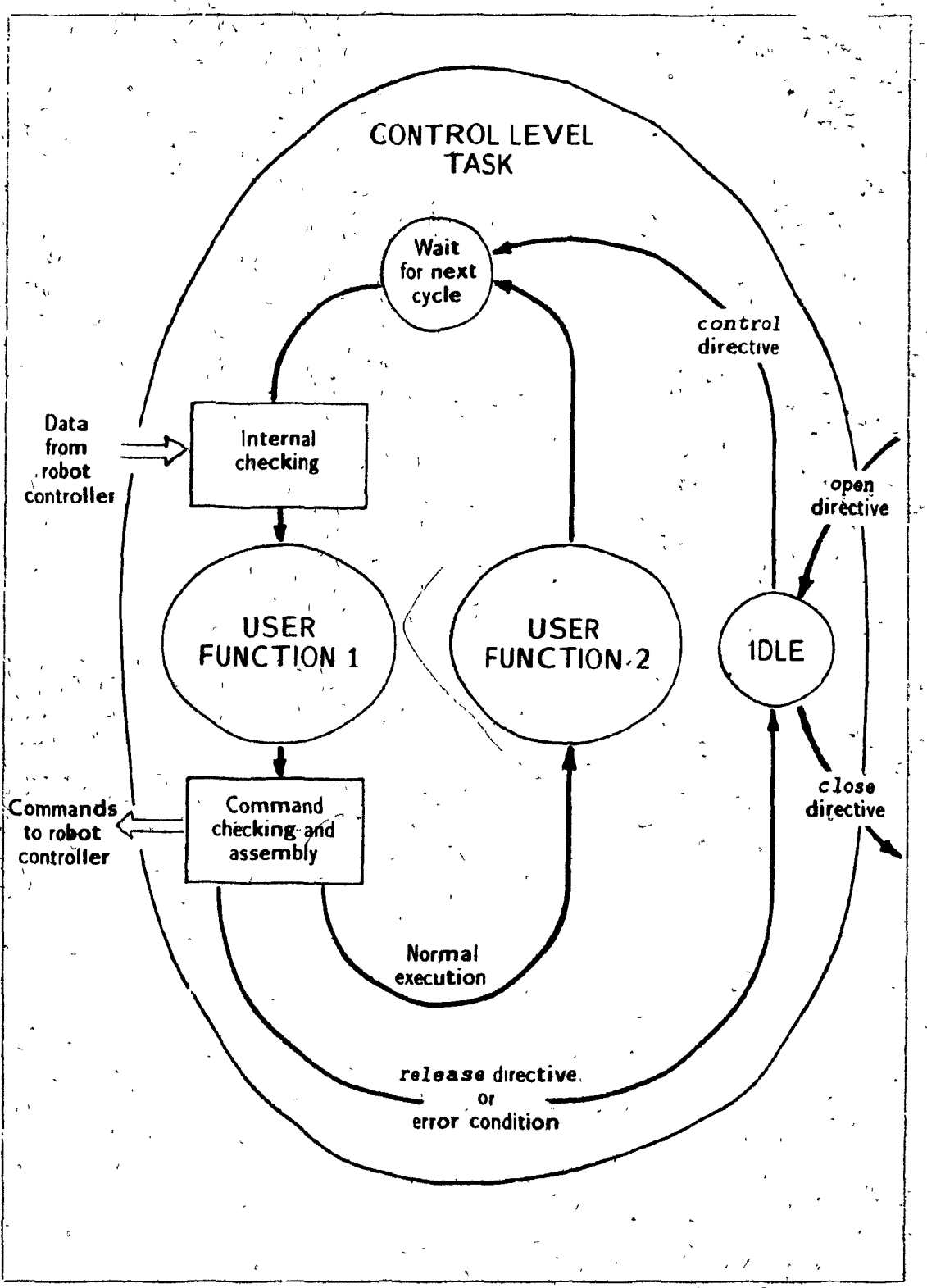


Figure 2.2 The RCI control cycle.

cycle begins anew. The two control functions provide the ability to both respond quickly (function 1) and perform computations at the same time the robot controller is executing its commands (function 2). A full sequence of exchanging data with the robot controller and calling the user's functions defines one *control cycle*. Control cycles are executed at the sample rate associated with the control level. This rate may be set by the user's program to any of the available values; the rate of 36 Hz is commonly used in the present implementation.

The control task is set up and managed, at run time, by the user's main program, using a set of RCI primitives. The directive `RCIopen()` initializes the control task and puts it into "idle", waiting for the control cycles to start. Another function call, `RCIcontrol()`, is used to select the two control functions and begin the control cycles; while these are executing, we say that a *control* is in progress. A control will be *released* either by a call to `RCIrelease()`, or upon the occurrence of a error condition; in both cases, control is released, the robot is notified, and the control task goes back into idle, waiting for another control to be initiated. Finally, the control task may be shut down permanently with a call to `RCIclose()`. If the user's program exits without first closing the control task, the task is closed automatically.

2.2.2 An Example

An example is now given of a control program written using RCI. The purpose of this program is to put the robot into a "zero gravity" mode, where the current on each joint is maintained at a level just sufficient to counteract the gravity loading presently acting on that joint. This means that the manipulator can be moved about freely, but will not fall down under its own weight. To do this, the joints are operated in "current mode" instead of position servo mode.

A requirement of the present implementation is that control functions and their data structures be defined in modules separate from the main program.

main program definition file *(planning level)*

```
1 #include <stdio.h>
2 #include <robot/rci.h>                /* RCI definitions                */
3
```

```
1  extern dummy(), weightless();          /* the control routines */
2
3  main()
4  {
5     RCIopen();                          /* Initialize RCI system */
6     RCIcontrol (dummy, weightless);     /* Start control using these */
7                                         /*  two functions */
8
9     chg_power_on_com = YES;             /* Turn on the arm power */
10
11    printf ("Type <CR> to exit:\n");
12
13    while (getchar() != '\n')           /* and then wait for */
14                                        /*  carriage return */
15
16    RCIrelease(1);                      /* Release control task */
17    RCIclose();                         /*  and shut off RCI system */
18 }
```

control routine definition file *(control level)*

```
1  #include <robot/rci.h>
2
3  dummy()                                /* A dummy function which */
4  {                                      /*  does nothing */
5  }
6
7  weightless()                           /* Control function for */
8  {                                      /*  zero gravity computation */
9
10    double angles[6];                   /* Robot joint angles */
11    double gload[6];                    /* Gravity loading torques */
12    short current_setting[6];           /* Output current values */
13    int i;
14
15    /* Compute the joint angles from the optical encoder counts */
16    /* which are available in the "how" structure. Use these angles */
17    /* to compute the gravity loading torques. Then calculate the */
18    /* required joint current settings from the torque values. */
19
20    enctoang (angles, how.pos);
21    compute_gravity_loadings (gload, angles);
22
```

```
23      torqtodac (current_setting, gload, 0, 0);
24
25      /  Finally, output a joint level command that sets the current  /
26      /  of each joint to the required level.                         /
27
28      for (i=0, i<6, i++)
29       { chg.motion[i].com = CUR,
30         chg.motion[i].value = current_setting[i],
31       }
32  }
```

The main program file starts by including a set of RCI parameter and variable declarations from the file *<robot/rci.h>* (line 2). The program begins by initializing the RCI system and starting a control task which uses the functions *dummy()* (it is not desired in this case to make use of the first control function), and *weightless()* (lines 8 - 9). The sample rate for this task is the default rate of 36 Hz. Once the control has been started, the arm power is switched on by writing the necessary command into the *chg* structure (line 12). Then the planning level simply waits for the person running the program to request an exit by typing a carriage return. When that is done, control is released and the RCI system is closed down (lines 19 – 20).

We now look at the file defining the control functions. The first control function, *dummy()*, does nothing. The second function performs the actual computation. The gravitational loadings on the joints of the robot are a function of the joint positions and certain fixed robot parameters. The function begins with a function call to determine the joint angles from the optical encoder values contained in the *pos* field of the *how* structure (line 20). A second function call determines the gravity loading torque, and a last function computes the motor currents necessary to counteract this load (lines 21 – 23). A command is then issued to each joint, via the *chg* structure, instructing it to exert this required current (lines 28 – 31). While this control is in progress, the joints of the arm will be free and "weightless", and an operator may put the manipulator in any desired position.

2.2.3 Other Aspects of the System

The present implementation requires that RCI programs be compiled and loaded in a special way. Since the necessary specifications to the C compiler and the loader are

somewhat tedious an interface command called rcc was developed by the author which hides the details from the user This command is essentially identical to the C compiler command (cc under UNIX) except that it ensures that the control functions are compiled and loaded in a way compatible with the RCI system, and also references by default the RCI and math libraries The rcc command is described in slightly greater detail in Appendix A, section A.6

Because of the real-time requirements which the control functions must satisfy they are subject to certain restrictions they must complete execution within the sample time interval, and they may not perform any UNIX system calls There is also a small possibility that a memory reference bug in the functions will result in a system crash. These limitations are summarized in Appendix A, section A 2

RCI performs a certain amount of run-time checking (Figure 2 2) for such errors as a joint moving out of bounds or traveling at too great a speed. If such problems are observed, an error condition will occur* Errors may also be reported by the robot controller or caused by a loss of communication In each case, the global variable *RCIterminate* (whose normal value is 0) is set to a code describing the nature of the error condition, an implicit *release* is performed, and a *signal*† is sent to the planning task The arm power may or may not be turned off, depending on the severity of the error By default, the signal is caught by RCI software which prints a diagnostic message and causes the program to exit

It is possible for the programmer to specify his/her own error handler which catches the error signal and performs recovery operations without causing the program to exit The handler can examine *RCIterminate* to determine what occurred and what corrective action should be taken Since the error causes an implicit control release, any further control of the manipulator requires that control be reinitiated through a call to *RCIcontrol()* Although this may be done from within the interrupt handler, it is generally unwise to do so since the RCI signal mechanism is not recursive (*i e* if another error occurs while the program is executing inside an error handler, further error signals will be blocked and the program will "hang") Instead, it is recommended that control be redirected to another part of the main program and the recovery actions be performed there This can be done quite

---

* These checks can be turned on or off by the programmer

† A UNIX software interrupt

easily through the use of the *longjmp* facility available under UNIX. Details about the error handling mechanism and *longjmp* are discussed in Appendix A, section A.5.

Debugging real-time programs can be tedious* since when the control task is executing at elevated priority it is often not possible to perform diagnostic output or run in an interactive debugging mode. There are two ways around this problem:

- Write appropriate diagnostic information into a buffer at run time and then later dump this buffer out for examination.
- Provide a simulator on which to run the software in a non-real-time environment.

The first approach tends to be required when problems arise that are highly dependent on real world conditions that are complex and not possible to simulate. On the other hand, major software errors may not even permit the system to run at all in real-time or otherwise, and in these circumstances a simulator/debugger can be extremely useful. It has been this author's experience that simulation is very convenient for performing the initial "checkout" of a piece of code, while the more obscure programming defects tend to surface only under real-time conditions, and must be ferreted out with information dumps.

An information buffer is easy for a user to implement in RCI; it can be defined simply as a global data structure accessible by both the control and planning tasks into which information is written at control time, and later examined by the planning level which can then print the relevant information on the screen or write it into a file.

A simulator facility was written by the author which makes it possible for a robot control program to attach to and interact with a simulator program instead of with the robot (Figure 2.3). All of the user's software and most of the RCI software, remains unchanged, only the communication link which nominally runs to the robot controller is altered. The programmer uses the simulator by linking the control program with a different RCI library. The simulator program is started and run independently of the RCI program. When the user's program is run, the RCI software establishes a link with the simulator program, and communication proceeds in a manner identical to the real-time version:

- the simulator interrupts the control program (using a UNIX signal)

* and lots of coffee

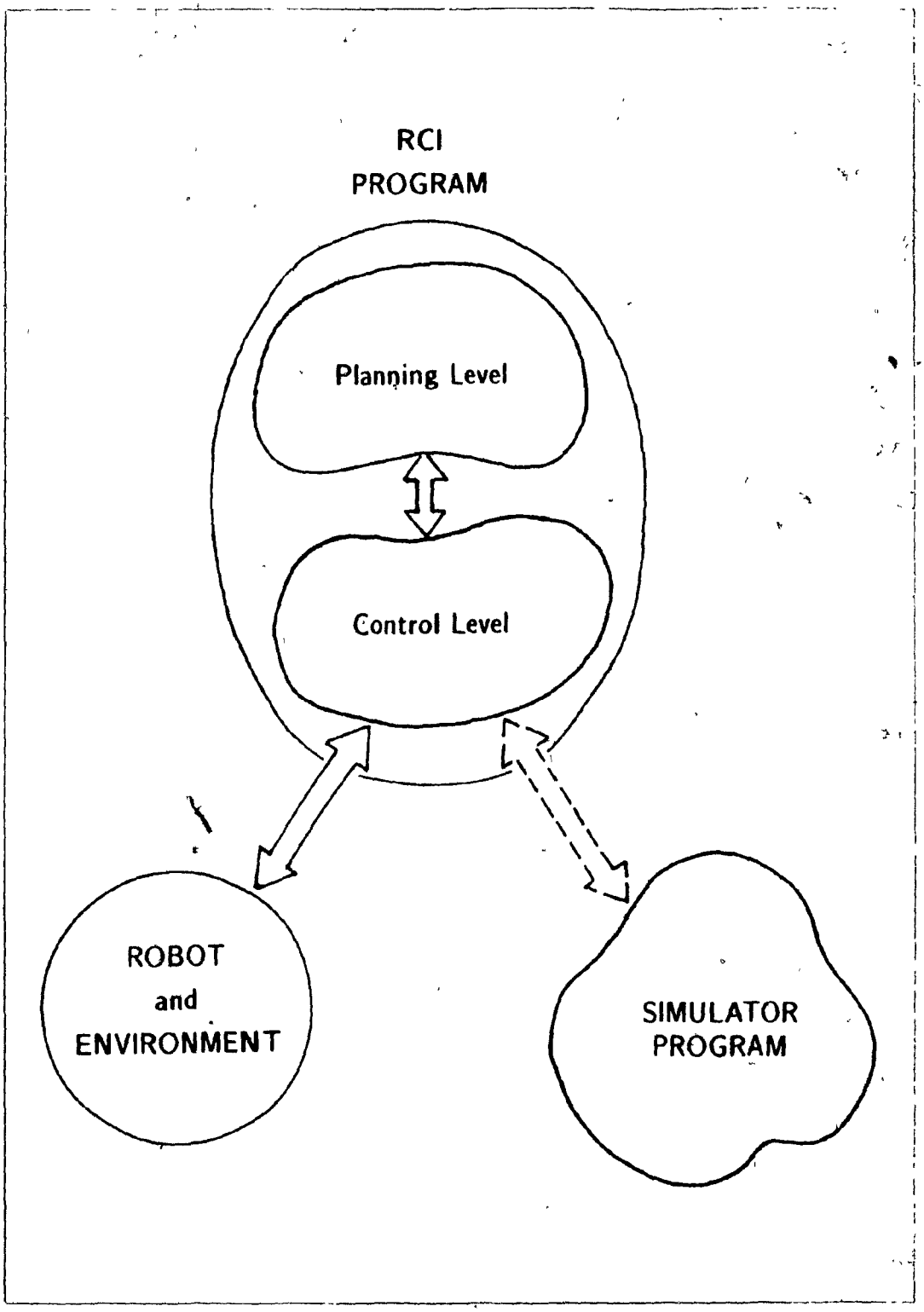


Figure 2.3 A simulator program may be substituted for the actual robot

- the control program reads state information from the simulator, invokes the usual control level functions, and writes command information back
- the simulator digests the commands and modifies its internal state accordingly

Since the simulated control program is executing completely in a normal UNIX timesharing context, the programmer is free to make use of standard I/O routines and other UNIX system facilities from within the control functions. More details on the simulator program are found in Appendix A, section A.7.

It should be noted that the only assumptions RCI makes about the simulator program concern the communication link between it and the control program. Hence different simulators may be developed and interchanged for different applications, including graphic animation. The simulator itself may be quite simple or very complex. It turns out that it is rather straightforward to model the free space behavior of a single arm, providing the desired joint velocities are not too great. This is because the dynamic terms of the manipulator equation have, at low velocities, small values compared to the friction and gravity loading terms, and hence may be ignored (see Chapter 4, section 4.3). The present simulator available for RCI makes this assumption, which classifies it as a kinematic simulator. More complex *dynamic* simulators can be useful for investigating the detailed behavior of an arm and testing the validity of the control algorithms themselves; the development of these simulators has been the subject of recent research [Goldenberg 82]. A couple of such simulators are available at CVaRL, although they have not yet been interfaced to the RCI system. Another aspect of the simulation problem involves the modeling of the external environment around the robot (*e.g.*, obstacles in the workspace), and the interaction of the robot with that environment [Wesley, et al. 80, Lozano-Perez and Wesley 79]; this is central to the off-line programming problem. The difficulties involved are a function of the complexity of the robot environment to be simulated. Such a simulator is presently not available at CVaRL.

2.2.4 Implementation

This section describes the implementation of RCI under UNIX 4.2bsd on a VAX 750. More detailed information about the VAX architecture may be found in [Digital 82A], more detailed information about UNIX 4.2bsd may be obtained from [Berkeley 83].

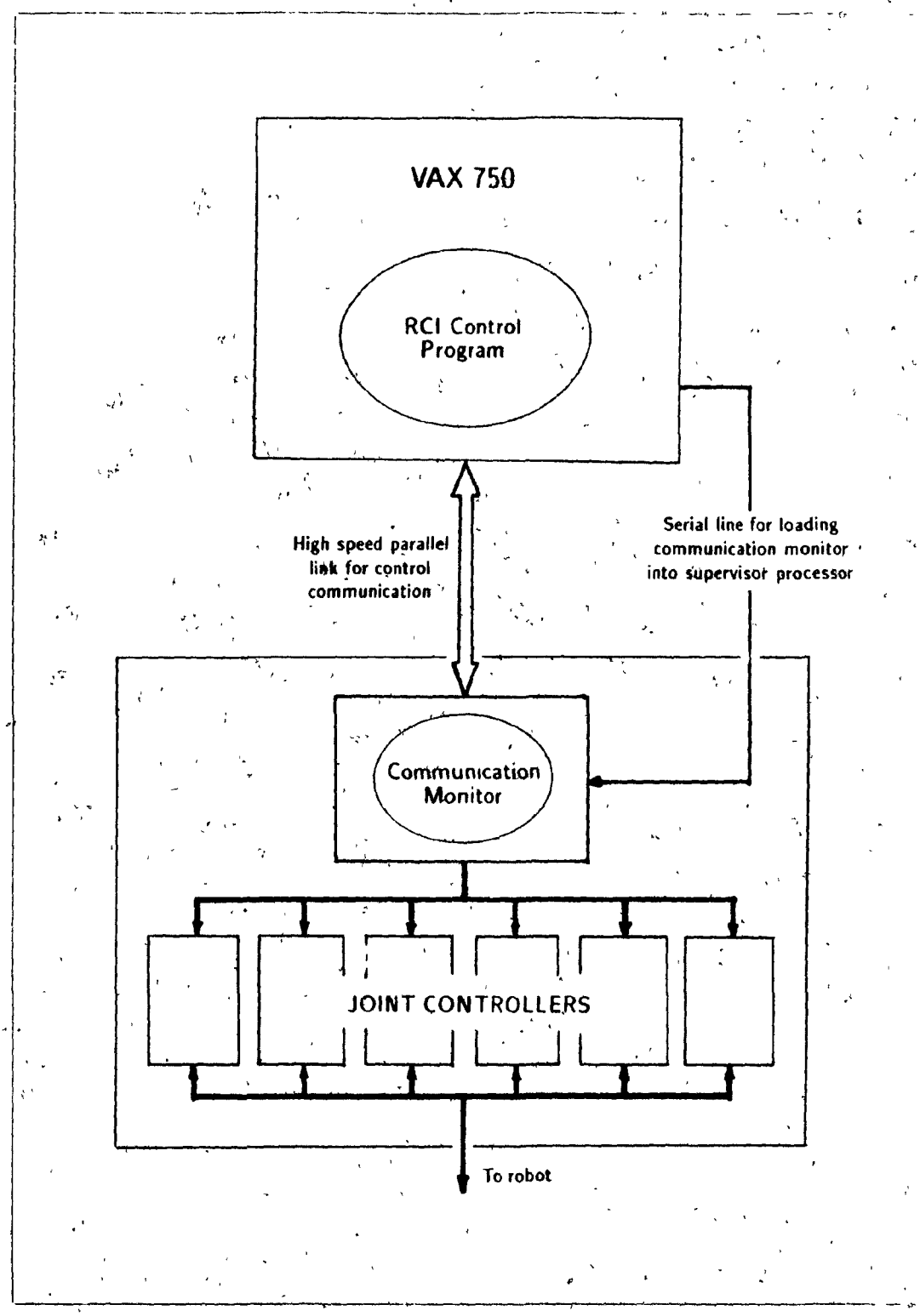


Figure 2.4 Physical implementation of the RCI system

Figure 2.4 illustrates the physical arrangement. The robot controller and the VAX are connected by a high speed parallel link which is interfaced to the UNIX system through a special purpose device driver. The supervisor control program (Figure 1.3) in the robot controller is replaced by a simple communication monitor. the function of the supervisor is assumed by the RCI control task executing on the VAX. Each control cycle is initiated by the monitor, which after gathering feedback information from the joint microcontrollers and other peripheral devices, sends an interrupt to the VAX over the parallel link. The VAX then executes the control cycle (Figure 2.2) using the parallel link to transfer command and feedback information to and from the robot controller.

To obtain real-time response, the control task is executed at elevated hardware priority. This is accomplished as follows. The *RCIcontrol()* routine passes to the device driver the addresses of the two user-specified control functions. When an interrupt is received from the robot, the driver interrupt routine, which is executing in kernel mode*, takes control and blocks out all other hardware interrupts (Figure 2.5). It then quickly restores the memory context of the robot control program, performs I/O with the robot and calls the necessary RCI interface routines and user control functions associated with the control cycle. Memory context is then restored, and the interrupt routine returns, allowing the system to continue with its other business until the next interrupt. Needless to say, if the control functions consume too much time, there will be insufficient time available for the rest of the UNIX system to function properly, or even for the control cycle to finish. Because the user's control functions are invoked by the device driver, they are executed in kernel mode and consequently are not separated from the system itself by the conventional "firewall". This approach is necessary on the VAX since, by definition, *only* code executing in kernel mode may block hardware interrupts, and it is necessary to block all interrupts to make sure that the control level functions will execute on time.

The VAX is a virtual memory machine, meaning that the memory assigned to a program may be either present in physical memory, or *paged out* onto secondary storage (*i.e.* disk), to be brought into physical memory when needed. Since the control level is executed in kernel mode at elevated priority, all the code and data which it references must be resident in physical memory, since there is insufficient time to retrieve pages from the disk. This was

* The maximum privilege hardware mode on a VAX.

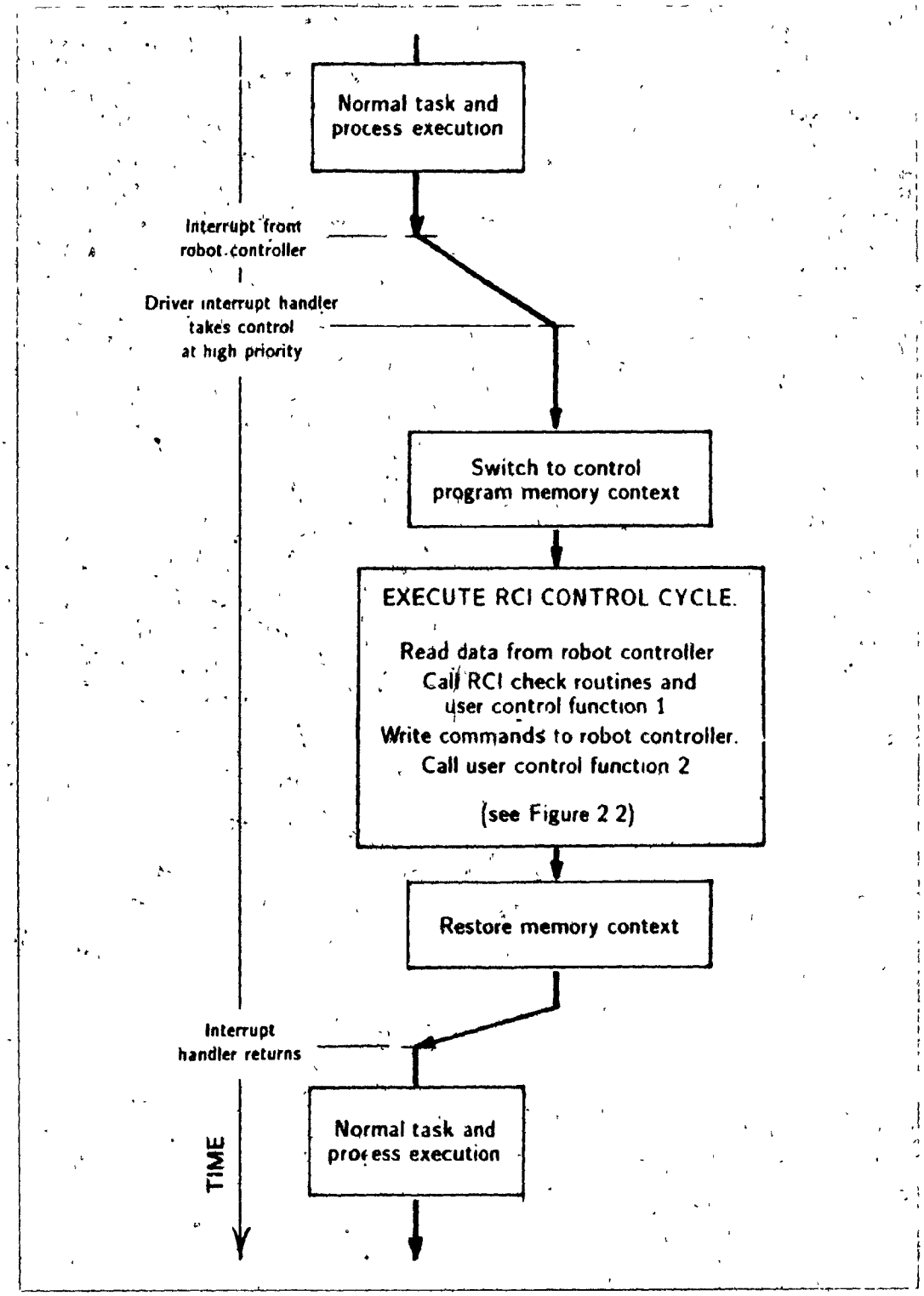


Figure 2.5 Internal VAX execution sequence for a control cycle

made possible by inserting into UNIX a memory locking mechanism which allows a user to request that a certain section of data in his/her program be locked into physical memory RCI takes care of this locking automatically when *RCIcontrol()* is called It determines which regions to lock by examining the location of special symbols which surround the control level code and data regions These symbols are inserted automatically at load time by the *rcc* command, their existence is the reason why RCI programs must be compiled and loaded in a special way

The system calls which have been added to provide the lock mechanism are defined as follows

```
status = memlock (address, size)
 int status, size,
 char *address,

status = memunlock (address, size)
 int status, size;
 char *address,
```

The call *memlock()* takes a region of program memory beginning at *address* and extending for *size* bytes and locks it into physical memory *memunlock()* unlocks the region The region *address* must be page aligned, *size* must be a multiple of the system page size, the region must be readable by the user's program and completely within the either the text or data segment, and no more than *MAX_LOCKED_PAGES* (presently 512) may be locked at once by any program When a process exits, all pages which it still has locked are implicitly unlocked.

A particular problem associated with running functions in kernel mode on VAX systems is that run-time hardware-detected errors will result in a system crash, this is because it is assumed that the operating system is by and large bug free and any errors that do occur should result in a system crash The execution of user-defined functions in kernel mode violates this principle A modification was made so that while the user functions are executing, a flag is set which indicates their presence to the kernel Hardware errors occurring at this time are hence recognized as belonging to the control program, and instead of causing a crash, result in a UNIX error signal being sent to the control program, which is the usual procedure for exceptions at the user level

The present (default) sample rate for the RCI system is 28 milliseconds. Many control programs of interest work well at this speed, although in the case of some more complex ones, such as RCCL applications, this can consume up to two-thirds of the available CPU time, and seriously restrict the number of other processes which can comfortably execute at the same time. Other available sample rates are 14, 56, and 112 milliseconds.

2.3 System Synopsis

2.3.1 Differences from the Original Implementation

The following is a summary of the features which have been incorporated into the present RCI system that were not present in the original Purdue system.

- Under the Purdue implementation, the robot control program was "restricted" in the sense that it could not call on any system routine which altered the memory context of that program while a control was in effect. In particular, this disallowed allocating new memory and opening files. This restriction was due to the way in which the control level code and data were locked in physical memory, and has been removed by the installation of the special kernel primitive *memlock()* (described above) which selectively locks portions of a process's memory space.
- The simulator program and interface.
- The error handling and recovery features.
- The *rcc* command.
- The communication protocol between the robot and the VAX was redesigned, and the user interface was encapsulated into the four RCI control routines.
- Increased control over the system and the robot itself, including the interfacing of external devices attached to the PUMA, the teach pendant, arm power control parameters in the joint microprocessors, and the ability to control checking done on positions and velocities.

2.3.2 System Advantages

The RCI VAX/UNIX system offers the following features

- The ability to write a robot control program, partioned into a planning and a control level, easily quickly, in one language, and as one executable module. Also the entire UNIX environment is available for program development, which has proven to be extremely useful

- The shared memory offers quick and convenient communication between the two control tasks

- The entire computing power of a minicomputer, in this case a VAX, is available for performing expensive control level calculations, such as the inverse kinematics or Jacobian of the manipulator

2.3.3 System Restrictions

The principal restrictions of RCI are

- Since the control level executes at a priority higher than any other process on the system, this can cause a serious load in cases where the computer is running a multiuser environment

- The user has to be wary of errors in the control level routines, since they may be capable of disrupting the system

- There is only one RCI control task which may control only one robot.

Resolving the first and last of these restrictions is the subject of the last part of this chapter

2.4 Extending the RCI System to Multiple Robots and Processors

Given the present trend towards research with systems involving both multiple robots and greater computational complexity, it makes sense to consider expanding RCI to handle multiple control tasks. It also makes sense to consider off-loading these computationally expensive tasks onto dedicated processors.

2.4.1 Multiple Control Tasks

The sort of place where a "multitasking" RCI would be useful is in a system where several actuator and sensory devices operate in close cooperation. Separate RCI control tasks could be used to run each manipulator, process sensory inputs, or serve as "watch-dogs" to monitor the activities of other parts of the system. The main characteristic of these tasks is that they are driven by an explicit external event, such as a timer or device interrupt, and must perform their computation within the time interval allowed by that interrupt. RCI may then be defined as *a tool that allows the creation of a set of interrupt triggered control tasks*. We can add generality to this definition by allowing the existence of software interrupts, although we will not consider this explicitly in the following discussion.

Defining such a mechanism, of course, is akin to creating a simple multitasking operating system where the burden of scheduling has been placed onto the timing hardware: the occurrence of an interrupt causes the function(s) associated with the corresponding task to be executed. This, coupled with the fact that the driving interrupt usually occurs repeatedly, removes the need to explicitly put a task to sleep and wake it up: a task can be made to wait for a condition simply by having it check for the condition at every invocation. Tasks can be given priorities to handle cases where interrupts are superimposed; these priorities can be enforced either by the hardware or the primary interrupt handler. The response time of each control task is determined by its interrupt rate.

What is gained with this approach is simplicity and ease of implementation. Obviously, there are limits to its applicability: we are mainly interested in computationally intensive tasks which run at rates of several Hz or more. If it were to be found that this interrupt driven task paradigm is not adequate for a particular problem, the next step would be to try using a more general real-time operating system, several of which are available

commercially (Intel's iRMX86 system [Intel 84B]) or distributed by research organizations (Harmony [Gentleman 85])

Assuming that it is desired to implement a multitasking RCI environment, attention must be given to the following points

Communication How shall the different tasks be connected, both in terms of hardware and software?

Hardware Architecture What sort of physical configuration is required?

Software Architecture How should the system functionality be defined, implemented, and presented to the user?

Development Environment What sort of programming environment and tool set is necessary for the development of application software?

These are now considered in more detail

2.4.2 Communication Issues

Before discussing hardware and software architecture, we examine some general issues related to communication.

2.4.2.1 Requirements

The inter-task communication facility must provide to the programmer the required types of communication at the necessary bandwidth Different classifications have been made to describe the sorts of communication required in a real-time control system At CVaRL, communication between the planning and control levels has been observed to fall into the following categories

c1 *Directives from the planning level* Commands and associated parameters are sent to the control level from the planning level, through either global variables or a message queue.

c2 *Feedback from the control level* Information computed by the control level is placed into a global data area where it may be sampled at the discretion of the planning level

c3 *Synchronization* The tasks synchronize their activities through the setting of global flags.

A characterization of the types of services needed to perform this communication has been described by the designers of a hierarchical control system for legged vehicles [Schwan, et al 85], who have grouped their communication needs into three types

t1 *Asynchronous communication with data loss* This is typically used at low levels in the control hierarchy Different tasks constantly feed information packets to each other Packets are not buffered, a new packet will overwrite an old one, whether it has been read by the receiver or not Since the information being exchanged generally describes the value of system variables which change slowly relative to the communication rate (such as joint angles), the occasional loss of a packet in this way will not adversely effect the system performance

t2 *Synchronous communication without data loss* Different tasks exchange packets of information, and wait for them if necessary. The individual packets comprise commands and responses, and consequently, the loss of data would interfere with system performance.

t3. *Synchronous or asynchronous communication with possible loss of data* This is a hybrid of types *t1* and *t2*. The data packets exchanged between processes are placed in a buffer of fixed capacity exceeding the capacity of the buffer results either in the loss of the oldest packet (asynchronous case) or in the blocking of the sending process (synchronous case)

What is interesting about this classification is the distinction between numerical control data (*t1*) and symbolic data (*t2*) if numeric control data is communicated at a rate sufficiently fast compared to the speed at which the data varies, it may be possible to tolerate data loss

The required communication bandwidths are difficult to specify exactly. Observations have been made of the communication rates required for different RCI applications at CMaRL, the bulk of which are RCCL programs (Chapter 5) RCCL is a collection of subroutines which control a robot in Cartesian coordinates, with built-in facilities to handle force compliance and adaptive trajectories The RCI control level is used to host the RCCL trajectory generator, to which individual motion requests are passed from the planning level The trajectory generator communicates to the planning level by setting event flags when motion requests are completed and supplying information such as the position of the

Application	Data rate (Kbytes/sec)	Number of individual messages per second
RCCL conveyer tracking program	52	19
RCCL pick and place program	40	21
RCCL guarded motion program	23	12
RCCL compliant edge detection program	26	10
Gravity loading measurement program	14	12
Manual teach mode program	03	13

Table 2.1 Operation — control level bandwidth requirements for various applications

arm. The inter-task communication rates for four different RCCL programs are given in Table 2.1*. These programs include applications where the robot tracks a conveyor belt performs a set of pick and place operations, does a "guarded motion" to detect a surface and uses compliance to infer the position of the edge of a tray. Two other RCI programs unrelated to RCCL are also profiled, these include a program which measures manipulator force control parameters (used to obtain the results of Chapter 4) and a 'teach' program where the operator moves the robot around using commands entered from the terminal or from a teach pendant. The overall transfer rates are on the order of 1 Kbyte/per second.

Because the present implementation supports only one control task, no direct experience has yet been had with communication between different control tasks. In some instances, very little communication may be necessary, this would be in keeping with the paradigm of a strict hierarchy, where the actions of sibling tasks are correlated only at the level of the parent. One the other hand, this will not always be the case. If two robots are operating in the same workspace, for instance, collision avoidance software might require the constant exchange of joint angle information between the control tasks driving each robot. If two robots are working in tight cooperation to hold and assemble a workpiece ([Alford and Belyeu 84]), the exchanged data may include the position and forces acting on each robot, as well as coordinating information. Some rough estimates of the required

* The actual communication took place through shared memory. The data rates given are equivalent to those which would be required if the information were instead transferred over a channel.

bandwidth may be made as follows. Assume that an information packet for each robot consists of the joint angles, their associated sines and cosines, and joint forces. Assuming 4 byte floating point numbers, a 6 joint robot, and a 40 Hz sample rate, then the required bandwidth, allowing for information transfer in both directions, is 7.7 Kbytes per second.

2.4.2.2 · Communication mechanisms

These may be classified into two general types: data channels, and shared data [Bowen and Buhr 80, Allworth 81], each of which has an interpretation in both hardware and software.

Communication through data channels

Data channel communication involves the existence of a data stream between two processes, through which messages may be sent. Data packets are written into the stream at one end by a *sending* process, and read out at the other end by a *receiving* process. Packets may be buffered inside the channel (Figure 2.6(a)).

In hardware, channels manifest themselves as communication lines and FIFOs*. In software (where they are often called *message passing* mechanisms), they are exemplified by the pipes and streams of UNIX [Kernighan and Pike 84], the mailboxes of VAX/VMS [Digital 82B], and many of the software devices associated with computer networks [Tanenbaum 81]. Processes read from the channel using some form of *receive* primitive, and write to the channel using some form of *send* primitive. The design of such primitives for robotics and other applications has been the subject of recent research [Shin and Epstein 85]. The communication modes described in the work of [Schwan, et al. 85] were all implemented using a channel-based mailbox facility.

Channel communication characterizes *loosely coupled* systems, and is associated, in general, with the following properties†:

- May be implemented on any communication medium, including serial lines. The communication speed is dependent on the hardware.
- Portability, given the wide range of supporting mediums.

---

* First in, first out memory.

† Data channels are sometimes known as "thin-wire" connections.

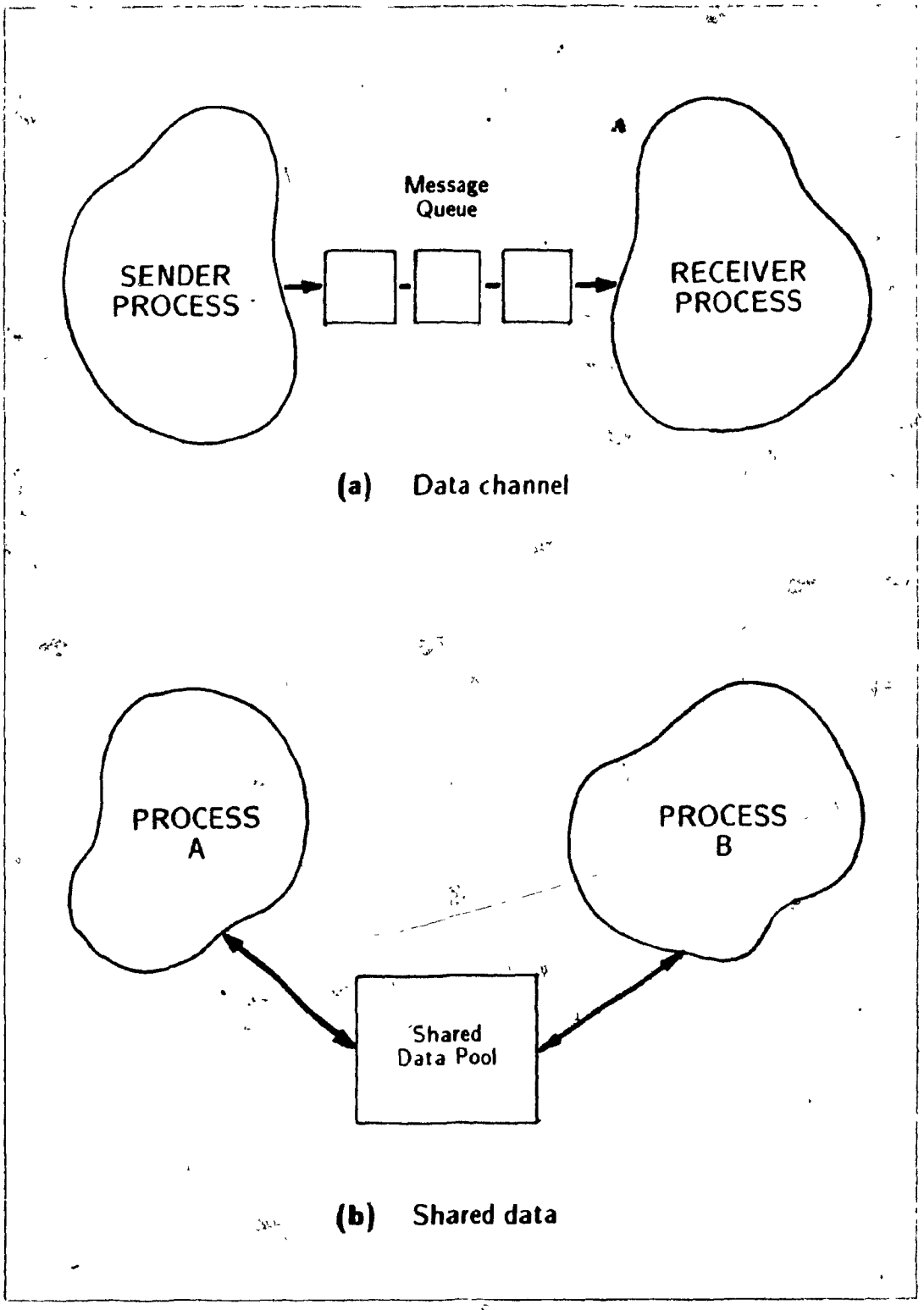


Figure 2.6 Process communication mechanisms

Message packets must be explicitly handled by both the sending and receiving processes By arranging for processes to block until certain packets are received interprocess synchronization may be achieved

There may be a significant software overhead in preparing and interpreting the data packets

Communication through shared data

A second method of process communication involves allowing different processes to access a shared data area (Figure 2 6(b)) In hardware this takes the form of common or multiport memory In software, this is exemplified by global variables and common data regions such as the interprocess sections of VAX/VMS [Digital 82B] Processes access the shared region either by directly referencing shared variables, or by calling intermediate routines which surround the shared data base The version of RCI in use at CVaRL is based on shared memory, which is easy to implement given the single processor implementation Shared data characterizes *tightly coupled* systems, and is associated with the following properties

- High speed
- Hardware dependent Implementation usually requires some sort of high speed parallel bus Although in theory it is possible to implement shared data mechanisms across any sort of communication medium (given time delays) this may be very tedious to accomplish It should be noted, however that shared data hardware may be used to implement either data channel or shared data software

  Processes may access the shared data areas at their leisure Synchronization may be achieved through the use of common event flags
- The software overhead associated with communication is greatly reduced

Access collisions and shared data

A particular problem associated with shared data is *access collision*, which occurs when one task accesses the data while another is in the process of writing it, possibly leading to undesirable results The usual away around this problem is to police access to the shared data, either explicitly using semaphores, or implicitly by encapsulating the access inside a

routine called a *monitor*. General purpose semaphores are straightforward to implement using a nondivisible *test-and-set* primitive, or somewhat less straightforward to implement using only nondivisible data transfers [Holt et al. 78].

Test-and-set primitives and nondivisible data transfers across processers require cooperation between the involved tasks and the communication medium. Hardware support for this is available on some parallel bus designs in the form of a *bus lock*.

In our use of shared memory with RCI, the only requirement has been the ability to transfer several bytes of data nondivisibly; the test-and-set primitive has not been needed. This stems from the fact that RCI type tasks perform specialized computations, with information originating in one *producer* task and being read by a *consumer* task (note communication types *c1* and *c2*, above). It may not be necessary to worry about collisions at all. Consider a process P which computes joint angles and stores them in an array for consumption by a process C. Given that each joint angle can be written nondivisibly and P and C both run at a fast rate compared to the speed at which the angles change, then any errors caused by failing to keep all the joint angle information entirely consistent will be small. This is an instance of communication mode *t1*, described earlier.

If some access control is necessary, then a simple token passing protocol may suffice: processes may access the region only when they possess a token, which is alternately passed back and forth between the producer and consumer. This protocol is modeled, for two tasks, by the Petri Net of Figure 2.7, for which it is easy to examine the reachability set and show that the data will never be accessed by both tasks at the same time. A similar protocol works for the maintenance of queues, providing that only one task adds to the queue and one task deletes from it. With the addition of more tokens, the Petri net of Figure 2.7 also describes a queue protocol (Figure 2.8), depending on the place they are in, the tokens represent either the number of items in the queue, or the number of empty spaces available. Again, it is possible to show that conflicts which corrupt the queue are not possible. It should be noted that such a queue is equivalent to a software channel mechanism.

2.4.3 Hardware Architecture

Hardware design issues associated with a multiprocessor RCI include:

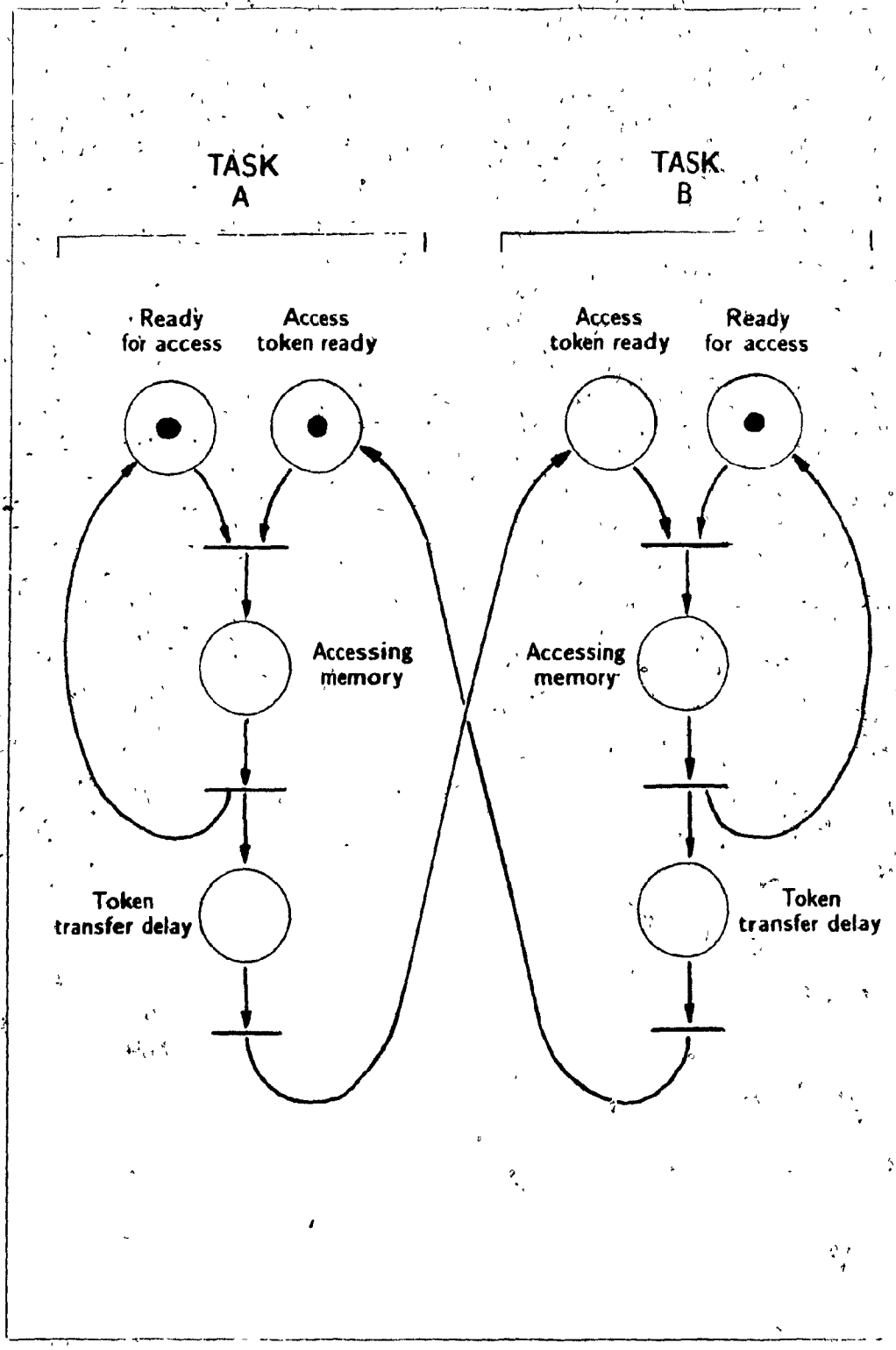


Figure 2.7 Petri net representation for a token passing protocol

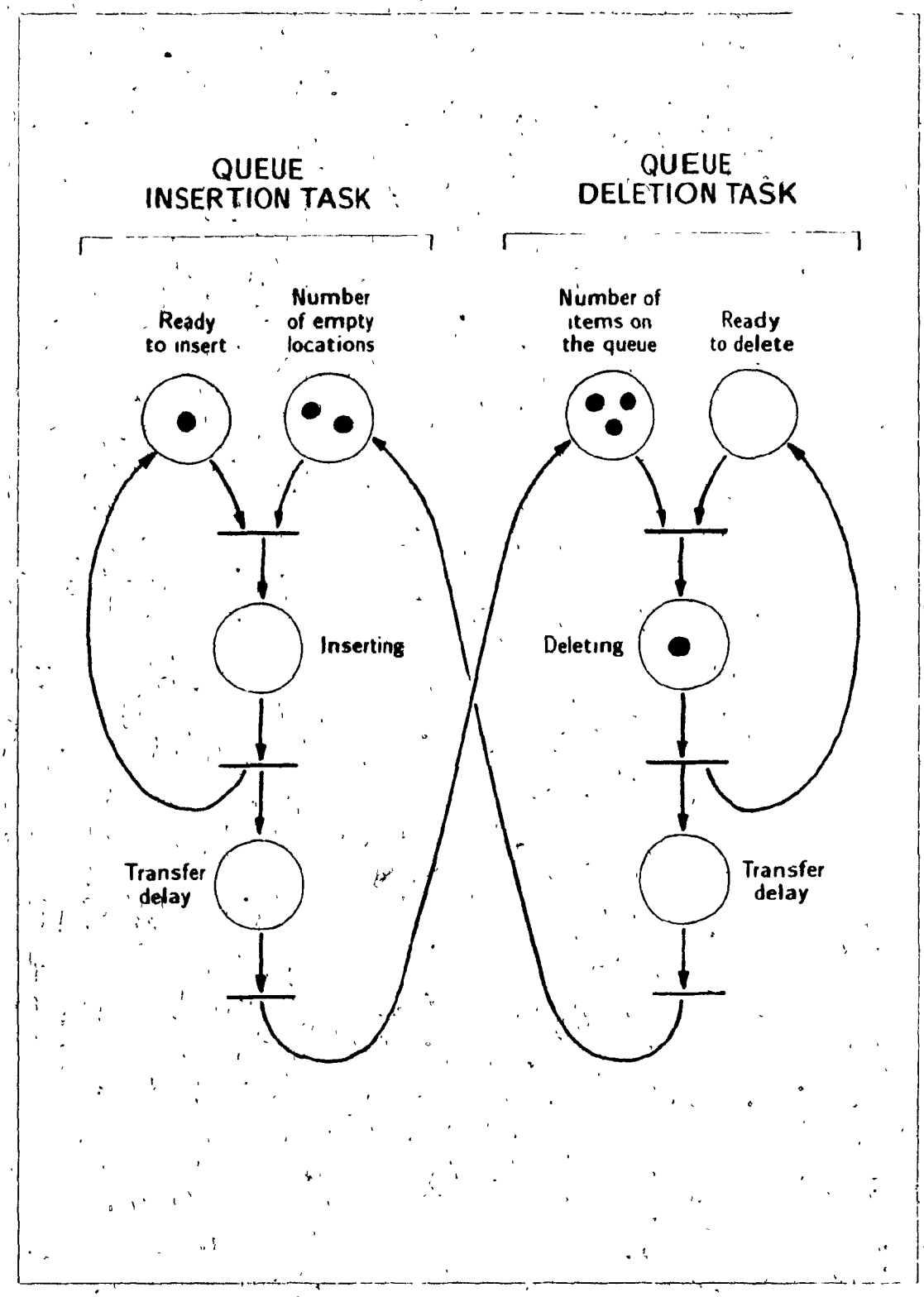


Figure 2.8 Petri net representation of a unidirectional queue

The host computer

Local characteristics of the control processors

Interconnection of the control processors

A general purpose computer is needed to host the individual control processors. RCI is a development system, and consequently a user needs access to an extensive and versatile programming and development environment with a large selection of software and a good file system, as well as peripheral devices such as graphics terminals and printers. The host is used to run the planning level task, and is interfaced to the various control processors which run the control tasks. The VAX/UNIX system currently in use at CVaRL has all of the features of a large host computer.

The next issue concerns the processors themselves. For the sorts of computations RCI is directed at, board level configurations of 16 or 32 bit microprocessors are suggested, preferably with floating point support. Two important requirements are that (1) the different processors be the same or from the same processor family, and (2) each processor or processor board contains local memory. The first requirement is based on a very practical reason: the overhead associated with "taming" any given processor is generally quite large, and in a development environment, particularly an academic one, this burden is often not tolerable. The second requirement is also a practical one in that the use of local memory greatly reduces the amount of interprocessor communication. For the same reason, the processor boards should, when possible, be interfaced directly to the sensory and actuator devices they are controlling. Although in some cases this may require private connections to separate I/O boards, it is important to localize this low level data flow.

The last issue is the physical interconnection of the control processors. The choice is driven by the following requirements:

1. *Generality* Given the usage of the system as a research tool, one cannot know in advance which processors (*i.e.*, control tasks) will have to intercommunicate, so a general and uniform topology is required.
2. *Extensibility* It should be easy to add processors as required.
3. *Availability* The required hardware should be commonly available.

Three well known interconnection topologies support the generality requirement: stars, busses, and rings. Of these, a bus configuration is recommended on the grounds of avail-

ability. (The term "bus" is used here strictly in the topological sense; the issue of whether high speed physically parallel busses are to be preferred over more loosely coupled network style busses is considered next.)

It should be noted that a bus is endorsed with regard to a research environment. Once a particular task communication structure has been demonstrated as useful, specific implementations may make use of special purpose interconnection arrangements.

A question remains as to what type of bus should be used, and how the host should be attached. There are essentially three choices (Figure 2.9). The first is the least expensive; it calls for everything to be interconnected on some sort of thin-wire (loosely coupled) bus. The second alternative calls for interconnecting everything on a parallel bus, to which the host is also attached using a bus adapter. If an appropriate bus adapter is difficult to obtain, then the host may remain attached through a serial bus, while the control processors are connected on a parallel bus (the last approach).

The main factors in choosing a bus arrangement are cost, bandwidth, and geographic proximity of the attached processors. Serial busses are presently available which can provide data transfer rates of up to 200 Kbytes/second (Intel BITBUS [MacWilliams, et al. 84]), and Ethernet hardware runs at over 1 Mbyte/second. However, communication overheads on such serial networks may reduce the effective transmission rate by as much as an order of magnitude or more. Parallel busses are significantly faster (transfer speeds of around 40 Mbytes/sec are available in both the VMEbus and Multibus II [Marrin 85, Intel 84A]), and provide the choice of implementing either shared memory or channel oriented communication software. This feature is notably exploited in Harmony [Gentleman 85], which is a distributed operating system which uses a parallel bus simply to provide a very fast message passing mechanism between processors with local memory. Parallel busses do require that the attached processors are located close together (such as within a single enclosure), but since RCI is intended for use with devices which are closely situated physically (within a robotics workcell, for instance), this is not a serious constraint.

From this discussion, and the bandwidth requirements presented in section 2.4.2.1, we note that a thin-wire interconnection scheme should be sufficient if (1) shared memory is not desired, and (2) most communication is between the planning and control levels only. Otherwise, a parallel bus is recommended, which provides (1) greatly increased speed, and

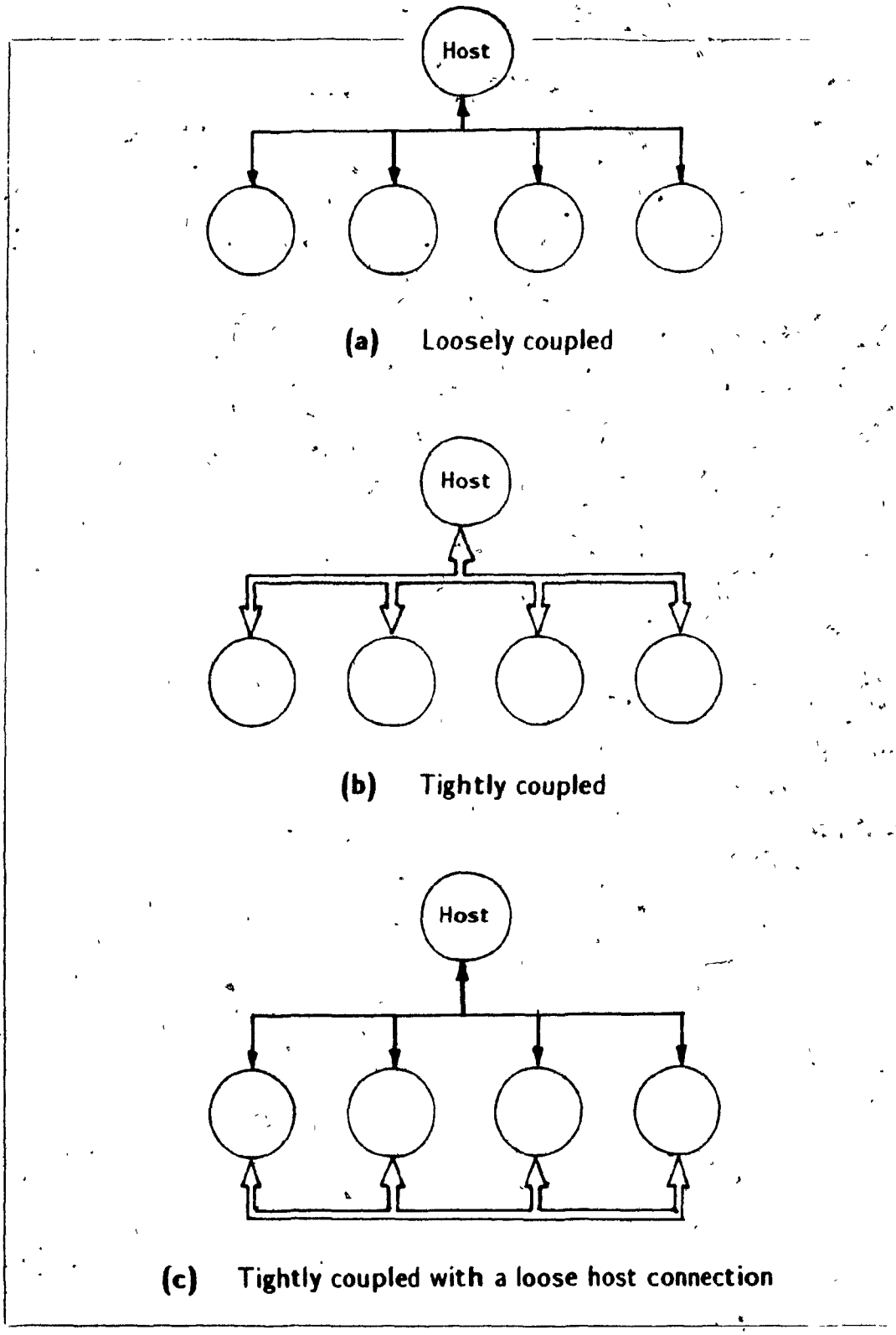


Figure 2.9 Possible processor connection schemes

(2) a richer environment for interprocess communication

2.4.4 Software Architecture

The software considerations in a distributed RCI include

Task Distribution – How shall different tasks be arranged on the various processors?

Task Characteristics – What properties will the control tasks have?

Task Control – How will the tasks be started and stopped?

Communication – What software inter-task communication mechanisms should be provided?

Implementation – What system software is necessary to implement this functionality?

2.4.4.1 Task Description and Control

With the hardware arrangement described above, the planning task would run on the host computer, in the context of a normal program, and invoke control tasks which would execute on the other processors. There could be several planning tasks running on the host, each attachable to control processes (Figure 2.10).

Program development can be done on the host, with the appropriate executable image for each processor being linked there and then downloaded. The mapping of tasks to processors would be done at load time by the programmer, which is reasonable since (1) most tasks will be bound to specific processors anyway (processor 1 is connected to robot 2, processor 2 is connected to the frame grabber, etc.), and (2) only the programmer will know best how to distribute the computational resources. Since these factors will not change much at run time, there is no need to consider complex issues such as the dynamic relocation of tasks.

Each control task will consist of one user-defined control function*. A predefined set of timer and device interrupts can be provided for each processor†, each associated with properties, such as sample rate and priority, which can be modified at run time. Each

* The present RCI task level provides two functions for both fast response and parallelism. However it was observed that function 1 was never used and only served to confuse things.

† As mentioned earlier we do not specifically consider software interrupts here.

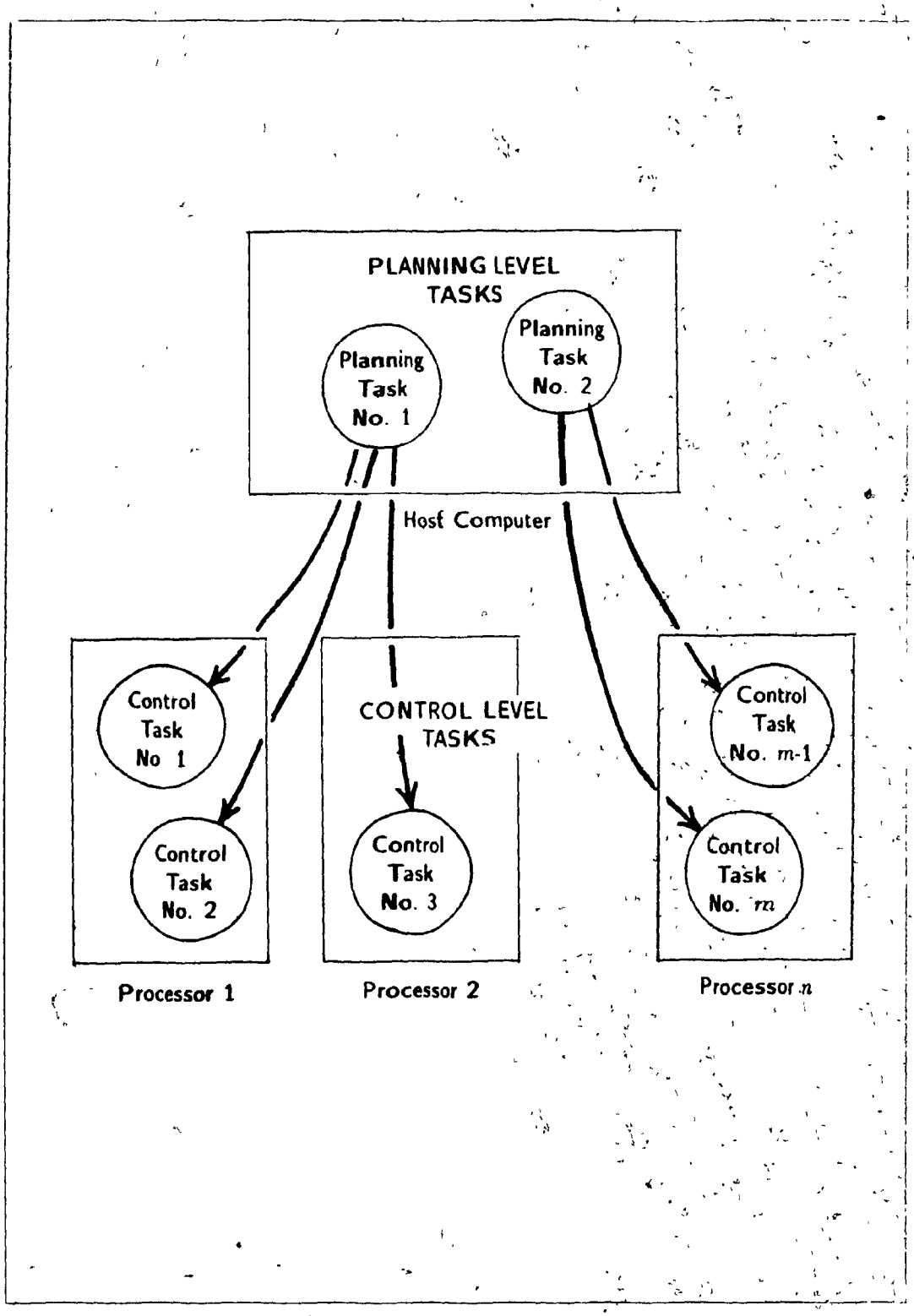


Figure 2.10 Extended RCI task structure

interrupt may be identified by a processor specific *event ID*. some event IDs may have internal pre- and post-processing functions to perform checking. actuator and sensory device I/O. and bookkeeping (as with the present RCI control task. Figure 2.2)

The executable images for each processor may be loaded from the host using a primitive of the form:

```
RCIload (image, processor_id);
```

which could effectively take the place of the existing *RCIopen()* primitive. Once in place. tasks could be controlled using the familiar *control* and *release* primitives:

```
RCIcontrol(event_id, function_name);

RCIrelease(event_id);
```

The argument *event_id* would specify both the processor and the interrupt. An error would be returned if the indicated function were not available in the currently loaded processor image. To simplify the implementation. these primitives would be issued by the planning level only. although error handling primitives would be provided by which the control tasks. could shut down either themselves or the whole control level. if necessary.

2.4.4.2 Communication Software

The most basic way of allowing the different tasks to communicate is to provide them with shared memory. which is possible assuming a parallel bus interconnection as described above. It has been the author's experience that shared memory can be very useful and sometimes necessary. particularly during system development. and this view has been echoed by other authors [Steusloff 84]. The implementation of shared memory across processors requires a single address space and a unified linking utility.

Shared memory places the burden of protocol management on the programmer. This may not always be fair. it is reasonable to also provide a more message oriented communication facility. particularly for packaging code into well defined modules which are to be used by "other people". Also. in the absence of parallel bus hardware. channel based communication primitives would provide the only means of communication. A rudimentary set of these could look like this:

```
send (receiver_id, buffer, size),

num_read = receive (sender_id, buffer, size),
```

The first primitive sends a packet of information described by *buffer* and *size* to the task denoted by *receiver_id*. Conversely, the *receive()* primitive reads in any messages which have arrived from *sender_id*, places them in the indicated buffer, and returns the actual size of the message. If the value of *sender_id* is set to 0, then *receive()* will read whatever packet is the most outstanding, and fill in *sender_id* with the sender's name. These utilities would encompass both planning and control tasks.

This description is preliminary, among the particular problems that would have to be considered in more detail would be (1) the well-known one of how to guarantee to a sender that the destination task is in fact active and received the message, and (2) the desired blocking behavior of the primitives. Since the control tasks operate under fixed time constraints, the primitives described above are assumed to be non-blocking. This also allows *receive()* to be used in conjunction with its return value as a testing primitive. If the implementation of the primitives is quick enough, however, blocking may be acceptable (the turnaround time for a *send-receive-reply* cycle in Harmony is presently about 1 ms [Gentleman 85]). Message passing mechanisms are discussed in detail in [Gentleman 81] and [Shin and Malin 85].

Summarizing the position on shared memory vs. message (channel) oriented communication, in the context of RCI, we state that

The ability to use shared memory should definitely be present

Message oriented communication is preferred when it is not inconvenient

In addition to this message passing scheme it might also be desirable to provide a means for control tasks to issue signals to their parent planning tasks, since in general the planning tasks do not run at a particular sample rate, and hence do not have as much opportunity to poll for events.

We should also mention communication with the external robot and sensory devices which the RCI system is controlling. This could be done either explicitly by the control tasks, which would have access to the physical device registers*, or implicitly by the task

* RCI tasks are assumed to run in the kernel mode associated with the target processor, avoiding memory

pre- and post-processing functions as is done presently in conjunction with the how and chg_structures. Alternatively if it proved necessary we could extend the message passing primitives described above and build I/O device abstractions into RCI.

2.4.4.3 Implementation

An implementation of a multiprocessor RCI could be structured as follows. A small run-time kernel would be loaded into the executable image for each processor. This kernel would contain (1) a primary interrupt handler and dispatch table that would assume the role of scheduler, (2) a background monitor that fielded control and release requests from the host computer and adjusted the interrupt handler accordingly, and (3) code to implement any built-in system services such as message passing. If the message passing was not based on shared memory, then these primitives would also require the existence of I/O drivers and handlers.

2.4.5 The Development Environment

Writing an RCI program under the version presently implemented is similar to writing any C application program where the programmer creates a main function and a collection of sub-functions. The source code can be kept conveniently in one set of files, and maintained by a single set of development tools (rcc, editor, etc.). It is important to preserve this ease of use in cases where the hardware is extended to include multiple processors. The only simple way to do this is to continue to do the software development on a single machine (the host) and have the executable code generated for the individual processors when needed, by cross development programs. The alternative is to maintain separate file systems and development tools on each processor, not only is this unwieldy in the applications we are considering, it is unnecessary.

Presumably it would be advantageous to develop some specialized tools to aid in maintaining software across different targets. Tools based on, or similar to, the UNIX facility make could be useful in this regard. If the system were to make use of shared memory, then some loader tools would have to be provided to enable the mapping of different data regions in the code for each processor onto the appropriate physical memory area. A simulator

---

access difficulties

environment similar in concept to the simulator described in section 2.2.3 which allowed a set of control tasks to be executed in a non real time context on the host would probably also be quite useful

2.5 Summary

We have presented a programming interface for real-time robot control applications which allows a programmer to easily create a periodic *control* task running in parallel with the main program or *planning* task. Communication with the robot is presented to the user through a pair a global data structures which remove the user from specific communication details.

The user writes this twin level program in a manner similar to any other application program using RCI system primitives to start the control level task and bind it to the desired control functions. Error conditions which occur at the control level or on the robot itself are reported to the planning level by means of a signal which may then handle the situation appropriately. A simple simulator is available by which the user may make dry runs of his/her program before actually operating the robot. A special program called *rcc* is available to simplify the compiling and linking of the user's program.

An extension of this system to multiple robots and processors is considered along the following lines. Multiple control tasks are used to operate different robots, process sensory data, and do related jobs. The planning level runs on a general purpose computer which serves as a host to a set of other processers which run the control tasks. Each control task is defined by a user-specified control function driven by an interrupt. The image for each processor containing the code for the tasks that are to be executed on it is compiled and linked on the host and then downloaded. A set of interrupts which are available for driving control tasks is defined by RCI for each processor, these are known as *event ids*. The mapping of tasks onto different processors is done by the user at link time.

Two principal inter-task communication mechanisms are discussed: data channels (slower, hardware independent, more software overhead) and shared data (faster, hardware dependent, less software overhead). Lower bandwidth serial hardware connections are feasible for communication between the control and planning tasks, but parallel hardware is preferred in general, as is provides greater flexibility, as well as the speed required for

control applications At the software level it is argued that (1) the ability to use shared data should be present, but that (2) it is cleaner to use message structured communication when it is not inconvenient

The hardware design includes a set of board level microprocessors, each with local memory (and private sensory and robot device I/O if possible), interfaced to the host computer and connected to each other using a bus topology Control tasks are initiated and released at run time from the planning level using *control* and *release* primitives similar to those in the current version Creating all software on the host using cross development facilities would give the user a simple and uniform environment in which to work and allow access to sophisticated tools such as simulators

Chapter 3 MANIPULATOR KINEMATICS

3.1 Overview

In this chapter we derive the manipulator kinematics for the Unimation PUMA 260 and Microbo Ecureuil robots, described in section 1.3.3.

The kinematic problem involves defining a mapping from the coordinate space defined by the joint variables of the robot into some more convenient coordinate space. Specifically we are interested in transforming joint variables into a Cartesian coordinate system, a task known as the *forward kinematics* problem, and determining the joint variables that correspond to a given Cartesian position (the *inverse kinematics* problem).

The *differential* kinematics of a manipulator are also of great utility in control applications, they represent the relationship between a differential change in joint coordinates $d\Theta$ and a corresponding differential change in Cartesian coordinates expressed by the vector dc. The manipulator Jacobian $\mathbf{J}$ is a $6 \times N$ matrix, N being the number of manipulator joints, defined such that

$$dc = \mathbf{J}\, d\Theta \tag{3.1}$$

If the Cartesian forces are denoted by $\mathbf{q}$, and the joint torques/forces by $\mathbf{t}_j$, then equation (1.19) can be used to relate these two using the transpose of the Jacobian

$$\mathbf{t}_j = \mathbf{J}^T \mathbf{q} \tag{3.2}$$

This chapter is organized as follows: a formalism for describing the position of a robot is reviewed and a coordinate system is defined for both manipulators. From this the forward kinematics may be obtained directly. We then derive the inverse kinematics. A procedure is discussed for obtaining the manipulator Jacobian, whose form is greatly simplified by expressing it relative to the Cartesian frame located in link 4 of the manipulator, instead of link 6 [Renaud 81]. The result is simple enough to allow symbolic inversion. Lastly, we obtain an efficient form for the Jacobian computations (3.1), (3.2), and their inverses, by expanding and then collapsing them, making use of recurrences in the solution.

A general introduction to this material is given in [Paul 81]. Kinematic calculations for the PUMA 600 robot are given in [Paul, et al. 81A] and [Bazerghi, et al. 84]. Differential kinematics are specifically treated in [Renaud 81] and [Paul, et al. 81B].

Some of the derivations presented in this chapter were obtained or verified with the use of MACSYMA, a computer algebra program distributed by Symbolics, Inc., Cambridge, Mass.

3.2 The Robot Coordinate System

The coordinate system for a robot is usually based on the conventions established by Denavit and Hartenberg for serial link manipulators [Denavit and Hartenberg 55], in which an orthonormal right-handed coordinate frame is associated with the base of the manipulator, and then with each of the individual joints. The transformation between frames is provided by a homogeneous transform known as an **A** matrix. There is one of these matrices defined for each link of the manipulator: the first matrix defines a transformation from the base coordinate frame to the frame in link 1, the second matrix defines a transformation from link 1 to link 2, and so on. Each **A** matrix is a function of 4 parameters, of which 3 are fixed and the other is the joint variable.

3.2.1 Link Coordinate Frames

An algorithm for establishing the coordinate frames in each link of an N link robot arm, according to the Denavit–Hartenberg nomenclature is given in [Lee 82]. For completeness, we quote the algorithm here. It should be noted that the coordinate system obtained by

this method is not unique. The unit vectors along the x_i, y_i, and z_i axes are denoted by $\mathbf{x}_i$, $\mathbf{y}_i$, and $\mathbf{z}_i$, respectively.

1. **Establish the base coordinate system.** Establish a right-handed orthonormal coordinate system (x_0, y_0, z_0) at the supporting base with the z_0 axis lying along the axis of joint 1.
2. **Initialize and loop.** For each i, $i = 1, \ldots, N$, perform steps 3 to 6.
3. **Establish the joint axis.** Align z_i with the axis of motion (rotary or prismatic) of joint $i + 1$.
4. **Establish the origin of the coordinate system.** Locate the origin of the i-th coordinate system at the intersection of the z_i and z_{i-1} axes or at an intersection of the common normal(s) between the z_i and z_{i-1} axes and the z_i axis.
5. **Establish the X axis.** Assign $\mathbf{x}_i = \pm(\mathbf{z}_{i-1} \times \mathbf{z}_i)/\|(\mathbf{z}_{i-1} \times \mathbf{z}_i)\|$ or along a common normal between the z_{i-1} and z_i axes when they are parallel.
6. **Establish the Y axis.** Assign $\mathbf{y}_i = \mathbf{z}_i \times \mathbf{x}_i$ to complete the right-hand coordinate system.
7. **Find the joint and link parameters.** For each i, $i = 1, \ldots, N$, perform steps 8 to 11.
8. **Find d_i.** This is the distance from the origin of the $(i-1)$-th coordinate frame to the intersection of the z_{i-1} axis and x_i, measured along z_{i-1}. It is the joint variable in the case of prismatic joints.
9. **Find a_i.** The distance from the intersection of the z_{i-1} axis and x_i to the origin of the i-th coordinate system, measured along the x_i axis.
10. **Find θ_i.** The angle of rotation from the x_{i-1} axis to x_i, measured counterclockwise about z_{i-1}. It is the joint variable for rotary joints.
11. **Find α_i.** The angle of rotation from the z_{i-1} axis to z_i, measured counterclockwise about x_i.

This algorithm is quite general and applies to any serial link manipulator. Figure 3.1 illustrates its application to two links with rotary joints.

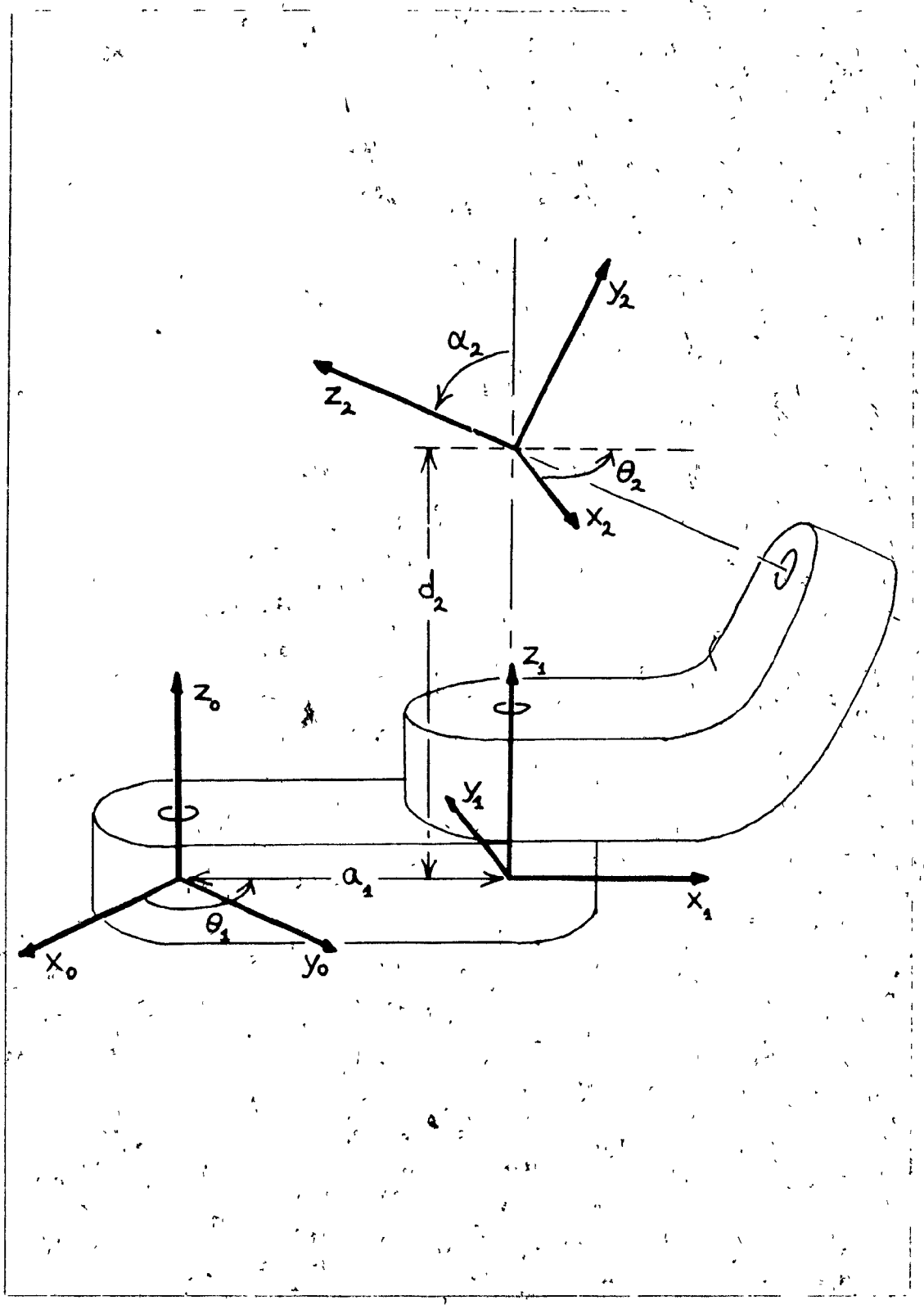


Figure 3.1 A Denavit Hartenberg representation for two links with rotary joints

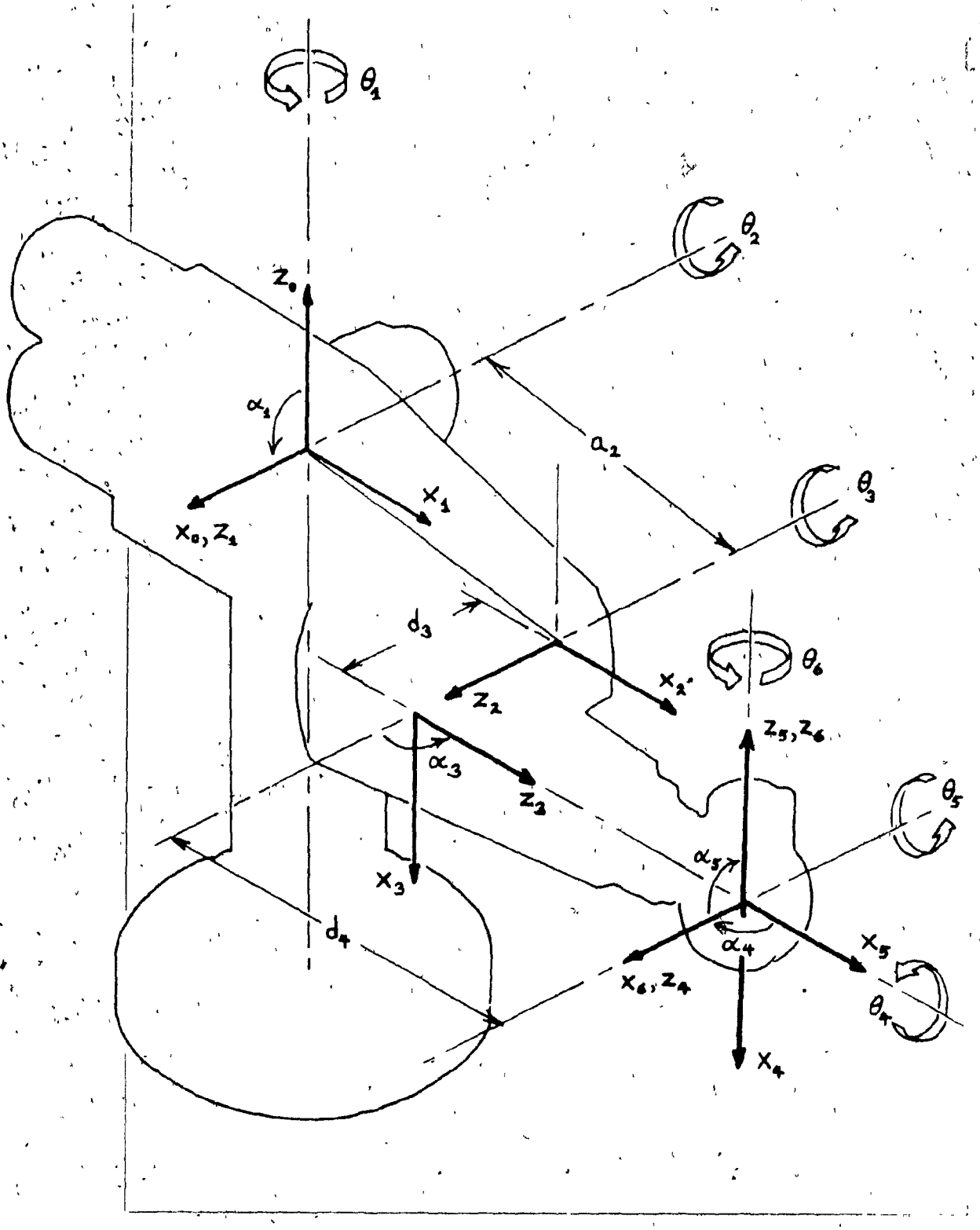


Figure 3.2 Coordinate system for the PUMA 260

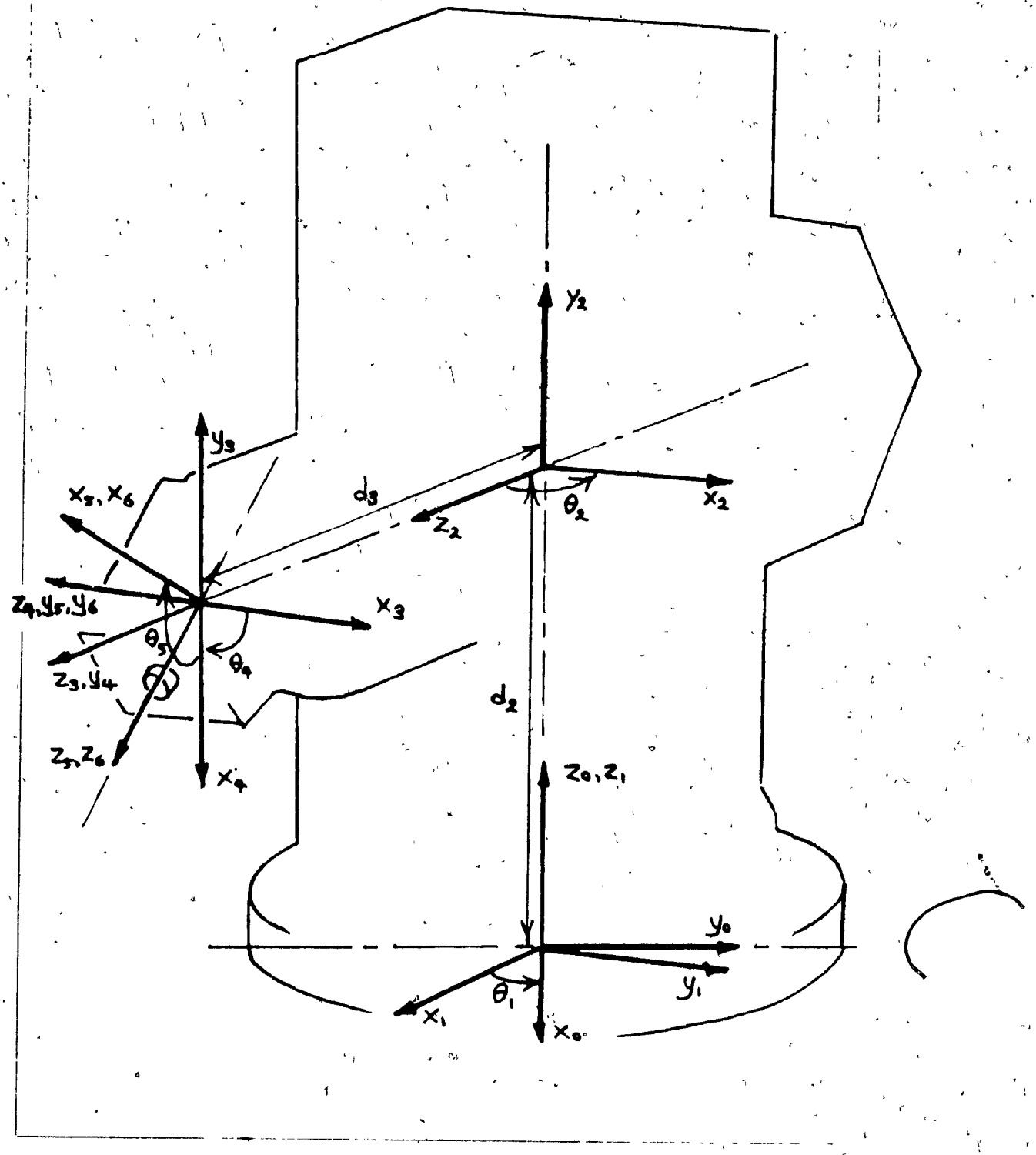


Figure 3.3 Coordinate system for the Micrōbo

Joint i	d_i	θ_i	α_i	a_i
1	0	θ_1	90	0
2	0	θ_2	0	203.20 = .38 m
3	126.24 = .50	θ_3	-90	0
4	203.20 = .64	θ_4	90	0
5	0	θ_5	-90	0
6	0	θ_6	0	0

Table 3.1 Coordinate parameters for the Puma 260 (Distances are in mm.)

Joint i	d_i	θ_i	α_i	a_i
1	0	θ_1	0	0
2	d_2	90°	90°	0
3	d_3	0	0	0
4	0	θ_4	90°	0
5	0	θ_5	90°	0
6	0	θ_6	0	0

Table 3.2 Coordinate parameters for the Microbo

Applying this algorithm to the PUMA 260 (Figure 1.5), we obtained the coordinate system shown in Figure 3.2. Applying the algorithm to the Microbo (Figure 1.6) yielded the coordinate system shown in Figure 3.3. The link parameters are given for the PUMA in Table 3.1 and for the Microbo in Table 3.2.

3.2.2 The A Matrices

The matrices $\mathbf{A}_i$ describe the transformation from the frame in joint $i-1$ to the frame in joint i. These matrices may be obtained in a straightforward manner from the general

relationship

$$\mathbf{A}_i = \begin{pmatrix} \cos\theta & \sin\theta\cos\alpha & \sin\theta\sin\alpha & a_i\cos\theta \\ \sin\theta & \cos\theta\cos\alpha & \cos\theta\sin\alpha & a_i\sin\theta \\ 0 & \sin\alpha & \cos\alpha & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

where d_i, a_i, θ_i and α_i are the joint parameters

Substituting the appropriate values from Tables 3.1 and 3.2, we arrive at the following sets of **A** matrices for the PUMA 260 and Microbo (A notation is used throughout the rest of this chapter where $S_i = \sin\theta_i$, $C_i = \cos\theta_i$, $S_{ij} = \sin\theta_i + \sin\theta_j$, and $C_{ij} = \cos\theta_i + \cos\theta_j$.)

A matrices for the PUMA 260

$$\mathbf{A}_1 = \begin{pmatrix} C_1 & 0 & S_1 & 0 \\ S_1 & 0 & -C_1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.4}$$

$$\mathbf{A}_2 = \begin{pmatrix} C_2 & -S_2 & 0 & a_2C_2 \\ S_2 & C_2 & 0 & a_2S_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.5}$$

$$\mathbf{A}_3 = \begin{pmatrix} C_3 & 0 & -S_3 & 0 \\ S_3 & 0 & C_3 & 0 \\ 0 & -1 & 0 & d_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.6}$$

$$\mathbf{A}_4 = \begin{pmatrix} C_4 & 0 & S_4 & 0 \\ S_4 & 0 & -C_4 & 0 \\ 0 & 1 & 0 & d_4 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

$$\mathbf{A}_5 = \begin{pmatrix} C_5 & 0 & -S_5 & 0 \\ S_5 & 0 & C_5 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

$$\mathbf{A}_6 = \begin{pmatrix} C_6 & -S_6 & 0 & 0 \\ S_6 & C_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.9}$$

A matrices for the Microbo

$$\mathbf{A}_1 = \begin{pmatrix} C_1 & -S_1 & 0 & 0 \\ S_1 & C_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.10}$$

$$\mathbf{A}_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & d_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.11}$$

$$\mathbf{A}_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.12}$$

$$\mathbf{A}_4 = \begin{pmatrix} C_4 & 0 & S_4 & 0 \\ S_4 & 0 & -C_4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.13}$$

$$\mathbf{A}_5 = \begin{pmatrix} C_5 & 0 & S_5 & 0 \\ S_5 & 0 & -C_5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.14}$$

$$\mathbf{A}_6 = \begin{pmatrix} C_6 & -S_6 & 0 & 0 \\ S_6 & C_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.15}$$

3.3 Forward Kinematics

If we multiply together the $\mathbf{A}$ matrices, we obtain the transform $\mathbf{T}_6$ that relates the base of the manipulator to the position of the end link.

$$\mathbf{T}_6 = \mathbf{A}_1 \mathbf{A}_2 \mathbf{A}_3 \mathbf{A}_4 \mathbf{A}_5 \mathbf{A}_6 \tag{3.16}$$

This allows us to determine the position/orientation of the end of the robot given the values of the joint variables. We can now multiply the **A** matrices together to determine T_6 for both robots, using the notation for homogeneous transforms described in equation (1.3)

Forward kinematics for the PUMA 260

$$T_6 = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.17}$$

where

$$n_x = C_1(-C_{23}S_4S_6 - C_6S_{23}S_5 + C_{23}C_4C_5C_6) + S_1(-C_4S_6 - C_5C_6S_4)$$

$$n_y = S_1(-C_{23}S_4S_6 - C_6S_{23}S_5 + C_{23}C_4C_5C_6) - C_1(-C_4S_6 - C_5C_6S_4)$$

$$n_z = -S_{23}S_4S_6 + C_{23}C_6S_5 + C_4C_5C_6S_{23}$$

$$o_x = C_1[(S_{23}S_5 - C_{23}C_4C_5)S_6 - C_{23}C_6S_4] + S_1(C_5S_4S_6 - C_4C_6)$$

$$o_y = S_1[(S_{23}S_5 - C_{23}C_4C_5)S_6 - C_{23}C_6S_4] - C_1(C_5S_4S_6 - C_4C_6)$$

$$o_z = -(C_{23}S_5 + C_4C_5S_{23})S_6 - C_6S_{23}S_4$$

$$a_x = -C_1(C_{23}C_4S_5 + C_5S_{23}) + S_1S_4S_5$$

$$a_y = -S_1(C_{23}C_4S_5 + C_5S_{23}) - C_1S_4S_5$$

$$a_z = C_{23}C_5 - C_4S_{23}S_5$$

$$p_x = C_1(a_2C_2 - d_4S_{23}) + d_3S_1$$

$$p_y = S_1(a_2C_2 - d_4S_{23}) - C_1d_3$$

$$p_z = a_2S_2 + C_{23}d_4$$

Forward kinematics for the Microbo

$$T_6 = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.18}$$

where

$$n_x = C_1C_6S_5 \quad S_1(S_4S_6 + C_4C_5C_6)$$
$$n_y = C_1(S_4S_6 + C_4C_5C_6) - C_6S_1S_5$$
$$n_z = C_5C_6S_4 \quad C_4S_6$$
$$o_x = -S_1(C_6S_4 - C_4C_5S_6) - C_1S_5S_6$$
$$o_y = C_1(C_6S_4 - C_4C_5S_6) - S_1S_5S_6$$
$$o_z = -C_5S_4S_6 - C_4C_6$$
$$a_x = -C_4S_1S_5 - C_1C_5$$
$$a_y = C_1C_4S_5 - C_5S_1$$
$$a_z = S_4S_5$$
$$p_x = C_1d_3$$
$$p_y = d_3S_1$$
$$p_z = d_2$$

When actually performing these calculations, it should be remembered that the information contained in n, o, and a is redundant. Since they define a right-handed orthogonal matrix, each of these vectors is equal to the cross product of the other two. This can be used to simplify the computations by explicitly computing only two of the vectors and from them determining the third.

3.4 Inverse Kinematics

It is desired to find the solution of (3.17) and (3.18), that is to ask, given a certain transformation matrix defining the desired position of the end effector, what are the corresponding joint variables? The solutions obtained here are based on solutions discussed in [Paul 81] and [Paul, et al. 81A].

Equating the individual matrix elements in equation (3.16) gives a set of 12 simultaneous equations which can be used in obtaining a solution. Not all of these equations however, are in a simple enough form to be very helpful. We can obtain more simultaneous equations by successively post multiplying (3.16) by the matrices $\mathbf{A}_1^{-1}$ through $\mathbf{A}_N^{-1}$.

where N is the number of manipulator joints. This yields a series of $N - 1$ matrix equations of the form

$$\mathbf{A}_1^{-1} \cdots \mathbf{A}_1^{-1} \mathbf{T}_6 = \mathbf{U}_i \tag{3.19}$$

where

$$\mathbf{U}_i = \mathbf{A}_i \cdots \mathbf{A}_6, \quad 1 < i < N$$

For $N = 6$ this gives 60 additional equations. Elements on the right of (3.19) will be functions of the variables of joints 1 to 6. Elements on the left hand side will be in terms of $\mathbf{T}_6$ and the variables of joints 1 through $i - 1$. We shall proceed by examining equations with successively higher values of i, solving for each joint variable in terms of fixed parameters and previously solved joint variables.

Inverse kinematics for the PUMA 260

The PUMA is definitely the more difficult manipulator in terms of solving the kinematics. In doing so, we shall make use of the **atan2** form of the inverse tangent function. This takes two arguments, a y and an x value, which are proportional to $\sin\theta$ and $\cos\theta$ respectively and returns θ in the range $-180 < \theta \leq +180$. The only time this function is undefined is when both arguments are zero. In particular, this function can be used to solve equations of the form

$$aS_i + bC_i = c \tag{3.20}$$

for which there is the general solution ([Paul, et al 81A])

$$\theta_i = \text{atan2}(b, a) + \text{atan2}(c, \pm\sqrt{a^2 + b^2 - c^2}) \tag{3.21}$$

We begin by considering equation (3.19) with $i = 2$,

$$\mathbf{A}_1^{-1} \mathbf{T}_6 = \mathbf{U}_2 \tag{3.22}$$

Expanding, and equating the fourth column on each side yields

$$\begin{pmatrix} C_1 p_x + S_1 p_y \\ p_z \\ S_1 p_x - C_1 p_y \\ 0 \end{pmatrix} = \begin{pmatrix} a_2 C_2 - d_4 S_{23} \\ a_2 S_2 + d_4 C_{23} \\ d_3 \\ 0 \end{pmatrix} \tag{3.23}$$

The solution for θ_1 comes from examining the bottom row. Using (3.21) we get

$$\theta_1 = \text{atan2}(p_y, p_x) + \text{atan2}(d_3, \pm\sqrt{r^2 - d_3^2}) \tag{3.24}$$

where

$$r^2 = p_x^2 + p_y^2$$

The $\pm$ in front of the square root corresponds to whether the robot is in a *right-handed* or *left-handed* configuration (Figure 3.4(a)).

Equation (3.23) also yields a solution for θ_3. Squaring and summing the top two rows gives

$$d_4^2 + a_2^2 - f_1^2 - p_z^2 = 2a_2 d_4 S_3 \tag{3.25}$$

where

$$f_1 = C_1 p_x + S_1 p_y$$

This is also equivalent in form to (3.20) in the case where $b = 0$. Substituting into (3.21) and making use of the identity

$$\text{atan2}(a, b) = \frac{\pi}{2} - \text{atan2}(b, a) \tag{3.26}$$

gives

$$\theta_3 = \frac{\pi}{2} - \text{atan2}(\pm\sqrt{f^2 - d^2}, d) \tag{3.27}$$

where

$$f = 2a_2 d_4$$
$$d = d_4^2 + a_2^2 - f_1^2 - p_z^2$$

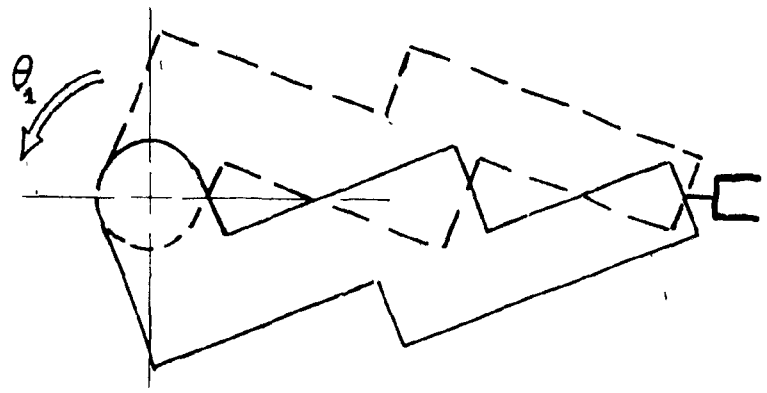
The $\pm$ in front of the square root corresponds to whether the robot is in an *elbow-up* or *elbow-down* configuration (Figure 3.4(b)).

The next angle solved for is $\theta_2 + \theta_3$. This is obtained by considering (3.19) with $i = 4$

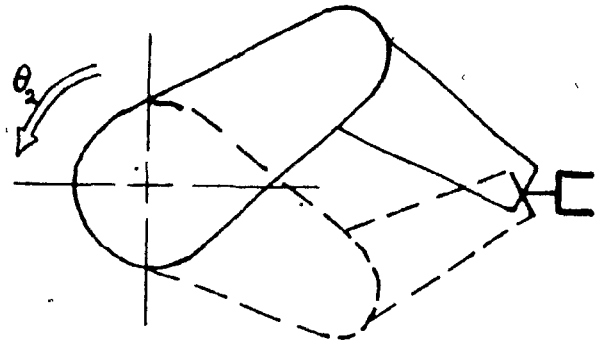
$$\mathbf{A}_3^{-1}\mathbf{A}_2^{-1}\mathbf{A}_1^{-1}\mathbf{T}_6 = \mathbf{U}_4 \tag{3.28}$$

Equating the fourth column on each side gives

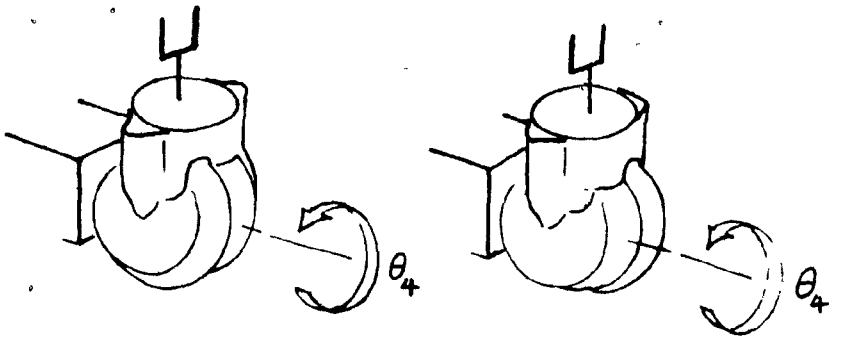
$$\begin{pmatrix} S_{23}p_z + C_{23}(S_1 p_y + C_1 p_x) - a_2 C_3 \\ C_1 p_y - S_1 p_x + d_3 \\ -S_{23}(S_1 p_y + C_1 p_x) + C_{23} p_z + a_2 S_3 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ d_4 \\ 0 \end{pmatrix} \tag{3.29}$$



(a) Right-handed / Left-handed



(b) Elbow-up / Elbow-down



(c) Wrist-flip / Wrist noflip

Figure 3.4 Redundancies on the PUMA 260

The top and bottom rows may be solved simultaneously to yield

$$\theta_2 + \theta_3 = \text{atan2}(p_z u_1 \quad f_1 u_2 \quad p_z w_2 + f_1 w_1) \tag{3.30}$$

where

$$w_1 = a_2 C_3 ,$$

$$w_2 = d_4 \quad a_2 S_3$$

If we next equate the third columns of (3.28) we have

$$\begin{pmatrix} S_{23} a_z + C_{23}(S_1 a_y + C_1 a_x) \\ C_1 a_y \quad S_1 a_x \\ S_{23}(S_1 a_y + C_1 a_x) + C_{23} a_z \\ 0 \end{pmatrix} = \begin{pmatrix} C_4 S_5 \\ S_4 S_5 \\ C_5 \\ 0 \end{pmatrix} \tag{3.31}$$

The first and second rows may be substituted into the **atan2** function directly, yielding a solution for θ_4. It must be noted that the S_5 term does not strictly divide out, i.e. atan2$(a, b) = \pi +$ atan2$(-a, -b)$. Hence if S_5 is negative this will result in a 180° phase shift in the solution. This means there are two possible solutions, one with S_5 negative and one with S_5 positive, corresponding to whether the robot is in a *wrist-flip* or *wrist noflip* configuration (Figure 3.4(c)). The solution shall also become undefined when both arguments to **atan2** approach zero. This will occur if $S_5 = 0$, which happens when $\theta_5 = \pm n\pi, \quad n = 0, 1, \ldots$. When this occurs, joints 4 and 6 are in alignment and the robot is at a singularity. The physical range of joint 5 reduces the number of singularities to one, at $\theta_5 = 0$.

In solving for joint 5, we continue to examine the equations defined by (3.19) for successive i. We obtain a good pair of equations with $i = 5$.

$$\mathbf{A}_4^{-1} \mathbf{A}_3^{-1} \mathbf{A}_2^{-1} \mathbf{A}_1^{-1} \mathbf{T}_6 = \mathbf{U}_5 \tag{3.32}$$

Looking at the third column of each side, and letting

$$f_{a1} = a_y C_1 \quad a_x S_1 ,$$

$$f_{a2} = a_y S_1 + a_x C_1$$

we get

$$\begin{pmatrix} f_{a1} S_4 + C_4(a_z S_{23} + f_{a2} C_{23}) \\ f_{a2} S_{23} + a_z C_{23} \\ (a_z S_{23} + f_{a2} C_{23}) S_4 \quad C_4 f_{a1} \\ 0 \end{pmatrix} = \begin{pmatrix} S_5 \\ C_5 \\ 0 \\ 0 \end{pmatrix} \tag{3.33}$$

Substituting the first and second rows into the **atan2** function yields a solution for θ_5.

Lastly, we look for a solution for joint 6 to do this we consider (3.19) with $i = 6$

$$\mathbf{A}_5^{-1}\mathbf{A}_4^{-1}\mathbf{A}_3^{-1}\mathbf{A}_2^{-1}\mathbf{A}_1^{-1}\mathbf{T}_6 = \mathbf{U}_6 \tag{3.34}$$

Letting

$$f_{a1} = o_y C_1 - o_x S_1,$$
$$f_{a2} = o_y S_1 + o_x C_1$$

and equating the second column on each side, we find

$$\begin{pmatrix} S_5(-f_{a2}S_{23} + o_z C_{23}) + C_5[f_{a1}S_4 + C_4(o_z S_{23} + f_{a2}C_{23})] \\ C_4 f_{a1} - S_4(o_z S_{23} + f_{a2}C_{23}) \\ C_5(-f_{a2}S_{23} + o_z C_{23}) - S_5[f_{a1}S_4 + C_4(o_z S_{23} + f_{a2}C_{23})] \\ 0 \end{pmatrix} = \begin{pmatrix} S_6 \\ C_6 \\ 0 \\ 0 \end{pmatrix} \tag{3.35}$$

which also yields a solution for θ_6 by direct substitution into **atan2**.

We now present the complete inverse kinematic solution for the PUMA 260

$$\theta_1 = \text{atan2}(p_y, p_x) + \text{atan2}(\text{d}_3, \pm\sqrt{r^2 - \text{d}_3^2}) \quad \text{with} \tag{3.36}$$
$$r^2 = p_x^2 + p_y^2$$

$$\theta_3 = \frac{\pi}{2} - \text{atan2}(\pm\sqrt{f^2 + \text{d}^2}, \text{d}), \quad \text{with} \tag{3.37}$$
$$f = 2\text{a}_2\text{d}_4,$$
$$\text{d} = \text{d}_4^2 + \text{a}_2^2 - f_p^2 - p_z^2$$
$$f_p = C_1 p_x + S_1 p_y$$

$$\theta_2 = \text{atan2}(p_z w_1 - f_p w_2, p_z w_2 + f_p w_1) - \theta_3, \quad \text{with} \tag{3.38}$$
$$w_1 = \text{a}_2 C_3,$$
$$w_2 = \text{d}_4 - \text{a}_2 S_3$$

$$\theta_4 = \text{atan2}(f_{a1}, S_{23}a_z + C_{23}f_{a2}) + \pi, \quad \pi \le \theta_5 \le 0 \tag{3.39a}$$
$$\theta_4 = \text{atan2}(f_{a1}, S_{23}a_z + C_{23}f_{a2}), \quad -\pi \le \theta_5 \le 0 \tag{3.39b}$$

$$f_{a1} = a_y C_1 - a_x S_1$$
$$f_{a2} = a_y S_1 + a_x C_1$$

$$\theta_5 = \text{atan2}(-f_{a1} S_4 \quad C_4(a_z S_{23} + f_{a2} C_{23}), -f_{a2} S_{23} + a_z C_{23}) \tag{3.40}$$

$$\theta_6 = \text{atan2}(S_5(f_{o2} S_{23} \quad o_z C_{23}) \quad C_5[f_{o1} S_4 + C_4(o_z S_{23} + f_{o2} C_{23})],$$
$$C_4 f_{o1} \quad S_4(o_z S_{23} + f_{o2} C_{23})) \quad \text{with} \tag{3.41}$$
$$f_{o1} = o_y C_1 - o_x S_1,$$
$$f_{o2} = o_y S_1 + o_x C_1$$

The three discrete redundancies (*right/left-handed, elbow-up/down, wrist-flip/noflip*) must be specified to define a unique solution

Inverse kinematics for the Microbo

The solution for the first three Microbo joint variables is very simple: since the robot is a cylindrical type, these variables define a cylindrical coordinate system. Examining the rightmost column of equation (3.18), we obtain the following

$$p_x = C_1 d_3$$
$$p_y = S_1 d_3 \tag{3.42}$$
$$p_z = d_2$$

From the first and second equations we get

$$\theta_1 = \text{atan2}\,(p_y, p_x) \tag{3.43}$$

The third joint variable is obtained by squaring and adding the first two equations

$$d_3 = \sqrt{p_x^2 + p_y^2} \tag{3.44}$$

Although this in principle yields both a positive and negative solution for d_3, there is no practical way on this robot for d_3 to assume a negative value (this in effect precludes the Microbo from having both *left-handed* and *right-handed* solutions)

The second joint variable is seen to be equal to p_z

The solution method of the last three joint angles is exactly analogous to the solution method of the last three angles for the PUMA considering the equation defined by (3.28) and examining the third column on either side, yields

$$\begin{pmatrix} a_y S_1 + a_x C_1 \\ a_y C_1 \quad a_x S_1 \\ a_z \\ 0 \end{pmatrix} = \begin{pmatrix} C_5 \\ C_4 S_5 \\ S_4 S_5 \\ 0 \end{pmatrix} \tag{3.45}$$

The second and third rows may be substituted into the **atan2** function to yield θ_4, with a 180 phase shift when S_5 is negative and no solution when $S_5 = 0$. Due to the physical range of joint 5, there is only one singularity, located at $\theta_5 = \pi$

The solution for joint 5 is found by examining the third columns of the equation defined by (3.32)

$$\begin{pmatrix} a_z S_4 + C_4(a_y C_1 \quad a_x S_1) \\ a_y S_1 + a_x C_1 \\ (a_y C_1 \quad a_x S_1) S_4 \quad a_z C_4 \\ 0 \end{pmatrix} = \begin{pmatrix} S_5 \\ C_5 \\ 0 \\ 0 \end{pmatrix} \tag{3.46}$$

The first and second rows yield an **atan2** solution for θ_5

Finally for joint 6 we look at the second column on each side of the equation defined by (3.34)

$$\begin{pmatrix} (o_y S_1 + C_1 o_x) S_5 + C_5[o_z S_4 + C_4(C_1 o_y \quad o_x S_1)] \\ (C_1 o_y \quad o_x S_1) S_4 - C_4 o_z \\ [o_z S_4 + C_4(C_1 o_y \quad o_x S_1)] S_5 \quad C_5(o_y S_1 + C_1 o_x) \\ 0 \end{pmatrix} = \begin{pmatrix} S_6 \\ C_6 \\ 0 \\ 0 \end{pmatrix} \tag{3.47}$$

which gives an **atan2** relationship for θ_6

We now summarize the inverse kinematics for the Microbo

$$\theta_1 = \text{atan2}(p_y, p_x) \tag{3.48}$$

$$d_2 = p_z \tag{3.49}$$

$$d_3 = \sqrt{p_x^2 + p_y^2} \tag{3.50}$$

$$\theta_4 = \text{atan2}(a_z \quad a_y C_1 \quad a_x S_1), \qquad \pi \quad \theta_5 \quad 0 \qquad (3.51\,a)$$

$$= \text{atan2}(a_z \quad a_y C_1 \quad a_x S_1) + \pi, \qquad \pi \quad \theta_5 \quad 0 \qquad (3.51\,b)$$

$$\theta_5 = \text{atan2}(a_z S_4 + C_4(a_y C_1 \quad a_x S_1) \quad (a_y S_1 + a_x C_1)) \qquad (3.52)$$

$$\theta_6 = \text{atan2}(\quad ((o_y S_1 + C_1 o_x) S_5 + C_5[o_z S_4 + C_4(C_1 o_y \quad o_x S_1)]),$$
$$(C_1 o_y \quad o_x S_1) S_4 \quad C_4 o_z) \qquad (3.53)$$

There is one discrete redundancy (*wrist-flip/noflip*) which must be specified to allow a unique solution

3.5 The Jacobian

The manipulator Jacobian relates differential changes in joint coordinates to differential changes in the Cartesian frame located in the last link (*i.e.* link 6) of the robot. The form of the Jacobian may be considerably simplified if this relationship is represented with respect to a coordinate frame located in a more central robot link [Renaud 81]. In particular, we choose link 4, situated at the beginning of the wrist. If the Jacobian is described with respect to link 4 (indicated by ${}^4\mathbf{J}$), we then can use the transformations discussed in sections 1.4.2 and 1.4.4 to map the results into link 6. For clarity the meaning of ${}^k\mathbf{J}$ is reiterated. ${}^k\mathbf{J}$ relates a differential change in joint coordinates to a differential change in the Cartesian frame located in the last link (link 6) of the robot with this relationship being described with respect to the frame in link k. Using the mapping $\mathbf{D}_{Q,V}$ this can be expressed as

$${}^k\mathbf{J}\,d\Theta = \mathbf{D}_{6,k}({}^6\mathbf{J}\,d\Theta) \qquad (3.54)$$

To determine the Jacobian ${}^k\mathbf{J}$ with respect to link k, we begin by noting that a differential change in Cartesian coordinates is due to contributions from all the differential changes incurred at each joint i. This can be expressed by the equation

$${}^k\mathbf{J}\,d\Theta = \sum_{i=1}^{N} {}^k\mathbf{da}_i \qquad (3.55)$$

where ${}^k\mathbf{da}_i$ is a vector representing the differential change (section 1.4.2) due to joint i, as observed in link k

$d\mathbf{a}_i$ will be an infinitesimal motion either along or about the z axis of the preceding link $i-1$* depending on whether the joint is prismatic or revolute. The value of $d\mathbf{a}_i$ as observed in the coordinate frame of link $i-1$ hence takes the form

$$ {}^{i-1}d\mathbf{a}_i = \mathbf{c}_i \, d\theta_i \tag{3.56} $$

where the differential change has been factored into a vector $\mathbf{c}_i$ denoting a unit change along or about the z axis of link $i-1$, and a differential change $d\theta_i$ in the joint variable. $\mathbf{c}_i$ is given by $\mathbf{c}_i = (0,0,0,0,0,1)^T$ for a revolute joint, and $\mathbf{c}_i = (0,0,1,0,0,0)^T$ for a prismatic joint.

We can now combine (3.55) and (3.56) in conjunction with the transformation $\mathbf{D}_{i-1,k}$ which maps the observed value of a differential change vector from the frame in link $i-1$ to the frame in link k.

$$ {}^{k}\mathbf{J}\, d\Theta = \sum_{i=1}^{N} \mathbf{D}_{i-1,k} (\mathbf{c}_i \, d\theta_i) \tag{3.57} $$

This can be broken up into a set of equations defining each column ${}^{k}\mathbf{j}_i$ of the Jacobian.

$$ {}^{k}\mathbf{j}_i \, d\theta_i = \mathbf{D}_{i-1,k} (\mathbf{c}_i \, d\theta_i) \tag{3.58} $$

and the $d\theta_i$ may then be factored from each side to yield

$$ {}^{k}\mathbf{j}_i = \mathbf{D}_{i-1,k} (\mathbf{c}_i) \tag{3.59} $$

If we substitute the values of $\mathbf{c}_i$ appropriate for either a revolute or prismatic joint into equation (1.13), and the coordinate transformation from link $i-1$ to link k is given by the matrix $\mathbf{X}_i$ with component columns $\mathbf{n}_i$, $\mathbf{o}_i$, $\mathbf{a}_i$, and $\mathbf{p}_i$, then the following expressions for the Jacobian columns are obtained.

$$ {}^{k}\mathbf{j}_i = \begin{pmatrix} (\mathbf{p}_i \times \mathbf{n}_i)_z \\ (\mathbf{p}_i \times \mathbf{o}_i)_z \\ (\mathbf{p}_i \times \mathbf{a}_i)_z \\ n_{z,i} \\ o_{z,i} \\ a_{z,i} \end{pmatrix} \text{(rotary joint)} \tag{3.60} $$

* The axis of motion for a link is the z axis of the preceding link from the algorithm in section 3.2.1 step 3.

$$ {}^4\mathbf{j}_i = \begin{pmatrix} n_z \\ o_z \\ a_z \\ 0 \\ 0 \\ 0 \end{pmatrix} \text{(prismatic joint)} \tag{3.61} $$

These match the expressions presented in [Paul 81] The $\mathbf{X}_i$ can be determined by multiplying the appropriate $\mathbf{A}$ matrices, or their inverses

PUMA 260 Jacobian

In addition to deriving the Jacobian of the PUMA with respect to link 4, we may obtain an additional simplification by combining rows 2 and 3 We thus define a modified Jacobian in link 4, ${}^4\mathbf{J}'$, such that

$$ \begin{pmatrix} dx \\ dy \\ dz \\ d\upsilon \\ d\rho \\ d\phi \end{pmatrix} = {}^4\mathbf{J}' \begin{pmatrix} d\theta_1 \\ d\theta_2 \\ d\theta_2 + d\theta_3 \\ d\theta_4 \\ d\theta_5 \\ d\theta_6 \end{pmatrix} \tag{3.62} $$

This is equivalent to saying that

$$ {}^4\mathbf{J} = {}^4\mathbf{J}' \mathbf{Y} \tag{3.63} $$

where

$$ \mathbf{Y} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

We can determine the corresponding force transformation equation from (1 19)

$$ \mathbf{t}_j = ({}^4\mathbf{J}' \mathbf{Y})^T \mathbf{q} \tag{3.64} $$

$$ \mathbf{Y}^{1T} \mathbf{t}_j = {}^4\mathbf{J}'^T \mathbf{q} \tag{3.65} $$

from which we get

$$ \begin{pmatrix} \tau_1 \\ \tau_2 - \tau_3 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \end{pmatrix} = {}^4\mathbf{J}'^T \begin{pmatrix} f_x \\ f_y \\ f_z \\ \tau_x \\ \tau_y \\ \tau_z \end{pmatrix} \tag{3.66} $$

Applying equation (3.60), and combining the second and third rows yields the manipulator Jacobian

$$ {}^4\mathbf{J}' = \begin{pmatrix} (C_2a_2 - S_{23}d_4)S_4 + C_{23}C_4d_3 & a_2C_4S_3 & -C_4d_4 & 0 & 0 & 0 \\ d_3S_{23} & a_2C_3 & 0 & 0 & 0 & 0 \\ C_{23}d_3S_4 + C_4(C_2a_2 - S_{23}d_4) & a_2S_3S_4 & -d_4S_4 & 0 & 0 & 0 \\ C_4S_{23} & 0 & S_4 & 0 & 0 & S_5 \\ C_{23} & 0 & 0 & 1 & 0 & C_5 \\ S_{23}S_4 & 0 & C_4 & 0 & 1 & 0 \end{pmatrix} \tag{3.67} $$

The inverse Jacobian may then be determined, with the aid of MACSYMA

$$ {}^4\mathbf{J}'^{-1} = \begin{pmatrix} \frac{S_4}{h} & 0 & \frac{C_4}{h} & 0 & 0 & 0 \\ \frac{d_3S_{23}S_4}{a_2C_3h} & \frac{1}{a_2C_3} & -\frac{C_4d_3S_{23}}{a_2C_3h} & 0 & 0 & 0 \\ \frac{C_2d_3S_4 - C_3C_4h}{k} & \frac{S_3}{C_3d_4} & \frac{C_3hS_4 + C_2C_4d_3}{k} & 0 & 0 & 0 \\ \frac{J_{41}}{kS_5} & \frac{C_5S_3S_4}{C_3d_4S_5} & \frac{J_{43}}{kS_5} & \frac{C_5}{S_5} & 1 & 0 \\ -\frac{J_{51}}{k} & -\frac{C_4S_3}{C_3d_4} & \frac{J_{53}}{k} & 0 & 0 & 1 \\ -\frac{J_{61}}{kS_5} & -\frac{S_3S_4}{C_3d_4S_5} & \frac{J_{63}}{kS_5} & -\frac{1}{S_5} & 0 & 0 \end{pmatrix} \tag{3.68} $$

where we have

$$ h = C_2a_2 - S_{23}d_4 $$

$$ k = C_3d_4h $$

$$ J_{41} = C_{23}C_3d_4S_4S_5 - C_2C_5d_3S_4^2 + a_2C_2C_3C_4C_5S_4 $$

$$ J_{43} = C_{23}C_3C_4d_4S_5 - a_2C_2C_3C_5S_4^2 - C_2C_4C_5d_3S_4 + C_3C_5d_4S_{23} $$

$$ J_{51} = C_2C_4d_3S_4 + C_3d_4S_{23} - a_2C_2C_3C_4^2 $$

$$ J_{53} = a_2C_2C_3C_4S_4 + C_2C_4^2d_3 $$

$$ J_{61} = C_2d_3S_4^2 - a_2C_2C_3C_4S_4 $$

$$ J_{63} = a_2C_2C_3S_4^2 + C_2C_4d_3S_4 - C_3d_4S_{23} $$

Microbo Jacobian

The Microbo Jacobian is much simpler and may be derived and symbolically inverted with relative ease

$$ {}^4\mathbf{J} = \begin{pmatrix} C_4 d_3 & S_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ d_3 S_4 & -C_4 & 0 & 0 & 0 & 0 \\ S_4 & 0 & 0 & 0 & 0 & S_5 \\ 0 & 0 & 0 & 1 & 0 & -C_5 \\ -C_4 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.69} $$

$$ {}^4\mathbf{J}^{-1} = \begin{pmatrix} \frac{C_4}{d_3} & 0 & \frac{S_4 C_5}{d_3} & 0 & 0 & 0 \\ S_4 & 0 & -C_4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ \frac{-C_4 C_5 S_4}{d_3 S_5} & 0 & \frac{C_5 S_4^2}{d_3 S_5} & \frac{C_5}{S_5} & 1 & 0 \\ \frac{C_4^2}{d_3} & 0 & \frac{C_4 S_4}{d_3} & 0 & 0 & 1 \\ \frac{-C_4 S_4}{d_3 S_5} & 0 & \frac{-S_4^2}{d_3 S_5} & \frac{1}{S_5} & 0 & 0 \end{pmatrix} \tag{3.70} $$

3.6 Jacobian Computations

The Jacobian is typically used in equations (3.1), (3.2), and their inverses. These computations may generally be simplified by expanding the entire expression and then collapsing the result, often making use of recurrences in the solution. As a useful illustration of this, the various Jacobian computations (relative to link 4) are presented for the two robots.

The vector $d\Theta$ is defined by $d\Theta = (d\theta_1, d\theta_2, d\theta_3, d\theta_4, d\theta_5, d\theta_6)^T$ for the PUMA 260, and $d\Theta = (d\theta_1, dd_2, dd_3, d\theta_4, d\theta_5, d\theta_6)^T$ for the Microbo. The joint forces are given by $\mathbf{t}_j = (\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6)^T$ for the PUMA, and $\mathbf{t}_j = (\tau_1, f_2, f_3, \tau_4, \tau_5, \tau_6)^T$ for the Microbo.

PUMA 260 Jacobian computations

The forward Jacobian calculation, ${}^4\mathbf{dc} = {}^4\mathbf{J}\, d\Theta$.

$$ dx = hS_4 d\theta_1 + C_4 k_c $$

$$
\begin{aligned}
dy &= -d_3 S_{23} d\theta_1 + a_2 C_3 d\theta_2 \\
dz &= -C_4 h d\theta_1 + S_4 k_c \\
dx &= C_4 S_{23} d\theta_1 - S_4 (d\theta_2 + d\theta_3) - S_5 d\theta_6 \\
d\rho &= C_{23} d\theta_1 + d\theta_4 + C_5 d\theta_6 \\
d\phi &= S_{23} S_4 d\theta_1 + C_4 (d\theta_2 + d\theta_3) + d\theta_5
\end{aligned} \tag{3.71}
$$

where

$$k_c = C_{23} d_3 d\theta_1 + a_2 S_3 d\theta_2 - d_4 (d\theta_2 + d\theta_3)$$

The force calculation, $\mathbf{t}_j = {}^4\mathbf{J}^T \, {}^4\mathbf{q}$.

$$
\begin{aligned}
\tau_1 &= h(S_4 f_x - C_4 f_z) + C_{23}(d_3 k_f + t_y) + S_{23}(C_4 t_x - d_3 f_y + S_4 t_z) \\
\tau_2 &= \tau_3 + a_2 (S_3 k_f + C_3 f_y) \\
\tau_3 &= -d_4 k_f - S_4 t_x + C_4 t_z \\
\tau_4 &= \tau_y \\
\tau_5 &= \tau_z \\
\tau_6 &= -\tau_x S_5 + C_5 \tau_y
\end{aligned} \tag{3.72}
$$

where

$$k_f = C_4 f_x + S_4 f_z$$

The inverse Jacobian, $d\Theta = {}^4\mathbf{J}^{-1} \, {}^4\mathbf{dc}$.

$$
\begin{aligned}
d\theta_1 &= \frac{d_x S_4 - C_4 d_z}{h} \\
d\theta_2 &= \frac{d_y + d\theta_1 d_3 S_{23}}{a_2 C_3} \\
d\theta_{23} &= \frac{-S_4 d_z - C_4 d_x + a_2 d\theta_2 S_3 + d\theta_1 C_{23} d_3}{d_4} \\
d\theta_3 &= d\theta_{23} - d\theta_2 \\
d\theta_4 &= d\rho - d\theta_6 C_5 - d\theta_1 C_{23} \\
d\theta_5 &= d\phi - d\theta_1 S_{23} S_4 - d\theta_{23} C_4 \\
d\theta_6 &= \frac{-dy - d\theta_{23} S_4 + d\theta_1 C_4 S_{23}}{S_5}
\end{aligned} \tag{3.73}
$$

The inverse force transformation, ${}^4\mathbf{q} = {}^4\mathbf{J}^{-1T} \, \mathbf{t}_j$:

$$f_x = S_4 k_{f1} - C_4 k_{f2}$$

$$
\begin{aligned}
f_y &= \frac{\tau_2 - \tau_3 + a_2 S_3 k_{f2}}{a_2 C_3} \\
f_z &= -S_4 k_{f2} - C_4 k_{f1} \\
\tau_x &= -\frac{\tau_6 - C_5 \tau_4}{S_5} \\
\tau_y &= \tau_4 \\
\tau_z &= \tau_5
\end{aligned} \tag{3.74}
$$

where

$$
k_{f1} = -\frac{S_{23} S_4 \tau_5 + C_{23} \tau_4 - \tau_1 - C_{23} d_3 k_{f2} + (C_4 \tau_x - d_3 f_y) S_{23}}{h}
$$

$$
k_{f2} = \frac{-C_4 \tau_5 + \tau_3 + \tau_x S_4}{d_4}
$$

Microbo Jacobian computations

The forward Jacobian calculation, ${}^4dc = {}^4J\ d\Theta$:

$$
\begin{aligned}
dx &= S_4\ dd_2 + C_4 d_3 d\theta_1 \\
dy &= dd_3 \\
dz &= d_3 S_4 d\theta_1 - C_4\ dd_2 \\
d\psi &= S_5 d\theta_6 + S_4 d\theta_1 \\
d\rho &= d\theta_4 - C_5 d\theta_6 \\
d\phi &= d\theta_5 - C_4 d\theta_1
\end{aligned} \tag{3.75}
$$

The joint force transformation, $t_j = {}^4J^T\ {}^4q$:

$$
\begin{aligned}
\tau_1 &= (\tau_x + d_3 f_z) S_4 + C_4 (d_3 f_x - \tau_z) \\
f_2 &= S_4 f_x - C_4 f_z \\
f_3 &= f_y \\
\tau_4 &= \tau_y \\
\tau_5 &= \tau_z \\
\tau_6 &= \tau_x S_5 - C_5 \tau_y
\end{aligned} \tag{3.76}
$$

The inverse Jacobian calculation, $d\Theta = {}^4J^{-1}\ {}^4dc$:

$$
d\theta_1 = \frac{d_z S_4 + C_4 d_x}{d_3}
$$

$$
\begin{aligned}
dd_2 &= d_x S_4 - C_4 d_z \\
dd_3 &= d_y \\
d\theta_4 &= \frac{d\rho S_5 - C_5 d\theta_1 S_4 + C_5 d\psi}{S_5^2} \\
d\theta_5 &= \frac{-d\theta_2 S_4 + d_3 d\phi + dx}{d_3} \\
d\theta_6 &= \frac{d\psi - d\theta_1 S_4}{S_5}
\end{aligned}
\tag{3.77}
$$

The inverse force transformation, ${}^4\mathbf{q} = {}^4\mathbf{J}^{-1T}\,{}^j\mathbf{t}$:

$$
\begin{aligned}
f_x &= S_4 f_2 + C_4 k_f \\
f_y &= f_3 \\
f_z &= S_4 k_f - C_4 f_2 \\
\tau_x &= \frac{\tau_6 + C_5 \tau_4}{S_5} \\
\tau_y &= \tau_4 \\
\tau_z &= \tau_5
\end{aligned}
\tag{3.78}
$$

where

$$
k_f = \frac{C_4 \tau_5 + \tau_1 - \tau_x S_4}{d_3}
$$

In working with the inverse Jacobian relations, it should be noted that there are certain configurations of the arm for which the Jacobian becomes singular, corresponding to when the denominators in the inverse Jacobian computations are zero.

3.7 Transforming the Jacobian Computations

Although in the foregoing discussion the Jacobian has been derived with respect to link 4, results concerning the Jacobian are typically desired with respect to some other link frame (such as link 6). To do this, it is necessary to transform the appropriate force and differential motion vectors between the Jacobian frame and the observation frame. The transformations which do this mapping from frame k to frame j were introduced in sections 1.4.2 and 1.4.4.

To get results in frame 6, with the Jacobian being calculated in frame 4, equations (3.1), (3.2), and their inverses, may be written as:

$$
{}^6\mathbf{dc} = \mathbf{D}_{4,6}({}^4\mathbf{J}\,\mathbf{d}\Theta)
$$

Instance	multiplies/divisions	additions
PUMA 260, best case	41	23
PUMA 260 worst case	46	23
Microbo, best case	24	13
Microbo, worst case	31	15

Table 3.3 Total expense of different Jacobian computations

$$\begin{aligned} d\Theta &= {}^4\mathbf{J}^{-1}\,\mathbf{D}_{6,4}({}^6\mathbf{dc}) \\ {}^6\mathbf{q} &= \mathbf{F}_{4,6}({}^4\mathbf{J}^{-1T}\,\mathbf{t}_j) \\ \mathbf{t}_j &= {}^4\mathbf{J}^T\,\mathbf{F}_{6,4}({}^6\mathbf{q}) \end{aligned} \tag{3.79}$$

The frames in links 4 and 6 are related by the matrix $\mathbf{X} = \mathbf{A}_5\mathbf{A}_6$ and its inverse. The $\mathbf{D}$ and $\mathbf{F}$ transformations are obtained by substituting this into equations (1.13) and (1.23). This yields, for both the PUMA and the Microbo, a solution of the form:

$$\mathbf{D}_{4,6} = \begin{pmatrix} \mathbf{B} & 0 \\ 0 & \mathbf{B} \end{pmatrix} \quad \mathbf{D}_{6,4} = \begin{pmatrix} \mathbf{B}^T & 0 \\ 0 & \mathbf{B}^T \end{pmatrix}$$

$$\mathbf{F}_{4,6} = \begin{pmatrix} 0 & \mathbf{B} \\ \mathbf{B} & 0 \end{pmatrix} \quad \mathbf{F}_{6,4} = \begin{pmatrix} 0 & \mathbf{B}^T \\ \mathbf{B}^T & 0 \end{pmatrix} \tag{3.80}$$

$\mathbf{B}$ for the PUMA is given by

$$\mathbf{B} = \begin{pmatrix} C_5C_6 & S_5C_6 & -S_6 \\ -C_5S_6 & -S_5S_6 & -C_6 \\ -S_5 & C_5 & 0 \end{pmatrix} \tag{3.81}$$

$\mathbf{B}$ for the Microbo is given by

$$\mathbf{B} = \begin{pmatrix} C_5C_6 & S_5C_6 & S_6 \\ -C_5S_6 & -S_5S_6 & C_6 \\ S_5 & -C_5 & 0 \end{pmatrix} \tag{3.82}$$

3.8 Summary

We have derived the forward and inverse kinematics for the PUMA 260 and the Microbo Ecureuil. The differential kinematics, when computed with respect to the 4th link of the

manipulator, reduce to a reasonably simple form. The Jacobian computations may be performed in link 4 of the manipulator and transformed to link 6 by using the equations in section 3.6 in conjunction with the transformations in (3.79). A summary of the total expense involved in these computations, excluding trigonometric function calls, is given in Table 3.3. (Each transform in (3.79) may be optimized so that it requires a total of 8 multiples and 4 additions.)

A brief mention should be made of errors in the kinematics. In practice, the **A** matrices for a robot are never known exactly. We may associate with each matrix $\mathbf{A}_i$ an error matrix $d\mathbf{A}_i$. The true value of $\mathbf{T}_6$ may then be computed from the equation

$$\mathbf{T}_6 = \prod_{i=1}^{6} (\mathbf{A}_i + d\mathbf{A}_i) \tag{3.83}$$

Expanding this and neglecting differential terms greater than first order allows us to separate the computation of $\mathbf{T}_6$ into an ideal component and a correction component:

$$\mathbf{T}_6 = \prod_{i=1}^{6} \mathbf{A}_i + \mathbf{M}\,\mathbf{e} \tag{3.84}$$

where **M** is a position dependent correction matrix, which may be found analytically, and **e** is a constant vector of error parameters. Measurements may in principle be made to determine **e** and find any error terms which are large enough to justify being used in the run-time kinematic calculations. A very comprehensive treatment of this subject is given in [Wu 85].

Chapter 4

MEASUREMENT of STATIC FORCE CONTROL PARAMETERS

4.1 Overview

It is often useful to control a robot manipulator so as to enable it to exert a specified set of static forces on its environment (*i e.*, forces not involving acceleration of the manipulator or the objects the manipulator is acting on). This process is known as "static force control", and is associated with a special case of the manipulator dynamics equation, discussed in section 4 3. The control itself requires knowing the parameters associated with the gravity loading and friction models of the arm

The measurement of these parameters for the PUMA 260 is the subject of this chapter The procedure discussed was not applied to the Microbo because the gearing mechanisms on that robot's prismatic joints make it unsuitable for the force control techniques that were used.

The chapter is organized as follows. We begin with a discussion of the principles underlying the measurement technique Next, the dynamics equation is reviewed, static force control is precisely defined, and the equations which model the gravity loading and friction are discussed The individual measurement techniques and their results are then presented Lastly, we describe transforming joint level force control into Cartesian coordinates in the robot's last link

An encompassing survey of the problems related to static force control is given in [Whitney 85].

4.2 Basis of the Measurement Technique

The measurement method is based on the ability to determine the work done by a joint motor by monitoring its current. This follows from the model of a permanent magnet DC motor [Slemon and Straughen 81].

$$\tau = k\phi i_a \tag{4.1}$$

$$\tau = \tau_{load} + \tau_{friction} \tag{4.2}$$

where

τ = the total torque produced in the armature

k = a motor parameter

ϕ = magnetic flux

i_a = armature current

τ_{load} = output load torque

$\tau_{friction}$ = mechanical and electromagnetic lossages

Since $k\phi$ is constant for these motors, the relationship simplifies to

$$\tau_{load} + \tau_{friction} = \sigma i_a \tag{4.3}$$

where σ is the *torque sensitivity* for the motor.

Now let the angular position of the motor shaft be given by the variable θ_s. The total work done by the servo motor as it moves the joint from θ_{s1} to θ_{s2} is

$$W = \int_{\theta_{s1}}^{\theta_{s2}} \frac{i_a d\theta_s}{\sigma} \tag{4.4}$$

A finite approximation to this can be computed by monitoring i_a and θ_s at a sufficiently high sample rate. If we then explicitly separate the work due to load and the work due to losses, we get:

$$W_{load} + W_{friction} = \sum_{\theta_{s1}}^{\theta_{s2}} \frac{i_a \Delta\theta_s}{\sigma} \tag{4.5}$$

The work associated with the motion of the manipulator along a given path can hence be measured. If a path is chosen for which the work depends on a particular parameter, it is then possible to solve for this parameter.

Another technique has been used at CVaRL to measure the inertia parameters of the robot; this involved observing the response of the different links to a sinusoidal input current [Aboussouan 85]. Both this latter method, as well as the methods detailed in this chapter, were implemented using the RCI system described in Chapter 2.

4.3 The Manipulator Dynamics Equation

We review briefly the formulation for the manipulator dynamics equation [Neuman and Murray 84, Paul 81]. The general form of this equation for a manipulator with N joints is

$$\mathbf{D}(\Theta)\ddot{\Theta} + \mathbf{c}(\Theta,\dot{\Theta}) + \mathbf{t}_g(\Theta) + \mathbf{f}(\dot{\Theta}) + \mathbf{t}_a(t) = \mathbf{t}_j(t) \qquad (4.6)$$

where

$\Theta \triangleq$ generalized joint coordinates

$\mathbf{D}(\Theta) = N \times N$ inertia matrix

$\mathbf{c}(\Theta,\dot{\Theta}) =$ centrifugal and Coriolis force vector

$\mathbf{t}_g(\Theta) =$ gravity loading vector

$\mathbf{f}(\dot{\Theta}) =$ joint friction

$\mathbf{t}_a(t) =$ externally applied joint forces

$\mathbf{t}_j(t) =$ joint torques/forces exerted by the motors

The $\mathbf{D}$, $\mathbf{c}$, and $\mathbf{t}_g$ terms are functions of Θ, and $\mathbf{c}$ and $\mathbf{f}$ are functions of the joint velocities $\dot{\Theta}$. The friction, $\mathbf{f}$, is in fact hard to characterize precisely, more will be said about this later on. In general, equation (4.6) can expand to great complexity. However, in cases where the required velocities and accelerations of the manipulator are small enough, it is possible to ignore $\mathbf{D}$ and $\mathbf{c}$ entirely, which reduces the problem to one involving only the gravity loadings, frictional characteristics, and external forces:

$$\mathbf{t}_g(\Theta) + \mathbf{f}(\dot{\Theta}) + \mathbf{t}_a(t) = \mathbf{t}_j(t) \qquad (4.7)$$

This defines the domain of the static force control problem.

The characterization of the terms of equation (4.7) for the PUMA 260 robot is the subject of the rest of this chapter.

4.4 Parameter Models

We now turn our attention to the models which describe the gravity loading, joint friction, and the implementation of equation (4.7) on the PUMA 260.

4.4.1 The Gravity Loading Model

A derivation similar to the one presented below is described in [Paul and Rong 83] for the PUMA 560.

When fully expanded, each element τ_{g_i} of $\mathbf{t}_g$ in equation (4.7) takes the form ([Neuman and Murray 84, Paul 81]):

$$\tau_{g_i} = \sum_{k=i}^{N} \left(-m_k \, \mathbf{g}^T \, \frac{\partial \mathbf{T}_k}{\partial \theta_i} \, {}^k\bar{\mathbf{r}}_k \right) \tag{4.8}$$

where

τ_{g_i} = torque/force joint i must exert to overcome gravity

N = total number of links for the robot

m_k = mass of link k

$\mathbf{g}$ = acceleration of gravity vector, measured in the base coordinate frame*

$\mathbf{T}_k$ = 4×4 transformation matrix from the base frame to link k

θ_i = joint variable for joint i

${}^k\bar{\mathbf{r}}_k$ = center of mass for link k, measured in that link's coordinate frame

$\mathbf{T}_k$ is given by the product of the $\mathbf{A}$ matrices for each joint up to k:

$$\mathbf{T}_k = \mathbf{A}_1 \cdots \mathbf{A}_k \tag{4.9}$$

The partial derivative term in (4.8) may be handled using the matrix representation for differential coordinate changes (section 1.4.2). Consider first a differential change in $\mathbf{T}_k$ due to a variation in joint i, as observed in link k.

$$d\mathbf{T}_k = \mathbf{T}_k \, \Delta_i \tag{4.10}$$

---

* Since the force of gravity is purely translational, it can be represented as a homogeneous 4-vector with the fourth element equal to 0.

The most convenient coordinate frame in which to express a differential change due to a joint variable θ_i is the frame attached to the preceding link $i-1$. In that frame the change is simply an infinitesimal motion along or about the z axis, depending on whether the joint is prismatic or revolute. Assuming that $d\mathbf{T}_k$ is observed in frame k, (1.4) and (4.10) may be combined to get

$$d\mathbf{T}_k = \mathbf{T}_k \mathbf{U}_k^{-1}\, {}^{i-1}\Delta_i\, \mathbf{U}_k \tag{4.11}$$

where $\mathbf{U}_k$ is the coordinate transformation from frame $i-1$ to frame k, and is given by

$$\mathbf{U}_k = \mathbf{A}_i \cdots \mathbf{A}_k$$

Given that ${}^{i-1}\Delta_i$ is represented in the frame attached to link $i-1$, the differential joint variation $d\theta_i$ can be factored out to obtain

$${}^{i-1}\Delta_i = \mathbf{Q}_i\, d\theta_i \tag{4.12}$$

where

$$\mathbf{Q}_i = \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{(rotary joint)} \tag{4.13}$$

$$\mathbf{Q}_i = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{(prismatic joint)} \tag{4.14}$$

We can now write (4.11) as

$$d\mathbf{T}_k = \mathbf{T}_k \mathbf{U}_k^{-1} \mathbf{Q}_i \mathbf{U}_k d\theta_i \tag{4.15}$$

from which the partial derivative may be obtained

$$\frac{\partial \mathbf{T}_k}{\partial \theta_i} = \mathbf{T}_k \mathbf{U}_k^{-1} \mathbf{Q}_i \mathbf{U}_k \tag{4.16}$$

Substituting this into equation (4.8) yields

$$\begin{aligned} \tau_{g_i} &= \sum_{k=i}^{N} \left(-m_k \mathbf{g}^T \mathbf{T}_k \mathbf{U}_k^{-1} \mathbf{Q}_i \mathbf{U}_k\, {}^k\bar{\mathbf{r}}_k\right) \\ &= \sum_{k=i}^{N} \left(-m_k \mathbf{g}^T \mathbf{A}_1 \cdots \mathbf{A}_k \mathbf{A}_k^{-1} \cdots \mathbf{A}_i^{-1} \mathbf{Q}_i \mathbf{U}_k\, {}^k\bar{\mathbf{r}}_k\right) \\ &= \sum_{k=i}^{N} \left(-m_k \mathbf{g}^T \mathbf{A}_1 \cdots \mathbf{A}_{i-1} \mathbf{Q}_i \mathbf{U}_k\, {}^k\bar{\mathbf{r}}_k\right) \end{aligned} \tag{4.17}$$

This rearranges to form

$$\tau_{g_i} = -\mathbf{g}^T \mathbf{T}_{i-1} \mathbf{Q}_i \sum_{k=i}^{N} (m_k \mathbf{U}_k {}^k\bar{\mathbf{r}}_k) \tag{4.18}$$

The first term in front of the sum can be further simplified:

$$-\mathbf{g}^T \mathbf{T}_{i-1} \mathbf{Q}_i = -(\mathbf{g}^T \mathbf{n}_{i-1} \quad \mathbf{g}^T \mathbf{o}_{i-1} \quad \mathbf{g}^T \mathbf{a}_{i-1} \quad \mathbf{g}^T \mathbf{p}_{i-1}) \mathbf{Q}_i \tag{4.19}$$

where $\mathbf{n}_{i-1}$, $\mathbf{o}_{i-1}$, $\mathbf{a}_{i-1}$, and $\mathbf{p}_{i-1}$ are the columns of $\mathbf{T}_{i-1}$. Making use of (4.13) and (4.14), we can express the final solution as

$$\tau_{g_i} = -\mathbf{h}_i \sum_{k=i}^{N} (m_k \mathbf{U}_k {}^k\bar{\mathbf{r}}_k) \tag{4.20}$$

where

$$\mathbf{h}_i = (\mathbf{g}^T \mathbf{o}_{i-1} \quad -\mathbf{g}^T \mathbf{n}_{i-1} \quad 0 \quad 0) \qquad \text{(rotary joint)}$$

$$\mathbf{h}_i = (0 \quad 0 \quad 0 \quad \mathbf{g}^T \mathbf{a}_{i-1}) \qquad \text{(prismatic joint)}$$

Equation (4.20) is valid for any manipulator. We now consider the case of the PUMA 260. It is first necessary to determine the vectors ${}^k\bar{\mathbf{r}}_k$. The coordinate frames for the PUMA 260 are illustrated in Figure 3.2. From the symmetry of the joints, certain components of these vectors can be approximated by zero. An illustration of the joints and their centers of mass is given in Figure 4.1.

The vectors are:

$$\begin{aligned}
{}^1\bar{\mathbf{r}}_1 &= (0 \quad 0 \quad \bar{z}_1 \quad 1) \\
{}^2\bar{\mathbf{r}}_2 &= (\bar{x}_2 \quad 0 \quad \bar{z}_2 \quad 1) \\
{}^3\bar{\mathbf{r}}_3 &= (0 \quad 0 \quad \bar{z}_3 \quad 1) \\
{}^4\bar{\mathbf{r}}_4 &= (0 \quad \bar{y}_4 \quad 0 \quad 1) \\
{}^5\bar{\mathbf{r}}_5 &= (0 \quad 0 \quad \bar{z}_5 \quad 1) \\
{}^6\bar{\mathbf{r}}_6 &= (0 \quad 0 \quad \bar{z}_6 \quad 1)
\end{aligned} \tag{4.21}$$

To complete the solution, the value of $\mathbf{g}$ as measured in the base frame must be known; this will be determined by the mounted orientation of the robot. Assuming here that the robot is mounted upright, we have:

$$\mathbf{g}^T = (0 \quad 0 \quad g \quad 0) \tag{4.22}$$

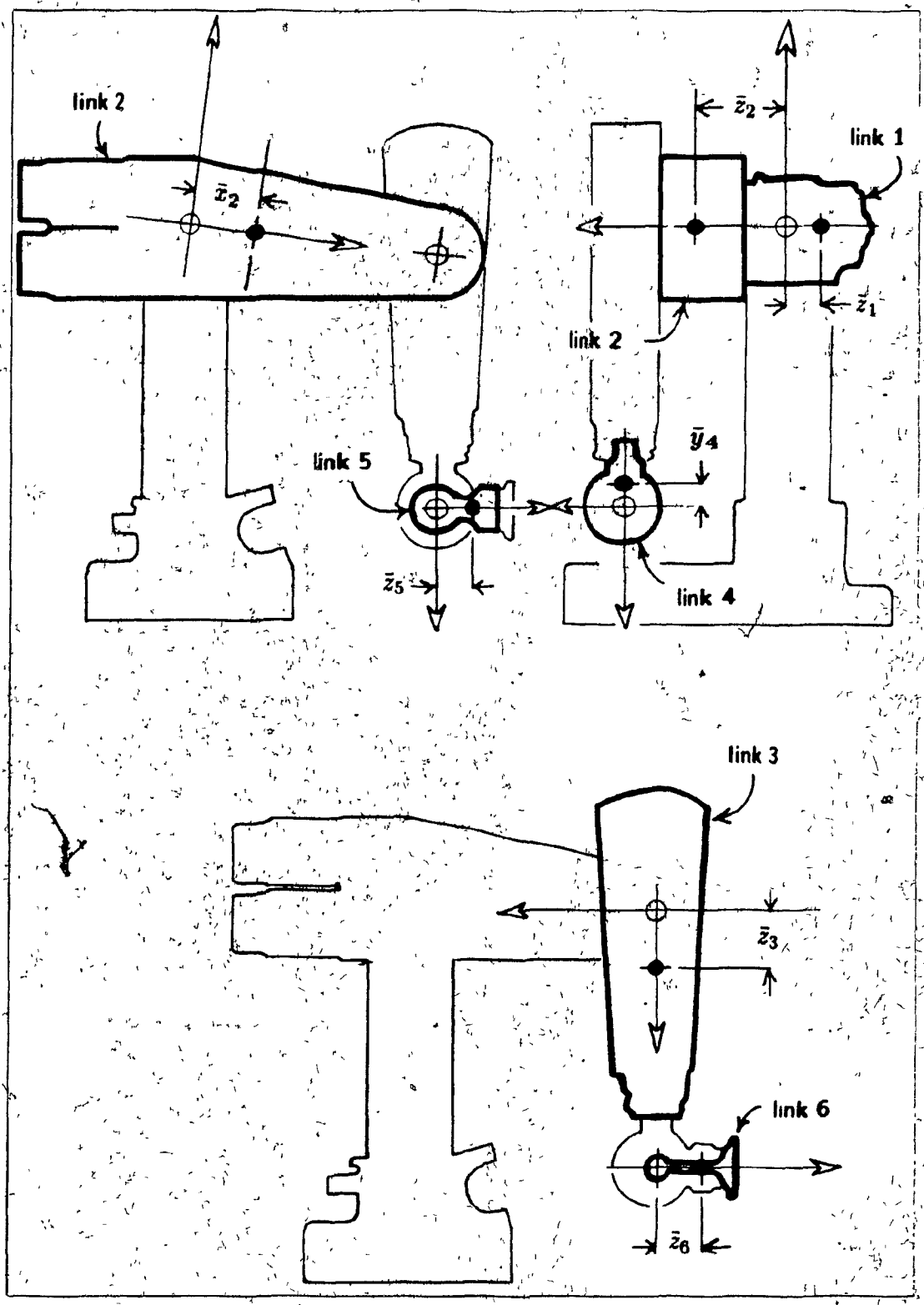


Figure 4.1 Center of mass locations for the PUMA 260 links.

This, taken with the fact that all PUMA 260 joints are rotary, means that $\mathbf{h}_i$ in equation (4.20) has the form

$$\mathbf{h}_i = (\, o_{z,i-1} \quad -n_{z,i-1} \quad 0 \quad 0 \,) \tag{4.23}$$

where $o_{z,i-1}$ and $n_{z,i-1}$ denote the (3,2)-th and (3,1)-th elements of $\mathbf{T}_{i-1}$.

Multiplying out equation (4.20) using the A matrices (defined for the PUMA in section 3.2.2) and factoring out expressions containing joint variables yields the gravity loading vector $\mathbf{t}_g$. (We use the notation $S_i = \sin\theta_i$, $C_i = \cos\theta_i$, $S_{ij} = \sin(\theta_i + \theta_j)$, and $C_{ij} = \cos(\theta_i + \theta_j)$.)

$$\mathbf{t}_g = \begin{pmatrix} 0 \\ (C_{23}C_4S_5 + C_5S_{23})\, c_{l5} + S_{23}\, c_{l3} + C_2\, c_{l2} \\ (C_{23}C_4S_5 + C_5S_{23})\, c_{l5} + S_{23}\, c_{l3} \\ -S_{23}S_4S_5\, c_{l5} \\ (C_{23}S_5 + C_4C_5S_{23})\, c_{l5} \\ 0 \end{pmatrix} \tag{4.24}$$

where

$$\begin{aligned} c_{l2} &= g[m_2\bar{x}_2 + a_2(m_6 + m_5 + m_4 + m_3 + m_2)] \\ c_{l3} &= -g[m_3\bar{z}_3 + m_4\bar{y}_4 + d_4(m_6 + m_5 + m4)] \\ c_{l5} &= -g(m_6\bar{z}_6 + m_5\bar{z}_5) \end{aligned}$$

The c_{li} are the joint gravity loading coefficients, and are independent of the arm position. The computation of $\mathbf{t}_g$ can be simplified by noting that

$$t_{g_2} = t_{g_3} + C_2\, c_{l2} \tag{4.25}$$

In many cases, the loading at the end of the robot varies, as either tools are changed or the robot picks up objects. We can model the tool loading (approximately) as a mass m_t acting about a point $\mathbf{d}_t$ along the z axis of link 6. The presence of this load will change the gravity loading coefficients from c_{li} to c'_{li}, as follows:

$$\begin{aligned} c'_{l5} &= c_{l5} - g(m_t\, d_t) \\ c'_{l3} &= c_{l3} - g(m_t\, d_4) \\ c'_{l2} &= c_{l2} + g(m_t\, a_2) \end{aligned} \tag{4.26}$$

These new values of c'_{li} may then be used directly in equation (4.24).

4.4.2 The Friction Model

The joint friction $\mathbf{f}(\dot{\Theta})$ is modeled as a combination of static friction, Coulomb friction, and viscous friction. Coulomb friction is a constant resistance which is in effect while the joint is moving. Viscous friction is velocity dependent, and static friction is the resistance that must be overcome to get the joint moving. Slight asymmetries in the mechanical system usually make these frictional values slightly different in the positive and negative directions of motion.

For a particular joint i, the friction model is given as follows, where $\dot{\theta}$ is the velocity of the joint variable:

$$\begin{aligned} f(\dot{\theta}) &= F_{c_+} + F_{v_+}\,\dot{\theta}\,, & \dot{\theta} > 0; \\ f(\dot{\theta}) &= -F_{c_-} + F_{v_-}\,\dot{\theta}\,. & \dot{\theta} < 0; \\ -F_{s_-} &\le f(\dot{\theta}) \le F_{s_+} & \dot{\theta} = 0 \end{aligned} \tag{4.27}$$

F_{c_+} and F_{c_-} are the values of Coulomb friction, in the positive and negative directions. F_{v_+} and F_{v_-} are the coefficients of viscous friction, and F_{s_+} and F_{s_-} are the coefficients of static friction.

Each joint i will have a different set of friction coefficients which can be used in (4.27) to determine the $\mathbf{f}(\dot{\Theta})$ term in equation (4.7).

4.4.3 Implementation Specifics for the PUMA 260

For any particular robot, the terms of equation (4.7) are generally not usable directly: joint torques must be converted into current signals; joint positions must be converted into radians, etc. Also, in the case of the PUMA robots, the last three wrist joints are mechanically coupled so that variables at the joint level are not directly proportional to variables at the motor level. We shall describe these relationships here. The values for the conversion constants and matrices referred to are given in Appendix D.

If we denote the angular positions of the motor shafts by the vector Θ_s, this is related to the joint angles Θ by the *gear ratio matrix* $\mathbf{G}$:

$$\Theta_s = \mathbf{G}\,\Theta \tag{4.28}$$

If the drive train for each joint was independent, then $\mathbf{G}$ would be diagonal. For the PUMA however, the last 3 links are in fact mechanically coupled so that $\mathbf{G}$ is lower triangular. From the relationship (1.19), it follows that the joint torques $\mathbf{t}_j$ are related to the motor shaft torques $\mathbf{t}_s$ by the equation

$$\mathbf{t}_j = \mathbf{G}^T \mathbf{t}_s \tag{4.29}$$

The motor shaft positions are measured using optical encoders, and hence are initially described not as the angles Θ_s, but as a vector of *encoder counts*, $\mathbf{e}$, which may be related directly to the joint angles Θ by the *encoder ratio matrix* $\mathbf{R}$:

$$\mathbf{e} = \mathbf{R}\,\Theta \tag{4.30}$$

Motor current values are measured as a vector of analog-to-digital converter (ADC) values, $\mathbf{v}_{adc}$, which describe the voltage drops across sensing resistors in the motor current return path (see Appendix C). Remembering that the motors on the PUMA are of the type described in section 4.2, it can be seen that these values will be proportional to the motor shaft torque (before frictional losses), and we express this proportionality with a diagonal matrix $\mathbf{K}_{adc}$:

$$\mathbf{t}_s = \mathbf{K}_{adc}\,\mathbf{v}_{adc} \tag{4.31}$$

The diagonal elements $k_{adc_{i,i}}$ of $\mathbf{K}_{adc}$ are given by

$$k_{adc_{i,i}} = \frac{V_g}{R_i}\,\sigma_i \tag{4.32}$$

where V_g is the ADC/voltage ratio, R_i is the value of the sensing resistor, and σ_i is the torque sensitivity of the motor.

Conversely, motor shaft torques are specified by sending to the particular joint a digital-to-analog converter (DAC) value. A vector of such values, $\mathbf{v}_{dac}$, is related to the motor shaft torques by

$$\mathbf{v}_{dac} = \mathbf{K}_{dac}\,\mathbf{t}_s \tag{4.33}$$

$\mathbf{K}_{dac}$ is also diagonal, and each element is related to $k_{adc_{i,i}}$ by a proportionality constant κ_i:

$$k_{dac_{i,i}} = \kappa_i\,k_{adc_{i,i}} \tag{4.34}$$

Equations (4.29), (4.31), and (4.33) may be combined to get

$$\mathbf{v}_{dac} = \mathbf{K}_{dac}\,\mathbf{G}^{-1T}\,\mathbf{t}_j \tag{4.35}$$

$$\mathbf{t_j} = \mathbf{G}^T \mathbf{K_{adc}} \mathbf{v_{adc}} \tag{4.36}$$

These equations are the ones used in determining the raw joint torques $\mathbf{t_j}$ when measuring the friction and gravity loading parameters as discussed below. Once these parameters have been determined, then the terms of equation (4.7) can be solved for and it is possible to determine the net torque exerted on a joint

$$\mathbf{t_a} = \mathbf{G}^T \mathbf{K_{adc}} (\mathbf{v_{adc}} - \mathbf{f_{adc}}(\dot{\Theta})) - \mathbf{t_g}(\Theta) \tag{4.37}$$

or, conversely, the current signal necessary to yield a particular net torque $\mathbf{t_a}$:

$$\mathbf{v_{dac}} = \mathbf{K_{dac}} \mathbf{G}^{-1T} (\mathbf{t_a} + \mathbf{t_g}(\Theta)) + \mathbf{f_{dac}}(\dot{\Theta}) \tag{4.38}$$

where $\mathbf{f_{adc}}(\dot{\Theta})$ and $\mathbf{f_{dac}}(\dot{\Theta})$ are the appropriate frictional values converted to either ADC or DAC values (it was computationally more efficient to handle the friction at this level).

4.5 Measurements

The actual parameter measurements are now described.

4.5.1 Basic Method

The measurement technique involved moving a particular joint along a path for which the total work done could be related to the desired parameter. The joint was moved by sending position requests to the appropriate joint microprocessor, whose PID control algorithm then dispatched the necessary current to the motor. Since the present PUMA 260 controller does not allow communication with the joint PID level, and also because the current control is imprecise, the joint current was determined separately by measuring the voltage drop across a sensing resistor, as mentioned above in section 4 4.3

It was discovered that the joint controllers employed an algorithm which issued a widely varying current signal, with a fluctuation typically as high as the average current value. However, since the joint controllers operate at 1 KHz, more than an order of magnitude faster than rate of observation for the experiments described here, it was possible to filter out this effect using an analog circuit interface (described in Appendix C), thus yielding an estimate of the motor current which could then be related to the physical torque on the joint with equation (4.36).

To obtain greater accuracy, multiple measurements were taken and then averaged. To ensure that kinetic energy did not enter into the measurements, the joint was moved at a constant velocity, and changes in direction were done at points outside the actual part of the path for which the observations were made. Since the joint friction tended to exhibit complex behavior, additional measurements were sometimes needed to cancel out the work done by friction, these are described below.

4.5.2 Torque Sensitivities

Although the torque sensitivities for the motors were available from the manufacturer, it became obvious, after some experimentation, that the given values were in error by as much as 25%. It was therefore necessary to measure the torque sensitivities directly

The technique used is quite simple: The joint in question is moved back and forth at constant speed through a fixed size arc (typically 60°), alternately raising and lowering a known mass which is attached to it by means of a pulley (Figure 4.2) The joint currents and positions are monitored in both directions, and from this the work done in each direction is calculated. The arc is kept small to keep the loading on the pulley relatively constant If W_+ is the work done in the positive direction, and W_- is the work done in the negative direction, these are equal to

$$W_+ = W_{f_+} + W_{fpulley_+} + W_{joint} + W_{mass} \tag{4.39}$$

$$W_- = W_{f_-} + W_{fpulley_-} - W_{joint} - W_{mass} \tag{4.40}$$

where W_{f_+} and W_{f_-} are the work done by the joint friction (in each direction). $W_{fpulley_+}$ and $W_{fpulley_-}$ denote the work done by the pulley friction. W_{joint} is the work due to any change in the potential energy of the joint; and W_{mass} is the change in potential energy of the known mass

To remove the friction terms, another set of measurements is performed without the attached mass, yielding a new set of work values, W'_+ and W'_-, which are equal to

$$W'_+ = W'_{f_+} + W_{joint} \tag{4.41}$$

$$W'_- = W'_{f_-} - W_{joint} \tag{4.42}$$

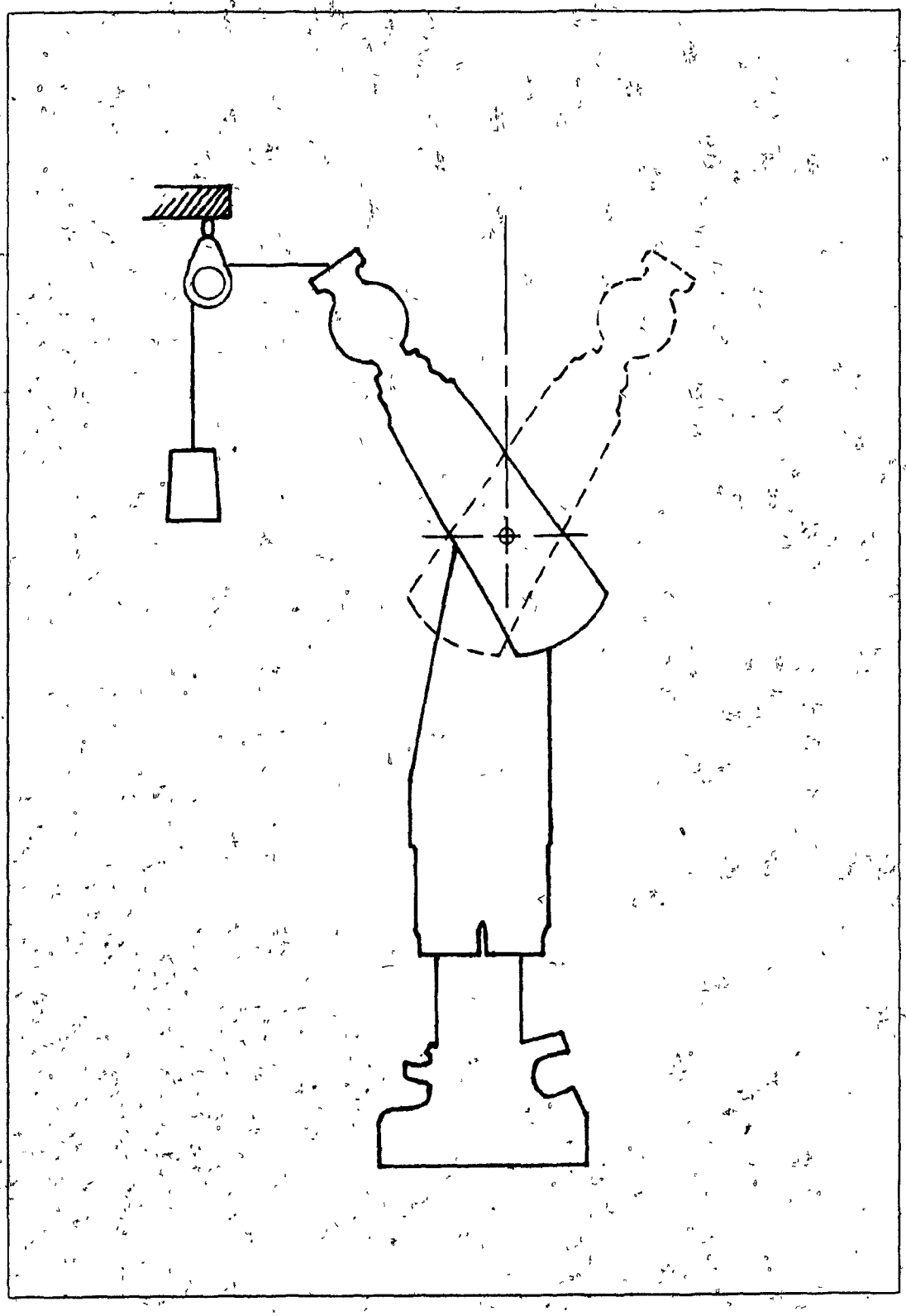


Figure 4.2 Measurement of the torque sensitivity for joint 3.

Joint	Number of trials	Mean	Standard deviation	Independent error estimate
1	22	.0732	0008	.002
2	25	.0751	.0006	.002
3	21	.0786	.0008	.003
4	13	.0358	.0008	.0015
5	15	.0349	.0006	0015
6	31	.0305	.0006	.0015

Table 4.1 Motor torque sensitivity measurements (Newton-meters/ampere)

The joint friction terms W'_{f_-} and W'_{f_+} may be slightly different from W_{f_-} and W_{f_+} because of the change in load conditions; we denote these differences by δW_{f_-} and δW_{f_+}. Since the load conditions on the pulley are constant in either direction, $W_{fpulley_+}$ and $W_{fpulley_-}$ are taken to be equal. Subtracting (4.41) and (4.42) from (4.39) and (4.40), and then subtracting again, yields

$$W_+ - W'_+ - W_- + W'_- = (\delta W_{f_+} - \delta W_{f_-}) + 2W_{mass} \tag{4.43}$$

All of the quantities on the LHS are directly measurable, and W_{mass} is known, leaving only a small uncertainity due to the δW_f terms.

For the measurements made at CVaRL, a large number of trials were run with different values of mass, arc length, direction, and joint velocity. The results were then averaged, helping to cancel out the δW_f variations. The overall accuracy appeared to be quite good, based on the small spread in the values for the different trials. The resulting torque sensitivities for all six joints are given in Table 4.1, including the number of trials, the average value, the standard deviation of the values, and an independent estimate of the error, based on known uncertainities in the system. These values may be contrasted with the values declared by the manufacturer, which were .092 (Newton-meters/ampere) for joints 1 to 3, and .035 for joints 4 to 6.

4.5.3 General Observations about the Friction

Some general experimentation with the joint friction showed that it depended on velocity

and direction, consistent with the model presented in (4.27). However, the friction was also found, to a certain extent, to depend on the following:

- ◇ *load*. This is to be expected since as the force between two surfaces changes, so does the force of friction.
- ◇ *position*. This is attributed to irregularities in the joint gearing mechanism.
- ◇ *time*. When a joint was first moved, after sitting idle for a certain period, the observed friction was higher (by as much as 20%) than when it had been in motion for a while. No precise explanation has been made for this, except that joint activity will increase the temperature of the gearing mechanism and presumably alter the characteristics of the lubricants. It was found, however, that once a joint had "warmed up", successive measurements of the work done between two points had a very high repeatability, usually less than 1%.

Since these effects were secondary compared to the model given in section 4.4.2, no attempt was made to characterize them precisely.

4.5.4 Coulomb and Viscous Friction

The Coulomb and viscous friction terms were determined by moving the joint in question at a constant speed along a path selected such that the potential energy of the arm did not change from the beginning to the end of the path. The total work done along the path was then equal to the work of friction. The value of friction in the negative direction was determined by doing the same measurement in reverse. These measurements were then done at different speeds, and the observations were fitted to the first two equations in (4.27) using a least square method. The measurements are plotted in Appendix B. The resulting Coulomb friction values are shown in Table 4.2, and the viscous friction coefficients are shown in Table 4.3. The errors given for each value were estimated by taking twice the standard deviation associated with the least square fit.

It should be noted that these values represent the average friction over a large range of joint travel. The friction values at a given point turned out to be far less predictable. At each point in the measurement path, the observed friction value could be expected to deviate from the average value by a percentage indicated in Table 4.4. This variation in

Joint	F_{c-}	F_{c+}
1	0.760 ± .05	0.640 ± .05
2	1.620 ± .05	1.620 ± .06
3	0.850 ± .03	0.750 ± .03
4	0.175 ± .012	0.194 ± .009
5	0.178 ± .003	0.187 ± .004
6	0.140 ± .004	0.159 ± .005

Table 4.2 Measured joint Coulomb friction values (Newton-meters).

Joint	F_{v+} (×1000)	F_{v-} (×1000)
1	12.03 ± .90	12.18 ± .91
2	8.43 ± 1.50	7.61 ± 1.70
3	4.78 ± .48	4.59 ± .45
4	0.50 ± .17	0.50 ± .14
5	0.70 ± .04	0.69 ± .05
6	0.31 ± .04	0.29 ± .05

Table 4.3 Measured joint viscous friction coefficients (Newton-meters-seconds / radian)

fact introduces a considerable unreliability into instantaneous measurements of the external torque acting on a joint.

4.5.5 Static Friction

An initial attempt to measure the static friction of the joints by measuring the work done at very slow speeds did not appear to produce meaningful results.

Instead, the static friction was determined using two other methods. The first was a simple "tuning" approach, where the joint in question was oriented so as to be free from gravity loading, the motor current value for that joint was set to be equal to the Coulomb

Joint	Variation
1	10%
2	20%
3	13%
4	27%
5	10%
6	5%

Table 4.4 Approximate variation of observed Coulomb friction values about the mean.

friction value for a particular direction, and then this value was adjusted until the joint could be moved, by hand, over its whole range, without appearing to "stick" very much The values determined in this way give an estimate of the static friction coefficients and are shown in Table 4 5.

The second method measured the typical static friction values by moving the robot through a set of randomly selected positions. At each position, the arm was brought to a halt and the static friction was computed using equation (4.37), given that the gravity loadings t_g were known (from measurements described in section 4.5.6), and t_a was 0 since the manipulator was unloaded. The measured values, when observed over a large number of samples, were characterized by a Gaussian distribution that was roughly zero meaned The results are given in Table 4.6.

It can be seen from this table that in practice the typical value of the static friction is somewhat less than the estimated coefficient values given in Table 4 5.

4.5.6 Gravity Loading

We begin the discussion by relating the loading torque on a joint to the work done against gravity as it travels through an angular distance. If all of the subsequent joints are held fixed, then those links plus the one under consideration form a single rigid body Figure 4.3 shows an arbitrary rigid body oriented so that it rotates in a vertical plane. A center of mass vector $\bar{r}$ is defined from the joint axis to the center of mass of the body.

Joint	Mean value (positive direction)	Mean value (negative direction)
1	0.88	0.88
2	2.23	2.04
3	1.36	1.36
4	0.179	0.180
5	0.197	0.195
6	0.197	0.211

Table 4.5 Static friction coefficients F_s hand measured (Newton-meters)

Joint	Maximum value	Minimum value	Mean	Standard deviation
1	0.429	-0.316	0.050	0.116
2	1.312	-1.103	0.118	0.375
3	0.895	-0.520	0.074	0.265
4	0.184	-0.217	-0.009	0.084
5	0.132	-0.167	-0.003	0.060
6	0.158	-0.185	-0.011	0.080

Table 4.6 Typical static friction values, based on 275 samples (Newton-meters)

This vector is initially oriented at some angle ϕ_0, and will rotate as the joint rotates.

The joint is then moved through a distance of 180° (Figure 4.4). The observations of this motion are divided into two segments, with segment 1 defined by the interval $[\phi_0, \phi_0 + 90°]$, and segment 2 defined by $[\phi_0 + 90°, \phi_0 + 180°]$. The work done against gravity is given by W_{g1} for the first segment and W_{g2} for the second segment. If m is the mass of the body, r is the length of $\bar{r}$, and g is the acceleration of gravity, it can be seen that

$$\begin{aligned} W_{g1} &= m\,g\,r\,(\cos\phi_0 - \sin\phi_0) \\ W_{g2} &= -m\,g\,r\,(\cos\phi_0 + \sin\phi_0) \end{aligned} \tag{4.44}$$

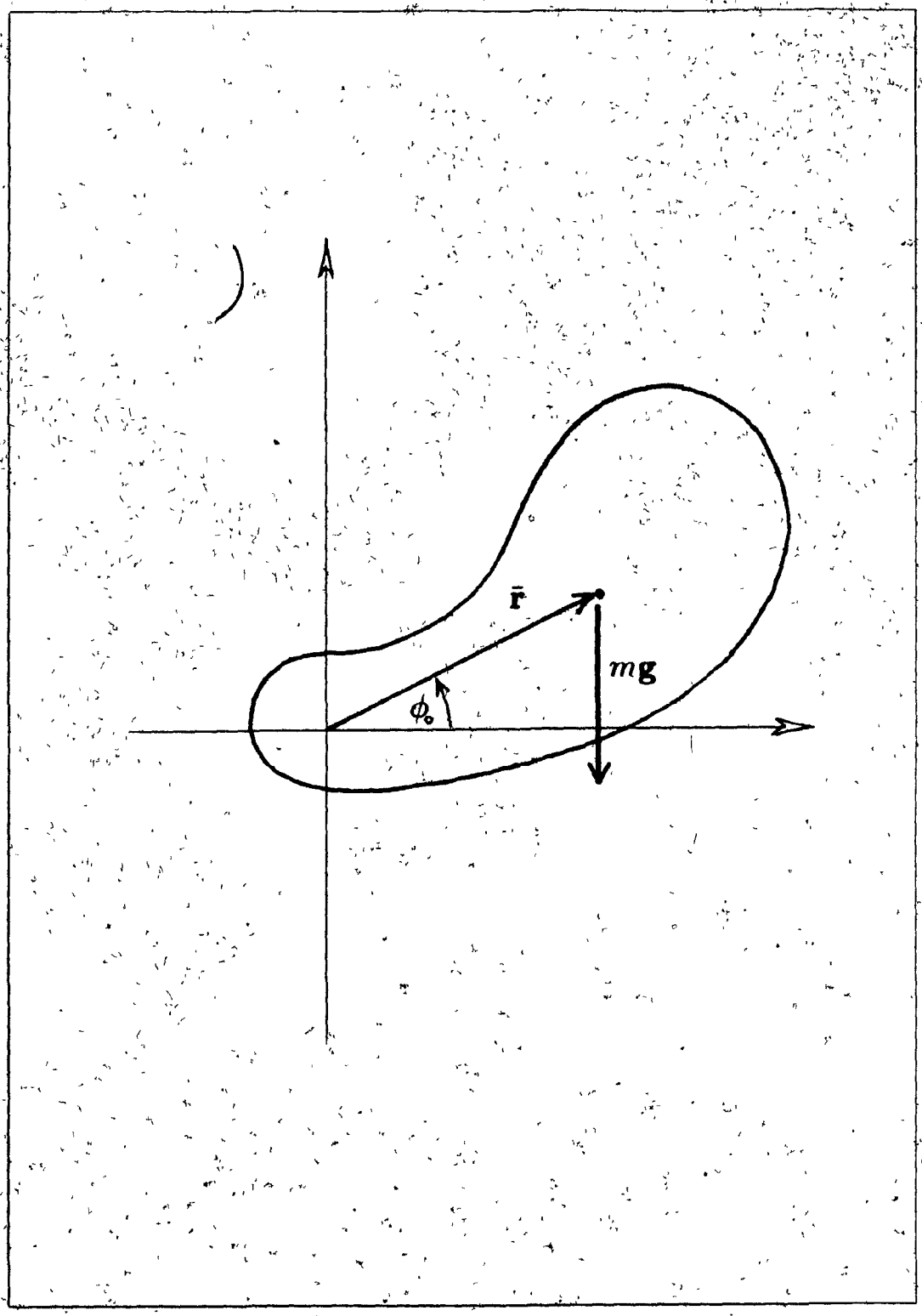


Figure 4.3 Gravity loading on an arbitrary rigid body.

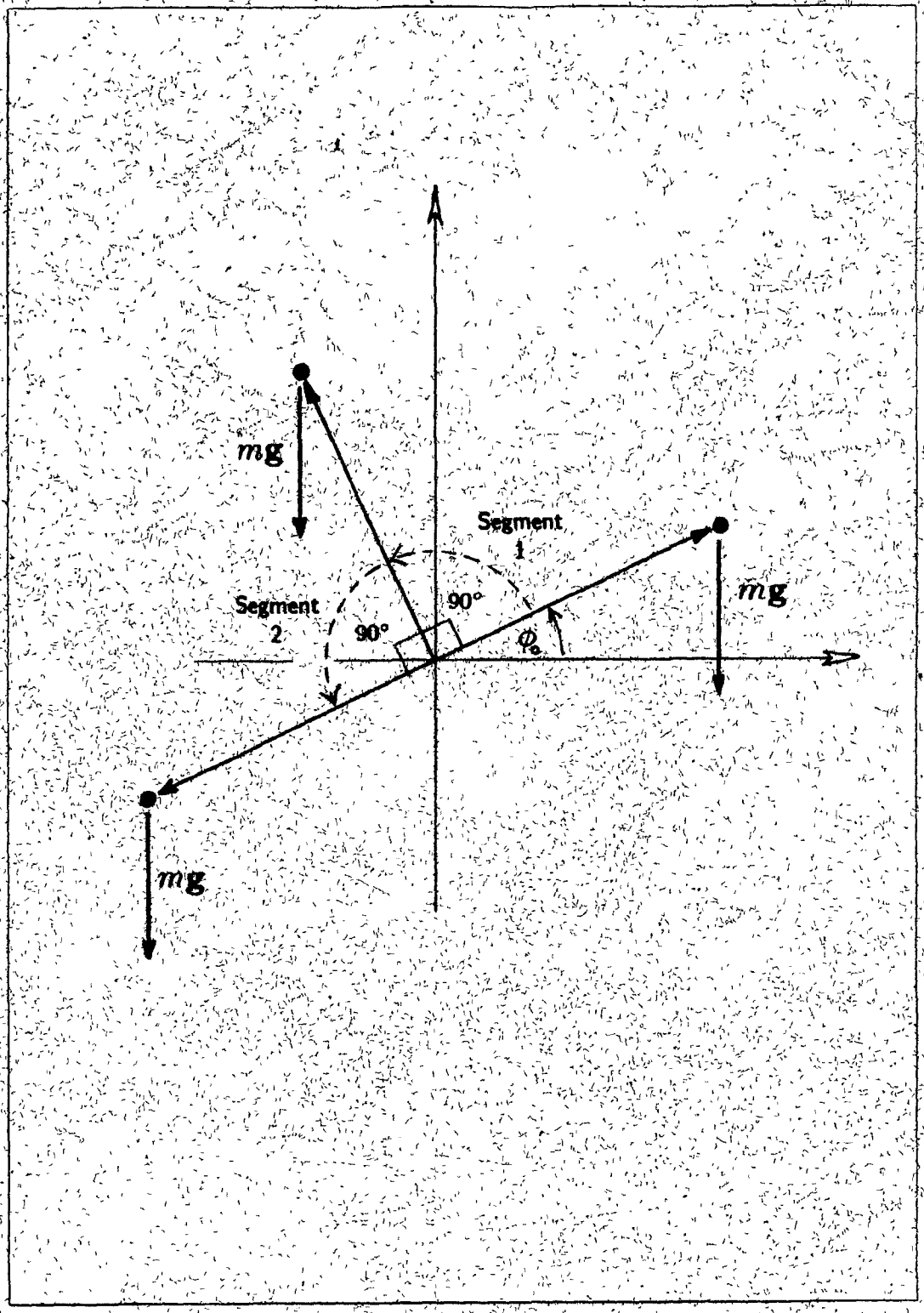


Figure 4.4 Gravity loading on the body as it is moved through two 90° angles

The work against gravity for the whole motion, $W_g = W_{g1} + W_{g2}$, is then

$$W_{gt} = -2m\,g\,r\,(\sin\phi_0) \tag{4.45}$$

The difference between the gravitational work done in each segment is

$$W_{g1} - W_{g2} = 2m\,g\,r\,(\cos\phi_0) \tag{4.46}$$

Now define τ_0 to be the torque loading on the joint at angle ϕ_0, and τ_1 to be the loading at the midpoint of the motion ($\phi_0 + 90°$). From $\tau_0 = -m\,g\,r\cos\phi_0$, and $\tau_1 = m\,g\,r\sin\phi_0$, we obtain*:

$$\begin{aligned}\tau_0 &= -2(W_{g1} - W_{g2})\\ \tau_1 &= -2(W_{g1} + W_{g2})\end{aligned} \tag{4.47}$$

For a particular joint, (4.24) may be used to find sets of joint angles for which the associated values of τ on the LHS of (4.47) are simple combinations of the gravity loading coefficients. The problem then becomes one of measuring W_{g1} and W_{g2}. The total work done along each segment can be measured, and is denoted by W_1 and W_2. This work is due both to the gravity loading discussed above, and the work done against friction, W_{f1} and W_{f2}, as expressed by

$$\begin{aligned}W_1 &= W_{f1} + W_{g1}\\ W_2 &= W_{f2} + W_{g2}\end{aligned} \tag{4.48}$$

If the assumption is made that $W_{f1} = W_{f2}$, then these equations can be subtracted,

$$W_1 - W_2 = W_{g1} - W_{g2}$$

which with (4.47) yields τ_0.

In practice, W_{f1} and W_{f2} are not exactly equal, as was mentioned in the discussion above on friction. This problem can be partly corrected for by estimating the friction work with an independent measurement. The same measurements are repeated, only with the system repositioned so that the work W_g' done against gravity is the same in each

* The author was reminded here of the slightly counterintuitive fact that torque and energy have the same units.

segment. (The most straightforward way to do this is to reorient things so that the joint axis is parallel to **g**, making the work done against gravity zero along any part of the path.)

Measurements in this new position give a set of new values W', expressed by

$$\begin{aligned} W'_1 &= W'_{f1} + W'_g \\ W'_2 &= W'_{f2} + W'_g \end{aligned} \tag{4.49}$$

Ideally, the work W_f done by friction in the first set of measurements is as close as possible to the work W'_f done by friction in the second set of measurements. Representing $W'_f - W_f$ as δW_f, and subtracting (4.48) from (4.49), yields

$$\begin{aligned} W'_1 - W_1 &= \delta W_{f1} + W'_g - W_{g1} \\ W'_2 - W_2 &= \delta W_{f2} + W'_g - W_{g2} \end{aligned} \tag{4.50}$$

If we assume that $\delta W_{f1} = \delta W_{f2}$, then (4.50) can be combined to get

$$W'_2 - W_2 - W'_1 + W_1 = W_{g1} - W_{g2} \tag{4.51}$$

Given that the δW_{f_i} are small to begin with, this approximation is not too severe*. $W_{g1} - W_{g2}$ can hence be estimated directly from the measured work and related to the initial torque τ_0 by equation (4.47).

The measurement procedure itself consists of finding appropriate joints to act as o_i and then fixing the other joint angles so that τ_0 is a simple combination of the gravity loading coefficients.

The start positions used for performing the CVaRL measurements are described in Table 4.7. The measurement results are given in Table 4.8. For each measurement, the robot was put into the indicated start position, the specified joint was moved back by 90° to ϕ_0, and was then driven along the interval $[\phi_0, \phi_0 + 180°]$. The third entry in the table is the value of τ_0 at angle ϕ_0, solved for in terms of the loading coefficients from equations (4.24). The measured value of $W_{g1} - W_{g2}$ is given, along with the computed c_{l_i} values.

It can be seen that all the measured coefficient values are quite close, even when the measurements are made using different joints.

* It is possible to arrange the experiment so that the gravity loadings in each segment change identically between the two measurements, in which case the assumption is very good.

Position name	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6
j5	90°	0	0	90°	0	45°
j4	180°	90°	-180°	0	90°	45°
j3A	270°	90°	-90°	90°	0	45°
j3B	270°	90°	-90°	90°	90°	45°
j2	270°	90°	0	90°	90°	135°

Table 4.7 Robot joint positions used in measuring gravity loadings

Position name	Joint (ϕ)	τ_0	$(W'_{g2} - W'_{g1})/2$	Result
j5	5	c_{l5}	−.191	$c_{l5} = -.191$
j4	4	c_{l5}	−.193	$c_{l5} = -.193$
j3B	3	c_{l3}	−1.744	$c_{l3} = -1.744$
j2	2	c_{l2}	5.509	$c_{l2} = 5.509$
j3B	2	$c_{l2} - c_{l3}$	7.288	$c_{l3} = -1.779$
j3A	3	$c_{l3} + c_{l5}$	−1.938	$c_{l5} = -.194$

Table 4.8 Gravity loading measurements

Coefficient	Value	Friction error
c_{l5}	$-.192 \pm .005$	.010
c_{l3}	$-1.762 \pm .04$	.010
c_{l2}	$5.509 \pm .15$	.080

Table 4.9 Final results for the gravity loading coefficients

The averaged values for the loading coefficients are given in Table 4.9. The uncertainity associated with each measurement was arrived at by considering a combination of (1) the basic uncertainities in the measurements, and (2) the differences in value that were found

when the same measurements were repeated in the opposite direction. The friction error is the amount that the measurement changed by when friction was *not* compensated for by the independent measurements described by (4.49).

4.6 Applications

The formulations described above can be used to

- ◇ Determine the net torque acting on a joint (presently implemented with equation (4.37)).
- ◇ Have the joint exert a desired net torque (presently implemented with equation (4.38)).

The use of motor currents to estimate torque results in large static friction errors whenever the joint is stationary, the magnitude of which are indicated by Table 4.6. Both torque observations and requests are also limited by the uncertainity in the Coloumb friction at any given instant (Table 4.4). To exert a desired torque when the joint is stationary, a correction equal to the maximum static friction (Table 4.5) must be applied to permit torque transmission.

An RCI program was written which used these methods to create any desired net joint torque while an observer manually held the arm under control*. Setting all the joint torques to 0 resulted in a "zero friction" effect, which caused the manipulator to "bounce around" unless held in one place. To users of the program, the effect was convincing except when the joints were stationary and the static friction caused some sticking.

Another application was known as the "zero gravity" program, where the gravity loadings of the joints were counteracted to permit a user to be able to easily move the "limp" robot into different positions. In this program, static friction was useful in keeping the joints still, and so was not compensated for.

4.7 Force Control in Cartesian Space

We conclude this chapter by discussing the use of joint torques to perform Cartesian force control.

* It should be noted here that the PUMA 260 is not a large robot.

Forces in joint space are related to forces in Cartesian space by the manipulator Jacobian J (equation (3.2)):

$$\mathbf{t}_a = \mathbf{J}^T \mathbf{q} \tag{4.52}$$

To exert a desired force $\mathbf{q}$ one may calculate $\mathbf{t}_a$ using (4.52), and then calculate the required current signals using (4.38).

It is also possible, in lieu of explicit Cartesian force sensors, to invert the calculation and determine the forces $\mathbf{q}$ from the joint torques. In some applications, one is concerned with determining whether or not the observed force $\mathbf{q}$ exceeds a certain limit value*. A method for doing this has been described in [Paul 81], Chapter 9:

Assume that the force limit can be described by a force vector $\mathbf{q}_l$. Determine, from (4.52), the torques $\mathbf{t}_l$ which correspond to this limit. Then for each observed torque vector, $\mathbf{t}_a$, the limit is said to be exceeded if the normalized projection of $\mathbf{t}_a$ onto $\mathbf{t}_l$ exceeds one:

$$\frac{\mathbf{t}_a^T \mathbf{t}_l}{|\mathbf{t}_a|^2} > 1 \tag{4.53}$$

This method has two drawbacks. First, combining the force limit information into one vector reduces the generality of the force limit that may be specified. The second problem is that since the mapping from Cartesian space to joint space is not orthogonal, the method is only approximate and prone to false results.

As an example, consider the simple two degree of freedom manipulator in Figure 4.5. A two dimensional Cartesian coordinate system is defined at the end. If we assume that each link has a length of one, then the relationship between Cartesian forces and joint torques, in the position shown, is

$$\begin{aligned} \tau_1 &= f_x + f_y \\ \tau_2 &= f_y \end{aligned} \tag{4.54}$$

$\mathbf{J}^T$ for this robot is hence given by

$$\mathbf{J}^T = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{4.55}$$

* This occurs, for example, in performing "guarded motions", where the robot is moved along a path until an obstruction is sensed by observing a large force in a particular direction. This is discussed in Chapter 5.

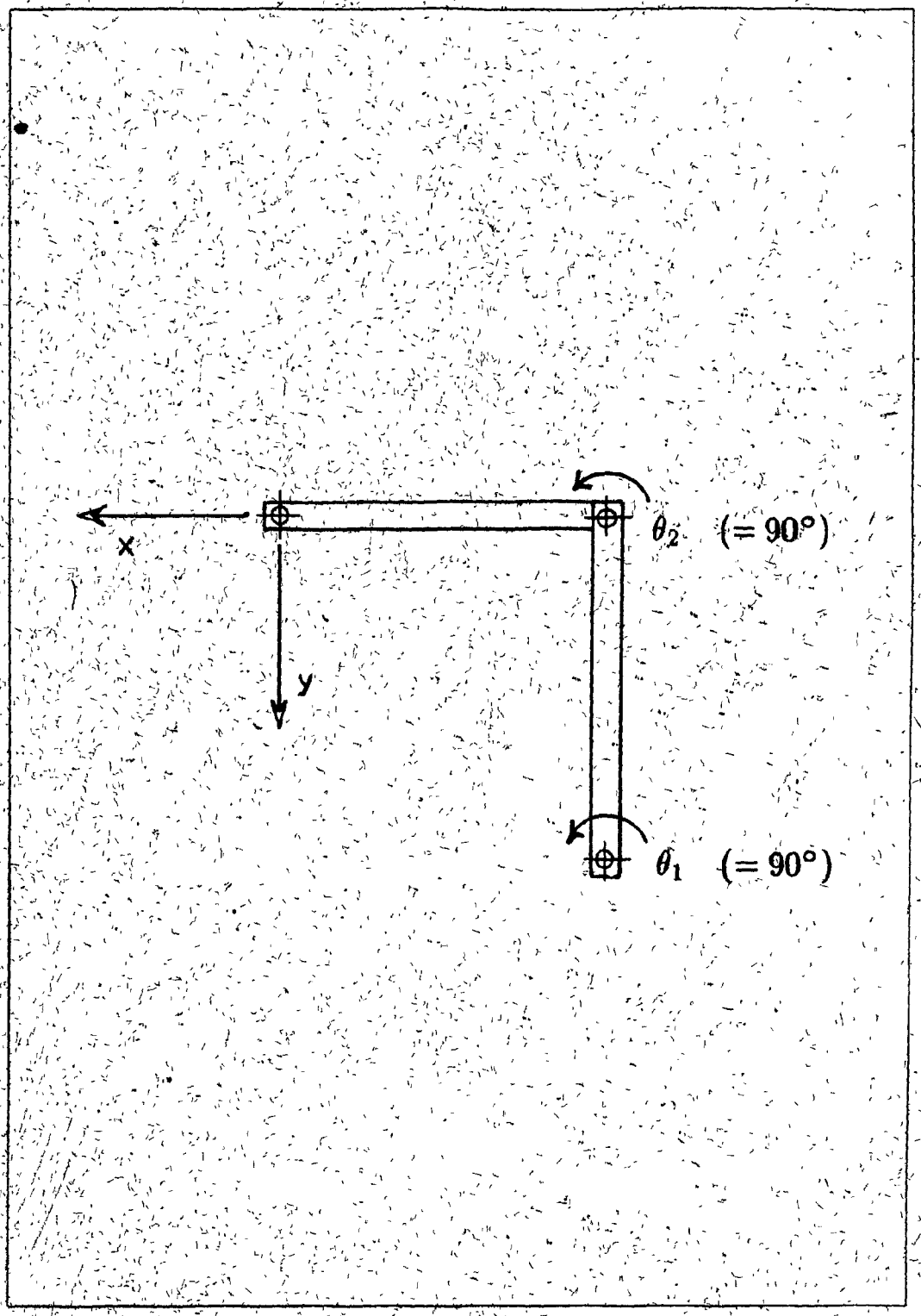


Figure 4.5 A simple manipulator

Now suppose we set a force limit of $f_x = -F_l$ and are unconcerned about f_y. The torque vector $\mathbf{t}_l$ corresponding to this is given by $(F_l, 0)$. However, it is possible to trigger condition (4.53) by exerting a force greater than F_l in the y direction alone.

A more straightforward way to determine if a force limit is exceeded is to perform a direct inverse of (4.52)

$$\mathbf{q}_a = \mathbf{J}^{-1T}\,\mathbf{t}_a \tag{4.56}$$

The observed force values $\mathbf{q}_a$ have now been recovered and can be used as a basis for any sort of decision. Computing (4.56) can be done reasonably efficiently, as was seen in the previous chapter. A problem that must be considered, however, is that as $\mathbf{J}$ approaches a singularity, the computation becomes much more sensitive to errors in $\mathbf{t}_a$. In general, if $\Delta\mathbf{t}_a$ is the error in the measurement of $\mathbf{t}_a$, and $\Delta\mathbf{q}$ is the corresponding error in the computed value of $\mathbf{q}$, we have

$$\frac{\|\Delta\mathbf{q}\|}{\|\mathbf{q}\|} \leq \mathrm{cond}\,(\mathbf{J}^T)\,\frac{\|\Delta\mathbf{t}_a\|}{\|\mathbf{t}_a\|} \tag{4.57}$$

where *cond* denotes the condition number of a matrix. The minimum condition number for $^4\mathbf{J}$ on the PUMA 260, based on a random sampling of positions, is about 1.8. As the matrix approaches a singularity, of course, the condition number becomes infinite. Since $\mathbf{J}^{-1}$ is known analytically, it is possible to minimize the error amplification by paying attention to the individual terms of equation (4.56), and scaling or discarding them if necessary.

It should be noted in passing that the accuracy of the first limit method (4.53) is also restricted by the condition of the Jacobian, in that this is a measure of the degree to which a matrix is non-orthogonal. However, the accuracy of the second method (4.56) is proportional to the precision of the initial torque measurements, which is not true for the first method.

The errors due to friction, coupled with the error introduced by the Jacobian, make the use of motor currents (equation (4.37)) inadequate for reliable Cartesian force sensing.

4.8 Summary

The "static force control" of a manipulator is defined by the dynamics equation in cases where the velocities and accelerations are small enough to justify ignoring the complex

inertia, centrifugal, and Coriolis terms. This leads to a very simple formulation involving only the applied torques, gravity loadings, and friction terms.

A simple method of measuring force control parameters for a robot has been described which is based on using the joint motor currents to estimate the work done as the joint is driven along a path at a constant velocity. The path may be chosen so that the work done is a function of the desired parameter. It was sometimes convenient to perform two such measurements and combine the results in order to remove the friction terms.

In particular, this method proved useful in measuring the torque sensitivity, friction, and gravity loading coefficients for the PUMA 260 to an accuracy which we estimate to be as good as 1 percent.

Applications of static force control include making the manipulator weightless for ease of teaching positions, having it exert forces and comply with its environment, and enabling it to detect forces which are acting on it. Local uncertainties due to the friction limit the effectiveness of basing this control on motor currents. Force control may be performed in Cartesian coordinates by using the manipulator Jacobian to map between joint space and Cartesian space, although the accuracy of detecting forces in this way is further restricted by the condition of the Jacobian.

Chapter 5

RCCL: CONTROL in CARTESIAN COORDINATES

5.1 Overview

RCCL* is a collection of subroutines and data structures specifically designed for the creation of robot motion control programs. The intended user is someone familiar with robotics concepts who wishes to implement higher level or more user oriented systems with the aid of a complete high level language and operating system.

Robot motion requests in RCCL are specified in a Cartesian coordinate system, in which the programmer uses homogeneous transforms to indicate positions. In this way, the system bears some resemblance to the language PAL [Takase, et al. 79], which was an earlier development at Purdue University. While the use of a Cartesian coordinate space to specify positions has become a commonplace feature in robot programming languages [Bonner and Shin 82, Lozano-Perez 82], a more distinguishing feature of RCCL is that it is not a language in itself, but a package of library routines.

This characteristic makes RCCL somewhat language independent: the package can be hosted by any language which supports function calls and the definition of data structures. Using a compiled language results in a considerable speed improvement over more commercially available interpreted environments. All of the features of the host language and operating system are available for use, and extensions may be added by defining other library routines. CVaRL's original interest in RCCL was stirred by a need to replace the

* Robot Control C Library

less powerful languages provided by the manufacturers of the Unimation and Microbo robot controllers

The present version of RCCL is written for the language C, and is implemented, for the PUMA 260*, using the RCI interface, discussed in Chapter 2. RCCL applications execute as an RCI control program (Figures 2.1 and 2.2) with a built-in trajectory generator which runs at the control level. This trajectory generator is implemented as a predefined RCI control function, and hence runs at a periodic sample rate† RCCL provides a set of library routines at the planning level which build motion requests and send them to the trajectory generator. Most of the user's code is written at the planning level, using these motion control routines, although he/she may define functions which are executed by the trajectory generator, at the control level, to monitor or modify robot motions. The RCI control level is referred to in this chapter as the *trajectory* level.

The objectives of this chapter are to

- ◇ Describe, somewhat abstractly, the features provided by RCCL.
- ◇ Give a brief illustration of the programming interface.
- ◇ Describe improvements which are needed to make the package more useful. In particular, a generalization to the motion specification mechanism is proposed which allows for conditional execution of motion segments, and a proposal is made for a new synchronization mechanism.

A discussion of the trajectory level implementation of the RCCL features is given in Appendix E. Additional descriptions of the philosophy of the present system may be found in [Hayward 83B], and the user interface is described in detail in [Hayward and Lloyd 85].

5.2 Description of the Motion Control Mechanism

RCCL allows a robot's motion to be specified as a series of path segments which are linear in some coordinate space (*e.g.*, Cartesian or joint coordinates). These path segments are determined by the planning level task, and then presented to the trajectory task, for execution. Each motion segment is specified by a set of parameters which form a *motion*

---

* An implementation for the Microbo is presently in progress [Kossman 85]

† The default RCI sample rate of 36 Hz is most commonly used

request packet. these packets are then placed on a *motion queue* from which they are removed and serviced by the trajectory generator (Figure 5.1). Using a motion queue has the advantage of allowing the planning and control levels to operate in parallel, with the disadvantage, of course, that explicit synchronization mechanisms (section 5.3.3) must be supplied to coordinate the two different levels.

An arbitrary path in space may be specified as a series of path segments. In cases where the manipulator does not stop at the intermediate points, the trajectory generator provides a transition phase between adjacent path segments (Figure 5.2), to avoid velocity and acceleration discontinuities. Different ways are known for computing this transition: one method involves forming the entire trajectory by splining together third and fourth order polynomials [Chand and Doty 83], though at the expense of straight line motion; the method used in RCCL involves fitting a quartic polynomial transition between the adjacent linear path segments [Paul 81]. This prevents the manipulator from actually moving through the intermediate points, except when it stops at one. Stopping may be accomplished by moving to the same point twice, as is done with point C in Figure 5.2.

When the trajectory generator is nearing the end of its current path segment, it grabs the next motion segment off the queue and computes the transition to it. If the motion queue is found to be empty, the previous motion request is simply repeated, which effectively causes the manipulator to halt at the position associated with that motion.

Each motion segment request is characterized by the following parameters:

1. A *position* in space for the robot to move to.
2. A *trajectory mode* specification, which determines the coordinate space in which the intermediate path is to be computed. In the present implementation, the choices are for either straight lines in Cartesian space, (Cartesian mode), or straight lines in joint space (joint mode).
3. A *velocity* specification, which may be given explicitly as a speed, or implicitly as the length of time the motion is to take.
4. A *transition time*, which determines the length of the inter-segment transition period, described above.
5. *Adaptive control* specifications. These are the "hooks" by which a path segment may be modified at run time, and will be discussed in detail below.

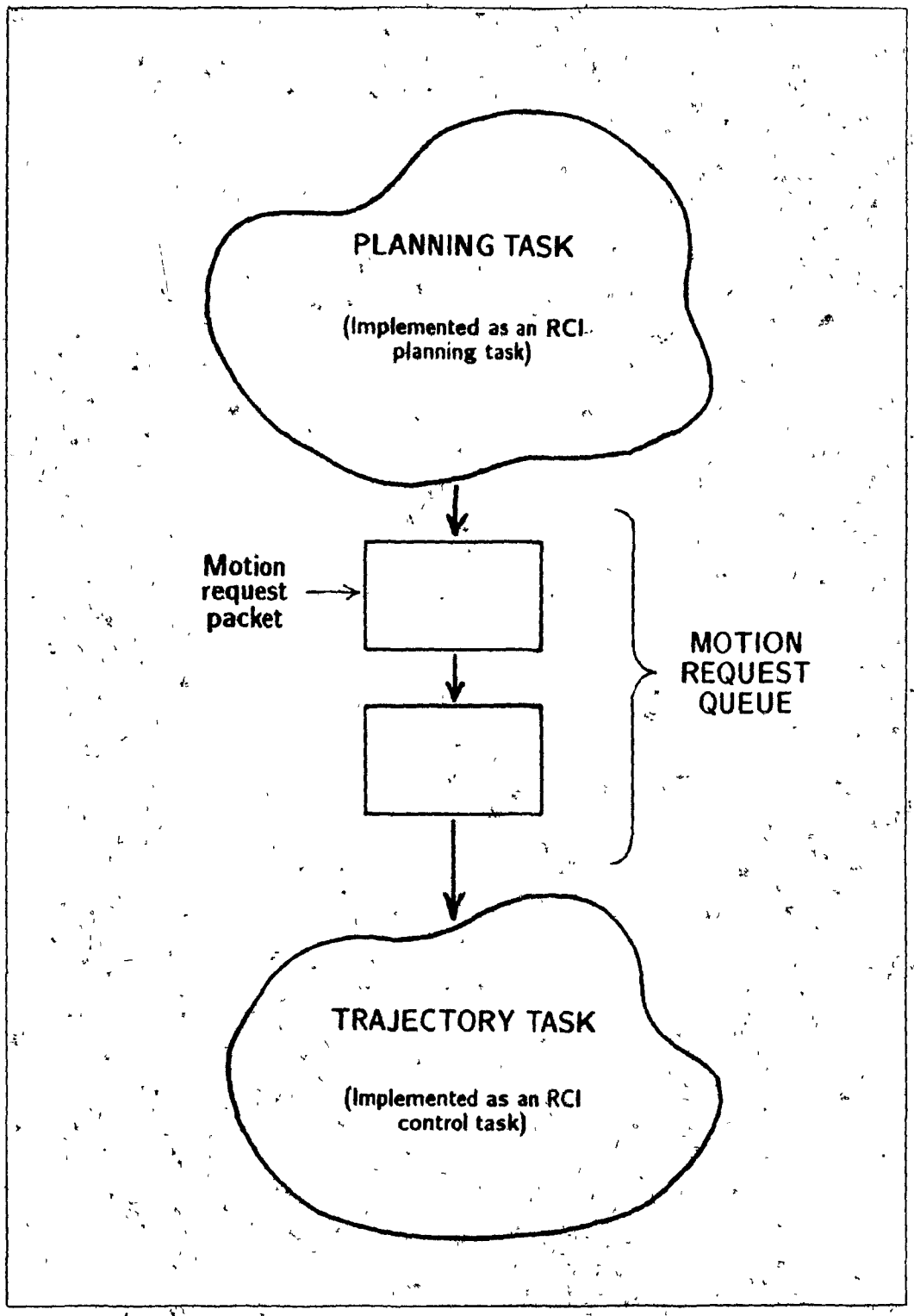


Figure 5.1 RCCL task structure.

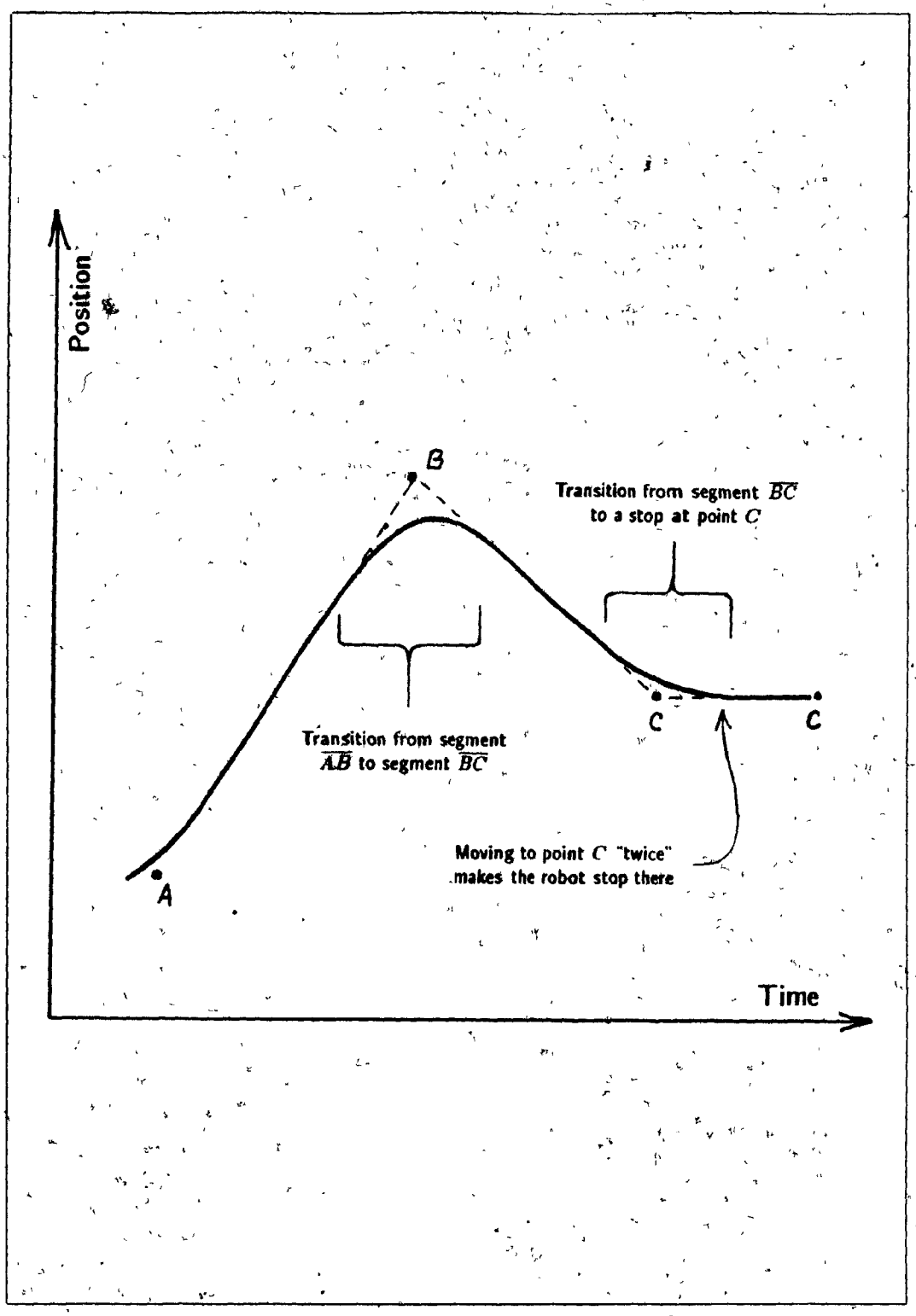


Figure 5.2 Path segments and the transitions between them

5.3 The Programming Interface

5.3.1 Specification of Positions

The data types used to determine locations in space are coordinate transforms (type *TRSF*) and position equations (type *POS*). *TRSF* data structures describe a homogeneous coordinate transform, and functions are available for creating and manipulating them. *POS* data structures are defined by an equation of transforms which includes the manipulator link 6 transform $\mathbf{T_6}$ (section 3.3). As an example, consider:

$$\mathbf{Z}\,\mathbf{T_6} = \mathbf{B} \tag{5.1}$$

If $\mathbf{Z}$ and $\mathbf{B}$ are both known, then $\mathbf{T_6}$ may be solved for to determine the corresponding robot position:

$$\mathbf{T_6} = \mathbf{Z}^{-1}\,\mathbf{B} \tag{5.2}$$

By breaking position definitions up into a series of transformations, it becomes possible to modify or redefine positions simply by changing an appropriate transform component.

One of the transforms in the equation defines the *tool* frame, describing the location of the robot tool, or end effector. For instance, if there is a transform $\mathbf{T_L}$ describing the position of a tool relative to $\mathbf{T_6}$, then this transform also defines the tool frame in the equation

$$\mathbf{Z}\,\mathbf{T_6}\,\mathbf{T_L} = \mathbf{B} \tag{5.3}$$

Aside from being useful in general, the tool frame is required by some RCCL features (such as compliance and force limit checking, section 5.4) as a reference frame for doing computations. In cases where the tool frame is the same as the link 6 frame, $\mathbf{T_6}$ is used to define the tool frame.

Positions are created using the function *makeposition()*; it takes a variable length argument list consisting of pointers to *TRSF* data types, along with the two "key" arguments *EQ* and *TL*, and returns a pointer to a newly allocated *POS* data structure. *makeposition()* calls to build positions such as those described in (5.1) and (5.3) would look like:

```
position1 = makeposition (z, t6, EQ, b, TL, t6);
```

```
position2 = makeposition (z, t6, t1, EQ, b, TL, t1);
```

where *position1* and *position2* are pointers to *POS* data structures, and *z*, *t1* and *b* are pointers to previously declared *TRSF* structures. The variable t6 is internal to RCCL and represents T_6. The special argument *EQ* is used to indicate the = sign in the transform equation, while *TL* is used to mark the end of the equation. The argument which follows *TL* is the transform used to define the tool frame.

5.3.2 Requesting Motions and Setting their Parameters

The primitive *move(position)* queues a motion request packet whose destination is specified by the *POS* structure *position*.

The parameters associated with the motion request packet are set prior to the issuing of the *move()* directive, using primitives such as

```
setvel (translational_speed, rotational_speed);
```

— explicitly sets the velocity

```
setime (transition_time, travel_time);
```

— implicitly sets velocity by giving the travel time for the whole motion, as well as the inter-segment transition time.

```
setmod (trajectory_mode);
```

— sets the trajectory mode (currently selectable to either *joint* or *Cartesian*).

5.3.3 Synchronization

Since motions occur asynchronously with respect to the main program, some explicit synchronization mechanism must be provided. RCCL presently does this by using an event flag attached to the destination position of a motion request. Every time the trajectory generator completes a motion request, it decrements this event flag. The planning level may synchronize itself with the completion of a particular motion request by using the primitive *waitfor()*, which increments an event flag and then waits for it to drop to a value less than one. This is done in the following example:

```
/* The event flag is contained in the "end" field of the */
/* position data structure                                */

move (position);                    /* Place the move request */

... ( computation ) ...             /* Do other things */

waitfor (position->end);            /* Wait until move completes */
```

Associating the event flag with the destination position rather than with the motion request itself is somewhat unwieldy; a more general synchronization scheme is proposed in section 5.6.3.

5.3.4 A Program Example

To illustrate the use of these primitives, a simple RCCL program, extracted from the manual [Hayward and Lloyd 85], is explained here.

```
1  #include <robot/rccl.h>              /* RCCL definitions          */
2
3  main()
4  {
5     TRSF_PTR t, r;                   /* transformation pointers   */
6     POS_PTR p0;                      /* position pointer          */
7
8     /* Allocate and initialize the transforms and position equation */
9
10    t  = gentr_trsl ("T",  250.,  250., 328.9);
11    r = gentr_rot ("R", 0., 0., 0., yunit, 180.);
12    p0 = makeposition ("P0", t, t6, EQ, r, TL, t6);
13
14    rccl_open (0,0),                 /* Initialize RCCL and ...   */
15    rccl_control (1);                /*  turn on the control      */
16
17    setvel (100, 100);               /* Set the velocity ...      */
18    move (p0);                       /*  move to the position ... */
19    move (park);                     /*  and return home.         */
20
21    waitfor (park->end);             /* Wait for motion to complete*/
22    rccl_close (1);                  /*  and then exit            */
```

```
23 }
```

This program moves the robot to a position *p0* and back again. Data types and predefined variables are declared in the file <robot/rccl.h>. The position *p0* is specified by the transform equation

$$T\,T_6 = R \tag{5.4}$$

The transforms *t* and *b* are defined, using the *gentr_* allocation primitives, to represent an $x - y - z$ translation of 250., 250., and 328.9 mm., and a rotation of 180° about the y axis (lines 10 - 11). The *rccl_open()* and *rccl_control* primitives (lines 14 - 15) call the necessary RCI primitives to initialize RCCL and start the trajectory generator. The velocity is set, and a motion is requested first to *p0*, and then back to the predefined home location *park* (lines 17 - 19). The program then waits for the trajectory generator to finish the motions, before shutting things down with *rccl_close()* and exiting.

The motion from *p0* to *park* is continuous. To make the robot stop momentarily at *p0*, two successive motions to that position could be specified:

```
move (p0);
move (p0);
```

To make it stop at *p0* for a certain length of time, the travel time for the second "stopping" motion should be explicitly set:

```
move (p0);
setime (transition_time, travel_time);
move (p0);
```

This will cause the robot to halt at *p0* for approximately *travel_time*. A primitive called *stop()* is available which combines these calls with a default *transition_time*.

```
move (p0);
stop (time);
```

5.3.5 Compiling and Linking

RCCL programs are compiled and linked in the same way as any other RCI application.

The `rcc` compiling command and the simulator program, described in Chapter 2 and Appendix A, may be used. The RCCL library is automatically referenced by `rcc`. The include file `<robot/rccl.h>` should be placed at the top of RCCL source programs to define the various data structures and system variables.

5.4 Adaptive Path Control

Much of RCCL's utility is derived from its adaptive path control features. There are four different mechanisms which permit run-time variations in the path segment trajectories.

5.4.1 Motion Termination on Force or Displacement Limit

This causes execution of the present path segment to be aborted when the end effector encounters a force exceeding a certain threshold, or wanders too far off the desired path. The next motion request packet is fetched from the motion queue. Force or displacement limits are set by the programmer using the primitive

```
limit ("specification_string", value_1, ... value_n);
```

The specification string indicates which forces, torques, or differential motions are to be monitored, and the limiting value for each is given by the remaining arguments. A limit specification is valid for the next motion only.

5.4.2 Motion Termination by a Programmer Defined Monitor Function

For each path segment, a function may be specified which monitors events in real-time and can abort the motion if desired. The next motion request packet will then be fetched from the motion queue.

The primitive

```
evalfn (monitor_function);
```

will cause *monitor_function* to be executed by the trajectory level, at the current sample rate, during the next specified motion segment.

5.4.3 Compliant Motion

Certain degrees of freedom in the tool frame of the manipulator may be designated as *compliant*, meaning that motion in these directions will be force servoed instead of position servoed. It is assumed that there is a corresponding real-world constraint against which the compliant force may act; otherwise, the manipulator may simply "take off" in the designated compliant directions. Compliance enables the robot to track a constraining surface, and apply a certain amount of force to it, rather than follow a fixed trajectory.

Compliance may be specified using the primitive

```
comply ("specification_string", value_1, ... value_n);
```

The specification string determines which degrees of freedom are to be compliance controlled, with the corresponding force or torque value given in the rest of the argument list. Degrees of freedom are taken out of compliance mode with the primitive.

```
lock ("specification_string");
```

One particular problem associated with compliant motion is the possibility of slippage. If the manipulator is tracking a surface that does not provide a stable constraint to the compliant force, the tool may "slip off" the surface. This can be detected by specifying a path deviation limit in the compliant direction; when slippage occurs, this limit will quickly be exceeded and the motion will be terminated. However, it may be desirable to then execute a different motion request other than the next one present in the queue. This problem of having the trajectory generator select different motion requests based on how the previous motion terminated requires an extension to RCCL and is examined in section 5.6.1.

5.4.4 Real-time Modification of the Destination Position

It is possible to attach functions to the destination position which alter it in real-time, causing a corresponding change in the trajectory. This is done by binding modifying functions to the individual transforms that define the position. The binding is accomplished by setting the *fn* field in the *TRSF* structure to the function address. Modifying functions

must be defined so as to take one argument, which is a pointer to the *TRSF* data type. When in use, the function is called by the trajectory generator once every sample period, using the transform it is bound to as an argument.

As a specific example, consider a case in which the `t` transform in the example of section 5.3.4 is bound to a function that makes it trace a circle. The function can be declared like this:

```
static time = 0;                    /* Sample time counter          */

trace_circle (transform)            /* Function takes 1 argument: a */
TRSF *transform;                    /*  pointer to a transform      */
{
  float angle;

  angle = 2 * 3.14159 * time;       /* Compute the circle angle and */
  transform->p.x = sin (angle);     /*  set the X and Y parts of    */
  transform->p.y = cos (angle);     /*  the transform accordingly   */

  time = time + 1;                  /* Increment the time counter   */
}
```

The binding is done in the program with the statement

```
t->fn = trace_circle;
```

In subsequent moves to *p0* (of which `t` is a component) the x and y translational components of `t` will trace out a circle at the rate of one degree per trajectory sample period, and a circular motion will be superimposed on the trajectory (Figure 5.3(a)). If the robot is "stopped" at *p0*, then the trajectory will be completely circular (Figure 5.3(b)). The function binding may be removed by setting the *fn* field to the predeclared value *const*.

The real-time modification of a transform may also be tied directly to sensory input; in this way, for example, it becomes easy to control the manipulator with a device such as a joystick.

Since transform modifying functions are called at the trajectory level, they are subject to the restrictions of all RCI control level functions, described in sections 2.2.3 and A.2.

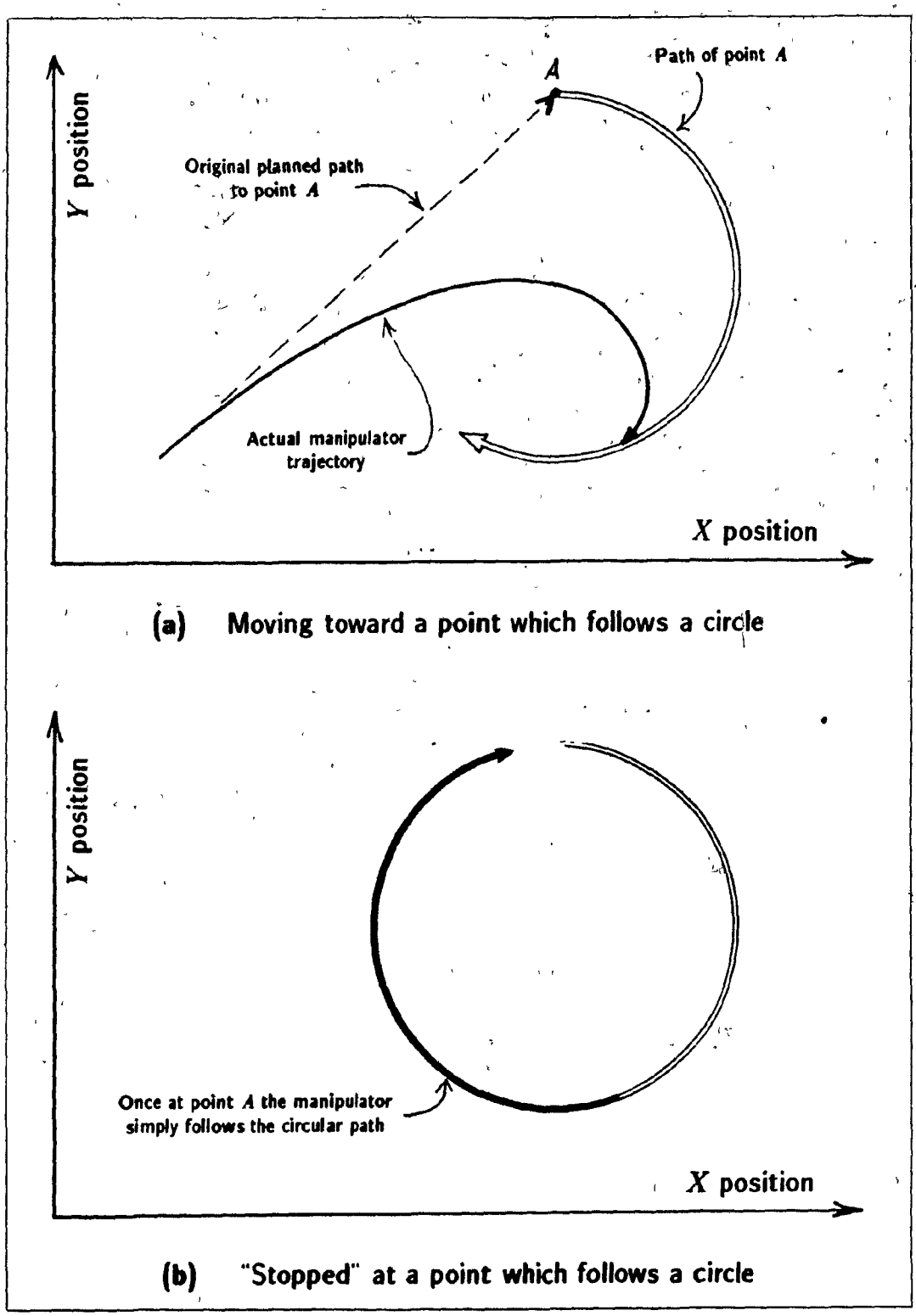


Figure 5.3 Projections of the manipulator trajectory in the XY plane as it follows a functionally defined path.

5.4.5 General Considerations

It is assumed that any adaptive changes made to a path segment are smooth enough to accommodate the dynamics of the manipulator. this is important because the trajectory generator has no control over the adaptive changes

When a transition is made to a new path segment from a path segment which has been adaptively modified. the trajectory generator takes the path modifications into account by examining the present position and velocity of the robot and using this to fit the actual path to the one originally planned (Figure 5.4)

A discussion of the implementation of the adaptive trajectory features is given in Appendix E

5.5 Assessment

At the time of this writing. RCCL has been in consistent use at CVaRL for six months The overall framework appears to be good. The primitive set is satisfactory. and can be easily extended when necessary.

Some particular restrictions are:

1. Users must be quite knowledgeable in programming and robotics concepts
2. The force control primitives presently suffer from a lack of adequate force sensing. Force control is done using joint motor currents (as described in Chapter 4 and Appendix E), which can result in large errors. mostly due to friction. Most seriously affected by this is the force limit sensing. although force feedback would also be useful in "hardening" the compliance features and making them more reliable
3. Reaction time can be slow the planning level may not always respond quickly to events occurring at the trajectory level This limitation is mainly a result of the present single processor / UNIX implementation.
4. The existing synchronization mechanism between the planning and trajectory level is somewhat awkward.

The computational resources required to generate RCCL trajectories are of interest. Table 5.1 lists the CPU time required per trajectory cycle for several different trajectory modes. A brief description of how these trajectories are computed is given in Appendix E.

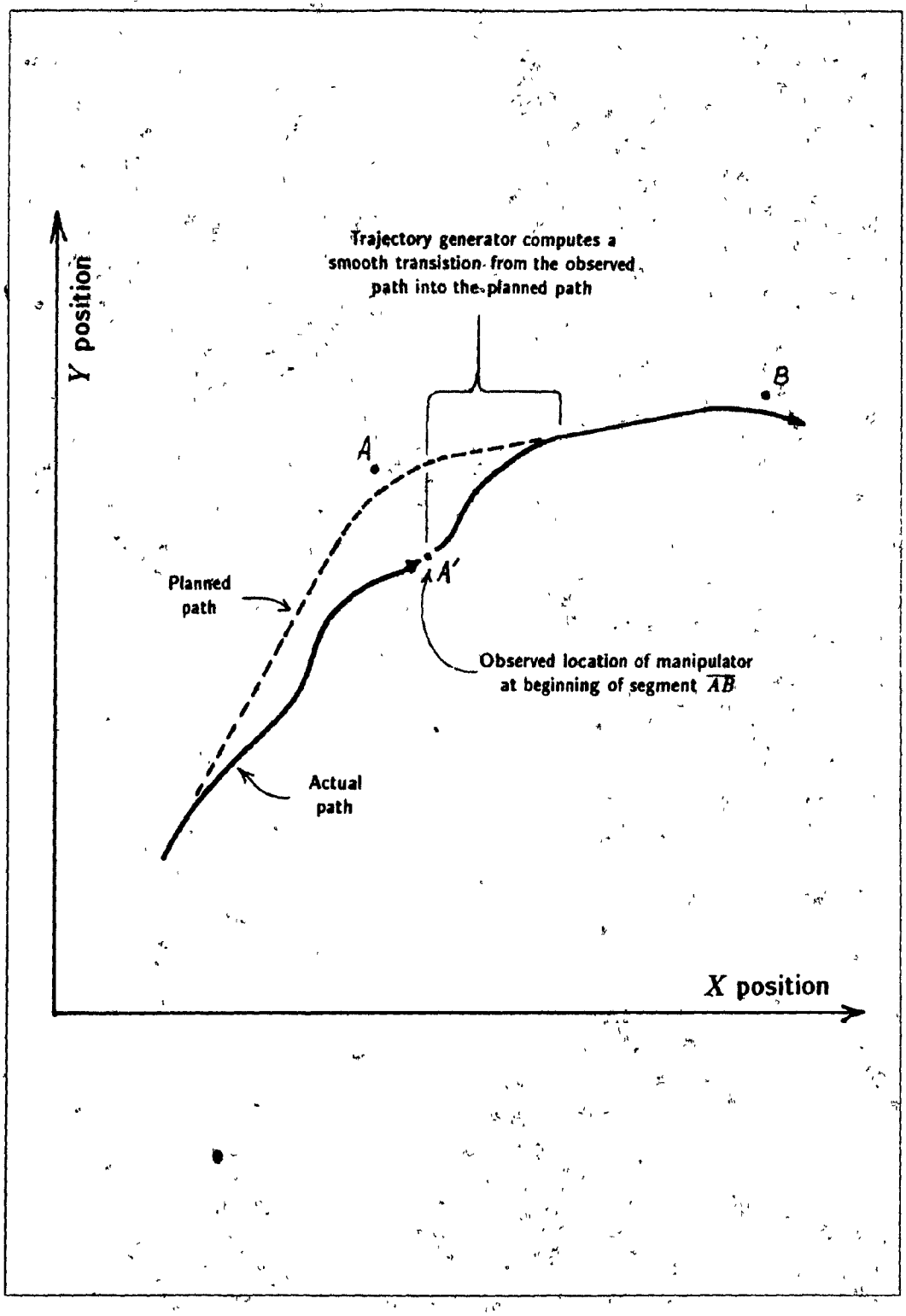


Figure 5.4 Transitioning from a modified path segment

Trajectory mode	Time
Joint	12.0
Cartesian	15.5
Cartesian with joystick-driven functional transform	18.0
Cartesian with force limit detection	19.0
Cartesian compliance with motion limits and 2 functional transforms	28.0

Table 5.1 CPU time per trajectory cycle for different trajectory modes, on a VAX 750 with floating point accelerator (milliseconds)

5.6 Proposed Improvements

Usage of RCCL has underscored the need for several enhancements, which are described here*:

1. *Conditional response* – enabling different motion segments to be executed depending on how the execution of the preceding motion segment terminated.
2. *Multiple robots* – generalizing the motion request mechanism to handle multiple manipulators.
3. *Synchronization* – improving the synchronization mechanism.

Each of these will be discussed in the following sections. In any robot control environment, these issues are intimately connected to the way in which the motion request mechanism operates. RCCL uses asynchronous, non-blocking motion requests, which are issued from a single thread of execution. It should be mentioned, however, that there are other approaches: for instance, in the language AL [Mujtaba and Goldman 79], motion requests do not return until the motion has completed. Asynchronous activity and simultaneous motions are instead made possible since the language itself is concurrent (*i.e.*, allows parallel execution of statements). This technique is not possible in RCCL unless its host language supports concurrency. To date there has not been enough experience at CVaRL using concurrent languages to permit a conclusion as to whether or not this approach is to be preferred

* These extensions are not implemented in the present version

5.6.1 Conditional Motion Requests

RCCL motion segment requests are queued and then executed. To be able to queue at least one request in advance is useful because the trajectory generator is then able to immediately perform a smooth transition to the next motion segment (as discussed earlier) without having to wait for instructions from the planning level. Known as *continuous path motion*, this feature is present in some commercially available robot control languages (*e.g.*, VAL II, [Unimation 83]).

The termination of each motion segment is associated with a particular condition, such as normal termination, exceeding force or displacement limits, or any termination invoked by a programmer defined monitor function. Having only a single queue of path segments does not address the fact that it may be desirable to invoke *different* motions depending on the conditions with which the previous motion terminated. This can be done if the planning level waits for the end of the motion in question and then selects the appropriate motion segment, but the time delay involved in invoking and waiting for the planning level, particularly under UNIX, may allow several control cycles to elapse and thus prevent a smooth transition.

The problem is related to the reflex mechanism in animal motor control systems. Suppose a motion segment trajectory is being executed. If the motion ends in one way, we wish to invoke motion request A. However, if the motion terminates in a different way, we may wish to invoke a different motion request B. It may be desirable to make this decision immediately at the trajectory level, without stopping, and without having to consult the planning level, which has a slower response time.

We could deal with this problem by allowing the planning level to specify, in advance, not simply a queue of motions, but a tree of possibilities (Figure 5.5(a)), along which the trajectory generator steps depending on how the various motions terminate. Unfortunately, it would be unwieldy to manage such a feature in the context of RCCL's current host language*: to ensure uninterrupted motions, it would be necessary for the planning level to constantly add motion requests to all leafs of the tree which were still active. It is also not certain that specifying an extensive motion request tree in advance is necessary

* Some consideration has been given to embedding RCCL in LISP, which may provide a more natural environment for this sort of specification, but that is beyond the scope of the present work.

or desirable.

Since the main objective of advance motion planning is to provide, in addition to parallelism, the continuity of trajectories, a simplified "pruned tree" structure (Figure 5.5(b)) could suffice. Motion requests are queued as usual. Each motion request, however, may be followed by *conditional motion requests* which specify different motion segments to be invoked upon different termination conditions. If a conditional motion is in fact chosen, then all other motion requests are deleted from the queue, and the planning level will have until the conditional motion completes to notice the fact and determine the next motion request. The tree is kept "pruned" since conditional motions may not have conditional motions specifications of their own. A primitive for requesting conditional motions is suggested in section 5.6.2.

Other robot control languages have incorporated features to execute fixed procedures when certain conditions are met. AL allows an arbitrary program statement to be executed whenever a certain condition is satisfied during motion execution. This is declared using the *ON* clause in the AL motion request statement:

ON <cond> *DO* <statement>

The condition <cond> will be watched for the duration of the motion; if it occurs, then <statement> is executed. Any subsequently desired motion request may be contained in <statement>.

An equivalent to the *ON* clause is difficult to provide under the existing RCCL system. Since the planning and trajectory levels operate asynchronously, an <action> would either have to be executed in the form of an interrupt at the planning level (whose response would probably be too slow under UNIX), or executed by the trajectory level, which would require allowing motion requests to be made from this level, with the ensuing problem of informing the planning level of the action taken. It is here, then, that the RCCL model becomes limited by the operating system and language it is based on (such as the somewhat slow software interrupt mechanisms of UNIX and the lack of concurrency in C).

5.6.2 Extending the System to Multiple Robots

In general, different methodologies can be considered for the programming of multiple

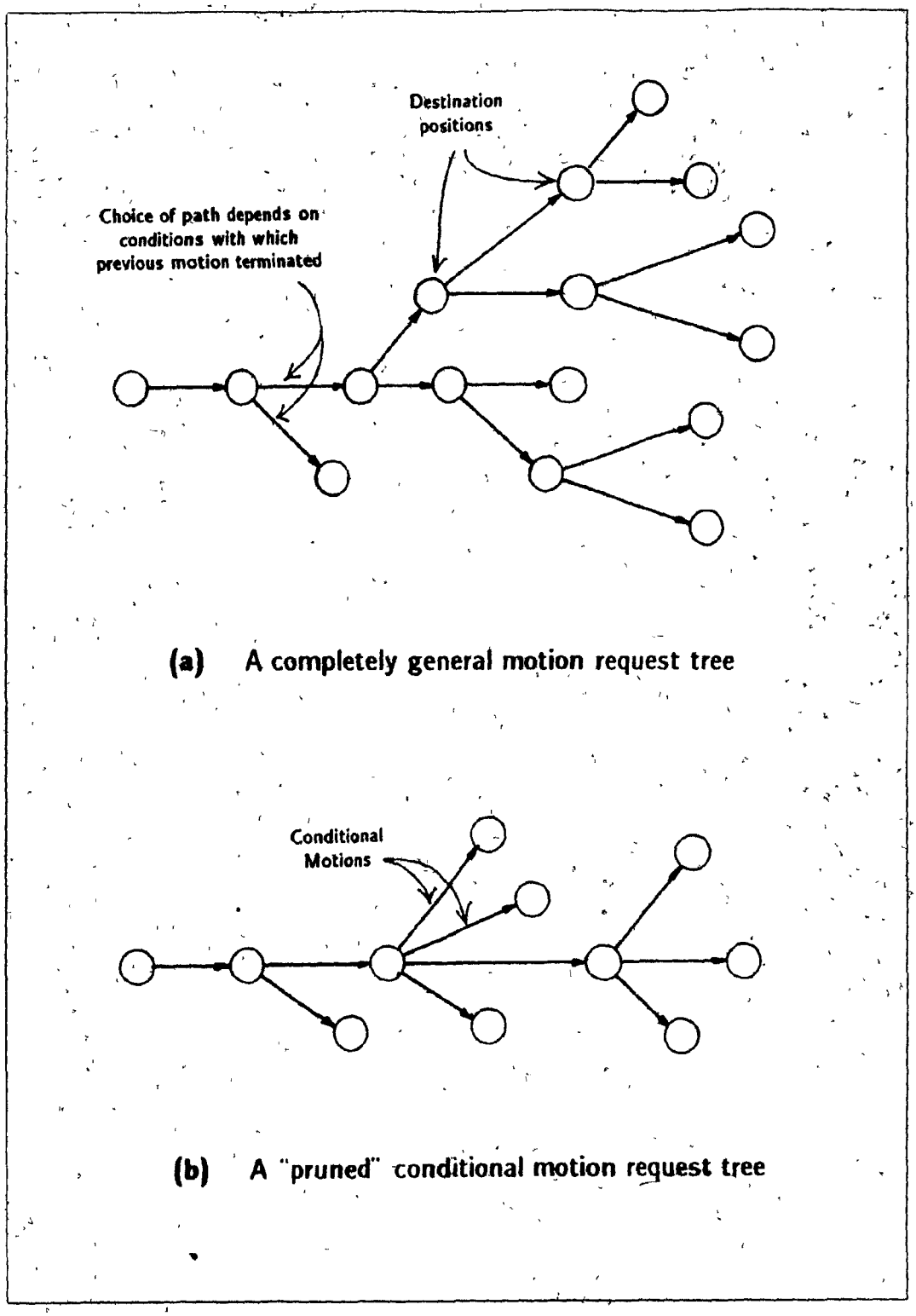


Figure 5.5 Motion request trees.

manipulators:

1. Using multiple programs, one for each robot, each having a single thread of execution and communicating with the other programs using standard interprocess communication facilities;
2. Using one program, with a single thread of execution, which issues asynchronous motion requests to the different robots.
3. Using one program, with multiple threads of execution, one for each robot, written in a concurrent language (such as AL).

The first method is available by default, given an operating system such as UNIX which readily supports multiple processes. Method 3 is not possible without basing RCCL in a concurrent high level language, whereas method 2 is rather straightforward to provide and can be combined with method 1 if desired. To implement method 2, RCCL must be modified as follows:

- ◇ Enhancement of the *move()* primitive to allow for multiple manipulators
- ◇ Enhancement of the synchronization primitives to allow for coordinated action between several manipulators.

The first item is discussed in this section. The synchronization problem is the topic of section 5.6.3.

The *move()* primitive can be expanded to take two extra arguments, one to specify the manipulator, and another pointing to a set of motion request parameters. Using an explicit data structure to store motion parameters would allow different parameter sets to be kept around for different manipulators and situations. The *move()* primitive would then take the following form:

```
move (robot, position, motion_params);
```

The primitives described in section 5.3.2 which set the motion parameters would remain unchanged, except for an additional argument specifying the parameter block, as in

```
setvel (motion_params, translational_speed, rotational_speed);
```

Conditional motion requests (section 5.6.1) can be specified by a separate primitive:

```
cond_move (condition, robot, position, motion_params);
```

where condition describes either a built-in RCCL condition value (such as "force or displacement limit exceeded"), or a termination code that can be set by a programmer defined monitor function If invoked, the conditional motion will travel to *position* with the motion parameters specified by *motion_params* The indicated conditional motion will be associated with the most recently specified standard motion For example, consider the sequence

```
move (robot_1, positionA, motion_params);
move (robot_1, positionB, motion_params);

cond_move (condA, robot_1, positionBackupA, motion_backup_params);
cond_move (condB, robot_1, positionBackupB, motion_backup_params);

move (robot_1, positionC, motion_params);
```

If the move to *positionB* terminated with either condition *condA* or *condB*, then the conditional motion to either *positionBackupA* or *positionBackupB* would be invoked, otherwise, the specified move to *positionC* would occur

The planning level could determine how and when each motion had completed by examining a status flag associated with the motion request (section 5 6 3)

In dealing with multiple robots, it remains convenient to describe locations using the transform equations discussed in section 5 3 1 To get two robots to move to the same location requires that the positional relationship between the two of them be known Suppose, for instance, that we define a position *p1* from the equation

$$\mathbf{Z}\,\mathbf{T}_6 = \mathbf{B} \tag{5.5}$$

with

```
p1 = makeposition (z, t6, EQ, b, TL, t6);
```

Assume that this is used to position one of the robots. A second robot may be moved to the same location* by defining an equivalent transform equation *p2* containing the transform

* Assumably later

$\mathbf{R}_{12}$ relating the base coordinate system of the first robot to the base coordinate system of the second robot

$$\mathbf{R}_{12}\ \mathbf{Z}\ \mathbf{T}_6 = \mathbf{B} \tag{5.6}$$

The equivalent position can be defined by

```
p2 = makeposition (r12, z, t6, EQ, b, TL, t6);
```

Moving the second robot to *p2* will place its end effector at the same location as that defined for the first robot by *p1*

Lastly, it should be noted that a multi-robot version of RCCL can be implemented on top of a multitasking version of RCI (Chapter 2), with the trajectory generation for each robot being performed by a separate RCI control task

5.6.3 A New Synchronization Mechanism

The present RCCL synchronization mechanism based on event flags associated with destination positions needs to be replaced In general, the synchronization of multiple manipulators requires that

- ◇ the motions of different robots may be started simultaneously;
- ◇ the motions of different robots may be stopped simultaneously.
- ◇ It is possible to determine the present status of a given motion request

To accomplish this, we propose removing the present synchronization mechanism completely In its place, a structure called an event flag triple (EFT) is defined, one of which may be associated with each motion request Each EFT consists of a *start* flag, a *stop* flag, and a *status* flag The *start* flag controls the beginning of the associated motion, the *stop* flag controls termination of the motion, and the *status* flag indicates the actual state of the motion The *start* and *stop* flags are set by default or by the user program, at either the planning or trajectory level, the *status* flag is read-only and is set exclusively by the trajectory level The operation of these flags is summarized by the following rules

1 The motion associated with the EFT may not begin until the start flag is set to a start code value Normally, this will be done by default when the *move()* request is issued, but an option may postpone this until some later time. After the flag has

been set, the trajectory generator will start the motion as soon as it is ready for execution. When the motion begins, the start code value is copied into the status flag.

2. Setting the stop flag halts the associated motion and initiates the next specified motion. The trajectory generator sets this flag to a default value if the motion terminates "naturally". After termination, the stop code value is copied into the status flag.

3. The stop flag has priority over the start flag. Setting the stop flag before the start flag aborts the motion completely, effectively deleting it from the queue.

4. Distinct values are used to indicate start and stop conditions.

5. All flags are initialized to a reserved value of NULL. No task may set a flag to this value.

Not wishing to place any restrictions on who may set the start or stop flags, it is important to ascertain that this protocol is safe, *i.e.*, that the motion is guaranteed to complete and the status flag will not become stuck at either a NULL or start value.

This is easy to demonstrate. The protocol is modeled with the Petri Net shown in Figure 5.6. The presence of a token in the first two places, p1 and p2, denotes the conditions where the start and stop flags, respectively, are set to non-NULL values. No assumption is made about what order the flag setting may take place, it is only assumed only that once set, flags are not reset to NULL (rule 5 above). This means tokens may appear in these places, but not disappear. It is assumed that flag settings are observed by the trajectory generator in the same order that they are set by the controlling tasks, and hence there is no need to explicitly model any time delay which may be involved. A token in places p3 through p5 depicts the three states of the motion itself, pending, active, and completed. Tokens in places p6 and p7 indicate whether the status flag has been set to a start or stop value, depending on whether p6 or p7 was transitioned to most recently. Again, it is assumed that the status flag settings are received by observing tasks in the same order as they are set by the trajectory generator, and so again any associated time delay does not have to be explicitly considered. From the reachability tree for this net (Figure 5.7), it may be seen by inspection that the two possible final markings, m8 and m9, both correspond to a completed motion, with the status flag having last been set to a stop code.

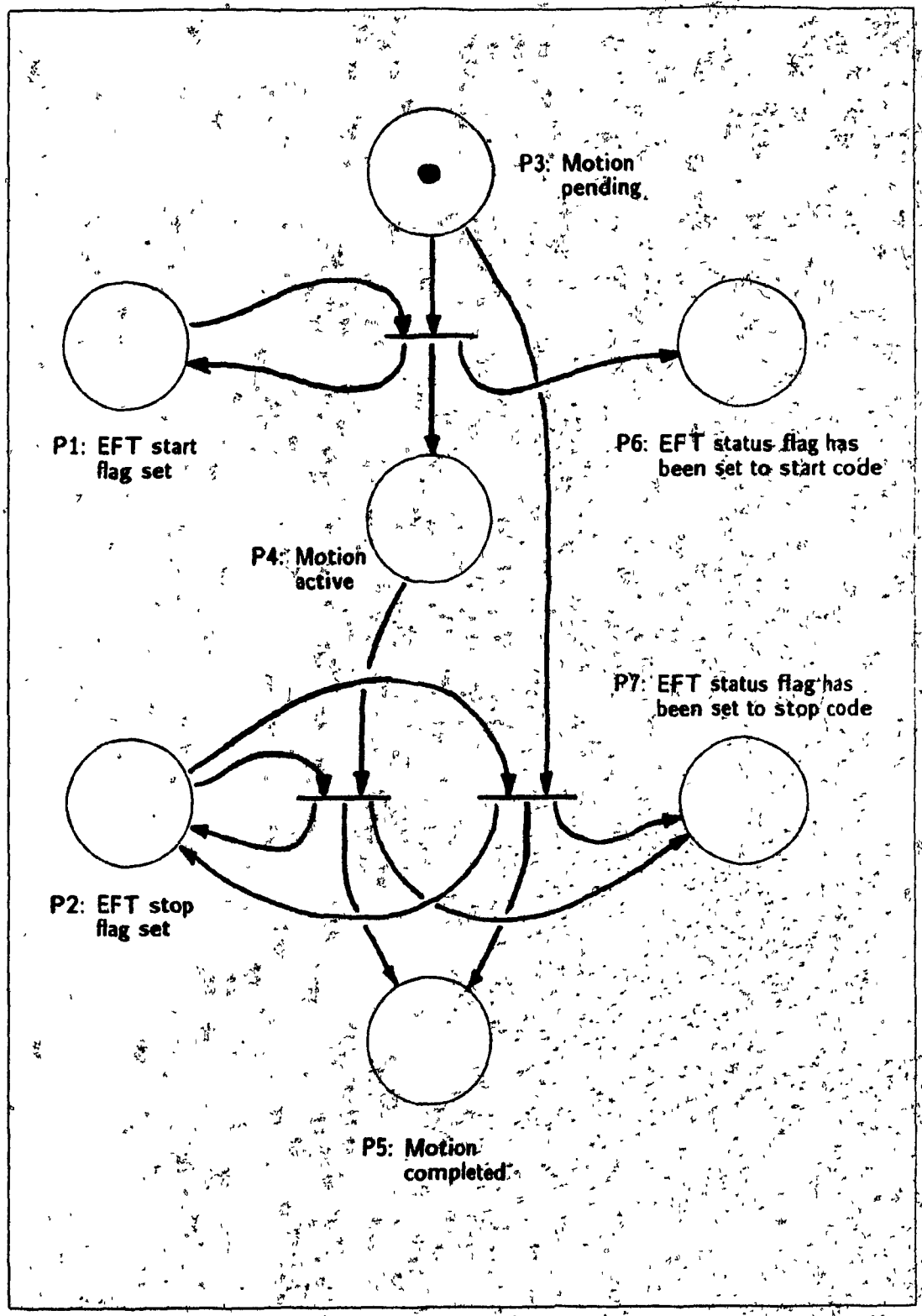


Figure 5.6 Petri net model for the three flag motion protocol.

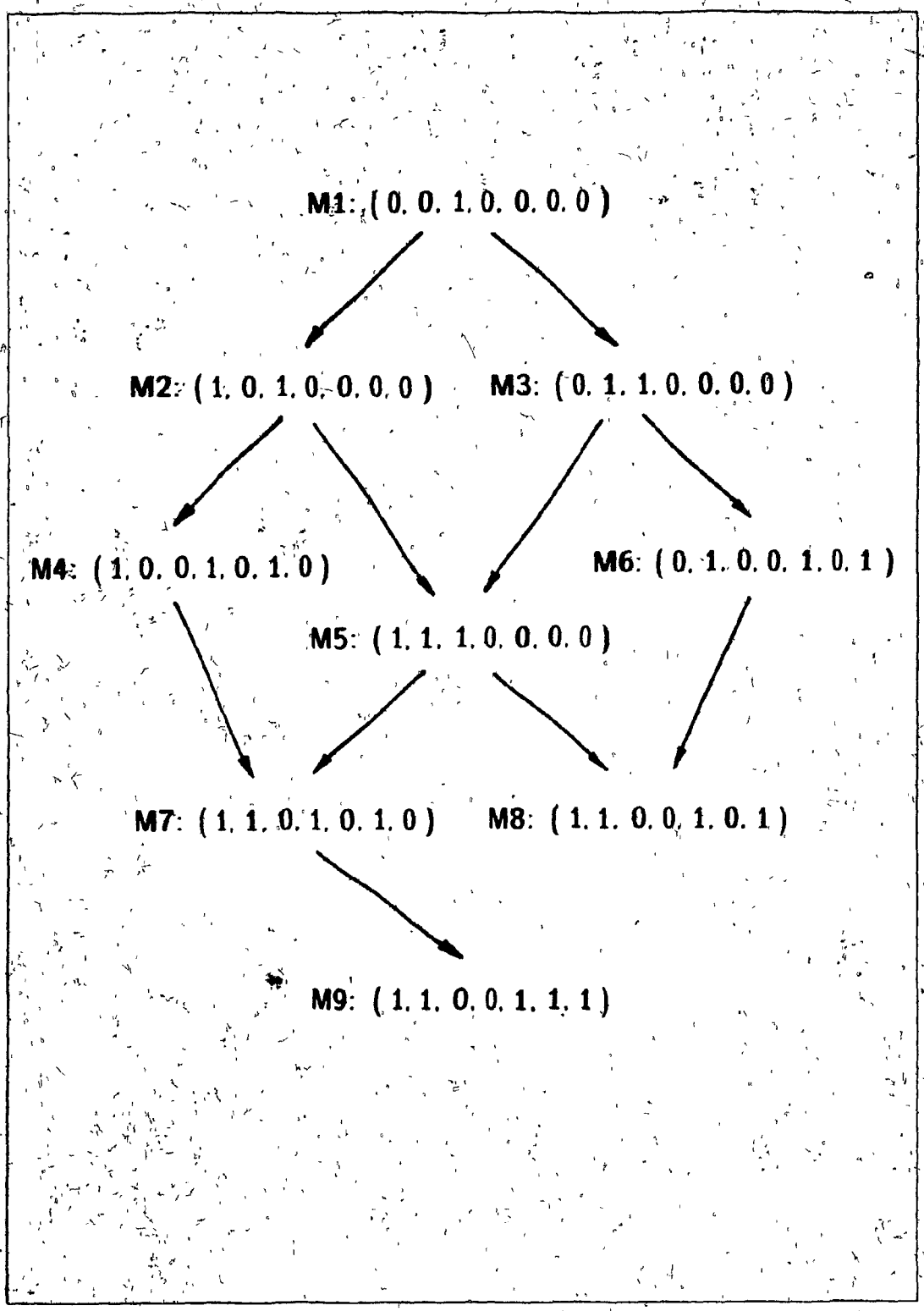


Figure 5.7 Reachability tree for the Petri net of figure 5.6.

We now define the synchronization primitives themselves. The programmer would allocate an EFT and associate it with a motion by using the directive

```
eft = move_ev (robot, position, motion_params),
```

in place of the usual *move()* primitive. A pointer to an EFT would be returned. After the programmer had finished with a particular EFT, such as when the motion has completed, it could be returned to the system with the call

```
free_ev (count, eft_1, ... eft_n);
```

which would release the number of EFTs indicated by **count**.

Conditional motions would be associated with EFTs implicitly, by inheriting, when appropriate, the EFT assigned to the nominal motion which was to follow. Consider, for instance:

```
move (robot_1, positionA, motion_params);
eft1 = move_ev (robot_1, positionB, motion_params);

cond_move (condA, robot_1, positionBackupA, motion_backup_params);
cond_move (condB, robot_1, positionBackupB, motion_backup_params);

eft2 = move_ev (robot_1, positionC, motion_params),
```

If the second motion segment terminates with either condition *condA* or *condB*, then the indicated conditional motion would occur in place of the default move specified to *positionC*, and the conditional motion would be controlled by *eft2*.

Setting the start and stop flags of an EFT may done using the following functions.

```
start_ev (code, count, eft_1, ... , eft_n),
```

— Set the start flag of the indicated EFTs to the value of *code*.

```
stop_ev (code, count, eft_1, ... , eft_n);
```

— Set the stop flag of the indicated EFTs to the value of *code*.

These primitives allow multiple arguments to permit different EFTs to be set simultaneously, which is necessary when synchronizing the start or stop of two motions.

The status of an EFT can be examined with the primitive

```
status_ev (eft);
```

which returns the status flag value for the indicated EFT. This function may be combined with logical operators and wait primitives to build any desired test or blocking operation.

Lastly, for completeness, a primitive should be provided which flushes an entire robot motion queue and brings the arm to a halt:

```
stop_abort (robot);
```

5.7 Summary

RCCL is a trajectory control package implemented using the RCI mechanism described in Chapter 2. It is intended primarily as a tool for developers and researchers, with robot control being determined through a set of library routines (currently written in the language C). Robot motions are specified by path segments, each parameterized in terms of their destination position, velocity, path interpolation mode, and path adaptation characteristics. Motion requests are built using special primitives and queued for servicing by a trajectory generator running as an RCI control task.

The path adaptation features permit a particular path segment to be altered dynamically by a user-specified function or compliance specification, or terminated when certain limit conditions are met.

Having used the original system for several months, it has been seen that certain improvements are necessary. It is suggested that the path specification be extended to allow conditional motions to be bound to different motion termination conditions, so that if a path segment is terminated, the trajectory generator can immediately start the next path segment without having to wait for instructions from the planning level. Supplemental arguments are suggested for the motion request primitive to allow different robots and sets of motion parameters to be selected. It is also necessary to improve the synchronization mechanism between the planning level and the trajectory generator. The recommended way of doing this calls for associating an *event flag triple*, consisting of a *start*, *stop*, and *status* flag, with each motion request for which synchronization is required.

Chapter 6 CONCLUSION

We recapitulate here the results of the preceding chapters, and discuss some areas for future work.

6.1 Robot Kinematics

Given the assistance of MACSYMA, the derivation of the robot kinematics and differential kinematics was a fairly straightforward procedure. It was possible to condense the solution for the Jacobian computations into a form that required, in the worst case for the PUMA 260, 46 multiplies/divisions and 23 additions, excluding trigonometric function calls. This has roughly the same cost as the forward nominal kinematic computations implemented in the existing version of RCCL, which require 41 multiplies/divisions and 17 additions. Similar results can be shown for the Microbo

For robots of this complexity, then, the symbolic determination of the nominal and differential kinematics yields solutions which can easily be handled by more advanced microprocessors*, without having to resort to special purpose systems such as those described in [Orin, et al 85]. Moreover, the symbolic formulation makes it easier to optimally handle cases where the manipulator is singular, since the terms which remain well-defined are known explicitly and may still be computed

---

* Such as the Motorola 68020 with floating point coprocessor

A method for improving the accuracy of the kinematic model was quickly mentioned at the end of Chapter 3. It would be useful to to actually perform the error measurements called for by this method, determine its practicality, and see whether or not it is possible to use information from such a model to significantly improve the run-time accuracy of the robot, at an acceptable cost in computation. Kinematic errors account for the well known difference between robot repeatability and accuracy, which is typically an order of magnitude.

6.2 Force Control Parameters

A simple method was discussed for determining force control parameters by measuring the work done as the robot was moved along a specially selected path. This proved to be a quick and reliable way to obtain the gravity loading and friction terms for a robot where the joint friction is not unduly large. The integration inherent in the work measurement reduced the effects of noise and random variations in friction. This method could be incorporated into a general self-calibration routine for manipulators, allowing the determination of both the intrinsic gravity loadings, and the load of any mass which is being carried.

Using the motor currents to estimate the instantaneous applied torque on each joint was less successful because of local uncertainties in the value of the friction, particularly the static friction. Mapping the observed torques into Cartesian space amplifies this error still further. In the best configuration, the condition number of the manipulator Jacobian for the PUMA 260 was determined to be about 1.8. In this position, a typical static friction error of .5 Newton-meters (Table 4.6) will result in an error of about 4.5 Newtons in the observed translational force.

A way around this problem is to perform the force/torque sensing at some point in the system *after* the joint gear train. One method involves placing torque sensors on the output shafts of the joints and determining the torques there. This has been done successfully at Purdue University with a Stanford manipulator [Paul, et al. 80]. Unfortunately, most commercially available robots are not equipped with such sensing devices, and their installation requires major mechanical work on the robot. A more common method involves using a force sensor to measure Cartesian forces directly at the wrist [Stepien, et al. 85]. This yields very accurate information for Cartesian force control, though at the expense of

direct knowledge about the joint torques

6.3 The RCI System

The RCI system provides an easy interface for writing trajectory level robot control software in the programming language C. By providing a couple of extensions to UNIX 4.2bsd, it was possible to create a convenient real-time software environment running under UNIX. Software development tools, including a simulator and compilation–link command, have been provided, and the programmer now has complete access to all devices and peripherals associated with the PUMA robot controller, including the teach pendant.

RCI is essentially an interrupt handling mechanism that provides the ability to do real-time control while the planning part of the program executes in the utility-rich domain of a large scale operating system. This concept can be extended to permit multiple control tasks, where each task is simply an interrupt driven procedure. The problem of computational load can be resolved by distributing the different control tasks among separate processors. The issues involved in doing this have been explored, including the nature of the inter-task communication mechanisms and suggested hardware and software architectures. It was recommended that the system be designed around a set of board level microcomputers, each with local memory, all interconnected using a bus topology. A high speed parallel bus would provide the necessary speed for the control applications under consideration, as well as the flexibility to implement software communication based on either shared data or message passing. Throughout the discussion, the author was guided by the following paradigm: in the development of real-time systems, multiprocessing can often be best realized in terms of a set of subordinate processors interconnected in a general way and in turn connected to a host. Program development is done on the host in a consistent environment and executable images are loaded into the processors. This principle is present in Harmony [Gentleman 85], it is also the basis for a distributed computing system for robot vision applications under development at Bell Laboratories [Selfridge 84].

The limits of such a system are determined at one end by the computational power of the individual processers and the bandwidth of the connecting bus. In principle, there is no reason why a sufficiently enhanced version of the system could not be used to perform joint level as well as robot level control (section 1.3.1). The limit in the other direction is-

the complexity of the processes that one would wish to implement using the RCI model. if a task is very complex and operates at a slow rate, it is probable that the limited RCI control task environment would be too restrictive. Such tasks, however, can be run at the planning level which has full access to the host operating system.

One obvious project would be to implement the multiprocessing RCI which has been described and test its usefulness. It is interesting to consider an environment that allows rapid creation and prototyping of different control algorithms and structures, where the top level of the program executes as an application program under a general purpose operating system. Two particular aspects to such a project would require strong attention. The first concerns the physical and logical interfacing of the multiple control processors to the host, particularly if shared memory is desired, a common address space would have to be provided across the system that does not interfere with the host's native operating system. A second aspect of the work would be establishing high level development tools for the easy creation and maintenance of the target images for the different processors. By this is implied both run-time tools such as simulators and debuggers, and compile-time tools, similar in kind to the UNIX `make` command. Such problems are not trivial, given that the development of multiprocessor multitasking software is itself not usually easy.

6.4 RCCL

RCCL offers a flexible approach to robot motion programming. The suggested improvements regarding the new synchronization and conditional motion primitives are straightforward to implement and install. The system may also be extended to control multiple manipulators, as illustrated by the prototype multi-robot primitives. The implementation of a multi-robot RCCL would require a multitasking RCI.

As a programming basis for implementing more sophisticated or specialized robot programming interfaces, RCCL is quite useful given the advantages in flexibility it has over commercially available robot control languages. The fact that it is compiled offers a considerable improvement in terms of speed alone. The user of RCCL, of course, is assumed to be versed in robot technology. There is some disadvantage in that the syntax of RCCL is restricted to function calls in its host language, but this is a cosmetic problem which is tolerable for a set of development primitives.

A particularly useful aspect of RCCL is the built in force control (compliance and force limit primitives), although improved force sensing is required, as noted in Chapter 5 and the conclusions above concerning force control, motor current feedback is not adequate for accurate instantaneous force/torque detection

RCCL's principal use is to serve as a trajectory generator interface. Work remains to be done both at the implementation level in terms of simplifying the method by which these trajectories are generated, and at the user level by generalizing the way in which trajectories can be specified One could, for instance, use velocity constraints, in addition to positions, in specifying trajectories Just how an RCCL interface would have to appear to a more advanced system which incorporated collision planning and avoidance also needs to be considered It should be noted that the use of coordinate frames alone to describe robot positions and trajectories is mainly useful only for 6 degree of freedom manipulators when more degrees of freedom are present, the robot is redundant, and extra information (or rules) must be provided in order to solve the inverse kinematics A simple case of this may be observed with six degree of freedom robots, which can have discrete redundancies which define the robot's *configuration* RCCL currently allows this configuration (the extra information) to be set explicitly, otherwise, the kinematics are solved assuming the existing configuration (a rule). The handling of truly redundant manipulators will involve much more thought

6.5 General Comments about Creating a Robotics Research Environment

Given that the work described in this thesis has been directed towards developing a robot research environment, some concluding remarks will be made toward this in general The necessary chores can be grouped under the following headings:

1. *Taming the robots* Deriving various equations and parameters to obtain models of the manipulator
2. *Taming the software.* Providing a programming environment in which the user has control over the manipulator at the level of his/her choice.
3. *Taming the hardware.* Although not greatly emphasized in this thesis, a certain amount of effort was required to modify the purchased equipment to support the required

communication and data acquisition capabilities This has been particularly true in working with the Microbo robot

Predictably, the job took longer than estimated to complete, during which time several points were noted First, the requirements of manufacturers to provide turn-key systems which insulate the user from technical details are contradictory to the requirements of research facilities to have access and control over those very details. Some manufacturers do not express much interest in helping to overcome this situation. Second, it was observed that in setting up a integrated robot system, a principal tenet should be to keep the different components reasonably uniform. This is simply in response to the fact, discussed in Chapter 2, that the resources are typically not available, in a laboratory setting, to master and interface to a multitude of components

References

[Aboussouan 85] Patrick Aboussouan, "Frequency Response Estimation of Manipulator Dynamic Parameters" (M Eng Thesis) Dept of Electrical Engineering, McGill University, Montreal, Canada, December 1985

[Albus, et al. 81] J S Albus, A J Barbera, and R N. Nagel, "Theory and Practice of Hierarchical Control" *Proceedings of the Twenty-third IEEE Computer Society International Conference* (COMPCON), Washington, D C , September 15 - 17 1981, pp 18 - 39

[Alford and Belyeu 84] Cecil O Alford and Stanley M Belyeu, "Coordinated Control of Two Robot Arms" *IEEE International Conference on Robotics*, Atlanta, Georgia March 13 - 15, 1984, pp 468 - 476.

[Allworth 81] S T. Allworth, *Introduction to Real-Time Software Design*. MacMillan Press, Ltd., London, 1981.

[Automatix 82] Automatix, Inc , *RAIL Software Reference Manual, Robovision and Cybervision* 1000 Tech Park Drive, Billercia, Mass , 01821, 1982

[Baird, et al. 84] Henry S Baird, Eleanore G. Wells, and Dianne E. Britton, "Coordination Software for Robotic Workcells" *IEEE International Conference on Robotics*, Atlanta, Georgia, March 13 - 15, 1984, pp 354 - 360.

[Bazerghi, et al. 84] A Bazerghi, A.A Goldenberg, and J Apkarian, "An Exact Kinematic Model of the PUMA 600 Manipulator", *IEEE Transactions on Systems, Man, and Cybernetics*, May/June 1984, pp 483 - 487 (Vol SMC-14, No 3)

[Berkeley 83] *UNIX Programmer's Manual* (revised for 4.2bsd) Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, California 94720, 1983.

[Bonner and Shin 82] Susan Bonner and Kang G Shin, "A Comparative Study of Robot Languages" *Computer*, December 1982, pp 82 - 96. (Vol 15, No 12)

[Bowen and Buhr 80] B.A Bowen and R J.A. Buhr, *The Logical Design of Multiple-Microprocessor Systems*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1980

[Chand and Doty 83] Sujeet Chand and Keith L Doty, "Cooperation in Muliple Robot Operation" *IEEE SOUTHEASTCON Conference Proceedings*, Orlando, Florida, April 11 - 14, 1983, pp. 99 - 103.

[Denavit and Hartenberg 55] J Denavit and R S Hartenberg "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices" *ASME Journal of Applied Mechanics*, June 1955, pp 215 - 221

[Digital 82A] Digital Equipment Corporation *VAX-11 Architecture Reference Manual Revision 6 1* Part Number EK-VAXAR-RM-001, Bedford, Mass 01730, May 1982

[Digital 82B] Digital Equipment Corporation *VAX Software Handbook* Part Number EB-21812-20 Maynard, Mass 01754, 1982

[Faro and Messina 83] Alberto Faro and Gaetano Messina, "Robot Internetworking" *IEEE Transactions on Industrial Electronics*, November 1983, pp 353 - 357 (Vol IE-30, No 4)

[Finkel, et al. 75] R Finkel, R Taylor, R Bolles, R P Paul, and J. Feldman, "An Overview of AL A programming System for Automation" *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi Georgia, USSR September 3 - 8 1975, pp 758 - 765

[Gauthier, et al. 85] David Gauthier, Gregory Carayannis, Paul Freedman, and Alfred Malowany, A Session Layer Design for a Distributed Robotics Environment', *COMPINT 85 - Computer Aided Technologies*, Montreal Canada, September 8 - 12, 1985, pp 459 - 465

[Gentleman 81] W M Gentlemen, "Message Passing Between Sequential Processes the Reply Primitive and the Administrator Concept" *Software - Practice and Experience*, May 1981, pp 435 - 466 (Vol 11, No 5)

[Gentlemen 85] W M Gentlemen, "Using the Harmony Operating System", Technical Report ERB-966 (NRCC No 24685), Division of Electrical Engineering, National Research Council of Canada, Ottawa, Canada, May 1985

[Goldenberg 82] A A Goldenberg, "A Simulation Model for Robotic Manipulators Application to the PUMA 560" Technical Report UL-RAL-TR-8201, Robotics and Automation Laboratory, Department of Mechanical Engineering, University of Toronto, Toronto, Ontario, 1982

[Goldman 82] Ron Goldman, "Design of an Interactive Manipulator Programming Environment" (Ph D. Thesis) Report No STAN-CS-82-955, Department of Computer Science, Stanford University, Stanford, California 94305, December 1982

[Goldstein 50] Herbert Goldstein, *Classical Mechanics* Addison-Wesley, Reading, Massachusetts, 1950

[Harmon 83] S.Y Harmon, "Coordination Between Control and Knowledge Based Systems for Autonomous Vehicle Guidance". *Proceedings of the 1983 IEEE Conference*

on Trends and Applications. Gaithersburg, Maryland. May 25 – 26, 1983. pp 8 – 11

[Hayward and Paul 83] Vincent Hayward and Richard P Paul. "Robot Manipulator Control Under UNIX" *Proceedings of the 13th International Symposium on Industrial Robots*. Chicago, Illinois. April 17 – 21 1983. pp 20.32 – 20.44.

[Hayward 83A] Vincent Hayward. "Robot Real Time Control User's Manual" Report TR-EE 83-42, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907, October 1983

[Hayward 83B] Vincent Hayward. "Introduction to RCCL: A Robot Control C Library" Report TR-EE 83-43, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907, October 1983

[Hayward and Lloyd 85] Vincent Hayward and John Lloyd. "RCCL User's Guide", Technical Report, Dept of Electrical Engineering, McGill University, Montreal, Canada, December 1985

[Holt, et al. 78] R C Holt, G S Graham, E D Lazowska, and M.A. Scott. *Structured Concurrent Programming with Operating System Applications* Addison Wesley, Reading, Mass., 1978

[Hong and Paul 85] Zhang Hong and Richard P Paul, "Hybrid Control of Robot Manipulators", *IEEE International Conference on Robotics and Automation*, St Louis, Missouri, March 25 – 28, 1985, pp 602 – 607

[Intel 84A] Intel Corporation, *Multibus II Bus Architecture Specification Handbook* Order number 146077-C, OEM Modules Operation, 5200 NE Elam Young Parkway Hillsboro, Oregon, 97123, 1984.

[Intel 84B] Intel Corporation, *iRMX86 Introduction and Operator's Reference Manual* Order number 146194-001, Santa Clara, California 95051, 1984

[Kernighan and Ritchie 78] Brian K Kernighan and Dennis Ritchie, *The C Programming Language* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1978

[Kernighan and Pike 84] Brian K Kernigham and R Pike, *The UNIX Programming Environment* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1984

[Kossman 85] Don Kossman, "A Multi-Microprocessor-Based Control Environment for Industrial Robots" (M. Eng Thesis), Dept of Electrical Engineering, McGill University, Montreal, Canada, March 1986

[Lee 82] C S G Lee, "Robot Arm Kinematics, Dynamics, and Control", *Computer*, December 1982, pp. 62 – 80. (Vol 15, No. 12)

[Lozano-Perez and Wesley 79] Tomas Lozano-Perez and Michael A. Wesley. "An Algorithm for Planning Collision-Free Paths among Polyhedral Obstacles". *Communications of the ACM*. October 1979. pp. 560 - 570. (Vol. 22, No. 10)

[Lozano-Perez 82] Tomas Lozano-Perez. "Robot Programming". A.I. Memo No. 698. M.I.T. Artificial Intelligence Laboratory. Cambridge, Mass. December 1982.

[MacWilliams, et al. 84] Peter D. MacWilliams, Peter I. Wolochow, and Randy H. Zasloff. "Microcontroller Serial Bus Yields Distributed Multispeed Control". *Electronics*. February 9, 1984. pp. 125--130. (Vol. 57, No. 3)

[Marrin 85] Ken Marrin. "Multibus II and VMEbus compete for real-time applications". *Computer Design*. August 15, 1985. pg. 24. (Vol. 24, No 10)

[Mujtaba and Goldman 79] S. Mujtaba and R. Goldman. "AL User's Manual". Report No. AIM. Stanford Artificial Intelligence Laboratory. Stanford University. Stanford, California 94305. January 1979.

[Neuman and Murray 84] Charles P. Neuman and John J. Murray. "Linearization and Sensitivity Functions of Dynamic Robot Models". *IEEE Transactions on Systems, Man, and Cybernetics*. November/December 1984. pp. 805 - 818. (Vol. SMC-14, No. 6)

[Nigam and Lee 85] Ravi Nigam and C.S.G. Lee. "A Mulitprocessor-Based Controller for the Control of Mechanical Manipulators". *IEEE International Conference on Robotics and Automation*. St. Louis, Missouri. March 25 - 28, 1985. pp 815 - 821

[Orin, et al. 85] D.E. Orin, H.H. Chao, K.W. Olson, and W.W Schrader. "Pipeline / Parallel Algorithms for the Jacobian and Inverse Dynamics Computations". *IEEE International Conference on Robotics and Automation*. St. Louis, Missouri. March 25 - 28, 1985. pp. 785 - 789.

[Paul and Shimano 76] Richard P. Paul and Bruce Shimano. "Compliance and Control". *Proceedings of the Joint Automatic Control Conference*. West Lafayette, Indiana. July 27 - 30, 1976. pp. 694 - 699.

[Paul, et al. 80] Richard P. Paul, J. Luh, S.Y. Nof, W. Fisher, H. Lechtman, and C.H. Wu. "Force Sensing and Joint Torque Sensor System". Sixth Report. Advanced Industrial Robot Control Systems. Technical Report TR-EE 80-31. School of Electrical Engineering. Purdue University. West Lafayette, Indiana 47907. 1980.

[Paul, et al. 81A] Richard P. Paul, Bruce E. Shimano, and E.G. Mayer. "Kinematic Control Equations for Simple Manipulators". *IEEE Transactions on Systems, Man, and Cybernetics*. June 1981. pp. 449 - 455. (Vol. SMC-11, No. 6)

[Paul, et al. 81B] Richard P. Paul, Bruce E. Shimano, and E.G. Mayer, "Differential Kinematic Control Equations for Simple Manipulators", *IEEE Transactions on Systems, Man, and Cybernetics*, June 1981, pp. 456 - 460 (Vol. SMC-11, No. 6)

[Paul 81] Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*, MIT Press, Cambridge, Mass., 1981.

[Paul and Rong 83] Richard P. Paul and Ma Rong, "The Dynamics of the PUMA Manipulator", Technical Report TR-EE 82-34, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907, October 1982.

[Peterson 81] James L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1981.

[Raibert and Craig 81] Marc H. Raibert and John J. Craig, "Hybrid Position/Force Control of Manipulators", *Journal of Dynamic Systems, Measurement, and Control*, June 1981, pp. 126 - 133 (Vol. 103, No. 2)

[Renaud 81] Marc Renaud, "Geometric and Kinematic Models of a Robot Manipulator: Calculation of the Jacobian Matrix and Its Inverse", *Proceedings of the 11th International Symposium on Industrial Robots*, Tokyo, Japan, October 7 - 9, 1981, pp. 757 - 763.

[Schwan, et al. 85] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee, "GEM: Operating System Primitives for Robots and Real-Time Control Systems", *IEEE International Conference on Robotics and Automation*, St. Louis, Missouri, March 25 - 28, 1985, pp. 807 - 813

[Selfridge 84] Peter G. Selfridge, "Distributed Computing for Vision, Architecture and Preliminary Results", *Seventh International Conference on Pattern Recognition*, Montreal, Canada, July 30 - August 2, 1984, pp. 996 - 998.

[Shimano, et al. 84] Bruce E. Shimano, Clifford C. Geschke, Charles H. Spalding, and Paul G. Smith, "A Robot Programming System Incorporating Real-Time and Supervisory Control: VAL II", *Robots 8 Conference Proceedings*, Volume 2 (Future Considerations), Detroit, Michigan, June 4 - 7, 1984, pp. 20.103 - 20.119.

[Shin and Malin 85] Kang G. Shin and Stuart B. Malin, "A Structured Framework for the Control of Industrial Manipulators", *IEEE Transactions on Systems, Man and Cybernetics*, January/February 1985, pp. 78 - 90 (Vol. SMC-15, No. 1)

[Shin and Epstein 85] Kang G. Shin and Mark E. Epstein, "Communication Primitives for a Distributed Multi-Robot System", *IEEE International Conference on Robotics and Automation*, St. Louis, Missouri, March 25 - 28, 1985, pp. 910 - 917.

[Shoemake 85] Ken Shoemake "Animating Rotation with Quaternion Curves". *Computer Graphics* (SIGGRAPH '85 Conference Proceedings), San Fransisco, California, July 22 – 26, 1985, pp 245 – 254 (Vol 19, No 3)

[Slemon and Straughen 81] Gordon R Slemon and A Straughen, *Electric Machines* Addison Wesley, Reading, Mass., June 1981

[Stephanou and Saridis 76] H E. Stephanou and G.E Saridis, "A Hierarchically Intelligent Method for the Control of Complex Systems" *Journal of Cybernetics*, July/December 1976, pp. 249 – 261 (Vol. 6, No. 3-4)

[Stepien, et al. 85] T.M Stepien, L.M Sweet, and M.C. Good, "Control of Tool / Workpiece Contact Force with Application to Robotic Deburring" *IEEE International Conference on Robotics and Automation*, St. Louis, Missouri, March 25 – 28, 1985, pp. 670 – 679

[Steusloff 84] Hartwig U Steusloff, "Advanced Real Time Languages for Industrial Process Control" *Computer*, February 1984, pp 37 – 46 (Vol 17, No 2)

[Takase, et al. 79] K. Takase, R.P. Paul, and E J Berg, "A Structured Approach to Robot Programming and Teaching" *IEEE COMPSAC*, Chicago, Illinois, November 6 – 8, 1979, pp 452 – 457

[Tanenbaum 81] A. Tanenbaum, *Computer Networks* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1981

[Taylor 79] R H Taylor, "Planning and Execution of Straight Line Manipulator Trajectories", *IBM Journal of Research and Development*, July 1979, pp 253 – 264, (Vol 23, No 4)

[Taylor, et al. 82] R H Taylor, P D Summers, and J.M Meyer, "AML – A Manufacturing Language" *International Journal of Robotics Research*, Fall 1982, pp 19 – 41 (Vol 1, No 3)

[Unimation 83] Unimation Inc *User's Guide to VAL-II The Unimation Robot Programming and Control System* Shelter Rock Lane, Danbury, Conn 06810, 1983

[Wesley, et al. 80] Michael A Wesley, Tomas Lozano-Perez, L I Lieberman, M A Lavin, D D. Grossman, "A Geometric Modeling System for Automated Mechanical Assembly", *IBM Journal of Research and Development*, January 1980, pp 64 – 74, (Vol 24, No. 1)

[Whitney 85] Daniel E. Whitney, "Historical Perspective and State of the Art in Robot Force Control", *IEEE International Conference on Robotics and Automation*, St Louis, Missouri, March 25 – 28, 1985, pp 262 – 268.

[Wu 85] Chi-haur Wu. "A Kinematic CAD Tool for the Design and Control of a Robot Manipulator". *International Journal of Robotics Research*. Spring 1984. pp 58-67 (Vol. 3. No 1)

[Zimmermann 80] H Zimmermann. "OSI Reference Model - The ISO Model for Open Systems Interconnection". *IEEE Transactions on Communication*. April 1980. pp 425 – 432. (Vol. COM-28. No 4)

Appendix A. RCI User's Manual

A.1 Synopsis

The RCI system allows a programmer to develop robot control software by defining *control functions*, which run in parallel with the main program at a selectable sample rate. These control functions define a high priority *control task*, which communicates through shared variables with the main program, or *planning task*. Because of the real-time constraints imposed on the control task, the control functions must meet certain restrictions (section A.2). A set of primitives is available at the planning level for initializing the control task, binding the control functions to it and activating the control, releasing the control, and closing the task (section A.3). Both the planning and control levels may interact with the robot through a pair of special data structures (section A.4). The RCI system watches for error conditions and acts on them by releasing any currently active control and sending a signal to the planning level (section A.5). Some software tools are available to assist in developing RCI applications, these include a command to assist in compiling and loading the program, and a simple robot simulator under which the program may be run to help find software errors (sections A.6 and A.7).

This manual describes the current version of RCI, which runs under VAX/UNIX 4.2bsd.

A.2 Function Restrictions

The RCI function restrictions arise from the fact that to meet real-time requirements, they must always be resident in the VAX's main memory, and must also be executed at elevated priority from within the UNIX kernel. The resulting restrictions are:

1. All modules containing code and data which is accessed by the control functions must be compiled and loaded into the main program using a special procedure. This is to enable the RCI system to isolate these code and data segments at run time and lock them in main memory. A command called `rcc`, described below, is provided to take care of this for the programmer automatically.
2. The control functions may not invoke any UNIX system call. This means that file and device I/O, memory allocation, and process management routines are not

accessible from the control level*

3. The control functions must complete execution in the time interval defined by the sample rate

4. Care should be taken to keep the functions free of bugs. Although arithmetic errors and access violations will most likely be caught by the UNIX system, there is a small chance that an invalid memory reference into the kernel memory space will cause a system crash. Since the functions are non-interruptable, infinite loops are understandably undesirable as they will cause the system to hang.

A.3 System Control Primitives

Each primitive described here returns 0 if it was able to execute successfully, and −1 otherwise, in which case an error number is placed in the global variable *RCIerror*. The error numbers are defined in the file <robot/errors.h>. RCI global variables and parameters are defined in the file <robot/rci.h>.

RCIopen()

— initializes the RCI control task.

usage:

```
status = RCIopen ();
 int status;
```

description:

Initializes the RCI control task. Must be called before a control is started using *RCIcontrol()*. The RCI system will remain open until a call is made to *RCIclose()*.

errors:

R_OPENP – RCI control task already open.

R_PAGELOCK – cannot lock RCI control code in memory.

R_BAD_OPEN – cannot open robot communication device.

R_BAD_IOCTL – cannot control robot communication device.

---

* This is not a great loss, since system calls are usually very time-consuming and one would wish to avoid them anyway.

RCIcontrol()

— activates RCI control

usage:

```
status = RCIcontrol (function1, function2)
 int status;
 int (*function1)(), (*function2)();
```

description

Activates a control session using the two user supplied control functions These functions will be executed in a non-interruptable context by the RCI system once every control cycle To satisfy the real-time requirements of the implementation, the functions must meet the restrictions detailed in section A 2. The control session will remain active until either an error condition at the control level forces a *termination*, or a call is made to *RCIrelease()*.

errors.

R_NOOPEN – RCI control task is not open

R_CONTROLP – RCI control already in progress.

R_NOFUNCTION – illegal function pointers

R_BAD_IOCTL – cannot control robot communication device

R_BAD_START – robot communication device hung.

R_TIMEOUT – control task timed out, no response from robot

RCIrelease()

— deactivates RCI control.

usage:

```
RCIrelease(power_off)
 int power_off;
```

description:

Deactivates any RCI control which is currently active: this must be done before any new control can be activated with *RCIcontrol()*. If the argument *power_off* is true

(non-zero), the robot arm power will be shut off, otherwise, the arm power will remain in the state it was in An error condition at the control level will automatically cause a *release* to be performed, shutting off the power if the error is severe enough.

errors

R_NOCONTROL – no RCI control is in progress

R_BAD_IOCTL – cannot control robot communication device.

RCIclose()

— closes down the RCI system.

usage:

```
RCIclose (power_off)
 int power_off;
```

description:

Closes down the RCI task if it is open. If a control is active, then control is released, and the robot arm power is either turned off, if *power_off* is true (non-zero), or left in the state it was in if *power_off* is false. If no control is active, the argument *power_off* is ignored.

errors.

R_NOOPEN – RCI control task is not open

A.4 Communication Data Structures

Both the planning and control levels may obtain information about the robot or send it commands by accessing the data structures *how* and *chg*, respectively, which are predefined by the RCI library

The *how* structure contains state information about the robot and must be treated as read-only Command and value fields in the *chg* structure are set in order to specify commands to the robot Both structures are serviced by the RCI control level once every sample period.

All of the parameters and structures given here are defined in the file <robot/rci.h>

how

— robot state information data structure

data structure definition

```
#define NJ      6       /* the number of robot joints                */
#define TPB     3       /* no. of bytes sent from teach pendant      */
#define ADC    16       /* no. of analog-digital converter channels */

struct how {
  short
    wc;                 /* reserved for internal system use          */

  unsigned short
    status,             /* robot error status word                   */
    state,              /* robot state descriptor                    */
    exio,               /* contents of robot external I/O register   */
    pos [NJ],           /* position of each joint (encoder values)   */
    inport1,            /* readings from parallel port1              */
    outport1,           /* outport sent to parallel port1            */
    pendant [TPB];      /* data sent from the teach pendant          */

  short
    adcr [ADC];         /* analog-to-digital converter readings      */
};
```

description:

NJ is the number of joints on the robot (currently 6)

TPB is the number of bytes of information that are returned from the teach pendant to describe which buttons are currently being pressed

ADC is the number of analog-to-digital converter channels available for data input The first 6 channels are reserved for RCI use

how.status contains a code describing any error condition which is detected on the robot The nominal (no-error) value of this word is 0 The error codes are defined in the file *<robot/cmdk.h>*

how.state contains flags which describe the state the robot is currently in The flags presently defined (in the file *<robot/cmdk.h>*) are.

CALIB_OK – the robot is calibrated.

TP_OK – the teach pendant is communicating properly

ZINDEX_OK – the robot calibration procedure was successful

how.exio contains information describing the external switch settings on the robot controller, as described in the file *<robot/addefs.h>*

how.pos is an array containing the optical encoder readings describing the position of each joint angle. Each optical encoder reading is a 16 bit unsigned integer. The allowed range of each optical encoder value varies from joint to joint, and is described in the file *<robot/pumadata.h>*

how.inport1 contains a 16 bit value equal to the input present on the robot controller parallel input port 1.

how outport1 contains the 16-bit value which is currently active on the robot controller parallel output port 1.

how.pendant is an array containing the three bytes last read back from the teach pendant. These bytes describe which keys, if any, are currently activated on the teach pendant. The protocol is described in the document "PUMA 260 Teach Pendant Protocol", in the file */local/doc/robot/teachPendant.txt*

how.adcr is an array of readings from the analog to digital converter attached to the robot controller. Readings are returned as 12 bit, two's complement values. The first 6 elements in the array are reserved to describe the motor current at each joint. The remaining elements are for use at the discretion of the programmer.

chg

— robot control data structure.

data structure definitions:

All of the parameters and structures given here are defined in the file *<robot/rci.h>*

```
typedef struct {          /* Valued command structure:          */
  char com;               /*   command code                     */
  unsigned short value;   /*   corresponding value              */
} CMDVAL;

typedef struct {          /* Unvalued command structure:        */
  char com;               /*   command code                     */
} CMD;
```

```
struct chg {
  CMDVAL motion [NJ];      /* control joint motion                    */
  CMDVAL setparam [NJ];    /* set joint control parameters            */

  CMDVAL setparamALL;      /* set control parameters for all joints */
  CMDVAL adco;             /* open analog-digital converter channel */
  CMDVAL hand;             /* control end effector                    */
  CMDVAL rate;             /* set control cycle rate                  */
  CMDVAL checking;         /* set checking options                    */
  CMDVAL outport1;         /* set parallel output port 1              */
  CMDVAL pendant;          /* set teach pendant display               */
  CMDVAL zindex;           /* put joints into calibration mode        */

  CMD    end;              /* end control session with robot          */
  CMD    stop;             /* stop the robot                          */
  CMD    power_on;         /* turn on the armpower                    */
  CMD    power_off;        /* turn off the armpower                   */
};
```

description

The robot may be controlled by writing appropriate entires into this data structure Once every control cycle, the RCI system examines this structure and sends the appropriate commands off to the robot controller

Each command entry in the data structure contains a *com* field, and possibly a *value* field Issuing a command is done by setting the *value* field, if present, and then setting the *com* field to the requested command Later, when the RCI system dispatches this command to the robot, it clears the *com* field Commands associated with a value are described by the *CMDVAL* structure; commands not associated with a value are described by the *CMD* structure.

chg.motion is an array of *CMDVAL* structures which direct the motions of the individual joints of the arm For each joint, the *com* field may be set to one of the following commands

POS – instructs the joint to move, during the next cycle, to the encoder count specified in the *value* field.

CUR – sets the current on the joint to the two's complement signed value specified

by the low order 12 bits of the *value* field

STOP – stops the joint motion The *value* field is ignored

STOPCAL – stops the joint motion and sets the optical encoder count to the value specified in the *value* field This command is used in robot calibration procedures

For all of the remaining *chg* elements, the only relevant setting for the *com* field is true (non-zero)

chg setparam is an array of *CMDVAL* structures which are used for setting various control gains and parameters in the microcontroller for each joint Definitions for these parameters are contained in the file *<robot/addefs.h>*

chg.setparamALL sets various control gains and parameters for all the joint microcontrollers at once

chg adco instructs the robot controller to open the analog-to-digital converter channel described in the *value* field and post the readings from this channel in the *how* structure

chg hand controls the robot end effector At present the instruction is entered in the *value* field, available instructions are *OPEN* and *CLOSE*

chg.rate changes the control cycle rate The new rate is specified by an integer entered in the *value* field, legal values are presently 0 (72 Hz), 1 (36 Hz), 2 (18 Hz), and 3 (9 Hz)

chg checking changes the nature of the robot state checking done by the RCI system The *value* field is interpreted as a bit map describing which checking features are to be enabled The various bit values are defined in *<robot/errors.h>*

chg.outport1 sets the robot controller parallel output port 1 to the contents of the *value* field

chg pendant sets the display of the teach pendant according to the contents of the *value* field The protocol is described in the document "PUMA 260 Teach Pendant Protocol", in the file */local/doc/robot/teachPendant.txt*

chg zindex turns on calibration mode for those joints whose corresponding bits are set in the *value* field In calibration mode, the joint moves as requested until it reaches a reference mark in the optical encoder, at which point it freezes until a *STOPCAL* command is received This is used in robot calibration procedures.

The remaining commands have no value associated with them

chg.end informs the robot controller that the control session has been released

chg.stop stops all robot joints

chg.power_on turns on the robot arm power

chg power_off turns off the robot arm power

A.5 Error Handling and Recovery

The RCI system watches for various run-time error conditions such as hardware or communication failure, illegal command specifications in *chg*, and range errors on various robot variables

While the control task is open, any error occurring at the control level will cause control to be released and a *SIGHUP* signal to be delivered to the main program By default, this signal is caught by RCI software, which then prints a diagnostic message and causes the program to exit. In instances where it is desired to have the program recover from such errors, it is possible for the programmer to specify his/her own handler for *SIGHUP* (section A 5 3) The RCI software also catches any *SIGINT* (interrupt) signal received by the program, and by default causes a message to be printed and the program to exit This was done since *SIGINT* is commonly used to abort programs from the keyboard in which case it is desirable to have the RCI system catch the abort and close the control task properly

Under no circumstances should the user directly assign handlers for *SIGHUP* and *SIGINT* using UNIX system calls, since this would interfere with internal "bookkeeping" done by the RCI software

A.5.1 Termination codes

When an error is generated by the RCI control level, it writes a diagnostic error code into the global variable *RCIterminate* This is a 32 bit word, in which the high two bytes contain some status flags describing the error condition and the low two bytes contain a specific error number Error codes are defined in the file `<robot/errors.h>`

The robot range checking reports the following error conditions for any joint Any of these checks may be disabled by using the *chg.checking* command (described above)

R_MAXVEL – Maximum velocity exceeded.

R_MAXCUR – Maximum current exceeded

R_MAXPOS – Maximum position exceeded

R_REQVEL – Requested velocity too great.

R_REQCUR – Requested current too great.

R_REQPOS – Requested position out of bounds.

The following general errors are also reported by the RCI software:

R_TOOMANY – Too many commands sent to robot at once

R_CHKS – Bad checksum on data from received from robot controller.

R_BADCMD – Bad command specification in *chg* structure.

R_BTEST – Test failure of the VAX – robot hardware communication link.

A.5.2 User-generated error conditions

Under certain circumstances, the programmer may wish to force an error condition from within one of the control functions. This may be done by setting the `RCIterminate` variable to any nonzero value reserved for use by the programmer (which is any code not defined in *<robot/errors.h>*) It is recommended that this be done only for errors discovered at the control level. if an error condition is discovered at the planning level, it is probably best to call `RCIrelease()` instead

A.5.3 Specifying an error handler

The user specifies his/her own error handler by setting the pointer *user_hangup* to the desired handling routine Thereafter, upon receipt of a *SIGHUP* signal, the RCI software will first release control and then call this routine The user's handler may be "unset" by assigning *user_hangup* a value of *NULL* (*i e* . 0)

Similarly, a user interrupt handler may be assigned or deassigned by setting the function pointer *user_interrupt* The RCI software does not perform an automatic release upon the receipt of an interrupt signal.

A.5.4 Error diagnosis

The user hangup handler can determine the cause of the error by examining the contents of *RCIterminate*. To assist in printing diagnostics based on this variable, there is a function called *print_terminate_code()*, defined as follows:

print_terminate_code

— print diagnostic messages based on RCIterminate.

usage:

```
status = print_terminate_code (code, message_str)
int status, code;
char *message_str
```

description:

Prints out a line of the form

TERMINATE/ <message> / <reason>

where *<message>* is the string given by the argument *message_str*, and *<reason>* is an information string appropriate to the termination condition described by *code*. If the error condition is not recognized, *<reason>* is not printed and the function returns -1. If *message_str* is *NULL*, then no *<message>* is printed.

A.5.5 Dispatching from an error handler

Once the error handler has determined the nature of the error condition, it is usually desirable to return to the main program for purposes of effecting recovery. Since control has been released, performing any further robot actions will require a call to *RCIcontrol()*; this is best done from the outside the error handler.

Control can be returned to the main program by using the UNIX *longjmp* package. This is a software "catch and throw" facility that saves a hardware context at a particular point in the code, and then allows control to later be returned to that point from a signal handler or some other place deeper in the code. A stylized example of its usage is given below:

```
1  #include <robot/rci.h>
2  #include <setjmp.h>
3
4  jmp_buf env;                    /* buffer for saving context      */
5
6  error_handler()                 /* user-specified error handler   */
7  {
8     int jump_code;               /* code describing recovery action */
9
10    ... examine 'terminate' and decide what to do ...
11    ... set jump_code to reflect the decision ...
12
13    longjmp (env, jump_code);    /* Return to main program         */
14 }
15
16 main()
17 {
20    ... other definitions ...
21
22    RCIopen();
23    user_hangup = error_handler; /* set up error handler           */
24
25    if ((jump_code = setjmp(env)) != 0)
26     { switch (jump_code)
27        {
28           ... different error recovery actions
29               and possible calls to RCIcontrol() ...
30        }
31     }
32
33    RCIcontrol(f1, f2);
34
35    ... main program ...
36 }
```

The call to *setjmp()* (line 25) performs the initial context save, at which point *setjmp()* will return with a value of 0. If an RCI error later occurs, the handler will first be invoked, and then control will be returned, via *longjmp()* (line 13) to the point where *setjmp()* first returned. *setjmp()* will then "return" again, though this time with the value specified by the last argument to *longjmp*. This value (assigned to the variable

jump_code) is then used in a switch statement to select the appropriate action routine The call to *RCIcontrol()* is made *after* the error handler is set, to avoid possible race conditions. For details on *setjmp()* and *longjmp()*, the user should consult the entry *setjmp*(3) in the UNIX Programmers Manual [Berkeley 83].

A.6 The RCC Command

RCI application programs must be compiled and linked according to a particular format. A special interface to the C compiler and linker, called *rcc*, is provided to insulate the user from most of these details

The *rcc* command is used just like the C compiler *cc*, with a few changes Programs are compiled and loaded according to a command line specification that looks like

rcc <planning level args> *REAL* <control level args >

The control level arguments include any object file, source file, or library which contains code or data structures that will be accessed at the control level The effect of placing these files to the right of the keyword *REAL* is to ensure that they are compiled and linked in such a way that the corresponding portion of the executable image may be locked in main memory when the program is run.

All the usual options to the C compiler, such as *-O*, *-g* *-c*, *-o*, etc , are applicable, and may be placed on either side of the *REAL* keyword One change from the C compiler is that the default output is named after the first source or object file in the command list, the *-o* option, of course, will override this A set of libraries is also referenced by default, including the RCI library and the math library, so the programmer does not need to specify these explicitly

We now give a couple of examples

A user writes an ordinary (non-RCI) program in the file *foo.c* and wishes to compile it, it contains no control code.

```
rcc foo.c REAL
```

will compile and load *foo.c* and place the output in *foo*. (This example does not specifically require the *rcc* command.)

Now we consider an RCI application written with two files defining the planning level, `foo1.c` and `foo2.o`, and one file containing control functions, called `ctrl.c`. To compile and load this, applying the C optimizer, the user may specify

```
rcc foo1.c foo2.o REAL ctrl.c -O
```

Control level object modules should be compiled using `rcc` and the `-c` option:

```
rcc -c REAL ctrl1.c ctrl2.c
```

will generate object modules `ctrl1.o` and `ctrl2.o`, which may be loaded later by placing them to the right of *REAL* in some subsequent call to `rcc`.

A.7 The Simulator

A simulator is available on which RCI applications may be run in non-real-time and without any of the control function restrictions. The principal use of this is in debugging control level software.

The simulator is a process which emulates the robot. When application programs are linked with the RCI simulator library instead of with the usual real-time RCI library, the RCI system will interface to this process, instead of the robot. Except for this, the program runs normally. The simulator library may be invoked by using the keyword *SIM* in place of *REAL* in the `rcc` command (section A.6), as in

```
rcc foo.c SIM track.o
```

The simulator itself is a program called ***simula***. It is necessary to start this program running before the RCI program is started. A typical invocation may look like

```
simula -p &
```

where the *&* is the shell directive to set the program running in the background. The *-p* option tells ***simula*** that the "robot" is initially in the *ready* position and is already calibrated. Another option is `-c`, which starts the program with the "robot" in the *nest* position and calibrated. If these two options are absent, the program will start with the

"robot" uncalibrated, and, as with the real robot, it will be necessary to run a calibration program before it will do anything else Lastly, there is the *-v* option (for *verbose*), which causes the simulator to print out information about its joint angles once every (simulated) second For this application, it is generally a good idea to run `simula` on a separate terminal, in the foreground

A succession of RCI programs may be run on the simulator, which will simply remember its state from one program to the next and continue to emulate the robot until it is explicitly killed

At the moment, the simulator is a kinematic one It operates in a simple way by parroting the joint level commands that the RCI system sends to it If a certain position is requested, then the simulator "moves" to that position, while exhibiting the required current. If a certain joint current is requested, then that joint current is "set" and the joint moves accordingly The model of the manipulator considers only the gravity loading and friction characteristics described in Chapter 4 Although Spartan, this is adequate to meet the testing needs of many RCI applications.

Appendix B. Friction Measurements

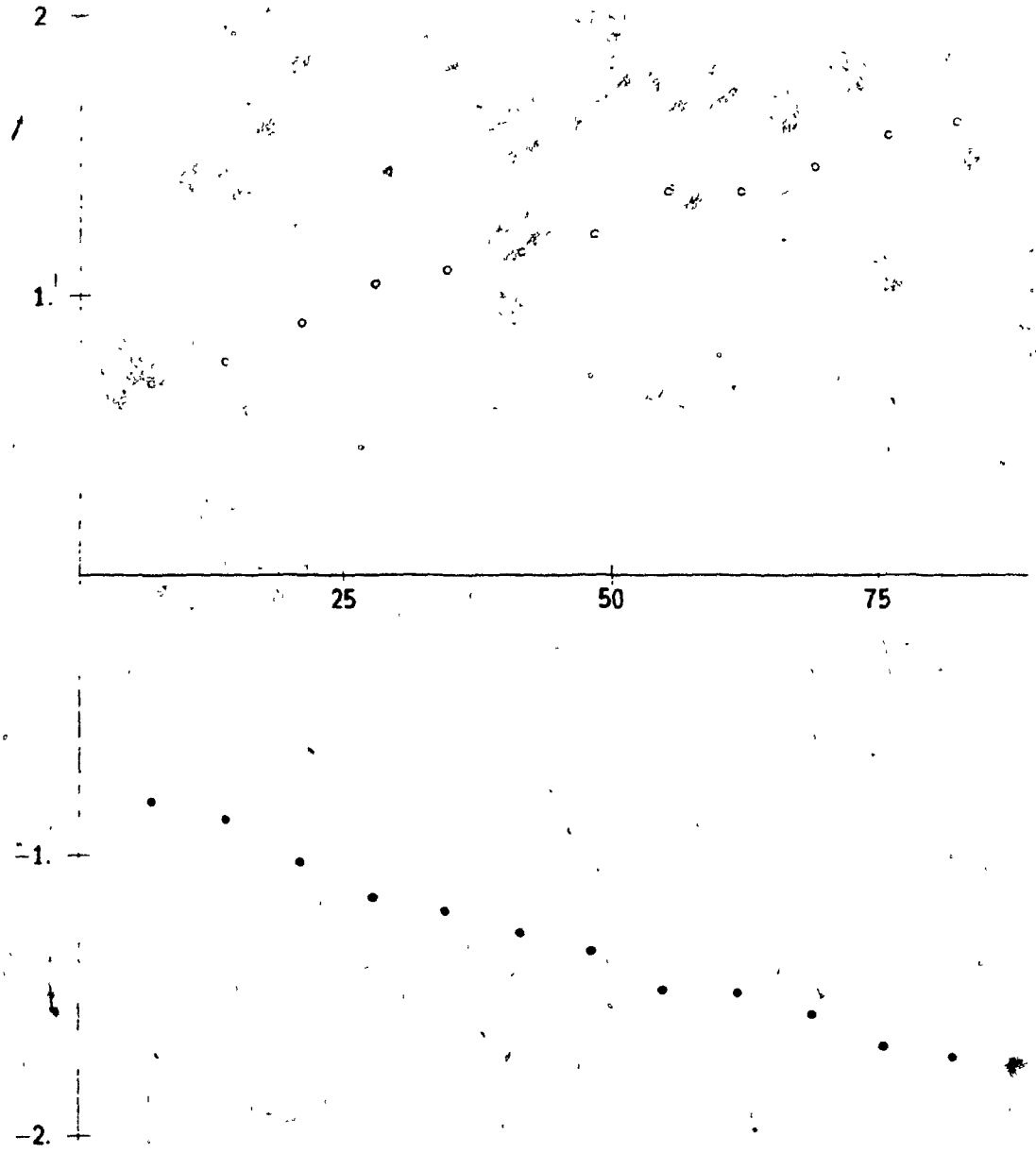


Figure B.1 Friction plotted against speed for joint 1 (Newton-meters vs degrees/sec)
o indicates (+) direction • indicates (−) direction

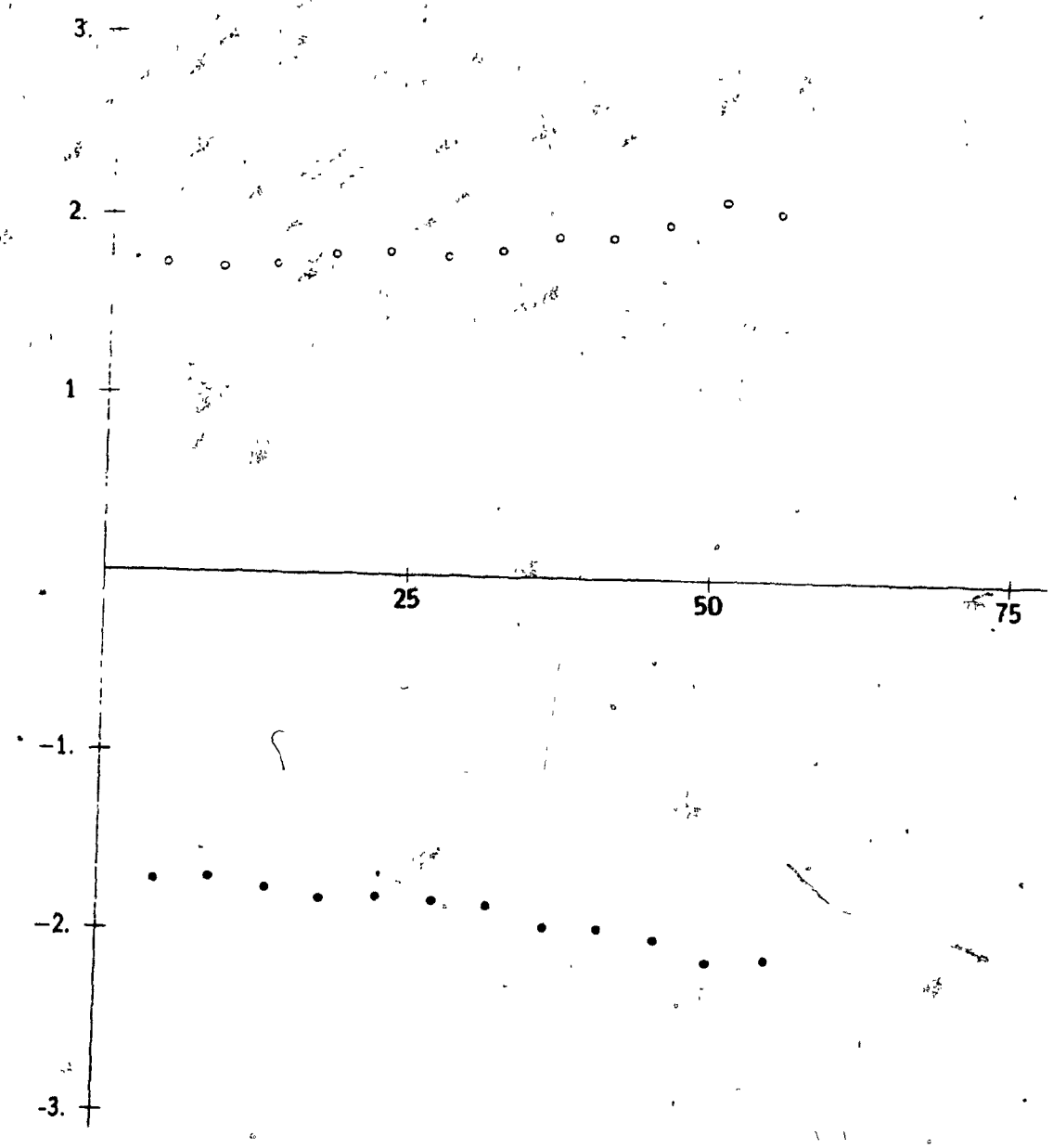


Figure B.2 Friction plotted against speed for joint 2 (Newton-meters vs degrees/sec)
o indicates (+) direction. • indicates (−) direction

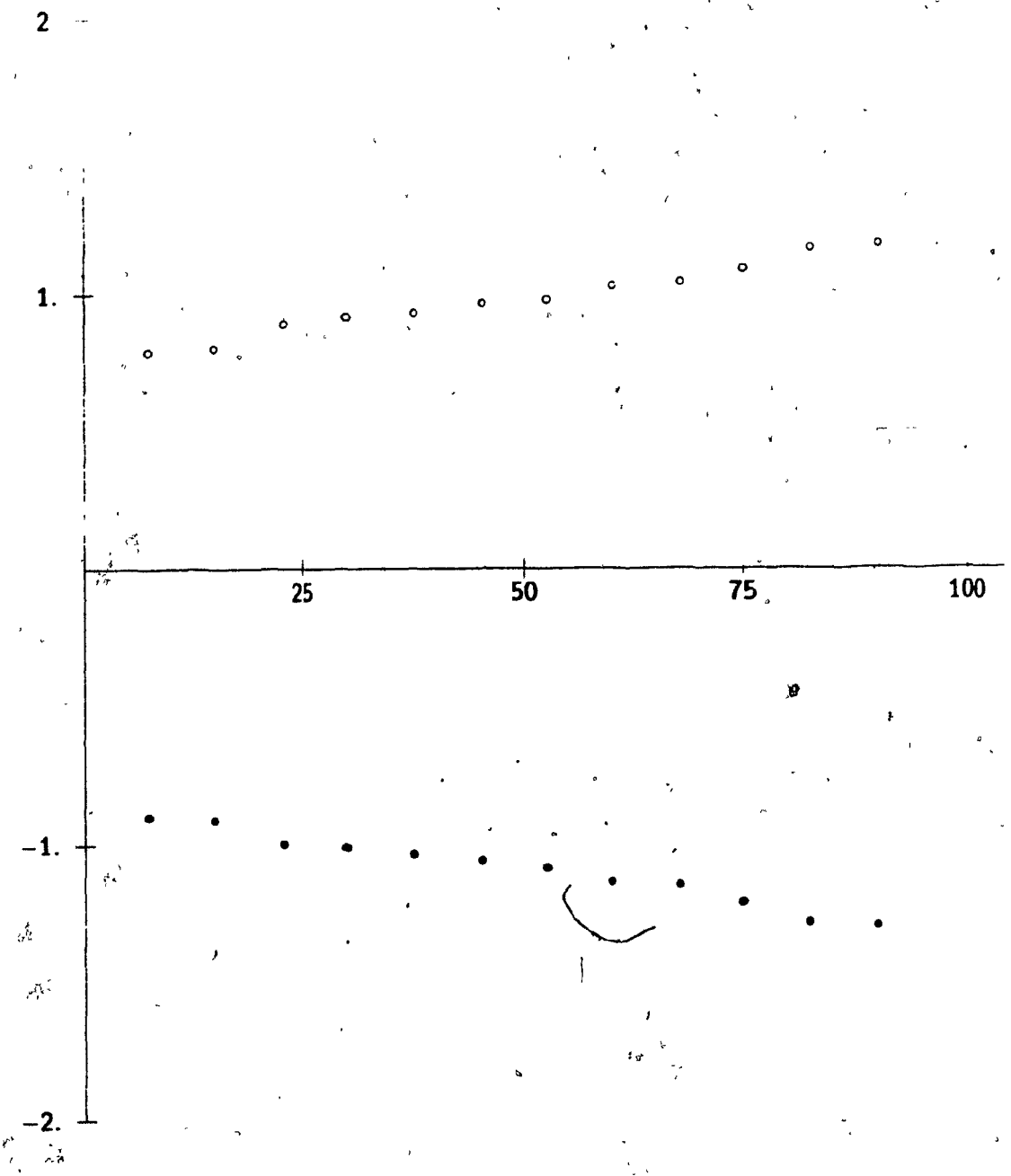


Figure B.3 Friction plotted against speed for joint 3 (Newton-meters vs degrees/sec)
○ indicates (+) direction. • indicates (−) direction

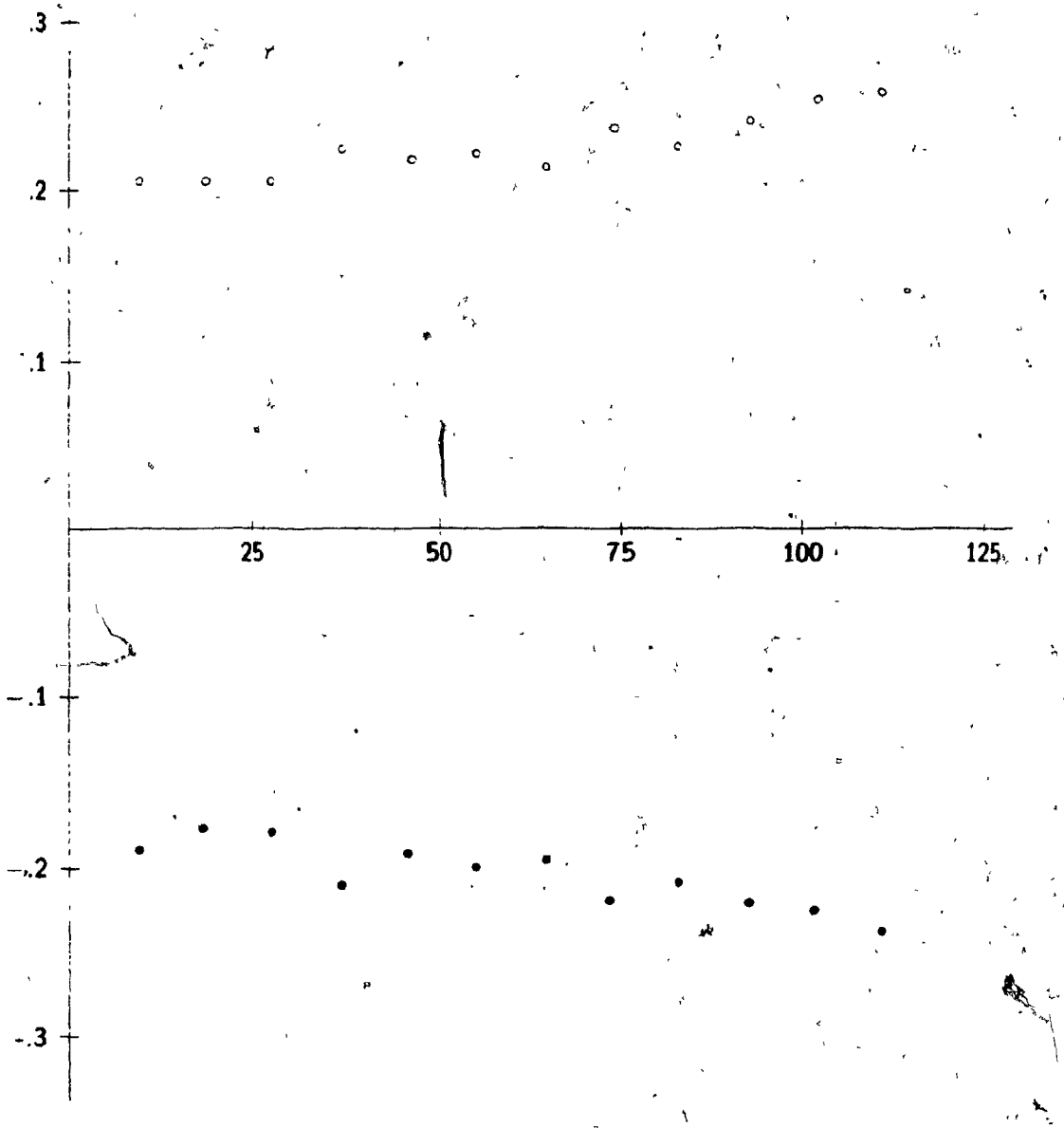


Figure B.4 Friction plotted against speed for joint 4 (Newton-meters vs. degrees/sec)
o indicates (+) direction. • indicates (−) direction

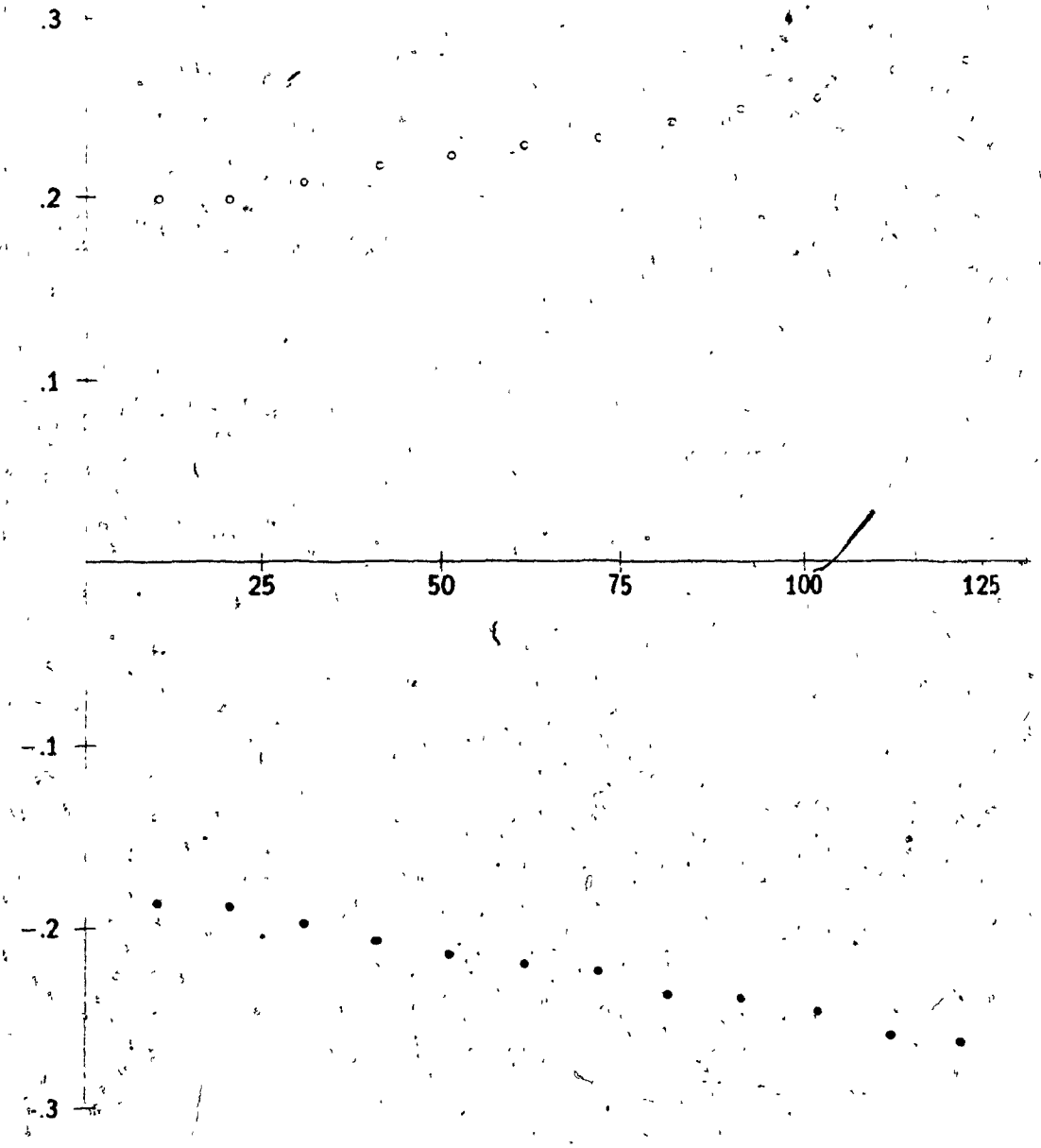


Figure B.5 Friction plotted against speed for joint 5 (Newton-meters vs degrees/sec)

o indicates (+) direction • indicates (−) direction

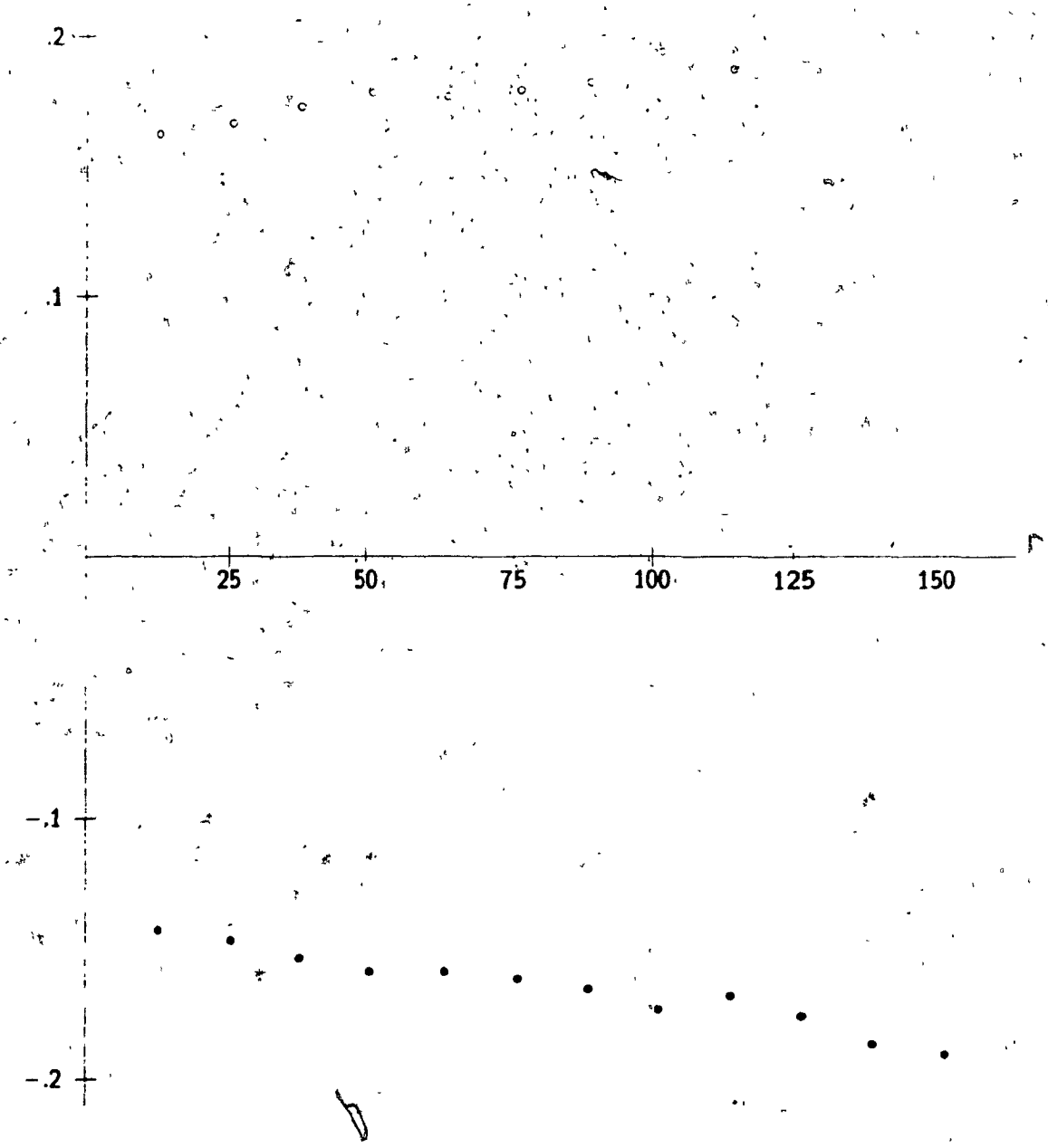


Figure B.6 Friction plotted against speed for joint 6 (Newton-meters vs. degrees/sec) ○ indicates (+) direction • indicates (−) direction.

Appendix C. Current Sensing Circuit

This is a description of the joint current measurement system

The motor current for each joint on the PUMA 260 is controlled by a separate microprocessor, which may operate in either a *position* mode, where the current is regulated so as to servo the joint to a desired location, or a *current* mode, where the current is simply set to some value specified by the supervisor controller (or RCI control task) The position mode uses a PID control algorithm which results in highly quantized output values for the joint currents Since the current values themselves are used as a measure of the joint torques, such quantization can lead to large errors in these values One alternative would be to replace the joint microprocessor software with a control algorithm developed to suit our needs This would however, have been quite costly A simpler solution was to make use of the fact that the joint controllers run at a sample rate of 1000 Hz, whereas the joint torque estimation operates at a (typical) rate of 36 Hz. It was therefore possible to put a low pass filter into the loop, between the current sensing resistor and the analog-to-digital converter, to provide averaging of the current measure

The whole current circuit is shown in Figure C 1, and the sensing filter itself is shown in Figure C 2 The current signal from the joint microprocessor is amplified, sent to the motor and returns through a low-valued sensing resistor, whose value is the R_s in equation (4 32) The voltage drop across this resistor can then be measured, and used to determine the current Because the system has a very low impedance, noise proved to be an important design consideration The voltage signal across R_s is measured with a differential amplifier, and then averaged by an active lowpass filter with a time constant of 15 milliseconds This averaged signal is then sampled by the analog-to-digital converter The gain of the averaging circuit, in conjunction with the gain of the conversion ratio of the analog-to-digital converter, determines the V_g in equation (4.32).

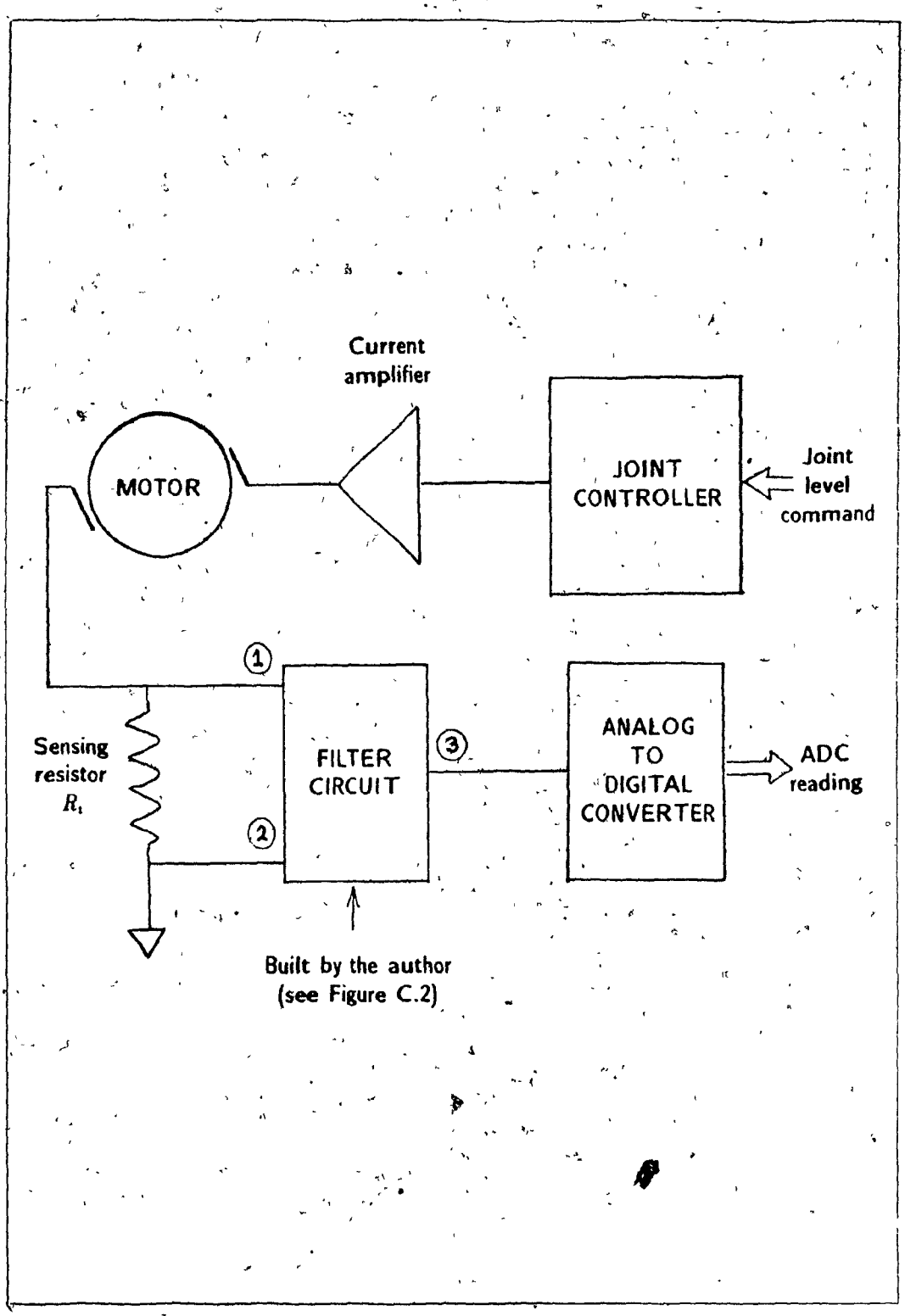


Figure C.1 Joint motor current circuit

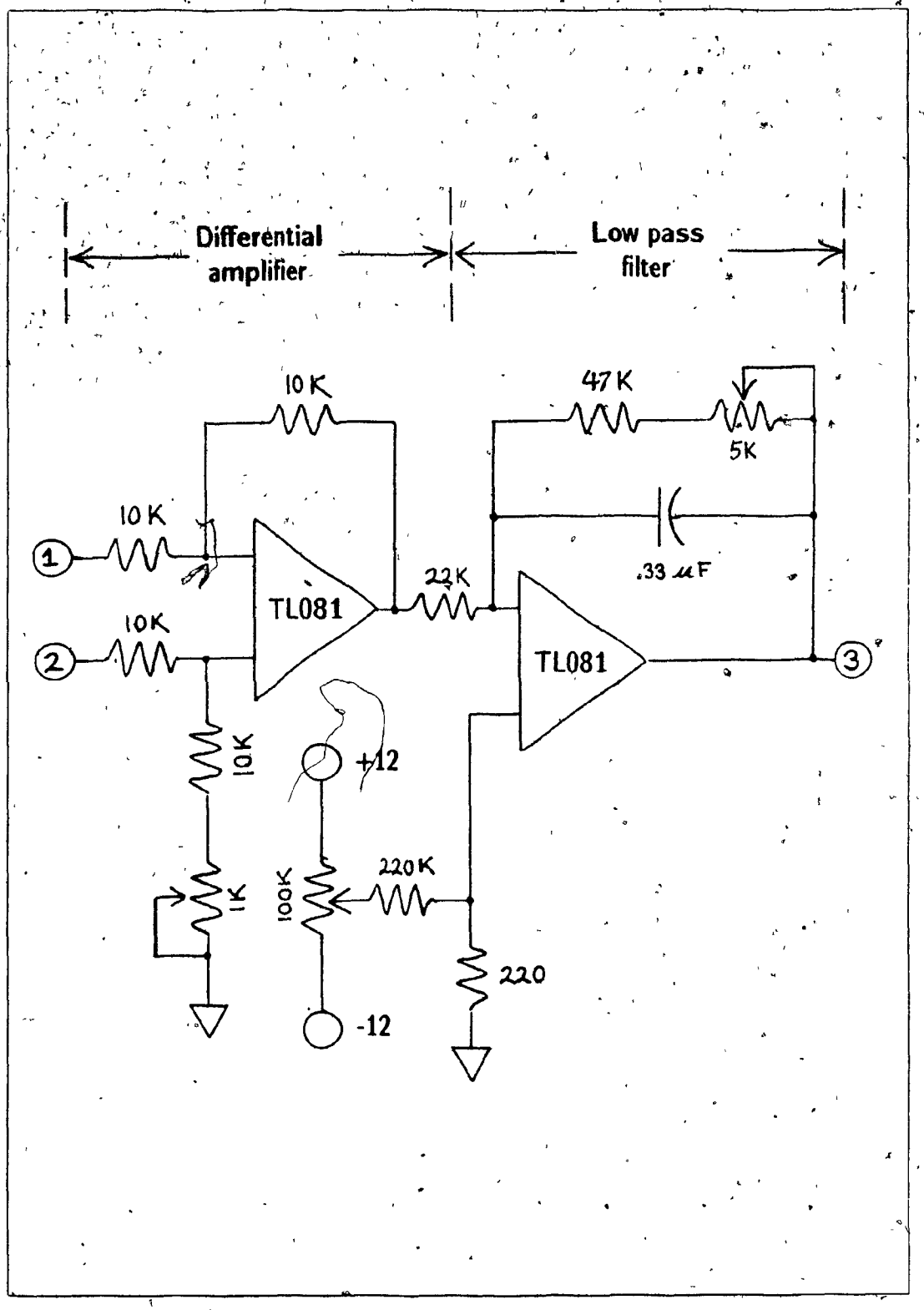


Figure C.2 Current sensing filter circuit

Appendix D. Miscellaneous PUMA 260 Constants

Mechanical parameters for the PUMA 260, as defined in section 4.4.3:

Gear ratio matrix, from equation (4.28):

$$\mathbf{G} = \begin{pmatrix} g_{1,1} & & & & & \\ & g_{2,2} & & & & \\ & & g_{3,3} & & & \\ & & & g_{4,4} & & \\ & & & g_{5,4} & g_{5,5} & \\ & & & g_{6,4} & g_{6,5} & g_{6,6} \end{pmatrix}$$

where

$$g_{1,1} = -46.7200$$
$$g_{2,2} = 69.9733$$
$$g_{3,3} = -42.9867$$
$$g_{4,4} = -43.5111$$
$$g_{5,5} = 39.3846$$
$$g_{6,6} = 31.7692$$

and

$$g_{5,4} = -9.8462$$
$$g_{6,4} = 1.0000$$
$$g_{6,5} = -6.9424$$

Encoder count matrix R (encoder counts per radian):

$$\mathbf{R} = \begin{pmatrix} r_{1,1} & & & & & \\ & r_{2,2} & & & & \\ & & r_{3,3} & & & \\ & & & r_{4,4} & & \\ & & & r_{5,4} & r_{5,5} & \\ & & & r_{6,4} & r_{6,5} & r_{6,6} \end{pmatrix}$$

where

$$r_{1,1} = -7435.72$$
$$r_{2,2} = 11136.60$$
$$r_{3,3} = -6841.55$$
$$r_{4,4} = 5540.00$$
$$r_{5,5} = 5014.60$$
$$r_{6,6} = 4044.98$$

and

$$r_{5,4} = -1253.65$$
$$r_{6,4} = 127.32$$
$$r_{6,5} = -713.01$$

ADC to current conversion constants, referred to in equation (4.32)

$$V_g = .00111$$

$$R_1 = .301$$
$$R_2 = .300$$
$$R_3 = .302$$
$$R_4 = .821$$
$$R_5 = .821$$
$$R_6 = .816$$

$$\kappa_i = 1.333 \quad \text{(for all } i\text{)}$$

SOURCE Unimation field service manual (for G and R); direct measurement (for R_i and κ_i), and the known gain of the current sense filter in conjunction with the analog-to-digital converter configuration (for V_g)

Appendix E. Implementation of RCCL trajectory features

E.1 Joint and Cartesian Trajectories

This appendix deals with the operation of the RCCL trajectory generator For the sake of brevity, the polynomial transitioning between path segments will not be dealt with. The method of trajectory generation is described in [Paul 81], and related methods are discussed in [Taylor 79]

If the trajectory is to be computed in joint mode then the joint position Θ_a of the manipulator at the start of the path segment is compared with the calculated joint position Θ_b at the end of the path segment Linearly interpolating the joint positions between these two values yields a straight line motion in joint space

Cartesian motions are slightly more complex since they require a transformation from Cartesian to joint coordinates The position of the end of the robot with respect to the base of the robot coordinate system is defined by the $\mathbf{T}_6$ matrix (section 3 3) Let $\mathbf{D}_a$ and $\mathbf{D}_b$ denote the values of $\mathbf{T}_6$ at the beginning and end of the path segment respectively If the motion begins at time t_a and ends at time t_b this means that

$$\mathbf{T}_6(t_a) = \mathbf{D}_a \qquad \text{(beginning of motion)} \qquad \text{(E 1)}$$

$$\mathbf{T}_6(t_f) = \mathbf{D}_b \qquad \text{(end of motion)} \qquad \text{(E 2)}$$

A special *drive* transformation $\mathbf{C}_{drive}(t)$ may be defined whose value varies in time and satisfies the boundary conditions

$$\mathbf{C}_{drive}(t_a) = \mathbf{I} \qquad \text{(beginning of motion)} \qquad \text{(E 3)}$$

$$\mathbf{C}_{drive}(t_f) = \mathbf{D}_a^{-1}\,\mathbf{D}_b \qquad \text{(end of motion)} \qquad \text{(E 4)}$$

The intermediate values of the drive transform are computed so that it traces a straight line in Cartesian space between its two end values ([Paul 81] Chapter 5) Hence if the value of $\mathbf{T}_6$ is tied to the drive function so that

$$\mathbf{T}_6(t) = \mathbf{D}_a\,\mathbf{C}_{drive}(t) \qquad \text{(E 5)}$$

then $\mathbf{T}_6$ will also trace out a straight line yielding the required straight line Cartesian motion The trajectory generator continuously uses (E 5) to evaluate $\mathbf{T}_6$ at its running

sample rate and then uses the inverse manipulator kinematics (section 3.4) to calculate the associated joint angles.

This method could be generalized to produce a trajectory that follows any curve in space for which a drive transform parameterization can be determined. A trajectory will be produced whose exactness is limited by the joint repeatability (a fundamental limitation), the accuracy of the inverse kinematic model and the control sample rate*. Computationally, however, the method is expensive, requiring an inverse kinematic calculation to be performed at each sample interval.

E.2 Adaptive Trajectory Features

The "stop on condition" features (sections 5.4.1 and 5.4.2) are straightforward to implement. Path deviation and force limits are specified in the Cartesian coordinates of the manipulator *tool* frame. As the motion proceeds, the observed errors in the joint angles may be transformed into Cartesian position errors at the wrist using the manipulator Jacobian **J** and equation (3.1), and these may in turn be transformed into Cartesian errors in the tool frame using equation (1.13). Similarly, force measurements may be made in Cartesian space using the methods described at the end of Chapter 4.

The compliance (section 5.4.3) is done using the Paul and Shimano algorithm [Paul and Shimano 76]. This involves extending the drive equation (E.5) to include a *comply* transformation $\mathbf{C}_{comply}(t)$ according to

$$\mathbf{T}_6(t) = \mathbf{D}_a \; \mathbf{C}_{drive}(t) \; \mathbf{C}_{comply}(t) \qquad (E.6)$$

As the robot moves along its trajectory, observed deviations in the path along/about the compliant axes are permitted by incorporating them into the comply transform; this effectively allows the path to vary freely in the compliant directions. At each position the joint best suited to provide a force/torque along/about each compliant axis is selected and force/torque servoed instead of position servoed. Computing the required force/torque is done using the methods described in section 4.7. The procedure is effective when the configuration of the manipulator is well suited to the compliance problem of interest. However, the configuration can often be such that no one joint is particularly well positioned to

* The control sample rate is of concern since the motions between each sample interval and the next are (necessarily) joint interpolated which causes a small amount of ripple in the computed path.

provide the necessary compliance Using more than one joint per compliant direction would render the manipulator unable to maintain its position profile along the non-compliant axes Several research efforts in the past several years have focussed on this problem in an attempt to resolve it using a hybrid control paradigm where each of the robot s joints is controlled in a unified way that considers both the positional and force requirements in Cartesian space [Raibert and Craig 81 Hong and Paul 85] Ultimately, the effectiveness of the compliant control is restricted by the condition of the manipulator Jacobian at points where the Jacobian is singular, the dimension of the range of the Jacobian is reduced and it becomes impossible to move or exert forces along certain Cartesian directions As described at the end of Chapter 4 some improvement in performance can be achieved by using external force sensors to close the force control loop in Cartesian space

The last adaptive feature is the ability of the programmer to tie a function to the position associated with a path segment (section 5 4 4) As discussed in section 5 3 1, the transform describing the path destination position ($\mathbf{D}_b$ in equation (E 1)) may actually be composed of several transforms

$$\mathbf{D}_b = \mathbf{D}_1 \mathbf{D}_2 \cdot \cdot \mathbf{D}_n \qquad (E.7)$$

These transformations may be bound to functions which modify them in real-time Each modifying function is called by the trajectory generator, once every sample interval during the time the motion segment associated with $\mathbf{D}_b$ is being executed

Care is needed when using functionally defined transforms to ensure that the changes do not cause abnormally large velocities or accelerations in the manipulator Since the actual change at any point along the trajectory is likely to be small, it may be related to the corresponding change in joint coordinates by the manipulator Jacobian

$$d\Theta = \mathbf{J}^{-1}\, d\mathbf{C} \qquad (E.8)$$

As long as the manipulator is not too close to a singularity, one may ensure the required behavior by limiting the size of $d\mathbf{C}$ to an appropriate value determined by the condition number of the Jacobian