A Multi-Microprocessor-Based Control Environment for Industrial Robots

Don Kossman

March 1986

Computer Vision and Robotics Laboratory
Department of Electrical Engineering
McGill University

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

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

Abstract

This thesis describes work done to move the Robot Control C Library, RCCL, and its underlying Real Time Control system, RTC, from a VAX/UNIX environment to a multi-microprocessor system, and to extend it to another robot, the Microbo Ecureuil. The task included: designing and implementing an interface between the Microbo's joint controllers and a Multibus system; solving the robot's forward and inverse kinematics; designing a multi-microprocessor architecture which has the processing capability to support RCCL's computational load; redesigning the RTC layer so that it runs under Intel's iRMX-86 real-time multi-tasking operating system on the multi-processor system; and creating a usable development environment for RCCL users. It is shown that this system is flexible and expandable, and opens the way to the implementation of a multi-robot programming and control environment for the McGill Computer Vision and Robotics Laboratory.

Résumé

Cette thèse décrit le travail fait en vue de transporter une bibliothèque de programmes de contrôle de robots écrits en language C, RCCL (Robot Control C Library), ainsi que son système de support en temps réel, RTC (Real Time Control), de son environnement VAX/UNIX à un environnement de multi-microordinateurs et d'en étendre son usage à un autre robot, soit l'Ecureuil de Microbo. Les tâches à effectuer furent les suivantes: la conception et la mise en oeuvre d'un interface entre les régulateurs d'articulations du Microbo et le système Multibus de Intel; la détermination des modèles géométriques et cinématiques directs et inverses; la conception d'une architecture multi-microordinateurs d'une puissance suffisante pour supporter RCCL; reconcevoir le niveau RTC de façon à ce qu'il fonctionne avec le système d'exploitation en temps réel multi-tâches iRMX86 de Intel sur le système de multi-microordinateurs; créer un environnement de travail adéquat pour les utilisateurs de RCCL. Cette thèse montre que ce système est flexible, facile à étendre et ouvre la voie à la mise en oeuvre d'un environnement de programmation et de contrôle pour multi-robots au laboratoire de vision par ordinateur et de robotique de l'Université McGill.

Acknowledgements

I would like to thank my thesis advisor, Dr. Alfred Malowany, for his advice and support. Thanks also to Paul Freedman and Gregory Carayannis, who wrote RCCL demo programs which were invaluable in finding and correcting various "features" of the system. Paul also proof-read the first draft of this work. Dr. Laeeque Daneshmend provided helpful insight towards understanding certain control algorithms, and mention must be made of the many valuable discussions held with John Lloyd and Dr. Vincent Hayward.

This work was implemented using equipment and software generously donated to the McGill Computer Vision and Robotics Laboratory by Intel Corporation of Santa Clara, California. Financial assistance was provided by the McGill Computer Vision and Robotics Laboratory.

March 1986

Contents

Chapter 5 Summary and Conclusions

List of Figures

Chapter 1 Introduction

1.1 Motivation

As the pace of technology increases, the use of robots in manufacturing becomes more and more pervasive. Whereas 10 years ago a telephone set manufacturer, for example, could confidently invest in hard automation equipment to produce consumer telephones, nowadays that product might significantly change form every year. The manufacturer is wise to invest in a roboticized factory which can be quickly re-programmed to produce new models, even if at a slower rate than the hard-automated factory [Isbister 84]

Whereas the first industrial robots were used for relatively simple and repetitive paint-spraying, welding, and "pick and place" kinds of tasks, manipulators may nowadays be used for complex and precise assembly operations requiring sensory interaction. Computer-aided manufacturing techniques mean that product information is available throughout the design/test/manufacture/repair cycle, and it is desirable that a robot programming system be able to take advantage of this information to minimize tedious on-line "teaching" time [Bonner and Shin 82]. Modern robot workcells, moreover, may consist of several robots in addition to support machinery, sensors, and vision systems.

With the increase in the complexity of robotics systems and tasks comes a need for robot programming and control environments in which researchers can test new algorithms and control methods quickly and easily. One such environment is known as the Robot Control C Library, or RCCL, and was introduced by Hayward and Paul [Hayward and Paul 83] at Purdue University in 1983. The RCCL environment consists of two levels, a control

level, called the Real-Time Control system or RTC, and the RCCL trajectory controller, which uses RTC as a substrate. RCCL was implemented at Purdue for a PUMA 560 manipulator connected to a VAX 780 minicomputer running the Unix 4.2Bsd operating system. This system has recently been installed and enhanced[†] in the Computer Vision and Robotics Laboratory (CVaRL) in the Department of Electrical Engineering at McGill University [Lloyd 85] for a PUMA-260 robot connected to a VAX 750 minicomputer again running Unix 4.2Bsd.

One of the drawbacks of this implementation is the intensive real-time computational load that it imposes on the multi-user VAX/Unix environment. Under some conditions the system may become unusable; and it is not possible to concurrently control a second manipulator for the same reason.

In this thesis, the above restrictions were addressed by re-implementing the RTC/RCCL environment for another robot, the Microbo Ecureuil, using a multi-microprocessor-based system. The task included studying the basic feasibility of using a microprocessor to handle the floating-point math involved in the robot kinematics; designing and implementing the hardware to interface the Microbo's joint controllers to a microprocessor system bus; solving the Microbo manipulator's forward and inverse kinematics and coding the resulting algorithms; designing a multi-microprocessor architecture which has the processing capability to support RCCL's computational load; redesigning the RTC layer so that it runs under Intel's iRMX-86 real-time multi-tasking operating system on the multi-processor system; and creating a usable development environment for RCCL users.

It is shown that this system is flexible and expandable, and opens the way to the implementation of a multi-robot programming and control environment.

1.1.1 CVaRL Research Goals

One of the CVaRL research goals is to use multiple robots cooperatively, along with vision and force feedback, to implement a robotics workcell. Ultimately, a world model and database will be established which application programs will access and modify as they work. Expert systems will interact with the database to perform high-level task planning.

[†] RTC was in this case renamed "RCI", for Robot Control Interface; for simplicity we will use the "RTC" acronym for all implementations of this software.

Research is oriented towards investigation of algorithms, control strategies [Studenny and Bélanger 84], robot languages, database models, and Local Area Networking [Freedman 85], which are needed to construct a multi-robot workcell [Michaud, et al 85] The status of current research is summarized in [CVaRL 85].

An example task of the workcell is to inspect and repair hybrid integrated circuits The database will hold information concerning the dimensionality and possible faults of these circuits Because of the small dimensions involved, high precision and fine control of robot motion are required, along with the capacity for various kinds of sensory feedback at the trajectory level

1.1.2 The Configuration of CVaRL

The lab currently contains three industrial assembly robots: a Unimation Puma-260, a Microbo-Castor Ecureuil, and an IBM 7565. The first two robots are arranged in a workcell such that their workspaces overlap (see Figure 1.1). The Puma is an anthropomorphic robot with 6 degrees of freedom, and the Microbo a cylindrical robot with 2 prismatic and 4 rotational joints In addition to the robots, there is an x-y stage, linear stage, rotary stage, and force sensor. The vision system interfaces with several types of cameras, including CCD devices which can be manipulated by the robots, there is also a computer-controlled microscope These are connected to a Grinnell GMR-27 image acquisition system on a VAX-780 minicomputer. The Puma robot has a a Unimation controller which is interfaced to a VAX-750 minicomputer via a high speed parallel link. The Microbo robot's Robot Control Unit (RCU) controller is connected to an Intel System 310 via a high-speed bus adapter. The Intel system is Multibus-based, and contains 80286/287 and 8086/87 CPU cards, high speed memory, a Winchester disk, and a floppy disk This system is currently connected to the VAX-750 by a serial link, although we plan to eventually install an Ethernet interface

The two VAX computers are connected together, and to the rest of the McGill Electrical Engineering computing resources, via an Ethernet link. Figure 1 2 shows the computer architecture of the workcell

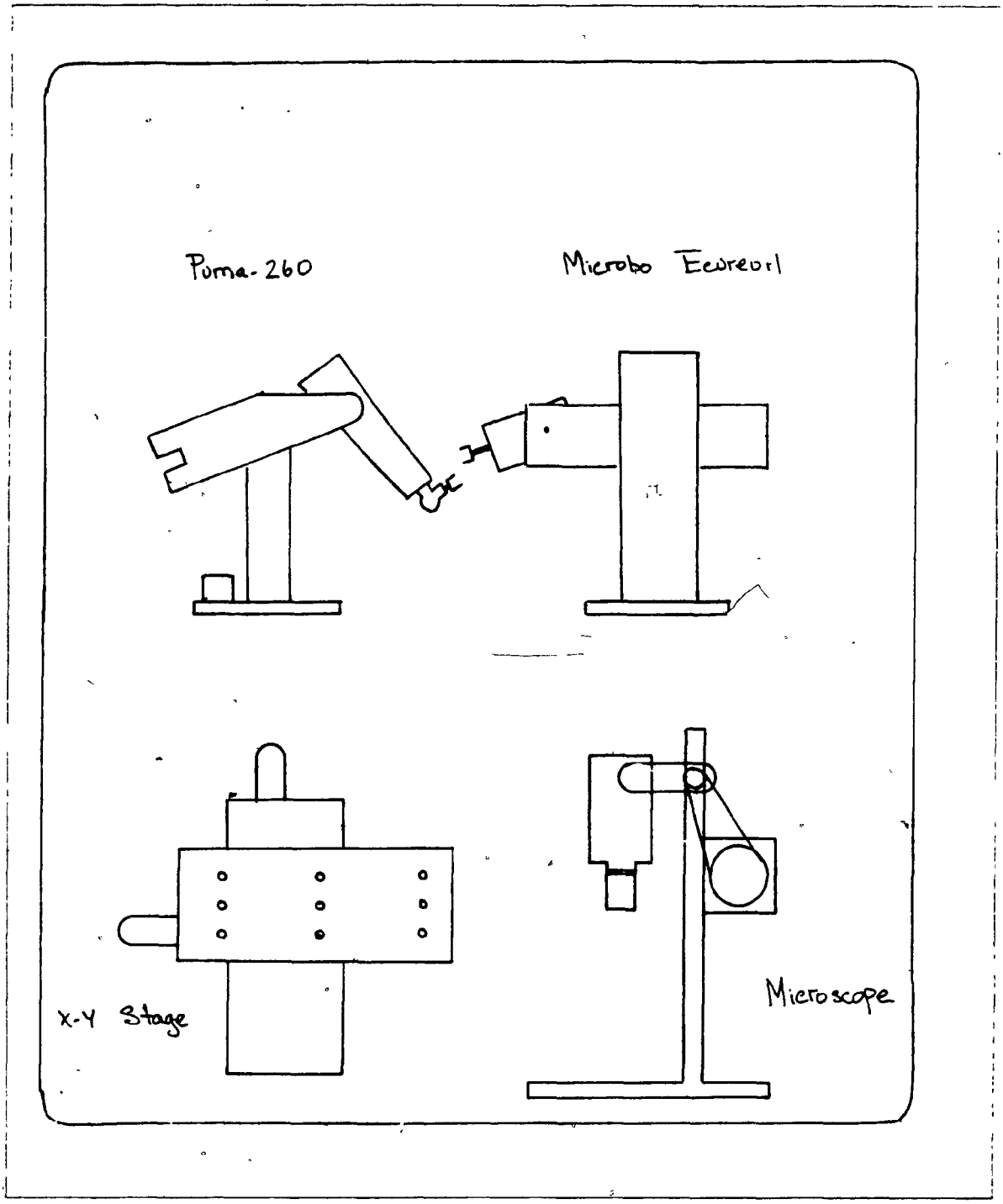

Figure 1.1 CVaRL Robot Workcell

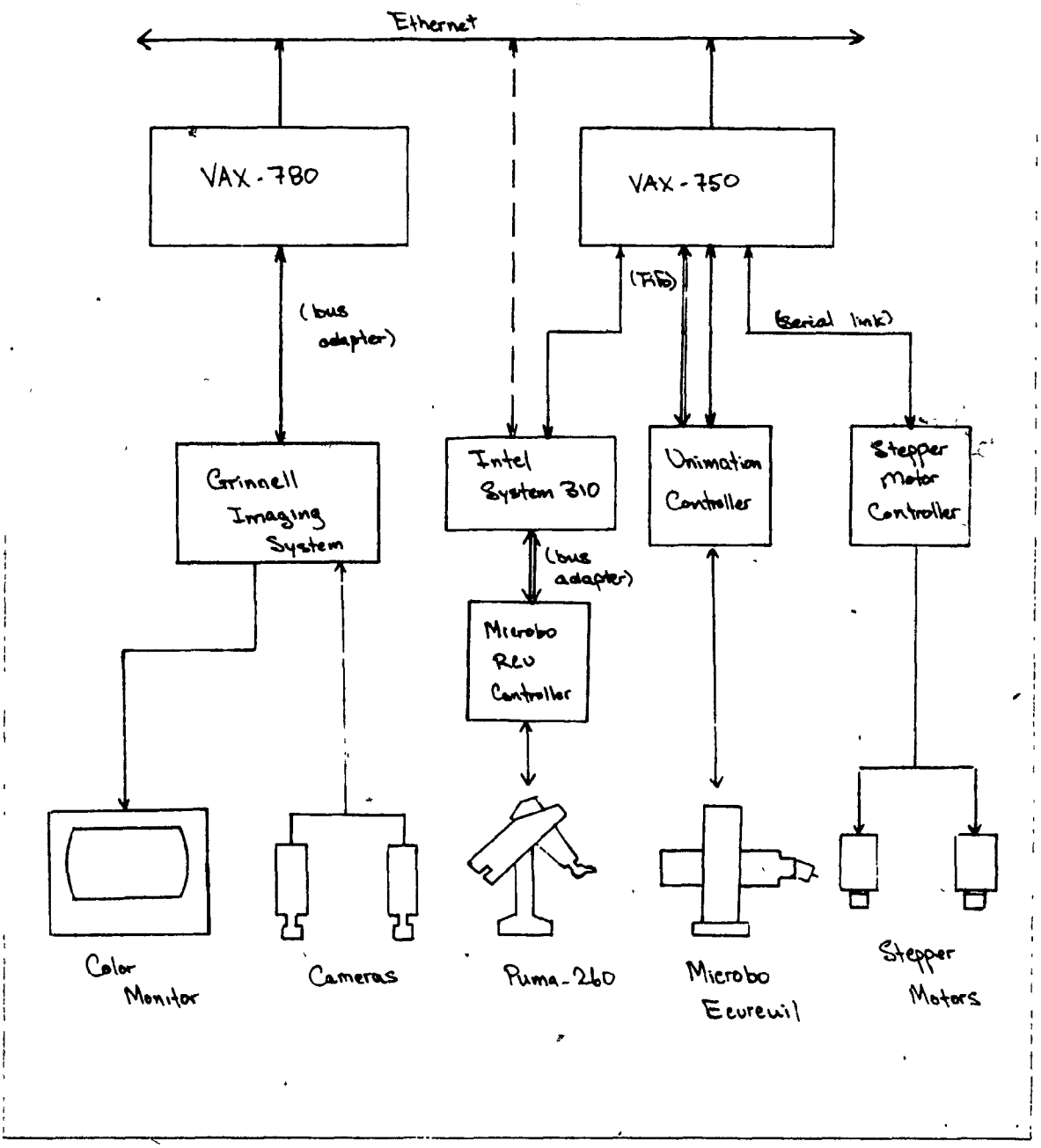

Figure 1.2 Current Robot Workcell Computer Architecture

1.1.3 History of Robot Control at CVaRL

The Puma robot was originally controlled using the VAL-1 language [Unimation 82], which ran on the Unimation (LSI-11/6502-based) controller Although VAL-1 provided Cartesian motion, it did not provide more advanced features such as force control, sensor integration, host computer interface, motion interruption, or multi-robot capabilities VAL-II [Unimation 83] does provide more functionality, but was still unsuitable, this will be described in the survey of robot languages presented in Section 1.3 As mentioned at the beginning of the introduction, the Robot Control Library, RCCL, has recently been installed for the Puma manipulator RCCL executes on a VAX-750 under the Unix 4 2Bsd operating system, and provides an excellent set of basic tools for developing robot software at different levels: it addresses most of the needs of CVaRL researchers.

The Microbo robot was controlled using the vendor's IRL language [Dupont 84], running on the 8085-based RCU controller. The IRL language is even more primitive than VAL, allowing motion to be specified only in joint coordinate space. IRL, like VAL-1, does not permit changing the trajectory control algorithms IRL, in fact, does not even provide coordinated straight line motion The individual joint processors perform an acceleration and deceleration function which is combined with the joint servoing function.

Thus, programmers wishing to develop coordinated robot software had two completely different environments to deal with: the RCCL system running on the VAX, and the primitive IRL interpreter running on an 8085-based controller. The initial solution was to connect the IRL controller to a VAX serial line and write terminal emulation programs which sent IRL commands in the form of ascii strings to the controller at 9600 baud [Michaud 85] Obviously, above and beyond the complete lack of features for trajectory control of the Microbo, there was no commonality between the two environments, making it difficult for programmers trying to coordinate the actions of the two robots and synchronize events in the workspace A more powerful and integrated programming and control environment was needed, and the obvious step was to investigate ways of extending RCCL to control the Microbo

1.2 Extending RCCL to the Microbo Robot

A major drawback of the VAX-based RCCL implementation is that the intensive real-

time computational load consumes the major portion of the VAX's CPU time RCCL is built around an RTC control routine (the "setpoint" function) which is interrupt-driven by the robot at the sample rate (approximately 30 Hz) User programs call RCCL routines which ultimately place motion requests, in the form of position transforms, on a "motion request queue" The setpoint routine, which runs in high priority "kernel mode", performs the real-time computations which convert the motion requests into joint angles These are then sent to the robot to actually implement the motion Although optimized, the computations involve lengthy matrix multiplications and trigonometric calculations in floating point.

Measurements showed that our VAX-750, even with a floating point accelerator, can generally execute RCCL for the Puma robot at a sample rate of 30 Hz; under worst-case conditions, the rate must be decreased further, or the overall performance of the machine decreases unacceptably Obviously, then, the VAX-based RCCL configuration would not be capable of concurrently handling the additional load of a second robot and maintaining the required control frequency. As one of the goals of our research lab is to develop a multi-robot system, this is a major limitation if we wish to use RCCL as a programming tool. Replacing the VAX with a more powerful machine and more costly was not possible, so a new architecture was adopted.

This thesis demonstrates that a reasonable alternative involves the use of a network of relatively powerful but inexpensive microcomputers We describe an implementation which supports the Microbo Ecureuil manipulator, using Intel 80286/80287 and 8086/8087 microprocessors.

1.3 Robot Programming and Control

Just about all Robot Programming and Control Environments (RPCEs) may be seen to consist of two major parts a real-time control system, and a programming system In some RPCEs the two parts are independent, in that the control system is not user-programmable; this tends to be true of most vendor-supplied systems, Unimation's VAL-1 [Unimation 82] is a typical example. Other systems (for example the RCCL/RTC environment) do allow the user to program the control level.

The programming system represents the interface between the robot and the rest of the world, usually in the form of a textual language or teach pendant, where input is at the

task planning level. Less often, the programming system is fed from the output of a more sophisticated task planner, which automatically generates planning level statements given a comprehensive world model and the task specification.

We designate the *control system* as that part of the RPCE which directly commands the robot's joint actuators In order to maintain the response of the manipulator according to goals set by a programmer, the control system must in general solve the so-called *inverse plant* problem. That is, it must compute, as a function of time, the actuator signals required for the physical plant (robot) to behave in the desired way

A survey of the literature reveals that researchers tend in general to focus their efforts in one of the two directions towards the programmability aspect, or towards the control aspect The former is the major issue for those involved with turn-key robot systems which must be programmed by production-level personnel, the control system is embedded, and usually unchangeable. As a result, researchers investigating control methods and algorithms often are forced to build their own programming environments so that they can access the control level of these systems

In the following sections, we present a representative survey of the approaches that have been taken by robot language developers The important aspects of the resulting RPCE's are categorized and their advantages and disadvantages are described. An attempt is then made to synthesize ideas from the various approaches, and some conclusions are drawn about the requirements of a good RPCE

1.3.1 A Survey of Robot Languages

The earliest RPCE's were simple, non-textual teach-and-repeat systems programmers used a teach pendant, joystick or similar device to lead the robot through a series of positions which could be recorded in terms of sets of joint angles The robot performed by moving in sequence through the memorized positions, generally with some kind of joint interpolated motion. An example of such a system was Cincinatti Milacron's T3 [Cincinnati 80]. This approach still finds wide use because of its inherent simplicity, production personnel have no difficulty teaching the robot motions. Operations such as spot welding and paint spraying, where the robot need simply repeat sequences with little chance of frequent modification and little environmental interaction are suitable applications. The

inadequacies of this approach, however, are easy to see

- deals badly with situations where there are very many points to be memorized or where the points are related in a regular way (eg palletizing).
- requires a very large number of intermediate points be recorded to achieve a predictable trajectory
- requires the robot to be taken off-line in order to be programmed.
- no integration with external databases
- no portability of programs between robots.
- any change in workspace configuration requires complete re-teaching.

For the above reasons, and because one of the basic justifications for using a robot instead of hard automation equipment is to add flexibility and allow short economical production runs, there was an obvious need for textual robotics languages. In answer to these needs, robot manufacturers, industrial researchers and universities have come up with a large array of languages. In [Bonner and Shin 82], robot languages are roughly classified by level as follows.

- task-oriented (most sophisticated)
- well-structured
- primitive motion
- point-to-point or joint-level (least sophisticated)

We have already given one example of a point-to-point language (T3). Some others are FUNKY [Grossman 77], RAPT [Popplestone, et al. 78] and ALFA [Wang 74]. We note that IRL, the vendor-supplied language for the CVaRL Microbo robot, falls into this category. These languages usually have an editing mode that allows the insertion or deletion of steps. There is only rarely provision for conditional branching, and there is generally no effective off-line programming mode. Motion is defined at the individual joint level only, and the

control systems associated with these languages are often correspondingly primitive, with no provision for Cartesian motion or force control

Languages at the primitive motion level begin to allow the user to describe the robot and its environment mathematically, solve the motion problem using algorithms expressed textually, and apply the solution to the robot. These systems may incorporate features such as subroutine capability, branching constructs, more sensor interaction, and the ability to synchronize with external events The programmer can specify motion at the manipulator level, sometimes in terms of coordinate systems grounded in the workspace Some examples of languages at this level are VAL, WAVE [Paul 77], and EMILY [Evans, et al 76], there are many others

Well-structured languages encompass the next step in sophistication. Structured control concepts, coordinate transformations, and complex data types such as vectors and frames are incorporated at the user level. We find the use of state variables, where the system automatically keeps track of important aspects of the workspace (such as the cartesian position of the manipulator end-effector) Motion may be constrained in terms of approach vectors, velocity, duration, acceleration, and force Some of these languages have advanced sensor capability, even vision A pioneering example of this level is the AL/POINTY [Mujtaba and Goldman 79] system, developed at the Stanford AI Laboratory. Some other systems of note are PAL [Takase 81] and IBM's AML [Taylor, et al 82] and AML/V [Lavin and Lieberman 82], again, there are many others.

Task-oriented languages hide the details of manipulator motion from the programmer, who can now concentrate on solving problems in terms of the items being manipulated as opposed to the manipulator itself Implicit in this approach is the use and maintenance of a complex world model, which must be updated in real time as actions take place. Such a database must maintain the geometrical relationships of parts within assemblies, assemblies with other assemblies, assemblies with the world, and the manipulator with the world. This is a difficult problem. IBM's AUTOPASS [Lieberman and Wesley 77] attempted this level of operation but apparently the work has been abandoned. Systems as abstract as this require the use of Artificial Intelligence techniques to infer low-level actions from the high-level task specifications. Researchers such as Alami [Alami 84] are looking at LISP environments as being suitable for this kind of work.

1.3.2 Conclusions from the Survey

Some of the topics that crop up again and again in the literature of robot languages are. portability. modularity. multi-robot capability. extensibility. usability. efficiency. and the ability to deal with sensory input The following discussion looks at each of these issues in turn.

Portability is in fact not often addressed by robot language designers. (perhaps due in part to vendor's desire to restrict the freedom of the user once a robot has been purchased) A language that is easily moved to a new robot or to a new computer and operating system will have a longer useful life than one which is robot and machine-specific This philosophy underlies the design of the RCCL environment. which has already been successfully ported to several machines and robots Closely associated with the portability aspect is modularity: a software system that is made up of a hierarchy of well-designed modules can generally be ported to a new environment fairly easily. because the required changes may be encapsulated in just a few places

Multi-robot workcells are becoming the rule rather than the exception. Often in the past. a distinct controller was required per workcell robot. and if these were of different genesis. communication was either impossible or limited to simple yes/no signaling. Obviously. systems are required which allow several manipulators to be controlled in a unified manner The main approaches appear to be

1. Concurrent programming languages Example: AL's *cobegin* and *coend* constructs allow the programmer to specify that the execution of different sections of code be started concurrently. An *event* variable. which has the same functionality as an operating system semaphore primitive. may be used to signal between the concurrent code sections and thus synchronize events The actual scheduling mechanism is implicit. AL will simply run one process until it blocks. then go on to the next It is critical to note that AL's *move* command blocks execution of the caller. thus allowing other code to execute.

2. Multi-tasking programming environments. Application code is divided into concurrent programs. or tasks. which control different manipulators. Here. inter-task communication primitives are supplied by the operating system to allow explicit synchroniza-

tion and scheduling of events The GEM [Schwan, et al 85] system, for example, supplies primitives for message passing between programs running concurrently on different processors In [Volz, et al. 84] an Ada-based implementation is described: task synchronization is done implicitly by the Ada run-time environment when procedure calls are made In the implementation described in the present thesis, we use the Intel iRMX-86 operating system's *mailbox* and *semaphore* primitives to synchronize tasks

The clarity, simplicity, and unity of a programming language will have an impact on its practical usability An associated issue is whether it is better to create entirely new languages, based loosely on lots of old languages, or to embed new constructs in an existing language There are arguments both ways. RCCL, for example, is simply a set of function calls provided for standard "C" language programs The advantages are clear: there is a concise, predefined syntax that is known to a wide base of potential users, there is a large array of development tools available, and the language is portable insofar as there are cross-compilers available for a wide variety of target machines. A major disadvantage is that the syntax of a function call may be less readable than the equivalent statement in a specialized language, however in providing wide functionality a specialized language may become almost unusably complex The AL language with POINTY, with its associated on-line teaching facility, has over 300 different reserved words

The efficiency of a language can be measured twice, during the creation of programs, and during their subsequent execution A compiled language will execute faster than an interpreted language, and produce more compact run-time executable code A compiler is not limited to a single pass, and thus has the ability to support fully structured code with forward references, separate compilation of modules, and all the programming conveniences that this entails However, the development cycle required to produce working programs tends to be longer, due to the associated compile, link, test, and debug cycle An interpreted program is often easier to debug unless the competing compiled system has a real-time source level debugger

Extensibility implies that new features and capabilities can be added as necessary without a complete overhaul of the system Languages (such as RCCL) which are basically subroutine libraries on top of a standard programming language are extensible simply by

adding new functions to the libraries The drawback is as before, that the syntax may become inconsistent over disparate functions It is also true, however, that to add features systematically to a specialized robot language requires system-level modifications, whereas in the other case this may be done at the user level The unity of the specialized language is lost, of course, by adding features through the subroutine mechanism

One of the major costs associated with workcell robotics (a driving force behind much research) has been the elaborate fixturing required so that the positions of parts which the manipulator deals with may be defined precisely before operations begin. If sensory-based systems can allow the run-time system to determine the exact positions of things, the need for fixturing will obviously decrease This is important because new fixtures may be required for every new part handled by the robot, whereas a sensory-based system need be installed only once. Mobile robots, of course, require comprehensive sensory ability in order to navigate in a changing or unknown environment, and in all cases the question of safety for both human and robot requires sensing of some nature.

Sensor technology is improving in many areas Touch sensors have evolved from simple contact switches to large tactile arrays with good dynamic resolution [Dario, et al 83]. Vision systems, of course, have improved tremendously, and with the development of powerful 32-bit microprocessors and special-purpose silicon there has been a significant drop in their cost. Thus, vision has become increasingly cost-effective for robotics applications in the real world; the *bin of parts* problem is now solvable in many cases.

Force sensing is an *a priori requirement* for systems which attempt to deal with manipulator dynamics, the control system needs a way of computing the torques at the robot joints in order to compensate in real time for gravitational, inertial, Coriolis, and centrifugal forces. This is especially true as joint velocities increase, and simple kinematic models become unacceptably inaccurate. RPCE's which allow programmers to specify compliance are another arena where force sensing is required In this case, the forces at the end-effector are of interest, and much work has been done designing wrist-mounted force sensors.

Obviously, then, it is of overriding importance that an RPCE be flexible enough to accommodate existing and new sensory input systems, both at the control level and at the planning level.

1.4 Real-Time Operating Systems

An operating system generally provides the following services:

- Task scheduling
- Memory management
- Intertask communication
- I/O services

A definition of *real-time operating system* seems hard to pin down. Various criteria have been proposed, but all commentators seem to agree that such a system must allow programs to respond very quickly to external events How fast *very quickly* is, is of course application-dependent - for example, a system which must create a flicker-free raster video image can be said to be *real-time* if it can keep up, without fail, with a frame rate of about 30Hz. A system executing robot control software is *real-time* if it can execute the control algorithms, without fail, at some chosen sample rate (the rate at which the software reads the joint positions and adjusts the control signals to achieve the desired motion). Typically, this rate is 1 kHz at the joint control level, and 10 to 100Hz at the path control level. The faster the sample rate, the finer the spatial control of the manipulator and the higher the joint velocities that the system can safely handle.

A real-time computer operating system must also provide programmers with the primitives that they need to build real-time applications In [Cole and Sundman 85], six requirements in addition to *very fast* response are listed:

- support for creating, deleting, and scheduling multiple independent software processes (*tasks*)
- the ability to communicate (send/receive data) asynchronously between tasks
- provision for sharing data between tasks
- ability to synchronize the execution of multiple tasks
- ability to synchronize task execution with external events (interrupts)
- an efficient I/O system which does not degrade system performance

1.4.1 Task Scheduling

Many different scheduling philosophies have been used to achieve *real-time* response, some of these will be described below. First, however, it is instructive to discuss the concepts of *tasks* and *task states*, because these will be key words in the discussion of scheduling to follow.

First of all, let's define *task* A task (or process), from the point of view of the operating system, is a unit of executable software. A task may request resources from the operating system, for example memory, CPU time, I/O services, timing services etc

Typically, a programmer links together his own application code with system code to create a task, assigns it a priority relative to other tasks, loads it, and requests the operating system to execute it Tasks may also be invoked by other tasks or by the operating system itself.

Associated with each task is a task *state* Although the names differ from operating system to operating system, the state of a task can roughly be described as either: running, asleep, ready, or idle. At any given moment in time, there is only one *running* task on the system (given that there is a single CPU). A task which is in the *running* state is actually in control of the CPU

A task in the *asleep* state has relinquished control of the cpu, usually to wait for some event, for example a message from another task, or an external action, such as a key being pressed on a keyboard

A task which is *ready* will become the running task as soon the operating system recognizes that its priority is higher than the priority of the currently running task

Finally, an *idle* task is one which is potentially executable, perhaps even resident in memory, but which has been removed from the operating system's list of tasks to be executed The operating system scheduler can be thought of as the mechanism which manipulates the states of the tasks on the system.

A time-slice scheduler allocates the cpu by checking the status of the various tasks in the system on a strictly periodic basis, a typical period might be 100 milliseconds Thus, each *time slice* is allocated to a particular task, and this allocation is always reconsidered at the end of the slice If the running task executes a system call which involves waiting for

some event (eg for an I/O operation to finish), the operating system generally reallocates the CPU to another task immediately

An event-driven scheduler responds to external events on an asynchronous basis. That is, the operating system pre-empts the currently running task at the moment that a higher priority task becomes *ready*. For example, if the running task goes to sleep to wait for something, (say I/O or a message from another task), the scheduler immediately changes the state of the highest priority *ready* task to *active*

Some schedulers work by *pre-allocating* the CPU to various tasks. In this scheme, a task may reserve a given amount of cpu time, at pre-determined future times. At those times, no matter what, the task is guaranteed to run without interference.

1.4.2 The iRMX-86 Real-Time Multi-Tasking Operating System

Intel's iRMX-86 [Intel 84] was designed for use by OEM's* as an embedded operating system for real-time applications. The system is configurable, in that the user need include only those system calls which he requires. In this way the size and speed of the resulting system can be optimized The iRMX-86 scheduler is *event-driven*.

iRMX-86 has the layering illustrated in Figure 1.3. The *Nucleus* lies at the core of the system. It performs the task scheduling, inter-task communication, memory management, and interrupt management. The *Basic I/O System or BIOS* supports asynchronous I/O operations on named, physical, and stream files. The *Extended I/O System, or EIOS* supports synchronous I/O and logical device connections The *Universal Development Interface or UDI* is a standard interface to the various supported high level languages The *Human Interface or HI* provides a development environment with a set of system commands to manipulate files, configure new systems, run compilers, text editors, object librarian, linkage editors, etc. The *Application Loader* allows executable programs to be loaded from mass storage devices

RMX-86 is an *object-oriented* operating system That is, the system supports a class of entities which include *jobs, tasks, mailboxes, semaphores, regions, connections* and *segments* Objects are created, deleted, and manipulated via system calls, and referred to

---

* Original Equipment Manufacturers

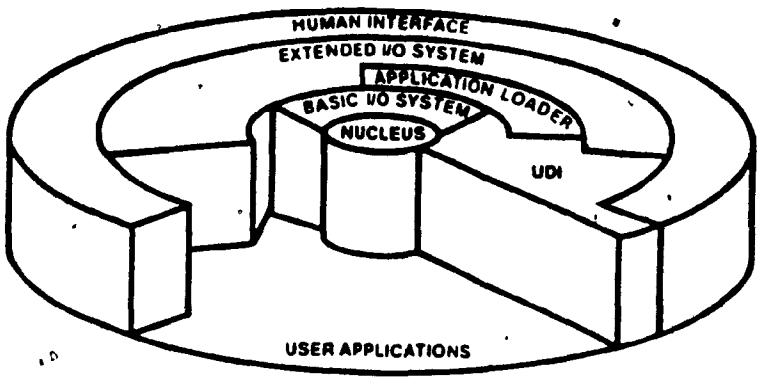

Figure 1.3 The RMX-86 Operating System

by their *tokens*, which are the system's way of tagging and identifying them. New kinds of objects may be created by the user and incorporated in the operating system.

All objects are created in the context of a *job*, which is an environment characterized by a set of limits. These limits allow restrictions to be placed on the number, type, and properties of objects that the system will create in response to system calls. For example, limits may be placed on memory allocation, maximum task priority, etc. Typically, a job contains a set of tasks which perform related functions and communicate with each other.

In a multi-user development environment, for example, each user works within a separate job, whose initial task is the Human Interface's Command Line Interpreter. When the user executes a system command (eg invokes an editor), a child job of that user's job is created by the Application Loader. If the editor crashes (for example because it runs out of memory) other user's jobs are not affected.

A *task* is an executable program which can be in one of a set of finite states, as shown in Figure 1.4. The task in the *ready* state which has the highest priority will become the running task. Task scheduling is *event-driven* as opposed to time-sliced. That is, once a task has control of the CPU, it retains control until some event (the arrival of an object in a mailbox, an interrupt occurring, a new task being created...) occurs which causes another task with higher priority to become ready. Once a task has been loaded and is memory-resident, it is *not* swapped out to a mass storage device.

A *mailbox* is a mechanism for passing objects between tasks. The tasks need not be in the same job. When the mailbox is created, it may be specified as having a *FIFO* or *Priority* mechanism. In the FIFO scheme, messages are simply retrieved in a first-in, first-out manner. Otherwise, messages are taken from the mailbox queue according to the relative priorities of the sending tasks. When waiting for a message at a mailbox, a task may elect to wait "forever" until a message arrives, or for a specified amount of time. The task is placed by the operating system in the *asleep* state while waiting.

A *semaphore* is similar to a mailbox, but rather than allowing arbitrary tokens to be passed, only *units* may be sent and received. Semaphores are normally used as a synchronization mechanism between tasks.

1.4.3 Unix

Unix is a now hugely popular operating system from Bell Labs (now ATT Bell Labs) written by Thompson, Kernighan and Ritchie using the C programming language. It was and still is an innovative approach to a multi-user program development environment, and has been ported to a great variety of computers, ranging from the Cray X-MP supercomputer to the Intel 80286 microcomputer (Xenix). There are many versions of Unix. CVaRL uses Unix 4.2Bsd from the University of California at Berkeley.

Unix consists of two parts. The *kernel*, consisting of approximately 10,000 lines of C

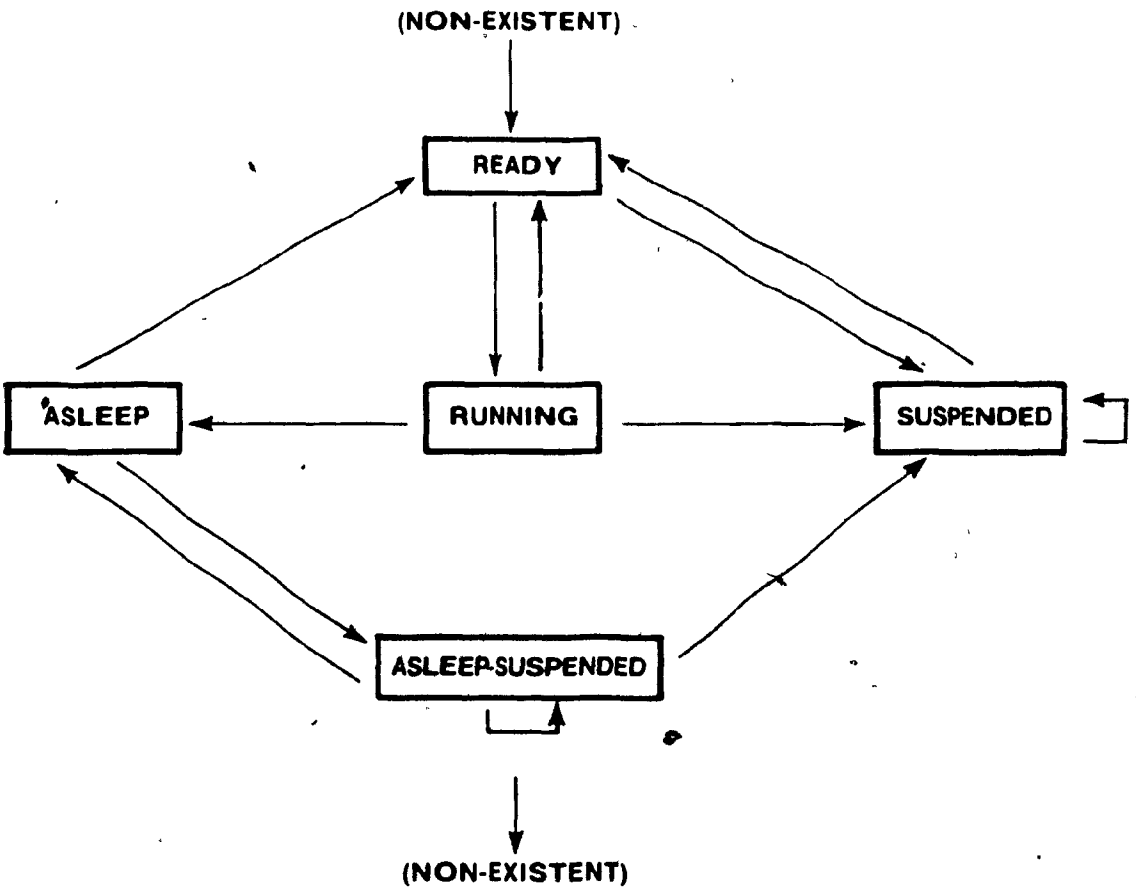

Figure 1.4 RMX-86 Task States

code and a few hundred lines of assembly code, and the utilities, which represent the other 95% of the system. The kernel is fixed, but the utilities can be manipulated by users as they wish. Unix is a very "open" environment: users, for example, are free to write their own *shell*, or command interpreter, and source code is available for the operating system and utilities

Unix supports the idea of I/O redirection at the shell level, and allows users to *pipe* the output of one program to the input of another This makes it very easy to build applications using *filters*, which each perform some operation on a stream of data and then pass it to the next filter using a pipe Often, commands drawn from the very rich set of utilities formed into pipes can immediately accomplish things that would otherwise require new programs to be written. This software reusability [Kernighan 84] is an important aspect of Unix's popularity.

The kernel controls process execution, I/O, swapping, and scheduling. Unix assumes a *virtual memory* environment, and the only limit placed on the size of user programs is the size of the virtual address space of the machine. The data and text associated with a process is *swapped* to and from secondary memory as required, depending on how long the process has been resident, and how long other processes have been swapped out which want to be swapped back in.

Unix has a time-slicing† scheduler which adjusts task (in Unix, *process*) priorities using a mechanism which was basically designed to maximize the system's response to multiple users typing sporadically at keyboards User process priorities are adjusted according to their recent ratio of compute time to real time consumed [Thompson 78]. The effect of this is that if a process uses its high priority to hog the computer, its priority drops. Similarly, low priority processes which have been ignored for a long time have their priorities increased.

Synchronization between processes is done using *signals*. There is no information associated with a signal, except that it has happened. A signal can be lost, processes can block signals, and any after the first which consequently arrive are **not** queued until the process unblocks that signal. Signals have no priority, most are pre-assigned, and there are only a finite number, which varies with the Unix implementation, available‡.

---

† The 4 2Bsd scheduler runs every 100 milliseconds

‡ 4.2Bsd allows 32 signals

Processes are created with the *fork* and *exec* mechanism A process which forks creates an exact copy of itself, but the child process cannot share data with its parent

Real-time Unix?

It may be appreciated from the previous discussion that standard Unix has a very limited appeal as a high-speed real-time operating system, due to the scheduling and signaling mechanisms used, and the inability of processes to share data

Attempts have been made to modify the Unix kernel to provide more support for real-time applications Two examples are Masscomp's Real-Time Unix, or RTU, and Charles River Data System's UNOS In the latter, the entire kernel was replaced, although users still operate using the Unix paradigm. The basic change made was to support the notion of multiple processes inside the kernel; in standard Unix, there is only one. Masscomp's RTU introduced something called the *Asynchronous System Trap* or AST This is a software interrupt that remedies the deficiencies of signals

A major problem that has to be overcome is to defeat the swapping mechanism. This was accomplished in Masscomp's version of Unix [Cole and Sundman 85] by special system calls with which programmers can lock particular pages of memory into core In order to implement RCCL, the McGill CVaRL version of Berkeley Unix was similarly modified so that the memory associated with the real-time control process would not be swapped to disk. Memory locking is fairly dangerous, because deadlock problems can easily occur, programmers must carefully precalculate the amount that they need, otherwise the system performance can be destroyed.

Another problem involves task synchronization. Signals are a poor mechanism because they are not prioritized and can be lost. AT&T's release of Unix (System V) introduced a semaphore primitive and also a primitive which allows processes to share memory, the latter is normally impossible in standard Unix.

Problems still remain in dealing with interrupts, providing asynchronous interprocess communication, and especially providing deterministic task scheduling. There is also the basic problem that the kernel is not easily configurable, as opposed to systems like RMX-86 where system calls can be included or excluded from the configuration on an as-needed basis.

It must be concluded that for high-performance embedded real-time systems, Unix is not a good choice, but as a multi-user program development environment, it has no peer.

1.5 Thesis Overview

Chapter 2 is a description and analysis of RTC, RCCL, and the Microbo manipulator. The chapter begins with RTC, and describes the control paradigm and the user interface. Next the RCCL trajectory level is examined, and the chapter concludes with a description of the Microbo Ecureuil manipulator and its RCU control unit. In Chapter 3 the forward and inverse kinematics for the Microbo robot are derived, and an analysis of the computational complexity of the resulting solutions is presented.

Chapter 4 describes the design and implementation of the hardware and software which make up the RTC system for the Microbo robot. RTC, which originally ran under the Unix 4.2Bsd operating system on a VAX minicomputer, was re-implemented using a set of Intel microprocessor cards and the iRMX-86 real-time multi-tasking operating system. There is also a list of the practical differences between the System 310 implementation of RTC/RCCL and CVaRL's VAX version.

Chapter 5 summarizes the results of the research and looks ahead at future enhancements of the RCCL control environment.

Appendix A is a brief user's guide to RTC/RCCL on the Intel System 310, concentrating on the practical aspects of booting the system, calibrating the robot, compiling and linking RTC/RCCL programs, etc.

Appendix B describes in detail the communication protocol of the Microbo joint controllers.

Chapter 2 RTC, RCCL, and the Microbo Robot

2.1 Introduction

RCCL, or the Robot Control C Library, is a library of functions and supporting data structures which allow applications programmers to create manipulator control programs using the C programming language [Kernighan and Ritchie 78]. Motion is specified in Cartesian coordinates, and homogeneous transforms are used to specify positions and spatial relationships between objects in the robot world

As shown in Figure 2.1, RCCL runs on top of the Real Time Control layer, called RTC This is a substrate which programmers can use to write joint-level control functions (for example the RCCL trajectory generator) which execute at some *sample rate* (typically, 10 to 100hz) in the background, while the user's planning level program executes in the foreground Interfaces are provided via which the individual joints of the manipulator may be queried and controlled Control of associated hardware (for example gripper open/close) is also provided.

RTC may be thought of as a kind of "robot operating system", it provides a standardized interface to the robots. This is analogous to a computer operating system's provision of a standard interface to I/O devices, with the added element of control over real-time aspects of the interface

A full description of the user interface to RCCL and RTC appears in [Lloyd 85] and [Hayward and Lloyd 85]; this chapter presents just a synopsis We concentrate here on the

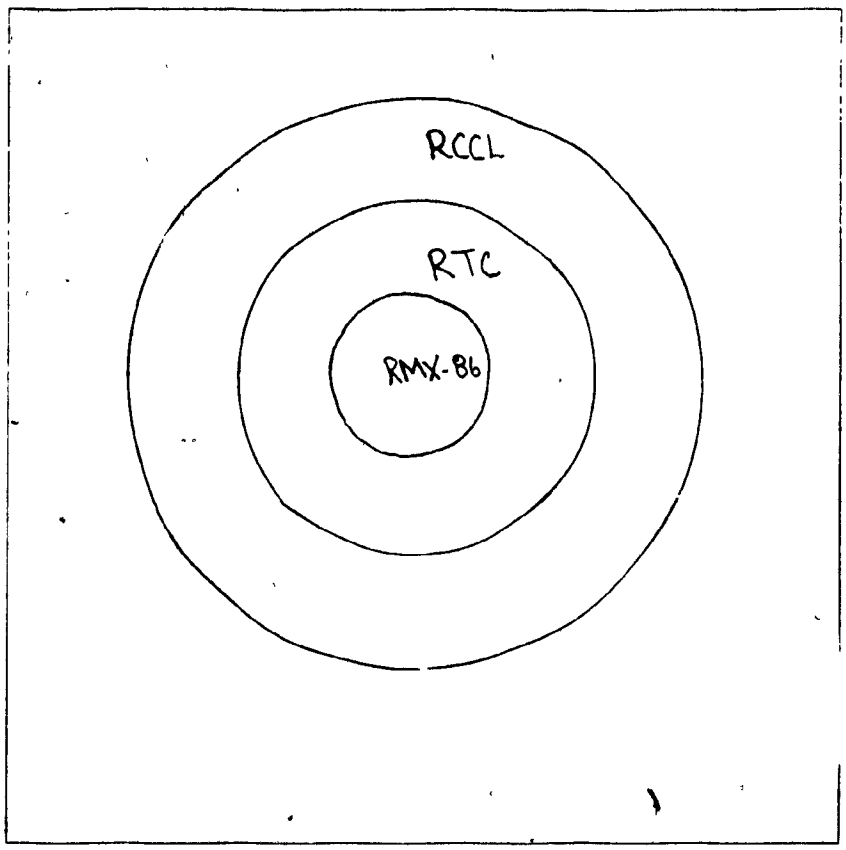

Figure 2.1 RCCL and RTC

robot interface to RTC and RCCL, as this is central to our work. The chapter begins with a brief outline of the genesis of RCCL and RTC and the previous implementations.

2.2 The Genesis of RTC/RCCL

The RCCL/RTC system was originally written by Hayward at Purdue University [Hayward and Paul 83], under the guidance of Paul. It supported a Puma-560 manipulator and ran on a VAX minicomputer running Unix 4,2Bsd. It may be noted that RCCL is an implementation of some of the ideas presented in Chapter 10 of [Paul 81]

The software was brought to CVaRL in 1983, and has been installed and enhanced at CVaRL [Lloyd 85] In this case a VAX-750 minicomputer again running Unix 4.2Bsd supports the smaller Puma-260 manipulator.

Both the original Purdue and McGill implementations use a custom-designed FIFO

(First In First Out) interface between a dedicated VAX parallel port and the Puma's Unimation controller This FIFO is under the control of a a Unix 4.2Bsd device driver, and allows the high speed transfer of small blocks of data to and from the Puma's joint controllers via the Unimation LSI-11 processor When RCCL is not running, the LSI-11 executes the native VAL-1 interpreter. The Unix device driver has a special mode wherein it executes the user-specified RTC control functions at high (system) priority Modifications must be made to the Unix kernel to prevent the operating system scheduler from swapping RTC data areas to disk. This would of course would be fatal to the real-time process's operation

RCCL has also been ported to a 68000-based Unix system. This work was done at Hewlett-Packard in California by Hayward in 1984 Unfortunately, there is no public reference for this implementation, but we do have the following details from a private communication [Hayward 86].

The target machine was an HP-900/200 running HP-UX, the Hewlett-Packard version of Unix. The CPU was a Motorola 68000 microprocessor with 16k of fast cache memory, and the robot a Puma-560. Apparently the code was ported very easily, but the sample rate could not be set faster than 9 Hz due to the inefficiency of the interface to the math co-processor. The trigonometric functions were done using table lookup. The system interface was significantly different from the VAX implementation, where the real-time control functions run inside the Unix kernel as part of the device driver for the Unimation controller In this case the HP-UX scheduler was modified so as to run the control functions as Unix "signal" routines Communication with the Unimation controller's LSI-11 processor and thus the joint microprocessors was via an HP-IB parallel interface, as opposed to the FIFO interface used in the VAX implementations at Purdue and CVaRL bb

2.3 RTC- Robot Real Time Control

RTC gives C language programmers the ability to develop robot joint level control procedures. It does this by providing

a) a set of function calls via which control algorithms (for example, the RCCL trajectory controller) may be executed at high priority in the background, at the sample rate.

b.) a set of data structures which reflect the state of the manipulator's joints and which may be used to control them

2.3.1 The RTC function calls

The RTC user interface consists of a set of procedures and data structures which are available to C programmers as a compile-time library. In what follows, the procedure names are printed in `typewriter font`

- `rtc_open()` initializes the RTC system and creates the background control task which runs at the sample rate
- `rtc_control()` is the major component of the real-time robot control The control task is started which collects information from the robot, and two user-specified control functions are activated Commands may be sent to the robot via global data structures (described below)
- `rtc_release()` stops a control session and re-initializes the system for another one A parameter may be set to turn off arm power.
- `rtc_close()` terminates the control session and deletes the background control task. A parameter may be set to turn off arm power.
- `print_rtc_error()` interprets the error codes which may be returned by the four above procedures, and prints an appropriate message The control procedures are arbitrary; a programmer working at the RTC level may specify the entry points of any two C language functions, with the restriction that they execute within the chosen sample period In the RCCL case there is essentially a single procedure- the trajectory controller's "setpoint" procedure

2.3.2 The RTC data structures

The important global variables available to the RTC programmer are as follows

- the how structure reflects the state of the manipulator
- the chg structure is the command request vehicle
- the `terminate` variable, if set by the user, will abort the control session If set by the system, it will contain information as to why the termination occurred
- the `rtc_message` variable allows the user to pass a string to the user level from the control level. This may be used to provide error or status information, etc.

- the user_hangup variable allows the user to set up a "hangup" handler of his choosing This function is invoked by the RTC system instead of the default handler when a "control-C" is typed at the user's terminal during a control session.

2.3.3 The RTC Control Paradigm

Figure 2 2 illustrates the RTC control cycle. This must remain basically the same for all RTC implementations.

The system is reset and enters the idle state via the `rtc_open()` function call The control functions may then be activated by a call to `rtc_control()`. Once control is active, the following cycle is repeated once per sample period data is collected from the robot and the how global data structure is updated The first user-specified control function is invoked The chg data structure is examined and appropriate commands are sent to the joints The second user function is then executed, and the cycle repeats. A call to `rtc_release()` will cause the system to re-enter the idle state, and `rtc_close()` shuts down the RTC system.

2.4 RCCL- The Robot Control C Library

RCCL is a manipulator control language, implemented as a C‡ function library. Because RCCL is not a language in itself, application programs can take advantage of all of the features provided by the host language and operating system, for example file I/O and user interaction The macro facilities of C in particular help to overcome problems of syntax and presentation

RCCL consists of two parts, a real-time trajectory function, called `setpoint()`, and the main user program, which we refer to as the *planning level* program The planning level communicates with the real-time function via a *motion request queue* in shared memory, and communicates with it indirectly via RCCL library functions.

The real-time part of RCCL uses the RTC system described in the previous section- `setpoint()` is just an RTC control function. Although RTC is closely bound to the host

‡ As long as calling conventions are respected there is in fact no reason why the RCCL functions may not be called from other languages

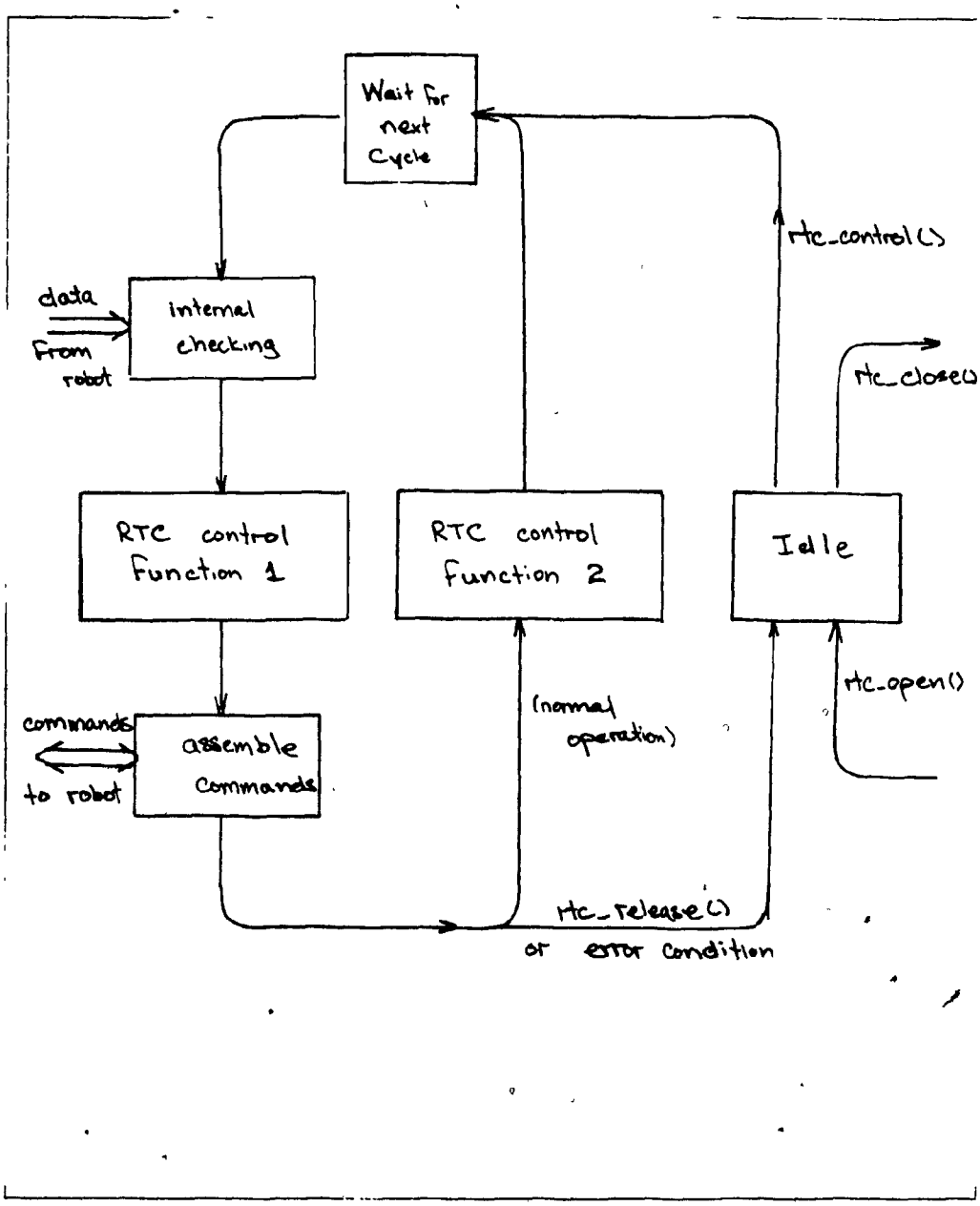

Figure 2.2 The RTC Control Paradigm

operating system and the robot hardware, RCCL is not; with the exception of the forward and inverse kinematic solutions, it is manipulator independent.

RCCL programs specify the motion of the manipulator using *motion equations* The elements of such an equation are homogeneous transforms representing the relationships

between coordinates frames attached to objects in the robot's world.

2.4.1 Motion Equations

A dedicated transform, T_6, represents the position of the end-effector with respect to some convenient reference frame. For the Microbo robot, for example, T_6 is taken with respect to the center of the base of the manipulator If **POS** represents a desired position and orientation of the end-effector, we can write

$$T_6 = \mathbf{POS}$$

If we use a transform **TOOL** to represent a tool attached to the end-effector and **GRASP** to represent the grasping position for an object **OBJ** that is on a conveyer **CONV**, we have

$$T_6\ \mathbf{TOOL} = \mathbf{CONV\ OBJ\ GRASP}$$

RCCL can now solve

$$T_6 = \mathbf{CONV\ OBJ\ GRASP\ TOOL}^{-1}$$

Transforms such as **OBJ** are known as *constant* transforms; they do not change during program execution RCCL also allows *varb* or *functionally defined* transforms which are re-evaluated every sample period; for instance, the **CONV** transform might thus represent a moving conveyer belt. By allowing an arbitrary function to be associated with a transform, RCCL can cause the manipulator to track moving objects or to react to arbitrary sensory information at run-time. A third type of transform, the *hold* transform, if modified by the user level program at run time, will be re-evaluated when the corresponding motion begins

Position equations are created using the `makeposition()` function The function's parameters are pointers to the transforms which make up the left and right-hand sides of the equation Run-time overhead is minimized by pre-multiplying any adjacent constant transforms, and the function returns a pointer to the result, which is a dynamic data structure

Transforms may be created and modified using an extensive family of function calls The basic ones are `gentr_rot()`, `gentr_trsl()` and `newtrans()`. The first two dynamically create *constant* transforms involving, respectively, a rotation and translation. The third function allows the creation of functionally defined or hold transforms.

The pointer returned by `makeposition()` may consequently be used as the parameter in `move()` function calls, to cause manipulator motion satisfying the position equation This is described below

2.4.2 The Trajectory Generator

Under RCCL, manipulator motion is specified as a series of path segments linear in either the joint coordinate space (*joint mode*) or Cartesian space (*Cartesian mode*).

Each `move()` call queues a *motion request packet* for the real-time `setpoint()` trajectory generator to service, then immediately returns control to the user program (see Figure 2.3). This has the advantage that the servicing of the motion request proceeds in parallel with the main program. Explicit synchronization mechanisms are supplied to coordinate the two levels, i e to let the planning level know when a the trajectory level has completed the path segment associated with a particular motion request

In Cartesian mode, the joint angles are controlled such that the coordinate frame attached to the tool moves along straight lines in Cartesian space. The `makeposition()` function in this case takes an additional argument to specify which transformation in the equation is to be taken as the *tool* transform. In this mode, the motion equation is re-solved each sample period during the motion segment, and the joint angles computed using the inverse kinematic formulations. This Cartesian mode imposes a heavy computational load on the system, but the path of the tool tip is now simple and predictable It should be also be noted that as the manipulator passes through any singularities, joint rates may become 'infinite'

In joint mode, the position equation is solved once at the beginning of the motion for the final set of joint angles. Intermediate values are then linearly interpolated during the motion using the initial and final values This has the advantages that a) joint velocities are limited only by the individual joint maximums, and b) manipulator degeneracies do not cause a problem The obvious disadvantage, of course, is that the path of the tool tip is not predictable in Cartesian space.

When the motion request queue holds several motion request packets, the RCCL trajectory generator performs a *transition* between the motion segments to prevent velocity

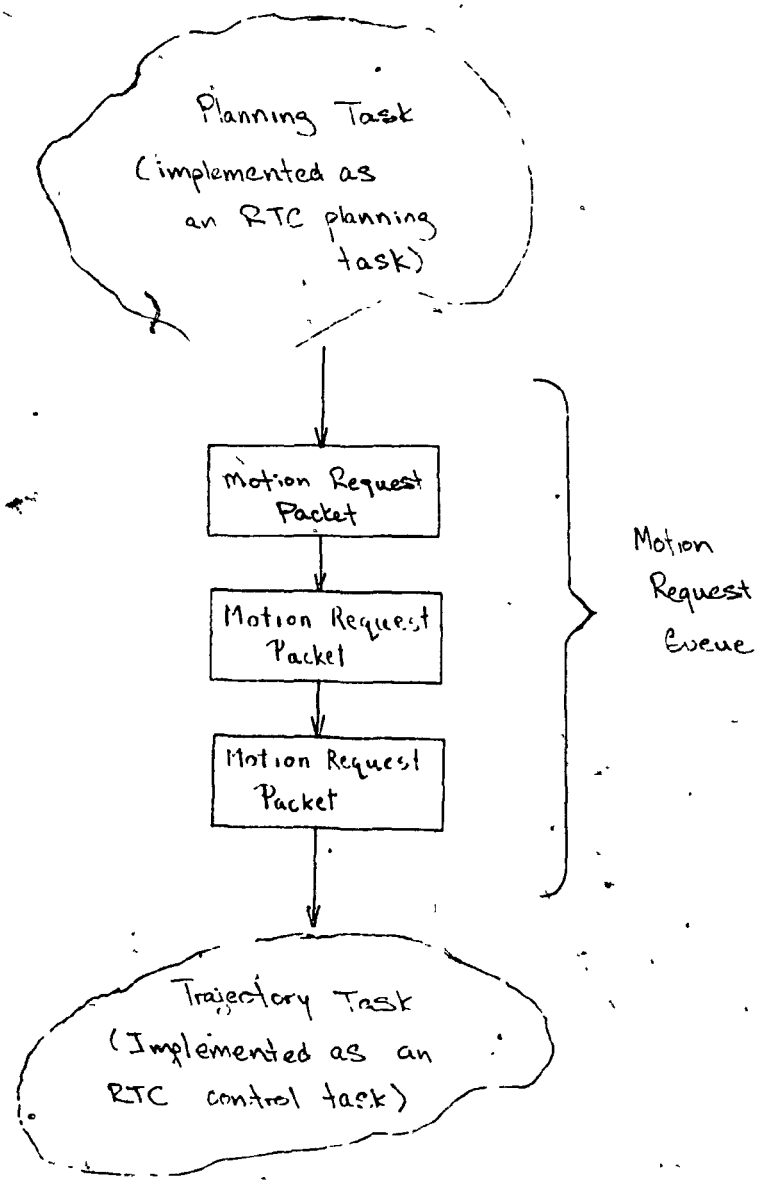

Figure 2.3 The RCCL Task Architecture

and acceleration discontinuities. The boundary conditions require that a quartic polynomial be fitted between the adjacent linear segments [Paul 81].

Each path segment is thus characterized by

- a *trajectory mode*, either Cartesian or joint (the `setmod()` function).
- a *position* to move to (the `makeposition()` function).
- a *velocity* which may be specified directly in terms of rotational and translational speed, or as a segment duration (the `setvel()` and `settime()` functions).
- a *transition time*, which is the duration of the transition to the next segment (the `settime()` function).

2.4.3 RCCL Programming

We present here a sample RCCL program by way of illustrating the preceding discussion. This program for the Microbo robot has been tested and is stored in /rccl/ex/washer.c on the System 310. For a more complete description of the RCCL user interface the reader is referred to the RCCL User's Manual [Hayward and Lloyd 85]

```
/*
        washer.c        the window washer program.
                        robot moves, with gradually increasing
                        translational and rotational velocity,
                        through 4 points arranged in a rectangle
                        in the yz plane.  The tool tip is maintained
                        orthogonal to the plane.
*/

#include "rccl.h"       /* the basic rccl include file */
#include "rtc.h"        /* the basic rtc include file */

main()
{
        TRSF_PTR b, c, d, e;
        POS_PTR  p1, p2, p3, p4;
        int      tvel = 100;
        int      rvel = 100;
        int      i;

                /*-- translations from robot origin to 4 corners --*/
                /*-- rotation so tool pointing towards table     --*/
```

```
b = gentr_rot("B", 325.0, 150.0, 300.0, xunit, 180.);
c = gentr_rot("C", 325.0,-150.0, 300.0, xunit, 180.);
d = gentr_rot("D", 325.0,-150.0, 250.0, xunit, 180.);
e = gentr_rot("E", 325.0, 150.0, 250.0, xunit, 180.);

        /*-- corresponding position equations --*/
p1 = makeposition("P1", t6, EQ, b, TL, t6);
p2 = makeposition("P2", t6, EQ, c, TL, t6);
p3 = makeposition("P3", t6, EQ, d, TL, t6);
p4 = makeposition("P4", t6, EQ, e, TL, t6);

if (rccl_open (0, 0))   /*-- start the RCCL control session --*/
 { print_rtc_error(rtc_error, 0);
   exit(-1);
 }
if (rccl_control ())  /*-- start the trajectory generator --*/
 { print_rtc_error(rtc_error, 0);
   exit(-1);
 }

setvel(tvel, rvel);        /*-- set trans, rot velocities --*/

move(park);                /*-- move to the park position --*/
waitfor (park->end);       /*-- and stop there            --*/
move(p1);                  /*-- move to p1               --*/
waitfor(p1->end);          /*-- and stop there            --*/

setmod('c');               /*-- cartesian mode  --*/
for (i=0; i<3; i++)
 { printf("velocity %d, %d\n", tvel, rvel);
   move(p2);               /*-- move to p2      --*/
   move(p3);               /*-- move to p3      --*/
   move(p4);               /*-- move to p4      --*/
   move(p1);               /*-- move to p1      --*/
   tvel += 5;
   rvel += 10;             /*-- faster...       --*/
   setvel(tvel, rvel);
 }

waitfor (completed);        /*-- wait for queue to empty   --*/

setmod('j');                /*-- back to joint mode        --*/
move(park);                 /*-- move to the park position --*/
waitfor (park->end);        /*-- wait till done            --*/
```

```
    if (rccl_close (0))        /*-- terminate control session --*/
     { print_rtc_error(rtc_error, 0);
       exit(-1);
     }
}
```

This program starts by creating four constant transforms, { b, c, d, e } and writing the associated position equations for { p1, p2, p3, p4 }. These positions correspond to the corners of a "window" in the y-z (vertical) plane The tool is rotated about the x-axis so as to point towards the table. In this way it will be physically possible for the manipulator to move so as to maintain orientation

The (rccl_open() and rccl_control() calls initialize and then start up the RCCL control session. It may be noted that these two functions eventually just call rtc_open() and rtc_control() We next set the translational and rotational velocities, specified in mm/sec and degrees/sec, and move to the predefined park position The waitfor() primitive allows us to synchronize with the end of this motion. We similarly move the robot to the p1 position. At this point we can change to Cartesian mode using the setmod() call, and begin the for() loop. This loop will in fact execute as quickly as the motion requests are queued The waitfor (completed) call will delay the program until the motion request queue has emptied. We finally restore joint mode (it is not possible to move in Cartesian mode from p4 to park) and close the control session

2.5 The Microbo Robot

The Microbo Ecureuil manipulator (Figure 2 4) is a Swiss-made 6 degree-of-freedom cylindrical robot designed for assembly tasks. It is installed in the McGill CIM-RL robot workcell alongside a Puma-260 manipulator, such that the workspaces of the two robots overlap.

The vendor-supplied "RCU" controller is shown in Figure 2 5 Each of the joints is driven by a D.C servo motor connected to a current amplifier, which is in turn controlled by a digital-to-analog converter Joint position feedback is via incremental optical encoders

The RCU controller has the hierarchical design typical of contemporary industrial controllers, with a processor for each joint, supervised by a coordinating master which may

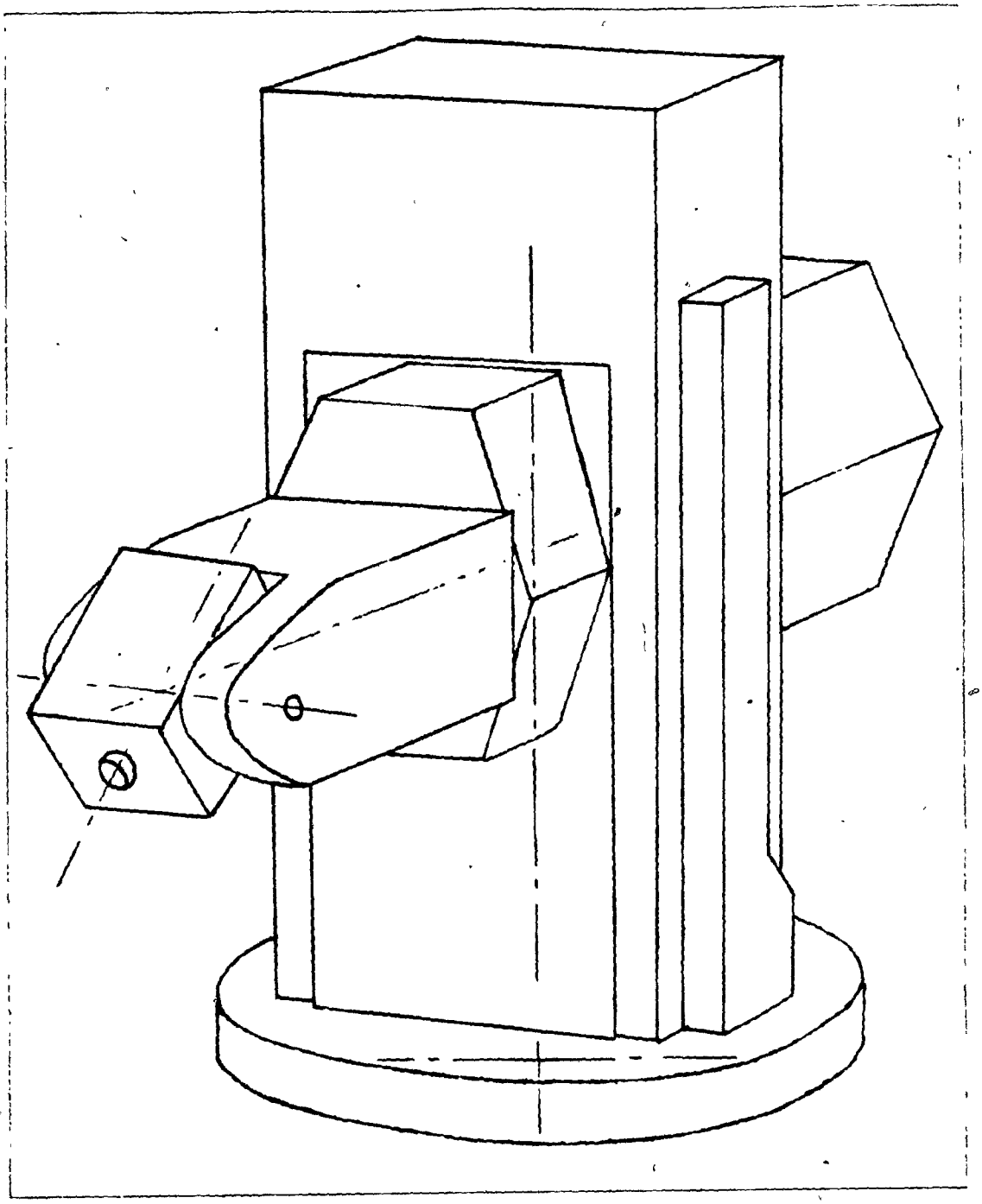

Figure 2.4 The Microbo Ecureuil Robot

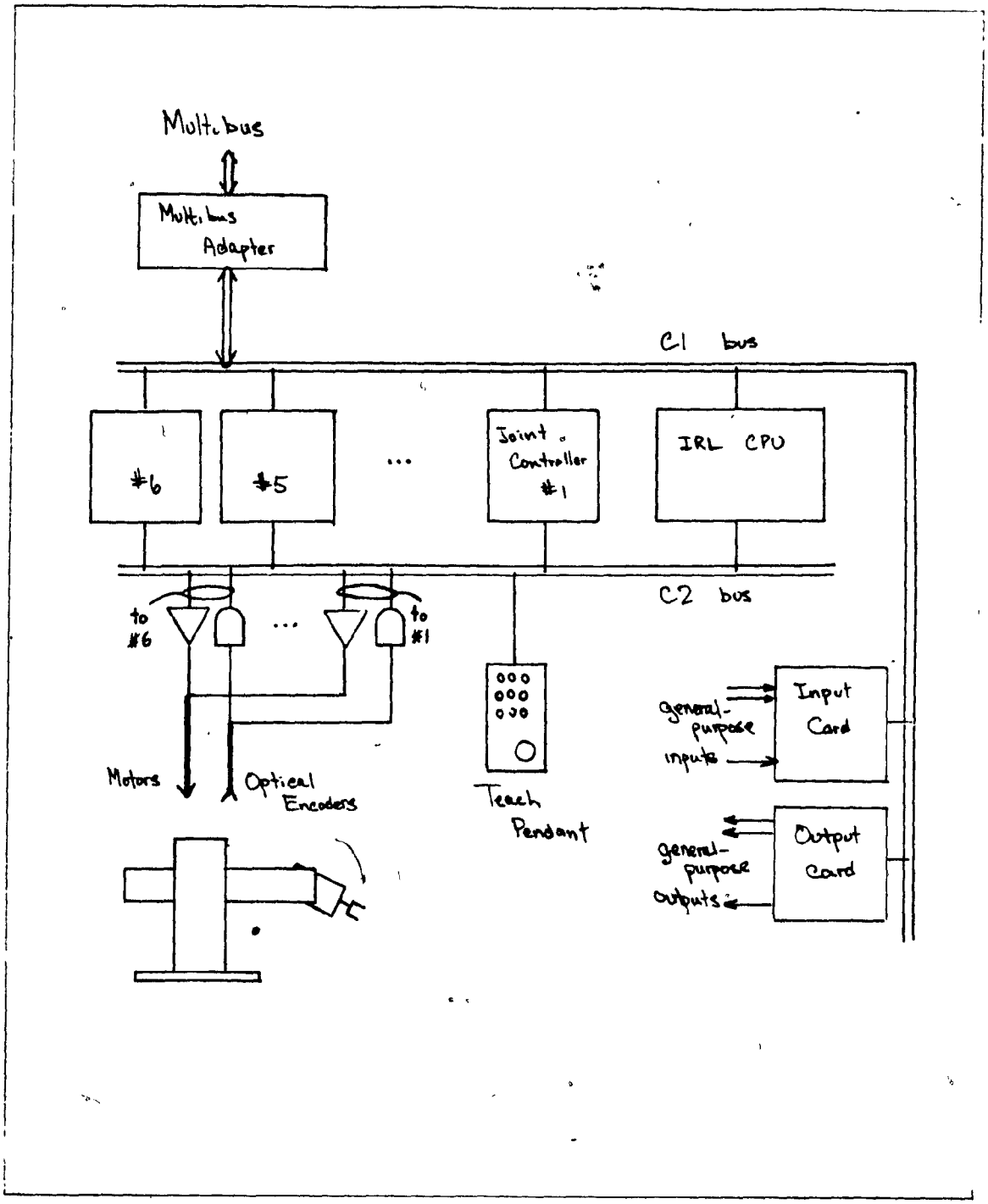

Figure 2.5 The Microbo RCU Controller

be programmed by the user using the IRL interpreted language which is resident in ROM (Read-Only Memory). The master processor also controls a teach pendant, an audio-cassette tape machine for program storage, a video terminal and a printer. The system supports up to 8 joints, in addition to the 6 robot joints, there is a rotary stage and a linear stage These are not currently being used, and are not supported by the RTC system

CVaRL's interest in the Microbo was spurred by this manipulator's stated capability for very fine motion, especially with the two prismatic joints This may be seen from the following table which shows physical motion per joint encoder count For comparison, we include the corresponding figures for CVaRL's Puma-260 manipulator

Encoder Resolution		
Joint	Microbo	Puma-260
1	0.0031°	0.0770°
2	0 0020 mm	0.0051°
3	0.0013 mm	0.0084°
4	0.0041°	0.0103°
5	0.0056°	0.0114°
6	0.0182°	0.0142°

It may be seen that the Microbo is theoretically capable of much finer motion increments than the Puma; it will be shown, however, that the vendor's control algorithms negate this advantage under actual working conditions

2.5.1 Controller Hardware

The master processor is an 8085-based single-card computer. On-board ROM is used for the IRL language interpreter, and static RAM (Random Access Memory) for non-volatile storage of programs and data This card is connected to two busses: a 16-bit/8-bit address/data bus (the C1 bus) and an input/output bus (the C2 bus). The latter is used for interfacing with the teach pendant and a VAX host, while the former allows this processor to control the 8 joint processors via their memory-mapped control registers This interface is discussed below.

The joint processors are identical and independent Each is an Intel 8085-based single-card computer with the joint control software stored in ROM. There are digital inputs for the incremental joint encoder, and an analog output which is routed to the appropriate power amplifier, which has a maximum current capacity depending on the joint's requirement The joint processors and master processor share a common address and data bus which is used for communication Each joint processor has a switch-selectable address on the bus which defines the location of a set of 8-bit *control*, *status*, and *data* registers. These are used by the master processor to query and control the joints The communication protocol is always initiated by the master processor It involves checking the status register before each read or write to the command or data register to determine that the joint processor is ready to receive or send data. For example, the sequence required to fetch the 16 bits of encoder data is as follows:

1 Read the status register until an input buffer empty condition is indicated.

2 Write the "read position" command code into the command register.

3 Read the status register until an output buffer full condition is indicated.

4. Read the data register, which contains the first byte of the two byte encoder position.

5 Wait until the status register again indicates that the output buffer is full

6. Read the second byte of the encoder position from the data register

A similar protocol is used to send data, for example a target position or a velocity, to a joint Each 16-bit data exchange of this type can thus be seen to require six register reads or writes, consisting of a read/write sequence to send the command followed by two read/read or read/writes to fetch or send the data A more complete description of the joint communication including definitions of the register bit patterns and a command dictionary, is presented in Appendix B.

2.5.2 Controller Software

The IRL Interpreter

IRL, a BASIC-like interpreted language, is the vendor-supplied software which executes on the Intel 8085-based master processor described above. It is fairly primitive, providing basically a one-to-one correspondence with joint-level commands. Programs are entered

via a CRT terminal and may be stored on cassette tape and printed on a line printer A teach pendant allows the user to move each of the robot joints separately and store resulting positions as sets of encoder counts The language was specifically designed for the Microbo Ecureuil manipulator's RCU controller and is bound closely to the hardware. It it written in 8085 assembly language

Motion commands do not block an IRL program. the MTARGET command simply sends single, *asynchronous*, joint-level "set target" commands to the appropriate joint processors The programmer may optionally delay until the requested target has been reached by using the MWAIT command, which causes the IRL processor to poll the joint(s) with "get joint status" queries until the appropriate bits are found set. It may be seen that a basic weakness of the IRL system is the lack synchronization between joints, which means that the path of the end effector is unpredictable for motions involving more than a single joint Also, because joint targets may only be specified in terms of encoder counts or via positions memorized using the teach pendant, it is impossible to compute positions off-line or from a program and then predict the resulting trajectory IRL programs work exclusively in the robot's joint coordinate system and not in any Cartesian coordinate frame which can be related to the workspace

The Joint-level Path Control Algorithm

Under normal operation, the Microbo joint controllers accept *target*, *velocity*, and *acceleration* commands. The joint control algorithm uses the latter two values to control the trajectory when moving the joint to the target position. When the target is reached (within a few encoder counts), the control law changes and the joint servos to maintain this position.

The control algorithm executing on the joint processors thus combines a primitive trajectory control with a joint servo function We note that except for the command protocol, the operation of the joint controller is not well-specified by the vendor. the explanation given here is based to a large extent on observation of the joints' characteristics Figure 2 6 is an educated guess at the block diagram

. When approaching the target position from afar, the joint controller is said to be, using the vendor's terminology, in the "dynamic regulation zone" The joint is accelerated (using its acceleration setting) until it reaches its velocity setting. It continues at this

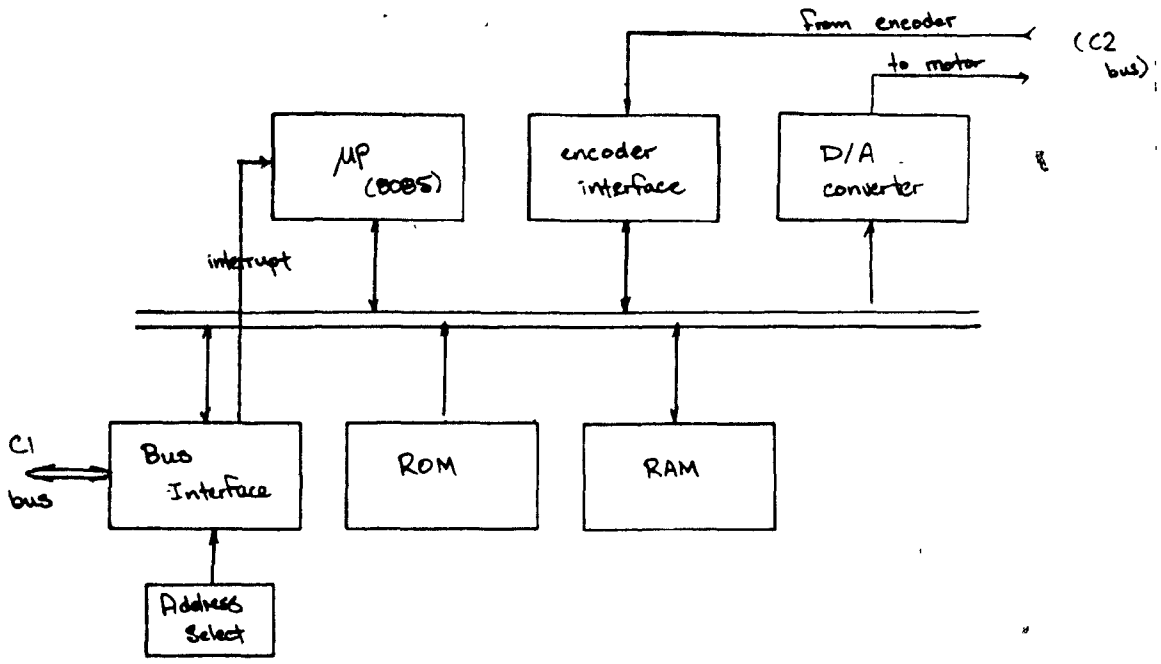

Figure 2.6 Microbo Joint Controller Block Diagram

velocity until it approaches the target position, then decelerates. It is then said to be in the "static regulation zone" At this point, the acceleration term is ignored, and the trajectory calculation is based solely on the velocity term See Figure 2.7

We introduce the following notation ·

X_n is the target position for the nth sample instant

X_{n-1} is the previous target position

a is the (signed) acceleration

v is the (signed) velocity

$\triangle t$ is the joint's sample period (1 msec)

G is the proportional gain

X_p is the current measured position

ϵ is the positional error, i e. $\|X_{n-1} - X_p\|$

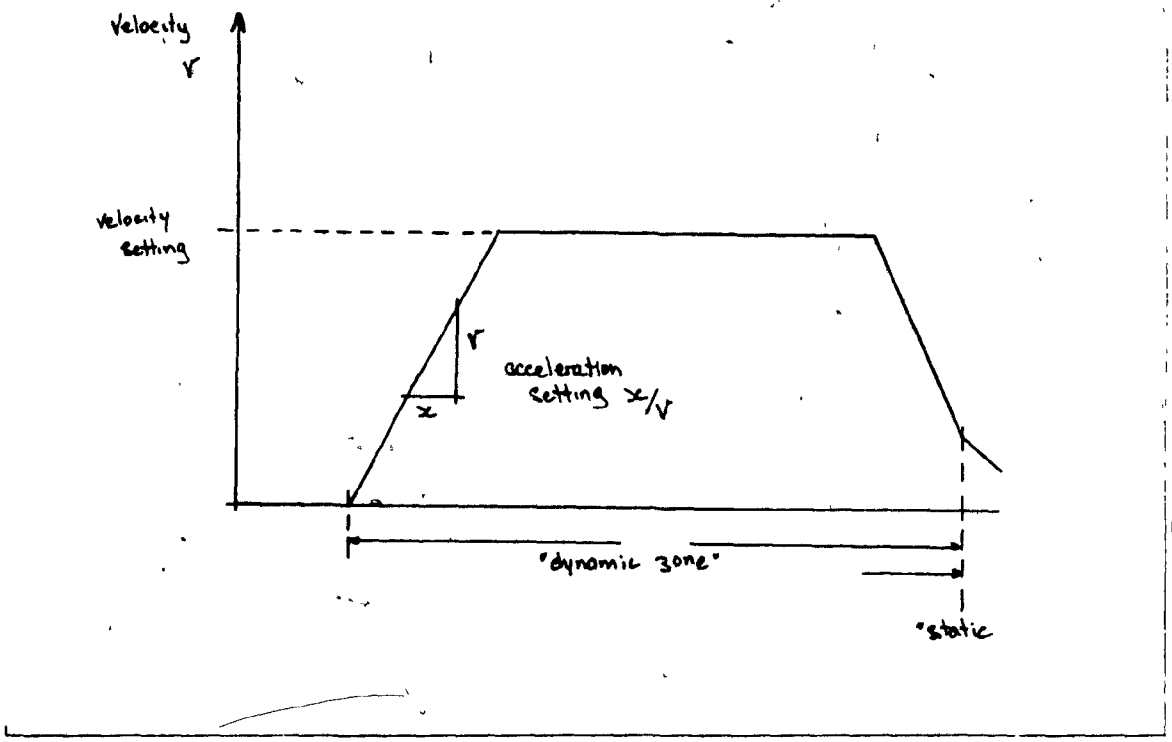

Figure 2.7 Microbo Joint-level Path Control

$\int (dyn)$ is the dynamic integral term

$\int (stat)$ is the static integral term

The vendor specifies the following equation for the "dynamic regulation zone"

$$X_n = X_{n-1} + a\ \Delta t^2 + G\ \varepsilon + \int (dyn)$$

We interpret the third term to be proportional path control, servoing based on the previous target position. The second term is the change in position due to acceleration over the sample period. This is the trajectory control term. The fourth term is possibly error compensation due to the proportional path control.

In the "static regulation zone", the following "control law" is used

$$X_n = X_{n-1} + v\ \Delta t + \int (stat)$$

The trajectory control term is now based on velocity rather than acceleration, and the proportional gain term disappears.

Unfortunately, this scheme as implemented by the vendor appears to have serious design flaws. It seems that the servoing, and thus the stiffness of the joint, is affected by the acceleration and velocity settings When the last velocity and acceleration sent to a joint are small, the joint loses its stiffness, and fails to maintain its position under even the most negligible load conditions when stopped Also, because there is no proportional control term in the "static zone", the integral term dominates and the joints may exhibit severely underdamped response when subjected to a small sustained force.

Work Areas

The precision of some of the Microbo joints (1 and 3) is such that the encoder word size (16 bits or 65535 counts) is too small to cover the full physical range of the joint. To overcome this, the vendor elected to implement "work areas" Under this scheme, if the encoder value is greater than 50000 or less than 15535, the joint will accept a "change work area" command. The result is that the current encoder setting is adjusted 50000 counts downwards or upwards by the joint controller It is up to the IRL programmer to keep track of which zones the joints are in; and of course it is impossible to move from the interior of one zone to the interior of another without stopping in the "transition zone", and sending the appropriate command to cause the joint processor to change work areas

It seems that adding another byte to the encoder word size would have been a far better solution from the IRL programmer's point of view Work areas, as will be seen in chapter 4, complicate the RTC implementation somewhat, and limit the velocities of the affected joints when they are under (synchronous) control of the RCCL trajectory generator

Chapter 3 Microbo Kinematics for RCCL

3.1 Introduction

Mapping the coordinate space defined by the joints of a manipulator into a Cartesian space is known as the *forward kinematic* problem. Conversely, the *inverse kinematic* problem involves determining the joint variables corresponding to a position and orientation defined in Cartesian space of the manipulator's end effector

This chapter begins with a review of the necessary mathematical background We first establish the mathematical nomenclature that will be used in the rest of the thesis, and then discuss the use of homogeneous 4 X 4 transforms in representing the transformation between coordinate frames associated with the links of a serial link manipulator We then present and apply an algorithm to establish a coordinate system at each link, and from this obtain the forward kinematic solution. This is followed by a derivation of the inverse kinematics. Note that joints 2 and 3 of this manipulator are prismatic while the remaining joints are rotational. A summary of the computational requirements of the kinematics is then presented, as this information was required during the design of the RTC system to establish the CPU power required for a real-time implementation.

3.2 Mathematical Background

4 X 1 vectors will be identified using bold face lower case, as in $\mathbf{a}$. 4 X 4 matrices will be shown in bold face upper case, for example $\mathbf{A}$. First and second derivatives, usually

corresponding to velocity and acceleration, will use dot notations, as in q and $\dot{q}$. Dotting also may be applied to vectors. Components making up a vector will be enclosed in curly brackets, for example $\mathbf{q} = \{q_1, q_2, q_3, q_4, q_5, q_6\}$. The transpose of a vector or matrix is indicated with a superscript as in $\mathbf{a}^T$ or $\mathbf{A}^T$. The use of a preceding superscript indicates that a vector or transform is defined with respect to some coordinate frame, for example, ${}^B\mathbf{A}$ ${}^B\mathbf{a}$ are a matrix and vector defined with respect to coordinate frame B.

The following notation may be used to more compactly represent sines and cosines: $S_i = \sin\theta_i$, $C_i = \cos\theta_i$, $S_{ij} = \sin\theta_i + \sin\theta_j$, and $C_{ij} = \cos\theta_i + \cos\theta_j$.

3.2.1 Homogeneous Transforms

The representation of an n-component vector by an $(n+1)$-component vector is known as the *homogeneous* coordinate representation. For example, the vector $\mathbf{p} = \{p_x, p_y, p_z\}$ becomes $\hat{\mathbf{p}} = \{wp_x, wp_y, wp_z, w\}$. The mapping from $\hat{\mathbf{p}}$ back to $\mathbf{p}$ is then $\mathbf{p} = \{wp_x/w, wp_y/w, wp_z/w\}$. It may be seen that there is no unique representation for the vector $\hat{\mathbf{p}}$, but that if w is unity, then the homogeneous coordinates are identical to the physical coordinates.

Say we attach an orthonormal coordinate frame F to a rigid body in space and wish to locate a point in F with respect to a reference frame G. If $\mathbf{p}$ is a vector representing the translation of the origin of G to the origin of F, and $\mathbf{R}$ is a 3 X 3 rotation matrix representing the rotation of F with respect to G

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{pmatrix} \tag{3.1}$$

then it may be shown (see for example [Lee 82]) that by using homogeneous coordinates a point $\mathbf{f}$ in frame F has coordinates in G obtained using

$${}^G\mathbf{f} = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} {}^F\mathbf{f} \tag{3.2}$$

The above result is known as a homogeneous transform, and is especially useful in representing the transformation between frames associated with each link of a robot manipulator. The four columns of the matrix are usually referred to as the **n**, **o**, **a** and **p**

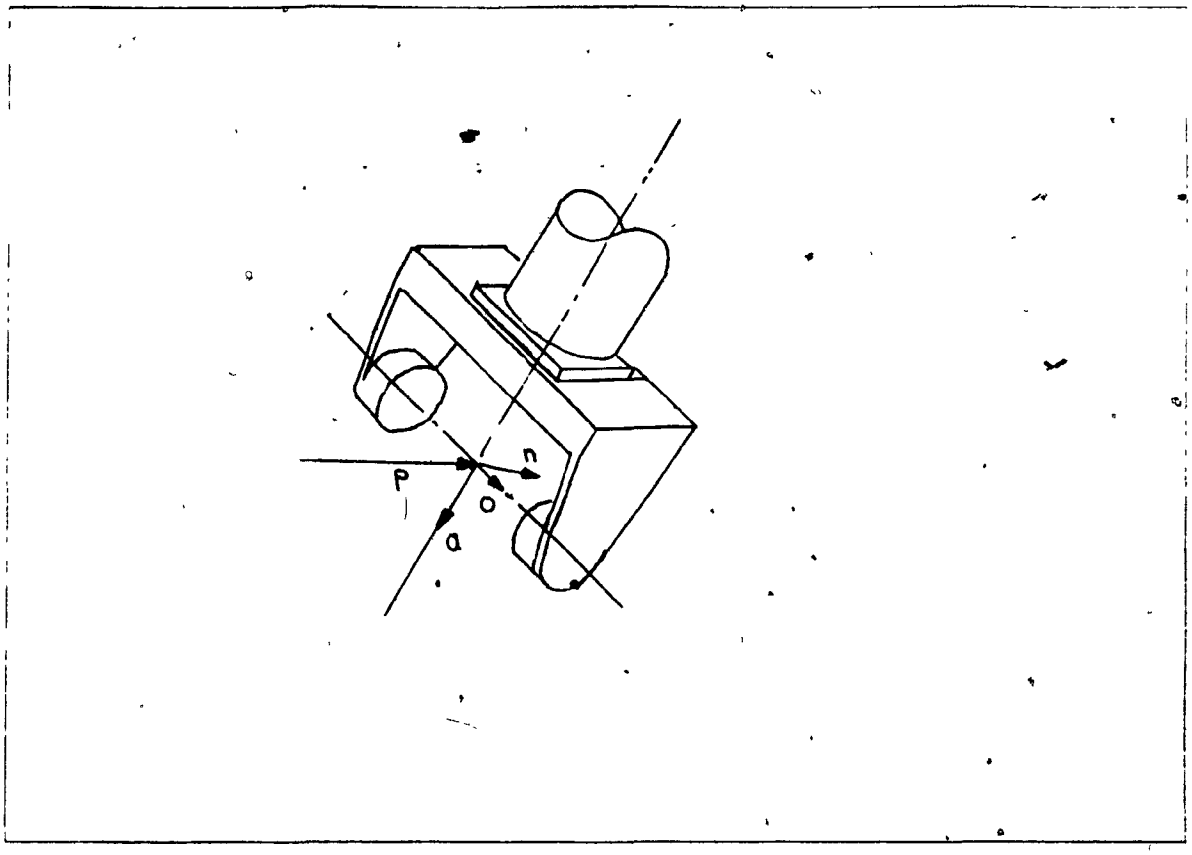

Figure 3.1 The **n o a** and **p** Vectors

vectors, and can be thought of in terms of representing a frame attached to a manipulator end effector. This is shown in Figure 3.1 The **p** vector locates the origin of the frame (a translation). The other three vectors are unit vectors, and the first, **a**, may be thought of as representing the direction from which the hand would *approach* an object. The **o** then specifies *orientation* of the hand, from finger to finger The *normal* or **n** vector completes a right-handed coordinate system; n is the cross-product of **o** and **a**

$$\mathbf{n} = \mathbf{o} \times \mathbf{a}$$

The result of multiplying two homogeneous transforms is another homogeneous transform; transform equations can be simplified by pre or post-multiplying both sides by the same transform. For example, if we have

$$\mathbf{X\,A = B\,C\,D}$$

we can post multiply both sides by

$$\mathbf{X} = \mathbf{B}\,\mathbf{C}\,\mathbf{D}\,\mathbf{A}^{-1}$$

because the product of a transform and its inverse is the identity transform

3.3 Defining the Coordinate System

Denavit and Hartenberg [Denavit and Hartenberg 55] established a convention in which an orthonormal right-handed coordinate frame is associated with each link of a serial link manipulator. The transformation between frames in consecutive links is done using homogeneous transforms known as **A** matrices. Each **A** matrix depends on four geometric quantities, a_i, d_i, θ_i, and α_i, associated with the link. Either d_i or θ_i varies and is known as the *joint variable*. This representation is sufficient for any serial link manipulator consisting of prismatic and revolute joints, and leads to a straightforward solution for the forward kinematics of a manipulator.

In [Lee 82], an algorithm is given for systematically establishing the **A** matrices. The method is as follows, where unit vectors along the x_i, y_i and z_i axes are shown as $\mathbf{x}_i$, $\mathbf{y}_i$ and $\mathbf{z}_i$. Please refer to Figure 3.2.

1. Establish a right-handed orthonormal coordinate frame (x_0, y_0, z_0) at the base of the manipulator, with z_0 lying along the axis of joint 1.
2. For each $i, i = 1, \ldots N$ establish the joint coordinate frame
   - 2.1 Align $\mathbf{z}_i$ with the axis of motion of joint $i+1$
   - 2.2 Locate the origin of the i-th coordinate system at the intersection of the z_i and z_{i-1} axes, or at the intersection of the common normals between the z_i and z_{i-1} axes and the z_i axis
   - 2.3 Assign x_i according to the cross product of the z_i and z_{i-1} axes or along their common normal if they are parallel $\mathbf{x}_i$ is the corresponding unit vector
   - 2.4 Assign $\mathbf{y}_i$ as the vector cross product of $\mathbf{z}_i$ and $\mathbf{x}_i$.
3. For each $i, i = 1, \ldots N$ find the joint and link parameters
   - 3.1 d_i 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} axis. d_i is the *joint variable* for prismatic joints.

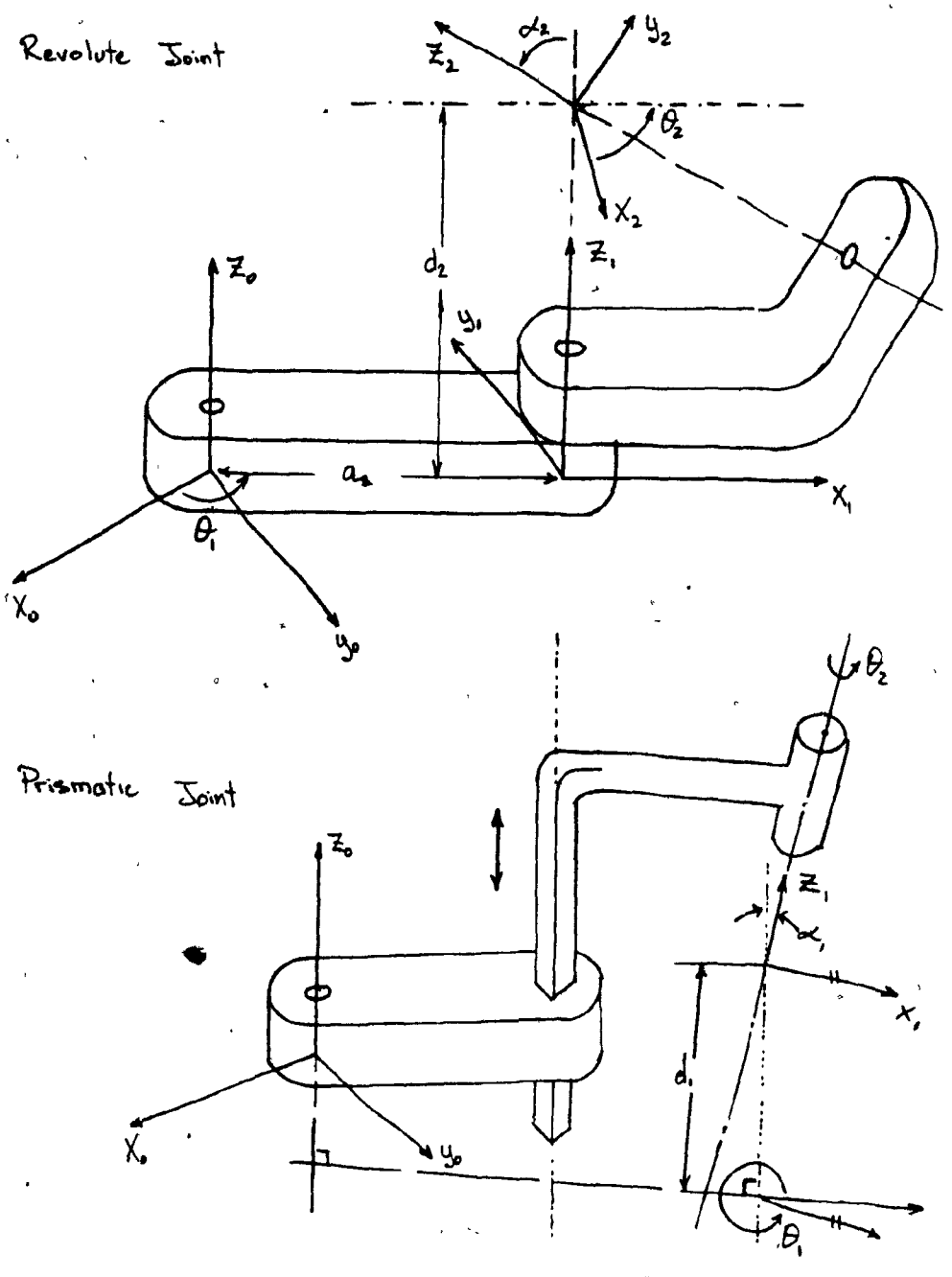

Figure 3.2 Denavit-Hartenberg Link Parameters

3.2 a_i is 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.

3.3 θ_i is the angle of rotation from x_{i-1} to x_i, measured counterclockwise around z_{i-1}. θ_i is the joint variable for rotary joints

3.4 α_i is the angle of rotation from the z_{i-1} axis to z_i, measured counterclockwise around x_i

Using this algorithm, we obtain the coordinate system for the Microbo illustrated in Figure 3.3. Note that the direction of the z axis in steps 1 and 2.1 may be chosen arbitrarily, for example, z_0 has been chosen pointing upwards though the opposite direction would also have been a valid choice

3.4 Determining the A Matrices

As shown in [Lee 82], by multiplying together homogeneous transforms representing, respectively, a rotation θ_i about the z_{i-1} axis, a translation of d_i along the z_{i-1} axis, a translation of a_i along the x_i axis, and a rotation of α_i about the x_i axis, the following general forms may be derived for the A matrices

$$\mathbf{A}_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{(revolute joint)} \quad (3.3)$$

$$\mathbf{A}_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & 0 \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{(prismatic joint)} \quad (3.4)$$

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

By observation of Figure 3.3, the link parameters for the Microbo are

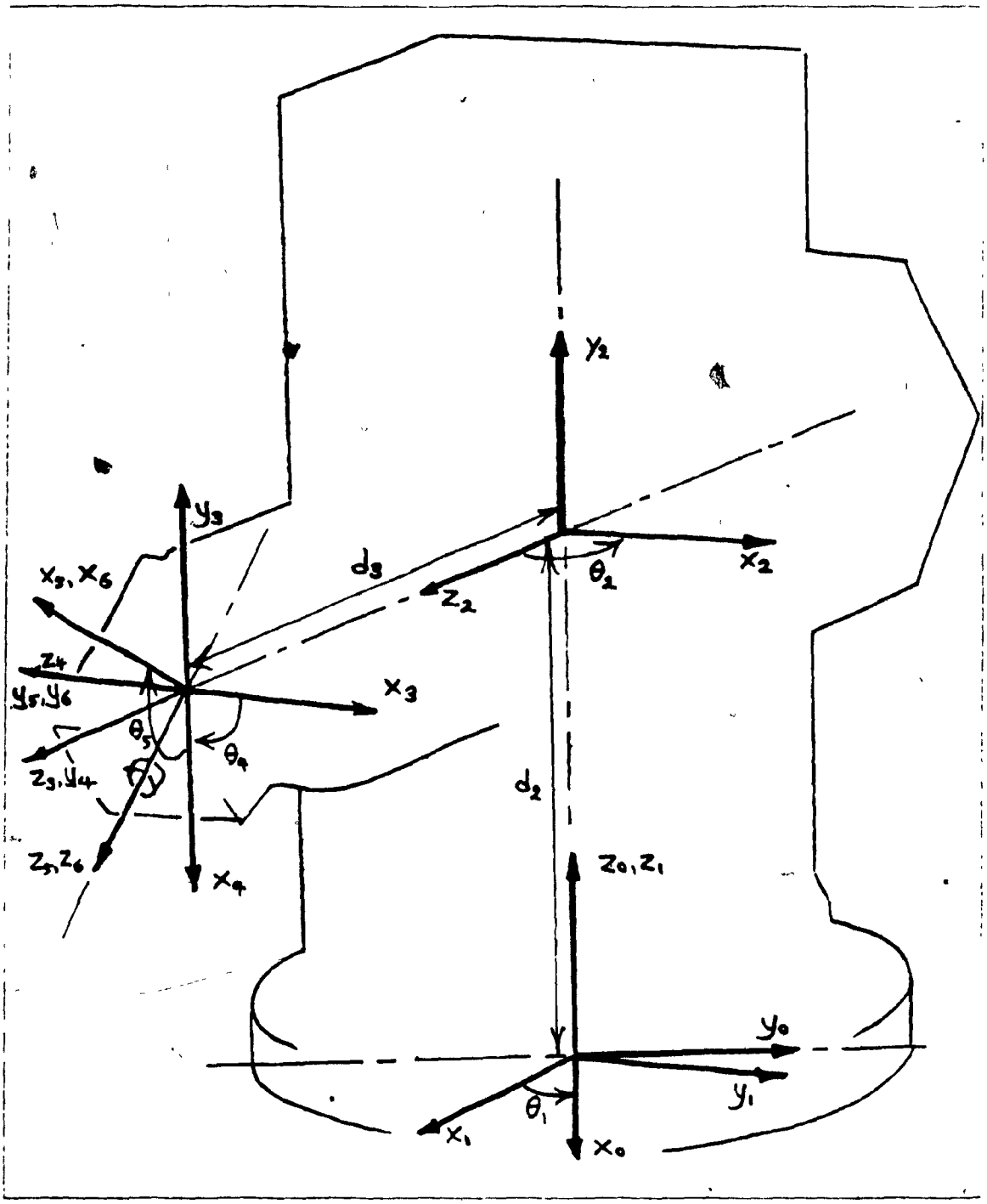

Figure 3.3 Coordinate System for the Microbo

Microbo Link Parameter Values				
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

and the joint variables are $\{\theta_1, d_2, d_3, \theta_4, \theta_5, \theta_6\}$

Substituting the values from the table into the formulae above. we get the following A matrices for the Microbo Ecureuil

$$\mathbf{A}_1 = \begin{pmatrix} C_1 & -S_1 & 0 & 0 \\ S_1 & C_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.5}$$

$$\mathbf{A}_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & d_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.6}$$

$$\mathbf{A}_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

$$\mathbf{A}_4 = \begin{pmatrix} C_4 & 0 & S_4 & 0 \\ S_4 & 0 & -C_4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

$$\mathbf{A}_5 = \begin{pmatrix} C_5 & 0 & S_5 & 0 \\ S_5 & 0 & -C_5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.9}$$

$$\mathbf{A}_6 = \begin{pmatrix} C_6 & -S_6 & 0 & 0 \\ S_6 & C_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.10}$$

3.5 Forward Kinematics

The transform which relates the base of the robot (the $\{x_0, y_0, z_0\}$ coordinate system) to a frame attached to the last link of the manipulator is known as the $\mathbf{T}_6$ transform It is obtained by multiplying together the A matrices

$$\mathbf{T}_6 = \mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\mathbf{A}_5\mathbf{A}_6 \tag{3.11}$$

Using the notation introduced earlier in this chapter we have

$$\mathbf{T}_6 = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.12}$$

Equating the terms of the result of (3 11) with (3.12) we get

$$\begin{aligned}
n_x &= C_1C_6S_5 - S_1(S_4S_6 + C_4C_5C_6) \\
n_y &= C_1(S_4S_6 + C_4C_5C_6) + C_6S_1S_5 \\
n_z &= C_5C_6S_4 - C_4S_6 \\
o_x &= -S_1(C_6S_4 - C_4C_5S_6) - C_1S_5S_6 \\
o_y &= C_1(C_6S_4 - C_4C_5S_6) - S_1S_5S_6 \\
o_z &= -C_5S_4S_6 - C_4C_6 \\
a_x &= -C_4S_1S_5 - C_1C_5 \\
a_y &= C_1C_4S_5 - C_5S_1 \\
a_z &= S_4S_5 \\
p_x &= C_1\mathrm{d}_3 \\
p_y &= \mathrm{d}_3S_1 \\
p_z &= \mathrm{d}_2
\end{aligned}$$

3.5.1 Computational Complexity of the Forward Solution

Because n, o, a defines an orthogonal system, there is some redundancy of information because each vector is the cross product of the other two. Such a cross product requires

6 multiplications and 3 additions or subtractions, so it is most efficient in this case to determine **o**, **a** and **p** and then do **n** = **o** X **a**. The required calculations are then 30 multiplications, 13 additions or subtractions, and 4 sets of sine and cosine operations

3.6 Inverse Kinematics

Given the terms of (3.12), which corresponds to a position and orientation of the manipulator end effector, the inverse kinematic problem is to find the joint variables.

It can be seen that (3.12) yields a set of 12 simultaneous equations, but that not all are in a simple enough form to be useful Using the method described in [Paul 81], we can obtain up to 60 additional simultaneous equations by successively post-multiplying (3.11) by $\mathbf{A}_1^{-1}$ through $\mathbf{A}_5^{-1}$ This results in a series of 5 matrix equalities of the form

$$\mathbf{A}_{i-1}^{-1} \cdots \mathbf{A}_1^{-1} \mathbf{T}_6 = \mathbf{U}_i \tag{3.13}$$

where

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

Elements on the right of (3.13) are functions of the variables of joints 1 through 6, while elements on the left are in terms of **n**,**o**,**a**,**p** and the joint variables 1 through $i-1$ If we examine equations with increasing values of i, we may solve for each joint variable in terms of previously solved variables.

The **atan2** form of the inverse tangent function is particularly useful here. It solves for θ_i given an equation of the form

$$\sin(\theta)/\cos(\theta) = f1/f2$$

which has the solution

$$\theta_i = \text{atan2}(f1, f2)$$

Thus we look for pairs of equations where the right hand sides may be divided to yield a similar form.

Examining the terms of the column vector **p** in (3.12), we have

$$p_x = C_1 d_3, \quad p_y = S_1 d_3, \quad \text{and } p_z = d_2 \tag{3.14}$$

which immediately gives us

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

By inspection, we have

$$d_2 = p_z$$

also, squaring and adding both sides of the first two equations of (3.14) gives us

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

Because of the geometry of the robot, d_3 is constrained to the positive square root value

If we consider (3.13) with $i = 4$ we have

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

which, if we equate the third column on either side, yields

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

We can solve for θ_4 by applying the atan2() stratagem to the second and third rows Note that there is a 180° phase shift when S_5 is negative, and a *singularity* (no solution) when S_5 is zero. Physically, this singularity corresponds to joints 4 and 6 being in alignment, and one way to deal with this when coding a computer algorithm is simply to use the previous value of θ_4. Note that for the Microbo robot, using the coordinate system that has been defined, the range of joint 5 is approximately $-\pi/2 > \theta_5 > -3\pi/2$. This means that the only singularity physically possible is at $\theta_5 = -\pi$, and gives us the following solution for θ_4

$$\begin{aligned} \theta_4 &= \text{atan2}(a_z,\ a_y C_1 - a_x S_1) && \theta_5 > -\pi \\ &= \text{atan2}(a_z,\ a_y C_1 - a_x S_1) + \pi && \theta_5 < -\pi \end{aligned}$$

To solve for joint 5, we use (3.13) with $i = 5$

$$\mathbf{A}_4^{-1}\mathbf{A}_3^{-1}\mathbf{A}_2^{-1}\mathbf{A}_1^{-1}\mathbf{T}_6 = \mathbf{U}_5 \qquad (3.15)$$

which gives us

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

We can divide the second row by the first to yield an atan() solution for for θ_5

$$\theta_5 = \text{atan2}(a_z S_4 + C_4(a_y C_1 - a_x S_1), -(a_y S_1 + a_x C_1))$$

Finally, for joint 6, we use (3.13) with $i = 6$

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

where equating the second columns gives us

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

We then divide the first row by the second and use atan2() which gives

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

3.6.1 Computational Complexity of the Inverse Solution

The inverse kinematics requires 4 sets of sine and cosine, 4 atan2's, 22 multiplies, 13 additions or subtractions, and 1 square root

We summarize the computational complexity of the Microbo forward and inverse kinematics in the following table. figures for the Puma-260 [Lloyd 85] are included for comparison. It can be seen that the Microbo kinematics are far simpler than the Puma's.

Kinematic Solution Complexity					
	mult	add/sub	sqrt	atan2	sin+cos
Microbo forward	30	13			4
Puma-260 forward	59	29			6
Microbo inverse	22	13	1	4	4
Puma-260 inverse	64	42	2	7	6

Chapter 4 RTC System Design and Implementation

4.1 Introduction

The user interface and operation of RTC was described in Chapter 2. In the present chapter we discuss the design of a new RTC system for the Microbo robot. We begin by enumerating the constraints which were placed upon the implementation. We proceed with a general discussion of the basic design alternatives that were open to us A feasibility study includes performance projections based on the kinematic analysis of the previous chapter, and the design and testing of an interface to the Microbo's joint controllers The chapter concludes with a description of the resulting RTC implementation

4.2 Design Constraints

The design of RTC for the Microbo robot was constrained by the following requirements.

1 Retain compatibility with the existing CVaRL VAX/Puma version of RTC, from the application programmer's point of view Beyond the obvious benefits of standardization, this would allow us to use the robot-independent RCCL library code without significant change

2 Achieve a useful sample rate (the initial goal was 28 milliseconds[1]) when running the RCCL trajectory generator as the control function This is important for smooth and precise motion of the manipulator.

[1] The default sample period for the VAX/Puma implementation is currently set to 56 milliseconds

3 Maintain and enhance the robot-independent aspects of the system as far as possible This simplifies extensions for other new robots

4 Avoid modifications to the existing Microbo RCU unit so that users can continue to use the IRL system and the robot while RTC/RCCL development is underway, as well as afterwards

5. Use the existing vendor-supplied joint controller software (in EPROM[‡]) This is the other aspect of ensuring IRL compatibility after RTC installation The constraint is also a function of an almost complete lack of documentation for the joint controller hardware and software, and a lack of the appropriate development tools to install new joint-level code

6. Use an available Intel "System 310" as a basis for the implementation This Multibus-based system is comprised of an Intel 80286/287-based single board computer, a 20 Mbyte Winchester disk plus 5¼ inch floppy disk, 896k of RAM, and free slots for additional Multibus boards. Extra CPU power is available in the form of an 8086/8087-based single board computer with 256k of on-board RAM, which may be booted from an 8 inch floppy disk unit controlled by an Intel iSBC-208 floppy disk controller The system runs the Intel iRMX-86 operating system, and provides a reasonable software development environment. A native C compiler is available.

4.3 General Considerations

The major computational load on an RTC system running RCCL is imposed by the real-time control functions. It may thus be seen that the first constraint listed above is ideally satisfied if we execute the RTC control functions on a separate microprocessor system, but retain the VAX/Unix development and execution environment

The VAX and this microprocessor would then need to communicate at the rate that motion or synchronization requests are generated by the user. Since an RCCL motion request is of the order of a hundred bytes, the communication between the VAX and the micro need not be very fast. This scheme would appear to be extensible to multiple robots, with one RTC processor per robot, and the single VAX running RCCL user programs.

‡ Electrically Programmable Read Only Memory

Synchronization between robots would be possible either by adding a *robot* parameter to the appropriate RCCL calls to identify a particular robot, or by having separate RCCL libraries for the various robots

It should be remembered that the motion equations created at the RCCL user level are solved in real time by *setpoint()*, an RTC control function, when a *move()* statement is executed This implies that the RTC and RCCL levels both need access to the linked list structure which represents the motion equation Unfortunately, the VAX and Intel conventions for storing floating point numbers differ†, so in addition to the problem of creating a data area concurrently accessible from both machines, we have the problem of changing data formats at a low level.

There are several additional difficulties with this scheme, the first of which involves the case of "functional" RCCL motion transforms, i e, those that are not constant but involve some function of time or of sensor data In the first case, where the transforms are simply functions of time, it should be possible to execute them on the microprocessor system, given real-time clock functionality, the second case implies more complex communication between the VAX and the microprocessor system to map sensor information between machines (if sensors are interfaced to the VAX).

A practical problem which arose at the time this work began was the lack of appropriate VAX/Unix-based cross-development tools to create and download executable code to the microprocessor system This meant that the RTC system and any control functions would have to be edited, compiled, and linked on the microprocessor system, obviously, for the RCCL trajectory generator this need only be done once, but functional transforms and would be ruled out, and the VAX user would lose the ability to use RTC by itself Also, source code for the libraries which make up the Intel iRMX operating system was not available, meaning that a major portion of the RTC support code development would have to be done on the System 310 anyway

The conclusion was that it is desirable, but impractical, to decompose the RCCL user level on the VAX and move the execution of the control function to a separate processor

The alternative solution is to port the entire RCCL/RTC environment to the Intel microprocessor system Since RCCL and its RTC control functions are written in C, and a

† Intel uses the I E E E standard DEC doesn't

compiler was available for the System 310, this seemed eminently possible. The problem could now be decomposed into the following steps.

1. design and implement an interface between the Microbo joint controllers and the System 310.
2. design a microprocessor system based around the System 310 which could support the computational and communication load imposed by RCCL and the Microbo robot.
3. design and implement a new version of RTC using the Intel iRMX-86 operating system.
4. port the RCCL library to the System 310
5. test and evaluate the resulting RTC/RCCL system.

4.4 Feasibility Study

This section describes preliminary work done to verify that the Microbo system was a suitable target for RCCL/RTC, and that the Intel microprocessor system outlined above was capable of satisfying the imposed restraints.

4.4.1 Execution Time Estimates

In order to determine whether the Intel 80286/80287 microprocessor chip set had the processing power required to support RCCL, some "ball-park" estimates were made, as follows

The performance of the RCCL implementation for the Puma-260 manipulator running on a VAX 750 with floating point accelerator was measured [Lloyd 85] in terms of the CPU time taken by the RTC control routine under various conditions. These results are summarized in the following table, where it should be noted that in Cartesian mode, the forward and inverse kinematics must be computed every cycle. In joint mode, however, the inverse kinematics need be computed just at the beginning and end of each motion segment, intermediate values are interpolated during consecutive cycles If the sample period is set to 28 milliseconds, it may be seen that under worst-case conditions the VAX will spend over 70% of the time available between sampling instants inside the RCCL control routine, which leads to unacceptably poor overall performance for other users. The solution was

to use a 56 millisecond sample period: for the Puma manipulator, this does not appear to cause noticeable performance degradation.

CPU Time per Cycle for VAX/Puma Implementation	
Trajectory Mode	**Time per Cycle**
Joint Mode	12.0 msec
Cartesian Mode	15.5 msec
Cartesian plus 2 functional transforms	20.0 msec

Keeping this information in mind, we look at the computational complexity of the forward and inverse kinematics of the Microbo in comparison to the Puma, summarized as follows (this table is repeated from Chapter 3).

Kinematic Solution Complexity					
	mult	add/sub	sqrt	atan2	sin+cos
Microbo forward	30	13			4
Puma-260 forward	59	29			6
Microbo inverse	22	13	1	4	4
Puma-260 inverse	64	42	2	7	6

From the above we can see that the Puma solutions are about twice as complex as those for the Microbo.

Next, we look at the floating point performance of a VAX-750 with floating point accelerator (FPA) versus a 6 MHz Intel 80286 with a 5 MHz 80287 math coprocessor. The table below includes results using both the vendor-supplied math libraries and a set of math functions written in 8086/8087 assembler language by the author.

† The Intel-supplied math library is very inefficient, apparently, little use was made of the 80287's 8-element floating point stack.

Floating Point Performance (time in μsec)		
	80286/80287	VAX-750 w/FPA
sin (C library)	2500	230
cos (C library)	2500	230
sin+cos (assembler)	540	
atan2 (C library)	2900	310
atan2 (assembler)	420	
4x4 transform multiply	2300	250
multiply (reg-reg)	19.5	2.4
multiply (mem-mem)	22.5	14.6

From the table it can be seen that the 80287 coprocessor provides the 80286 system with similar performance to the VAX for trigonometric calculations in assembler, but that floating point multiplication is almost an order of magnitude slower. We note also the benchmarks done by Hinnant [Hinnant 84] using the "Sieve of Eratosthenes", which is representative of non-floating point operations. These results showed the Intel processor achieving about 35% of VAX-750 speed.

The 80287 does not directly execute the sine or cosine functions but instead provides a tangent function which takes as input an angle θ and leaves y and x on the floating point stack. $Sin(\theta)$ and $cos(\theta)$ may then be computed using $y/\sqrt{x^2+y^2}$ and $x/\sqrt{x^2+y^2}$. From this it may be seen that once the sine has been computed, the cosine result is obtained with one additional divide operation. This technique was used in the author's assembly language implementation of the $sin + cos()$ function.

To summarize, the Microbo kinematics are about half as complex as those for the Puma, and the 80286/80287 processor has an overall performance of about 40% of a VAX-750 doing a mix of math and trigonometry. We therefore estimate that the Intel system will take about the same time to execute the RCCL trajectory control function for the Microbo as the VAX does for the Puma. Assuming similar overhead to communicate with the joint controllers, we should be able to achieve a 28 millisecond sample rate in joint mode, but may be forced to use a slower rate in Cartesian mode. It will be necessary to use the assembly-language versions of the trigonometric functions as the corresponding calls in the standard C math library are unacceptably slow.

4.4.2 Force Sensing

The VAX/Puma implementation of RCCL computes joint torques based on the assumption that these are linearly proportional to the joint motor currents There is a sensing resistor in series with each of the joint motors, and the resulting voltage is filtered and sent to the input of an analog-to-digital computer The inverse Jacobean matrix is used to map the forces from joint space to Cartesian space As has been shown in [Lloyd 85], even the relatively low static friction terms of the Puma-260 robot contribute major uncertainties to the determination joint torques The corresponding friction of most of the Microbo joints is, by observation, at least an order of magnitude higher, and rules out this method entirely Mounting torque transducers directly on the joint shafts would not overcome this problem The preferred solution would be to use a wrist-mounted force sensor Preferably, this force sensor would be "intelligent" that is, it would map the forces it detected onto a cartesian coordinate system attached to the wrist We can then transform forces to the tool tip using

$$\mathbf{F}_{4,6} = \begin{pmatrix} 0 & \mathbf{B} \\ \mathbf{B} & 0 \end{pmatrix}$$

where

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

This is of course computationally far less expensive than computing the forces at the tool tip from the motor currents via the inverse Jacobian transformation, and eliminates the problems of static friction Compensation would have to be made, of course, for gravity loading due to changes in the configuration of the wrist

4.4.3 The Interface to the Joint Controllers

The RCCL trajectory generator assumes *synchronous* control over the joints That is the path of the end effector is controlled by breaking motion up into segments of equal time duration and repeatedly computing and specifying *setpoints*, or target positions in joint space Each joint must therefore servo from setpoint to setpoint by computing the velocity and acceleration required so as to smoothly reach the new setpoint at the end of each sample period The joints must *not* decelerate as they reach the setpoint rather, they should assume that the velocity will remain the same during the next segment Stiffness

during motion and while at rest must be maintained We note in passing that the Puma s Unimation controller has the joint-level microprocessors executing a PID-based (Proportional, Integral, Derivative) control algorithm which satisfies the above requirements The details are proprietary information of Unimation

In the chapter describing the Microbo robot, we examined the joint controllers, their communication protocol, and the control algorithm that they implement We saw that as programmed by the vendor, the joints assume *asynchronous* control There is no notion of a sample rate, and thus their velocity and acceleration must be set explicitly Additionally some of the joints have more than one *work area*, since the word size of the joint encoder is insufficient to cover the entire physical range of the joint In order for these joints to travel from one work area to the next, a special command must be sent to explicitly cause a change of work area

If we wish to control the Microbo joints in a synchronous manner without reprogramming the controller EPROMs, we must verify that we are able to

- interrupt and communicate with the joint processors as fast as every 28 milliseconds[‡] without disrupting the joint servoing,
- accurately control joint velocities,
- defeat the joint-level trajectory control,
- deal with the work area problem

The Multibus Adapter Card

In order to communicate at high speed with the Microbo joint controllers, an interface was designed which maps a logical segment of the C1 bus into the I/O space of the Multibus In this way the joint processors' control and data registers may be directly accessed by a processor on the Multibus The Multibus I/O space rather than address space was used because the full 1 Megabyte Multibus address space is required for contiguous RAM and ROM memory

Figure 4 1 shows the Multibus Adapter card It resides on the Multibus, with a flat cable connection to the RCU C1 bus The cable is shielded and limited in length to about

[‡] This rate is a historical artifact of the Puma implementation We continue to use the 7 14 28 scheme to maintain compatibility with RCCL function calls which deal with the sample rate

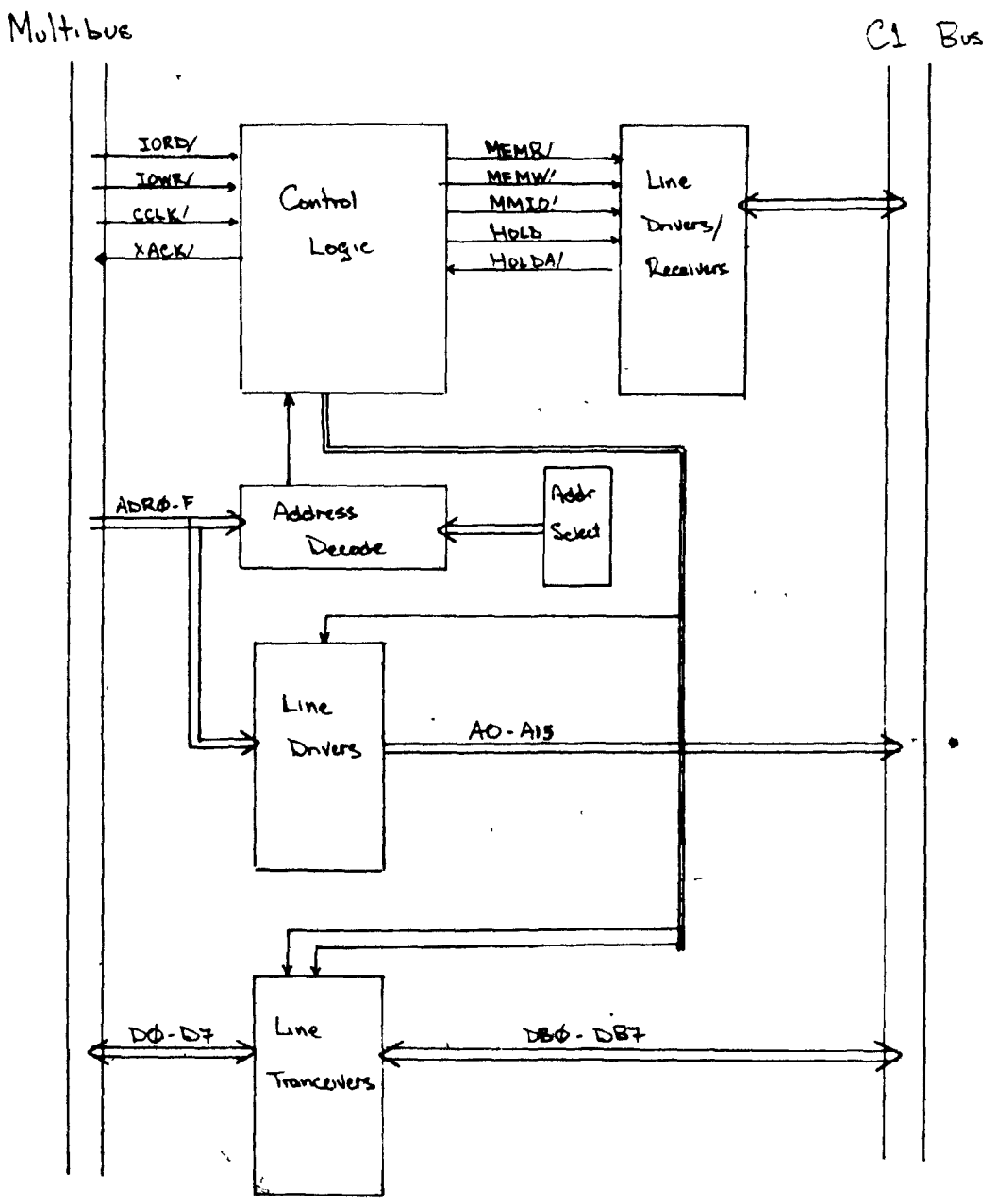

Figure 4.1 The Multibus Adapter Card

1 meter due to the finite current source capability of the bus driver chips Since one of the design constraints was that the IRL processor remain resident in the system, the adapter card asserts the IRL CPU's 8085 *hold* input and waits for the appropriate *hold acknowledge* (*HOLDA*) signal before taking control of the RCU C1 bus Once this low-level handshaking has taken place, the IRL processor's bus drivers are assured to be tri-stated and the adapter can safely drive the C1 bus The connection to the hold and hold acknowledge signals of the IRL processor, which was the only modification made to that card, has no effect on the operation of the card except when the Multibus adapter is present and active, and thus satisfies the constraint noted above

The timing of the bus interface is shown in Figure 4 2 The sequence is initiated by the address decoder detecting a valid address in conjunction with the Multibus */OWR/* or */ORD/* signals This initiates the *HOLD* signal onto the C1 bus The conjunction of *HOLD* and the resulting *HOLDA* from the IRL CPU then enables the adapter card's bus drivers to place an address onto the C1 bus, and initiate the acknowledge timing The bidirectional data bus drivers are enabled in one direction or the other depending on whether this is a read or write cycle We use a shift register clocked by the Multibus *CCLOCK* signal to generate the Multibus *XACK/* signal after a delay of 5 clocks, or about 1 microsecond The *XACK/* signal causes the Multibus master to remove the */OWR/* or */ORD/* signal and the cycle ends

We note that the adapter card currently does not control the RCU C2 bus, which contains the interface to the RCU power control relays, teach pendant, and other I/O devices We therefore rely on the IRL processor to power up the robot and control the teach pendant However, this tie to IRL is hidden from the user by the RCCL teach program, in this way, the RCCL programming environment is maintained for the Microbo robot (as described in Appendix A)

Communicating at the Sample Rate

The discussion in Chapter 2 concerning the Microbo RCU showed that each joint controller appears as a set of registers in the address space of the C1 bus We reiterate that every time a *data* or *command* register is read or written, the appropriate *status* register must first be read to verify that the joint controller is ready to receive or send the data (if not, the status register must be polled until the appropriate state is indicated) We

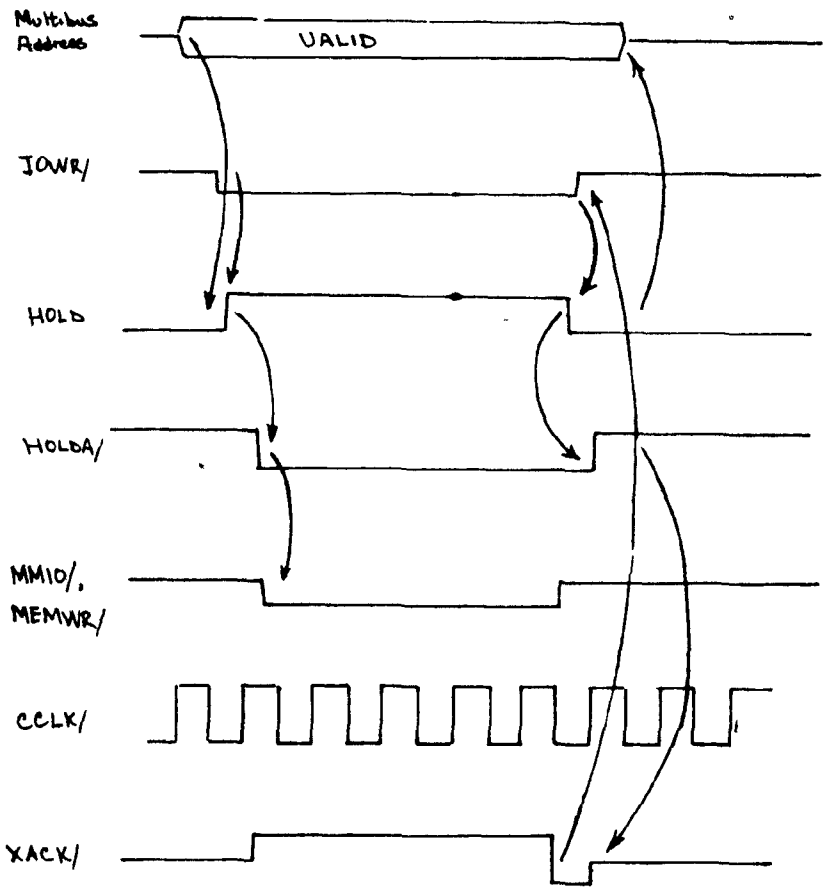

Figure 4.2 Multibus Interface Timing

call each such handshaking sequence a communication *exchange* A typical command with its 2 bytes of data thus requires 6 bus exchanges 2 for the command byte and 2 for each data byte

Every communication exchange involves *interrupting* the joint controller from its trajectory or servo control function Before synchronous control can be considered, we must confirm that the joint controllers can withstand being interrupted at the desired 28 millisecond rate We also need to measure the average duration of typical RTC communication

sequences

A test program was written which used the bus adapter described above installed in the Intel System 310. Using the iRMX-86 operating system, a real-time task was created which sent three commands to each joint every 28 milliseconds. The first command was a query for the current encoder value, the next was the command to set a position target, and the final command was to set the joint velocity. To begin with, the new target for each joint was simply set to the current encoder value, and the velocity set to a median value. The program was executed on the iSBC 286/10 processor and it was confirmed that the manipulator responded by going into "zero gravity" mode, i.e. if placed by hand into some configuration, it would stay there. The stiffness due to the joint position servoing was still present, of course.

This test confirmed the operation of the bus adapter and the viability of synchronous communication with the joints at the rate required by RCCL. With six joints, three commands per joint, and six bus exchanges per command, we have at least 108 bus register reads or writes per sample period. Oscilloscope analysis showed that the average time required for each bus exchange with a joint microprocessor is about 75 microseconds. This gives a total time of 108 × .075 or approximately 8 milliseconds to interact with the manipulator joints.

Synchronous Control of the Microbo Joints

Because it was impractical to re-program the joint processors, we were constrained to find a way of using them "as is". The previous experiment had shown that we could send targets and velocities to the joints every 28 milliseconds without disrupting their basic operation. Our next task was to demonstrate that we could establish manipulator-level trajectory control and achieve reasonably smooth motion.

As described in Chapter 2, the RCCL trajectory generator operates in an open-loop manner. RTC assumes joint-level position control. This control is expected to honour position setpoints generated at the sample rate such that targets are reached exactly at the end of each period.

Referring to Figure 2.7, it may be seen that if we simply send un-edited position requests to the Microbo joint controllers, then as we approach the "static" zone during

each motion, the joint's built-in trajectory control algorithm will cause deceleration, and the resulting motion will exhibit very high vibration at the trajectory sample frequency

A solution is to have the joints "follow" position targets which remain far enough away from the current position so that the deceleration phase is never entered and exercise control via velocity commands

This scheme was evaluated, using the iRMX-86 operating system, with the real-time test system shown in Figure 4 3 An RMX mailbox was used to queue a set of motion requests to an interrupt-driven "robot task" The latter serviced one such request each sample period by computing a target and velocity and sending the appropriate commands to the joint processors, while keeping trace records of actual and requested positions The main program created the series of motion requests using simple interpolation

The actual joint target X and velocity V to send to the joint each RTC sample period were computed as follows, where $X_{desired}$ is the position to be reached at the end of the next sample period, X_m is the last measured joint position, and K is the appropriate conversion factor (defined by the vendor)

$$X = X_{desired} + (X_{desired} - X_m)$$

$$V = K \times |X_{desired} - X_m|$$

The resulting motion of the manipulator was reasonably smooth, and judged to be good enough to proceed with an implementation of the full RTC layer using the existing joint controllers and this scheme

Dealing with Work Areas

A final issue concerned the Microbo joints' "work areas" (described in Chapter 2) It was desirable that RTC programmers be able to specify arbitrary position targets without having to consider whether a change of work area was required Using the test system described above, various algorithms were tried The one which seemed to work most reliably involved using two encoder count "thresholds" and associated "transition zones"

Referring to Figure 4 4, when the actual encoder position of a joint passes an upper or lower threshold, it is deemed to be in a "transition zone", and a command to change work areas is automatically sent to the joint controller The thresholds are set such that there

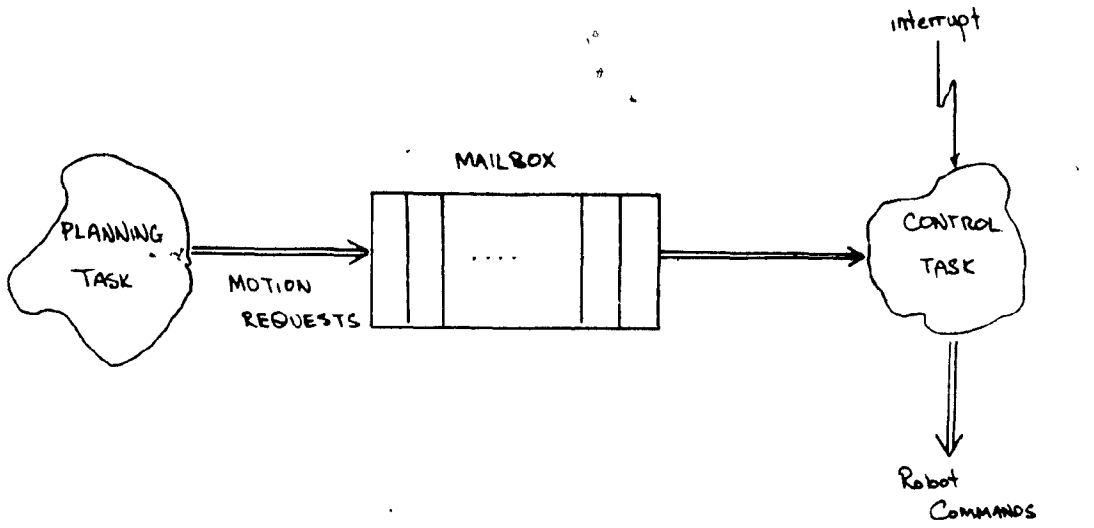

Figure 4.3 Test System

is some hysteresis, otherwise the system oscillates This scheme continuously maintains the joint inside the most appropriate work area, both when it is in motion due to RTC position setpoints, *and* when it is being manually moved (for example, in "zero gravity" mode during teaching) The only restriction is that if the joint speed becomes too high, we may pass all the way through the transition zone between samples, and the encoder may then "wrap around" undetected As the sample period is increased, the problem becomes more serious

Joint 3 has the finest resolution (see Table in Section 2 5) and therefore represents the worst case Referring to Figure 4 4, we see that the lower threshold is 7450 counts, and the upper threshold is 57550 counts (The hysteresis is thus 57550 - (7450 + 50000) = 100counts.) The lower transition zone is 7450 counts while the upper zone is 65535 57550 = 7985counts, so the lower zone represents the more dangerous case

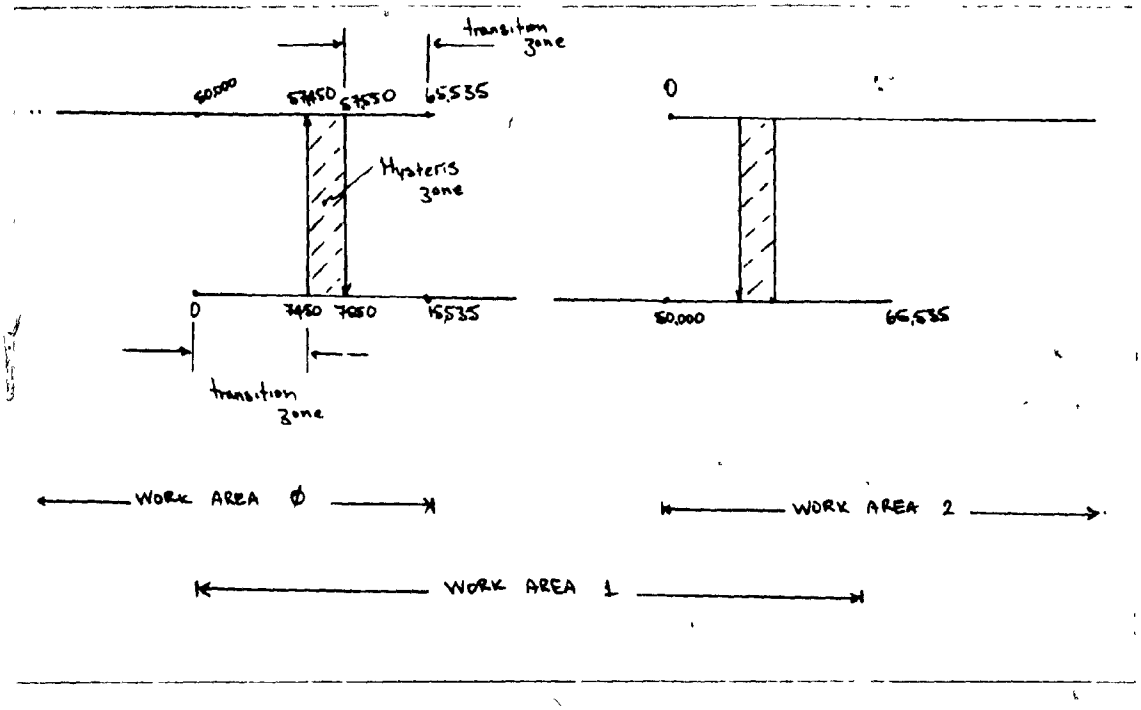

Figure 4.4 Work Area Thresholds

Now, using a 56 millisecond sample period and joint 3's encoder resolution of 0.0013 $\frac{mm}{count}$, we get a limit of

$$\frac{7450 \times 0.0013}{0.056} = 173 \frac{mm}{sec}$$

which represents a performance limitation[*] For a 28 millisecond sampling period this is relaxed to 346 $\frac{mm}{sec}$, which poses no problems

Some communication and computational overhead is added using this scheme, because we must check the encoder value against two limits each sample period, and possibly send an additional command to each joint Since the "change work area" command has no associated data, it requires just two communication exchanges (.075 milliseconds) to check the status register and sent the command In the worst case (all joints simultaneously changing work areas) this adds up to 6 × 2 × 0.075 or about 1 millisecond for 6 joints

Summary

Our feasibility study has shown that the communication overhead for 6 Microbo joints

[*] The maximum speed specified by the vendor for this joint is approximately 200 mm/sec

under RTC position control is approximately 8 milliseconds The worst-case cost of checking for and adjusting work areas is 1 millisecond

Using projections based on VAX versus 80286/287 performance and the relative complexity of the kinematics of the two robots, we estimated that the Intel microprocessor would take approximately the same time as the VAX, 20 milliseconds, to execute the RCCL setpoint function in Cartesian mode with 2 functional transforms

A sample period of 56 milliseconds seems to impose an unreasonable limit on the maximum velocity of joint 3 due to "work area" problems, but if we use a 28 millisecond sample period, joint communication overhead then represents a serious reduction of the time available Considering that we must leave a reasonable proportion of the CPU to run the planning-level program and the operating system, we conclude that an Intel 80286/287 processor is sufficient for trajectory control alone, but insufficient to implement the joint communication as well

4.5 The RTC Implementation

4.5.1 Overview

The solution taken was to implement the RTC system using two microprocessors, an Intel iSBC-286/10 and iSBC-86/30 The 86/30 is dedicated to the interface with the robot joint controllers, including the "work area" overhead The 286/10 is dedicated to executing the RTC control functions and planning level program Communication between the two processors is via the RTC chg and how data structures which are located in dual-port memory on the 86/30 The system is shown diagrammatically in Figure 4 5

Both processors use the iRMX-86 operating system described in Chapter 1 The 86/30 runs a minimally configured version which includes just the Nucleus and a single user job containing the robot communication tasks The 286/10 runs a fully configured version of the operating system, including the Human Interface, Application Loader, Universal Development Interface, Extended I/O System, Basic I/O System, and System Debugger The RTC and RCCL libraries along with a full complement of software development tools reside on a Winchester disk controlled by the 286/10 Users may thus edit, compile, link and load their RCCL or RTC programs from a video terminal connected to this processor

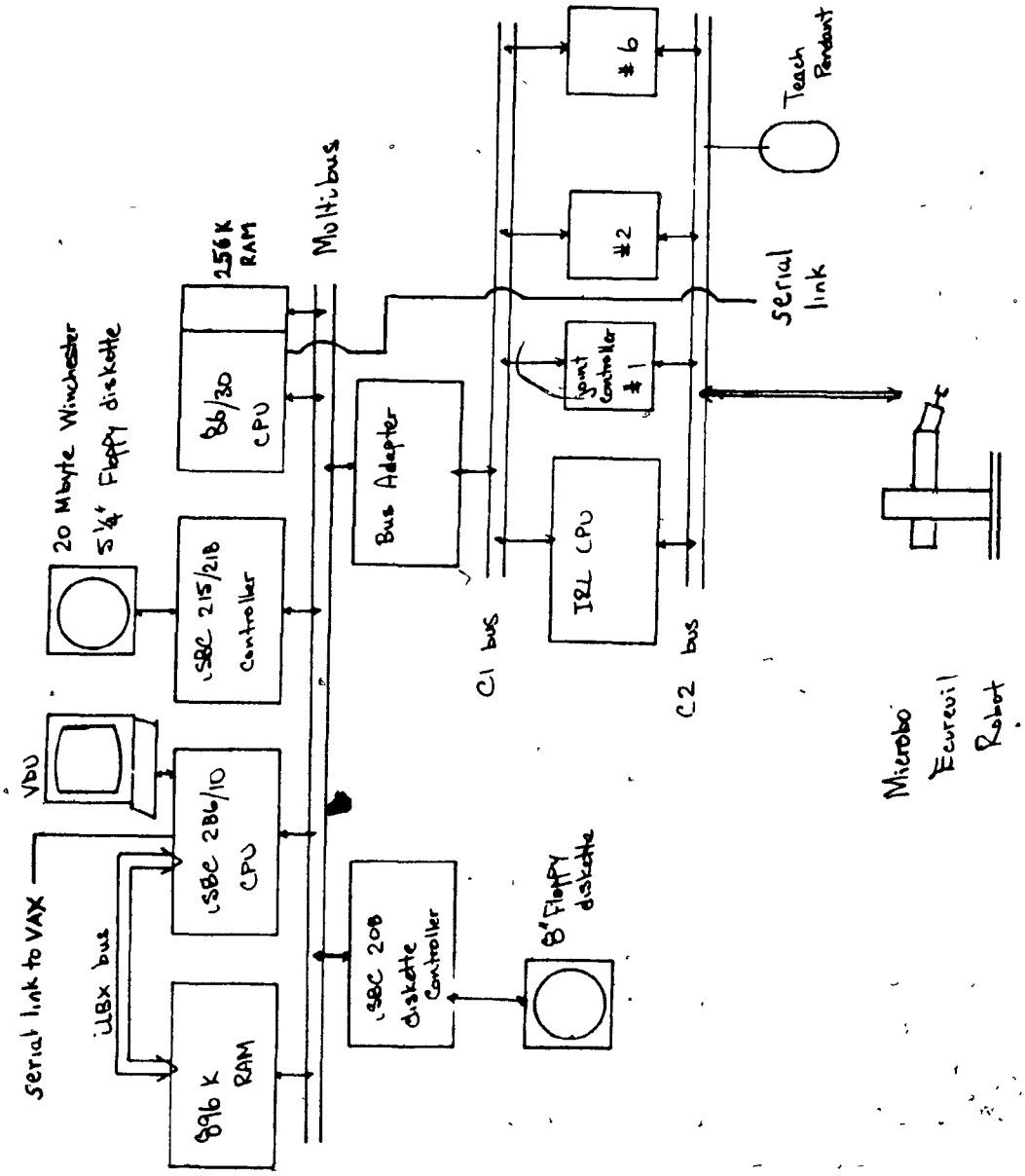

Figure 4.5 The RTC Hardware Implementation

An objective of this implementation was to achieve concurrency between the robot communication and the execution of the RTC control function. as soon as the setpoint information is available from one processor. the communication task on the other can begin to interpret it. send appropriate commands to the joints. and gather state information from the joints Meanwhile. the first processor can compute the next setpoint.

4.5.2 Task Architecture

The task architecture of the RTC implementation is shown in Figure 4.6. and the timing may be seen in Figure 4.7. The following discussion refers to these two figures.

The communication task on the 86/30 processor. which is automatically created by the operating system when the processor boots up. initializes a hardware timer such that an interrupt occurs every RTC sample period. The task is subsequently invoked each time the interrupt is signaled. and checks the current state of the RTC system by looking at a status flag in shared memory If there is no RTC control session active. the task does nothing Otherwise. it communicates with the robot joints to gather the current manipulator state information (the how data structure). and then interrupts the 286/30 processor.

When a user program makes an `rtc_open()` call. an RTC control task is created on the 286/10 processor This task first initializes the RTC system. basically by clearing the appropriate flags in the fields of the how and chg data structures

An `rtc_control()` call causes this task to set the status flag in shared memory for the 86/30 communication task The control task then waits for the interrupt from the communication task. The occurrence of the interrupt indicates that the how information structure has been updated. that it may be copied to local memory. and that the first RTC control function (f1) may be executed After the function call. the control task signals the communication task using the status flag in shared memory. and the chg structure is copied from shared memory to 86/30 local memory The communication task immediately signals back the control task via the status flag

The communication task then proceeds to process the chg information by generating appropriate Microbo joint commands. while the RTC control task concurrently executes the second RTC control function (f2) such as the RCCL "setpoint" function. At this point

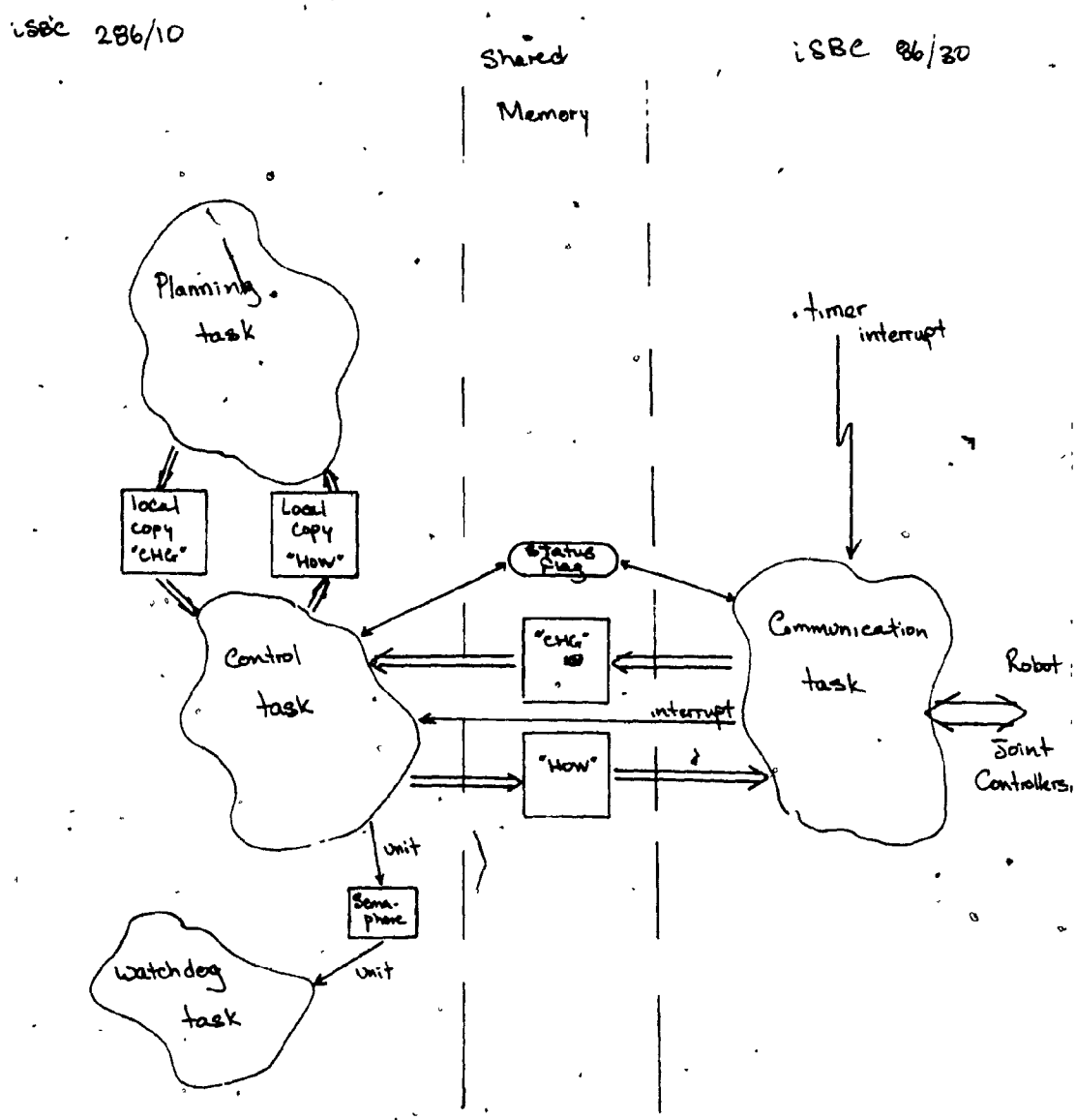

Figure 4.6 RTC Task Architecture

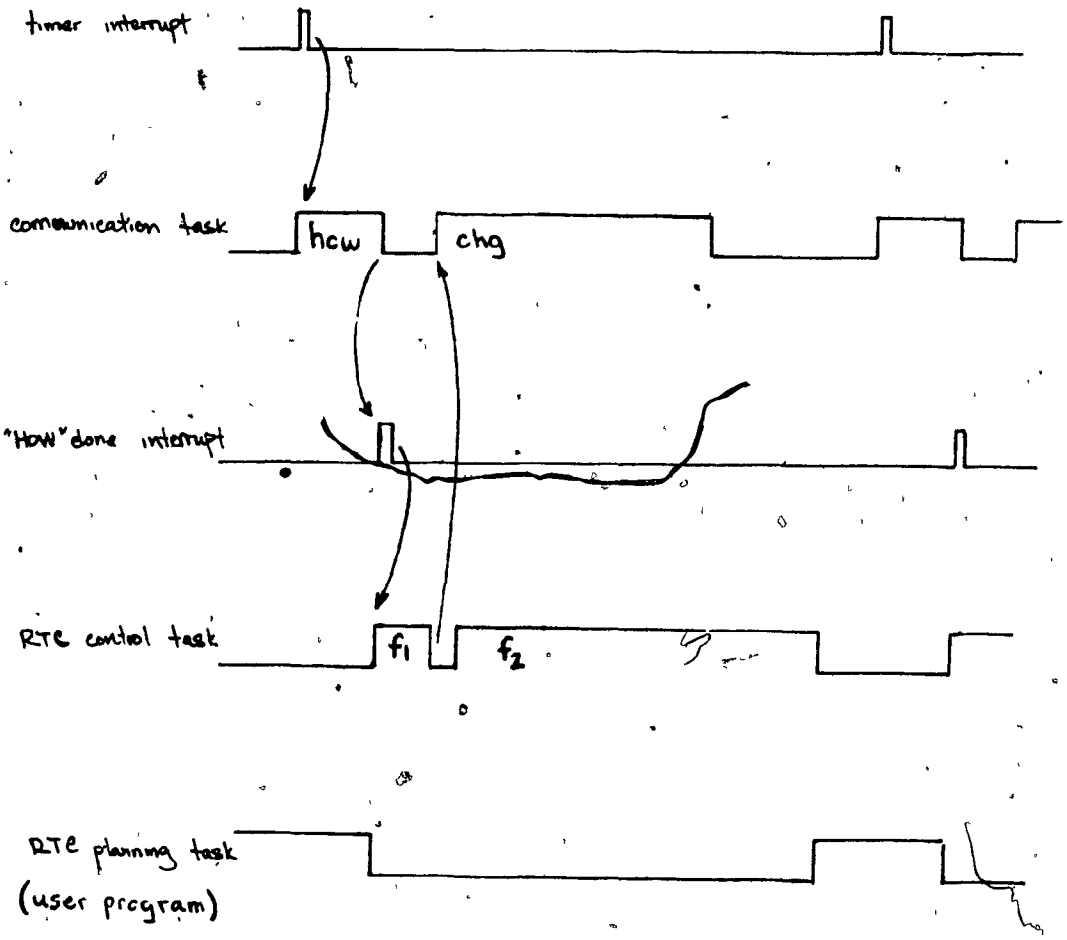

Figure 4.7 RTC Task Synchronization

the cycle repeats, the communication task waits for another timer interrupt and the control task waits for the "how data ready" interrupt

The priorities of the control and communication tasks are relatively high, since these are iRMX-86 *interrupt tasks*, the task priority is a function of the (prioritized) hardware interrupt level This was selected in both cases to be higher than that accorded to all other devices (eg the console device on the SBC 286/10) We thus guarantee that the tasks will always be invoked with minimum latency

Limit Checking and Error Handling

The robot communication task on the 86/30 handles all limit checking and error handling RTC control programs set the checking mode of the system via the chg data structure The robot communication task, according to the flags which are currently set, checks for observed position out of range, target position out of range, observed velocity out of range, target velocity out of range, joint calibrated, etc If a limit is exceeded, or a looked-for condition reached, a corresponding flag is set in the how data structure

The robot communication task also checks for basic problems such as loss of robot power, communication error while talking to a joint, etc Again, flags are set in the how structure according to the error

The RTC control task checks the flags in the how structure once each cycle, and the control session is terminated (control task deleted) if an error is detected As explained above, the control task has higher priority than that accorded to any other I/O tasks on the SBC 286/10, so it cannot do console I/O We therefore use the concept of a *watchdog task* to handle terminations This is done as follows

When the control task is created by the rtc_open() call, an additional task, called the watchdog task, is created This task creates an iRMX-86 *semaphore*, and then simply waits at the semaphore for a *unit* It is the control task's responsibility to send a unit once each control cycle If the unit does not arrive within the appropriate period, the watchdog task wakes up and looks at the state of the system

Depending on the error condition, the robot may be powered down In any case, the control task is deleted, an appropriate error message is sent to the user's console, and the rtc calling program (an iRMX-86 *job*) is terminated

The user can trigger a similar sequence via the "control-C" key. during an RTC control session it is bound to a special handler which gracefully terminates the session

The `teach()` Function

One prerequisite for a useful robot programming environment is a way of "teaching" robot positions using a teach pendant In the RCCL environment, it is useful to have a `teach()` function which allows the robot to be moved under teach pendant or keyboard control, then returns the corresponding $\mathbf{T}_6$ transform The latter may then be saved in a database or used in RCCL motion equations within the calling program

Lloyd, in his implementation of RTC/RCCL for the Puma manipulator, wrote a teach function which dealt directly with the Unimation controller's teach pendant [Hayward and Lloyd 85] In the present case, the IRL interpreter provides the basis of the Microbo teach pendant program

Because the 8085-based IRL processor remains resident in the RCU, it was possible to incorporate it into our implementation by connecting the IRL console port to a serial port on the 86/30 processor The teach function, called from the context of an RCCL program running on the 286/10 processor, just suspends the current RTC control session Then, using a field in the chg data structure, it causes the appropriate command string to be sent to the IRL interpreter via the serial port This activates the IRL teach program The robot may then be moved under control of the RCU teach pendant When this mode is terminated, the IRL teach program is interrupted the $\mathbf{T}_6$ transform computed, and the RTC control session resumed

The teach function also incorporates a "zero gravity" mode, such that the manipulator may be pushed into a desired configuration, and the corresponding transform computed and returned to the calling program (This corresponds in utility to the Puma's "zero gravity" mode)

Porting the RCCL Library

Once the RTC system described above was up and running, the RCCL library could be ported to the RMX system. The only prerequisite was to code the Microbo forward and inverse kinematics, verify them, and replace the appropriate modules in the library Given the results of Chapter 3, this was fairly straightforward.

A simple file transfer program (discussed in Appendix B) was written under Unix 4 2Bsd to transfer files from the VAX to the System 310 over a serial line, and the appropriate sources were then recompiled using the RMX utilities The only modifications which had to be made were due to the slightly different pathname syntaxes of the two operating systems (RMX-86 and Unix) The assembler-language math and trigonometric functions (sin+cos atan2, etc) discussed in Section 4 4 1 replaced the corresponding standard C functions in the library

4.5.3 Performance

Once the RCCL library was compiled and installed on the Intel system, a suite of test programs was written These were suitably modified versions of existing appropriate RCCL programs which had been written for the Puma (Because the Microbo is a cylindrical robot, it s workspace is considerably different from the Puma's) These confirmed that the system functioned as expected, and that all of the supported[†] RCCL function calls worked correctly Motion was reasonably smooth, although there was some vibration which was attributed to the Microbo's joint controllers' poor performance

The CPU time consumed by the RCCL control function was measured under various conditions, and the results are shown below

4.5.4 Summary of Departures from the VAX/Puma Implementation

4.5.4.1 RTC

The iRMX-86-based version of RTC has the same general functionality as the VAX version, the user interface is basically unchanged, both in terms of the function calls and the data structures

The internal implementation, however, is quite different, due to a) the different operating systems underlying the two versions, b) the different robots and c) the dual-CPU

[†] As discussed in Section 4 4 2 compliance-related calls are not currently supported due to the lack of appropriate force sensing

nature of the the microprocessor-based system In order to increase efficiency, some functionality, for example conversion of encoder values to RTC range values, has been moved to the second (86/30) CPU

The how and chg data structures are slightly different, the user should look at their definitions in /rccl/h/rtc h One basic change which will be noted is that the user of the Microbo robot no longer deals with encoder counts; both chg and how specify joint positions in terms of *range values* The latter coordinates express the joint variables such that they are zero at the joint minimum, and increase so that the maximum range value is the range of the joint, either in millimeters (joints 2, 3) or radians (all others)

Specifically, it should be noted that the `chg.i_motion vali` structure element, which is used to send position setpoints to the joint microprocessors, has been replaced by `chg i_motion[i].valir` and is a `float` as opposed to an `int` field as used on the VAX Similarly, the `how.pos` elements are now `float` instead of `int` values

The `chg i_motion[i].set` field supports 3 operations

- POS : simply causes the position specified to be translated to an encoder value and sent to the joint
- POSVEL : causes the system to compute velocities, accelerations, and encoder targets such that the specified position becomes the target position for the next sample period
- STOPCAL : causes the joint to be stopped and sent to its mechanical calibration position (via the joint-level calibration command)

The CUR command is not supported, due to lack of direct control of the D/A converters on the Microbo joint controllers The present joint controller firmware does support "read instantaneous current" and "set maximum current" commands, but the trajectory algorithm is still operative, so that these commands are of no practical use in terms of RTC current control requirements

As would be expected, the error messages returned by RTC are generally different from the messages of the VAX/Puma implementation In particular, there are two functions `print_rtc_error()` and `print terminate code()` which have been re-implemented to print appropriate error messages strings on the stdout device These functions are found in `/rccl/rtc/prt_error.c`, and the error code definitions are found in `/rccl/h/errors.h`

As has been noted in Chapter 4, the VAX and Intel floating point binary formats are

different. Intel uses the I E E.E standard. DEC doesn't This has no impact on the current implementation

4.5.4.2 RCCL

The Microbo implementation of RCCL supports all functions *except* those dealing with compliance or force The RCCL library was ported basically unchanged, except where compiler incompatibilities were encountered One such problem was that the 80286 processor is a 16-bit machine, so that the C `int` type is 16 bits as opposed to 32 bits on the VAX This does not cause any problems as long as programmers remember to use the `long int` type where required (eg the `terminate` global variable)

The `setpoint()` control function was modified to work with the new chg and how structures, and the force-related code was removed

Some of the mathematical functions were re-written in 80286/80287 assembly language, and the corresponding C functions were removed from the library They are `sincos()`, `atan2()`, `cross()`, `dot()`, `hypot()`, `invert()`, `reduce()`, and `trmult()` Note that `sincos()` had been previously been implemented as a C macro These functions may be found in `/rccl/math/`

4.5.4.3 Robot Calibration

Each of the Microbo's joints has a mechanical calibration position, when the joint controller receives a "calibrate" command it will seek this position Associated with the command is a 16-bit value, this becomes the encoder value at the calibration position The values used in the present RTC implementation are those originally used by the IRL system, and are defined in the `/rccl/h/constants.c` program

The ratios of encoder counts to joint motion was first taken from the vendor's specifications, but these proved to be mostly incorrect Measurements were then taken and the correct values incorporated

As with the VAX implementation, the `constants` program contains all the basic robot constants calibration positions, joint ranges, encoder counts per range unit, etc Compiling, loading, and executing this program results in the creation of a C include (`.h`) file In the VAX/Puma case this was called `pumadata.h`, it is now called `microbodata.h` This

file may then be included wherever the constants are needed

The Microbo manipulator may thus be calibrated via chg structure commands the set flag should be set to STOPCAL, and the how status word polled until ALL JOINTS is detected At this point the corresponding range values may be read from the how struct A calibration program, called `calib`, exists to do the Microbo calibration The source code for this program may be found in `/rccl/control/calib.c` Note that the encoder calibration values are internal to the iSBC 86/30 CPU's code, they are not sent from the RTC level

4.5.4.4 The Teach Function

As discussed in Chapter 4, a `teach()` function was written which makes use of the IRL teach program and the Microbo's teach pendant

The calling protocol of `teach()` is identical to the VAX/Puma implementation, there are also `teach_pos()`, `teach t6()` and `teach angles()` functions which call the basic teach function in different ways The source code for these is in `/rccl/teach`

The actual operation of the Microbo teach function is different from the VAX/Puma version, it is simpler and more restrictive

The name "zero gravity" is perhaps a misleading term, there is no gravity correction The joints are merely servoed to their observed position The prismatic joints are treated differently in that integration is turned off at the joint level and the velocity and acceleration these joints is set to zero, for all other joints, the velocity and acceleration are set to small values

The teach function has the following commands

- `Z` Enter "zero gravity" mode The current joint parameters are saved, and a the appropriate RTC control function is used
- `z` Exit zero gravity mode
- `m` Move joint using keyboard The program will query for a joint number and a target position (in RCCL coordinates)
- `p` Enter teach pendant mode The current control session is suspended, and the IRL teach program takes over control of the robot Users should consult the IRL manual

entry under "TEACH" Control may be returned to the RTC system by typing any character (followed by a carriage return)

xd. Examine position, degree coordinates The current positions of the six joints are fetched and displayed on the console, in units of degrees or millimeters The coordinate system is as defined in Chapter 3 of this thesis, see the `constants.c` for more information

xr. Examine position, radian coordinates Similar to the above, but coordinates are radians or millimeters

xR. Examine position, range coordinates As above, but values are in range coordinates zero at the joint's minimum, and expressed in radians and millimeters

X . Examine motor parameters The Microbo joint controller parameters for the requested joint are displayed (Appendix B)

h . Open hand There are two relays to open/close the hand, are operated in an "exclusive or" manner

H : Close hand As above

e . Exit teach function.

? Print menu of available commands

4.5.4.5 The Database Utility

This operates exactly as does Hayward's original implementation, except that the maximum length of the "name" string is now 32 characters (was 16) Also, the configuration field of the TRSF structure is now saved in the database The database program sources may be found in `/rccl/db/` the main module is `/rccl/db/dbot.c` An example program which uses the database functions is `/rccl/db/test_db.c`

4.5.4.6 Example RTC Programs

A number of example programs at the RTC level may be found in `/rccl/control/` These are listed below

`calib.c` This is the calibration program

`limp.c` This "limps" the robot joints by turning off the Microbo's arm power relay

`marsh.c` Puts robot in "zero gravity" mode

`report.c` Repeatedly reports the current joint positions, in degrees/millimeters

`test.c` A general-purpose test program to send commands to the individual robot joints, using the chg structure Most commands normally affect a single joint, the default is initially joint 1, and may be changed using the "j" command `test` includes code to set the joints' so-called "motor parameters" (Appendix B) The available commands are

- `a` : set joint acceleration
- `f` : set joint velocity factor
- `h` . open hand.
- `H` · close hand
- `j` · set joint number for subsequent commands.
- `k` . set joint dynamic integral
- `K` : set joint static integral
- `m` : move joint to target at current velocity
- `M` : servo joint to target, compute velocity
- `o` : output bit clear
- `O` : output bit set
- `p` : power off robot.
- `P` · power on robot
- `q` quit
- `r` : set sample rate
- `s` : stop joint at present position.
- `v` : set joint velocity.
- `V` : set static zone joint velocity.
- `x` . examine position
- `X` : examine motor parameters
- `z` : set static zone size
- `?` : print menu of available commands

CPU Time per Cycle for Intel 286/Microbo Implementation	
Trajectory Mode	**Time per Cycle**
Joint Mode	18 0 msec
Cartesian Mode	25 5 msec
Cartesian plus 2 functional transforms	33 5 msec

As predicted by the feasibility study, RCCL "joint mode" will run successfully using a 28 millisecond sample period, but "trajectory mode" timing is marginal at this rate, as not enough of the CPU is available to run the main user program and take care of system overhead In this case, a new robot interrupt may arrive before the previous one has been serviced The operating system will detect this and abort the RTC task (with an appropriate error message) A 56 millisecond sample period is thus required in this case.

In addition to the standard tests, a multi-robot demo program (Figure 4 8) was written by CVaRL researchers [Freedman, et al 85] This had both the Puma and Microbo robots, each under control of RCCL, cooperating to inspect and repair a small circuit board Communication between the Intel System 310 and the VAX network was via a serial link This test successfully exercised a good portion of the RCCL library

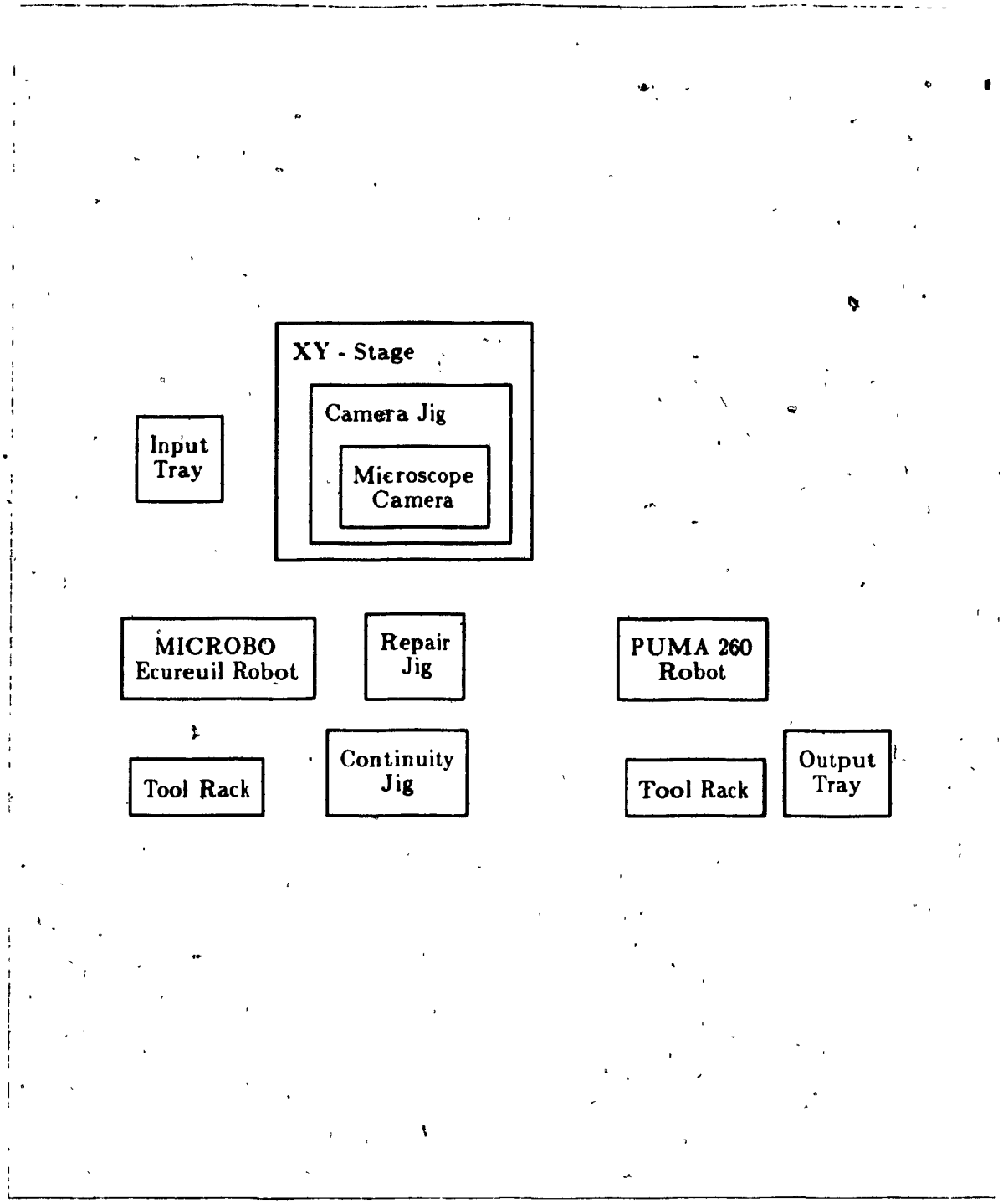

Figure 4.8 The Multi-robot Demo

Chapter 5 Summary and Conclusions

This thesis has described an implementation of the RTC/RCCL robot programming and control environment for a Microbo Ecureuil manipulator The system is based on a set of Intel microprocessors, resident on a Multibus, connected via a custom-built bus interface to the Microbo's RCU control unit

In this final chapter we evaluate the utility and performance of our implementation, and discuss possible future enhancements.

5.1 Evaluation of the RCCL/RTC Programming Environment

The RTC/RCCL environment provides programmers with a flexible set of tools for robot programming. The RTC layer allows users to write basic manipulator control programs (eg "zero gravity" and "teach" functions) by providing easily used mechanisms to access the joint level of the control hierarchy The RCCL function library, using RTC, provides Cartesian path control using the notions of homogeneous transforms and motion equations The RCCL "motion request queue" allows concurrency between the planning level and the control level, though the programmer must be careful to explicitly synchronize the two levels using the mechanisms provided RCCL, as claimed, proved to be easily transportable to a different CPU, computer architecture, and manipulator

Programming experience, such as the multi-robot demo described in Chapter 4, has shown that RCCL is really quite a low-level language For applications where programmers wish to be relieved of more of the details of path planning , a higher-level user interface to RCCL might be provided Implementing an intelligent task planner using a language such

as PROLOG or LISP is eventually planned as a CVaRL project. this idea was originally suggested in [Hayward and Paul 83] but to our knowledge has not been followed up

5.2 RCCL as a Multi-Robot Control Environment

A basic issue addressed by this work was the provision of a uniform programming environment for the CVaRL robots We can now control both the Puma and the Microbo using RCCL. but developing code on the Intel System 310 under RMX is definitely more painful than on the VAX under Unix 4 2Bsd Ideally, all code development should be done on the VAX. and the resulting executable files downloaded and executed on the Intel system Cross-development tools do exist. for example the "Amsterdam Compiler Kit" [Keizer 85]. which would allow this An associated issue is the communication link between the VAX and the Intel system The current solution is a 9600 baud serial link This is reasonable for transferring source files. but would be insufficient to transfer an RCCL executable image. which typically exceeds 100.000 bytes of code and data[†]

Extending the existing CVaRL Ethernet to the System 310 is possible using the Intel iSXM-552 Ethernet Communication board and iNA-960 Transport Software This package supports the ISO Transport protocol to level 4 (Transport Layer) and is available for the RMX-86/System 310 environment This would provide a 10 Mbit/second transfer rate. and would allow RMX *tasks* to communicate with Unix *processes*. for example via RMX *mailboxes* at one end and Unix *sockets* at the other The requisite hardware has been acquired and a project is currently underway at CVaRL to implement the Ethernet link It is noted that the iNA-960 Network Management layer also includes a "boot server". this would permit downloading and booting processors over the Ethernet

Given the network support described above. multi-robot programs could be written as sets of processes which communicated over the network for purposes of synchronization The practicality of this scheme has already been demonstrated by the multi-robot demo described in Chapter 4 Ultimately. it may be desirable to be able to control multiple manipulators from a single RCCL application program. this implies modifying RCCL function calls to take a *robot* identifier as an additional parameter (This is the solution used in

[†] Because of the use of C include files *all* RCCL library functions whether or not actually called are included in the linked RCCL executable module

the AL system [Mujtaba and Goldman 79], and is considered in [Lloyd 85]) This is an important area for further research.

5.3 Execution Speed Improvements

As seen in Chapter 4, the 80286/80287 CPU's major computational bottleneck is not trigonometric calculations (sine, cosine, atan2) but floating point multiplication and division: CVaRL's VAX-750 with its Floating Point Accelerator does a 32-bit floating point multiply in 2 8 microseconds versus the Intel 80287's 19.5 microseconds. When RCCL is operating in Cartesian mode, the motion equation is re-solved every sample period, depending on the makeup of the equation and the types of transforms involved, the number of multiplications varies Constant transforms are pre-multiplied together wherever possible to optimize run-time execution Typically, transform multiplications are a major component of the computational overhead, taking about as much time (on the Intel hardware) as the combined forward and inverse kinematics

One way of increasing the performance of the microprocessor-based implementation would be simply to upgrade the 80287 co-processor from the current 6 MHz part to 8 MHz, and similarly replace the 5 MHz 80286 with a 10MHz version. This should result in a 33% speed increase in floating point math performance, and a larger overall increase in execution speed . At this CPU clock rate, performance would be limited by memory access time

A more dramatic improvement would involve adding an array processor to the System 310, several are currently available for the Multibus [Numerix 85] This should result in an order of magnitude increase in the performance of transform multiplications and inversions

From the results of Chapter 4, we see that the communication task (on the iSBC-86/30) currently is busy during about 8 milliseconds of every control cycle, while the RCCL trajectory generator may use 25 5 milliseconds There is a major imbalance here, and it seems that another option would be to move the forward and inverse kinematics from the iSBC-286/10 to the iSBC-86/30 This would involve changing the how and chg data structures to include the T_6 transform In this way the RCCL trajectory generator need not compute the forward kinematics at the beginning of each control cycle, it could simply copy the current value of T_6 from the how structure in shared memory Similarly, after a target T_6 has been computed, this can be copied into the chg structure and the

appropriate flag set. The iSBC 86/30 communication task (now a "communication and kinematics" task) would then execute the inverse kinematic solution to find the appropriate joint variables. This scheme should redress the present execution-time imbalance between the two processors, and allow RCCL to use a faster sampling rate.

It is recognized that in "joint" mode the trajectory generator, at the beginning of a path segment, would still have to perform the inverse kinematics to compute the joint variables for the endpoint of the segment. This is because intermediate setpoints in this mode are interpolated between joint values computed for the start and end points of the segment.

5.4 Upgrading The Microbo Joint Controllers

As we have seen, the Microbo joint controllers' algorithm is inappropriate for the RCCL approach. It needs to be re-designed so as to provide a better match with RTC requirements. We reiterate that the basic change would be to an algorithm that expects to be driven *synchronously* at the RTC sample rate with position setpoints. A PID-based control scheme, as implemented by Unimation for the PUMA 260, would seem to be more appropriate, although this is an interesting area for research.

An improvement to be made during any future re-implementation would be to eliminate the vendor's "work area" scheme entirely. This could be done simply by increasing the word size used to store encoder values from the current 16 bits to 24 bits (18 bits are actually needed).

A current project at CVaRL involves modifying one of the joint controllers by replacing some of its ROM with RAM, and replacing the current firmware with a simple monitor that allows this RAM to be downloaded via the Multibus. In this way new control algorithms may be compiled on the VAX using 8085 cross-development tools, then downloaded and executed at the joint level.

5.5 Conclusions

This thesis has described the design and successful implementation of the Robot Control C Library, RCCL, and its underlying Real Time Control system, RTC, for the Microbo Ecureuil industrial robot, using a multi-microprocessor system. The design was based on a

previous implementation of RCCL/RTC for the Puma 260 robot and a VAX/UNIX environment. The task included designing and implementing an interface between the Microbo s joint controllers and a Multibus system, the Intel System 310; solving the robot's forward and inverse kinematics, designing a multi-microprocessor architecture, based on the Intel 80286/87 and 8086/87, which had the processing capability to support RCCL's computational load, re-designing the RTC layer to run under Intel's iRMX-86 real-time multi-tasking operating system, and creating a usable development environment for RCCL users. It was shown that this system is flexible and expandable, and opens the way to the implementation of a multi-robot programming and control environment for the McGill University's Computer Vision and Robotics Laboratory.

References

[Alami 84] Rachid Alami. "NNS. A Lisp-Based Environment for the Integration and Operation of Complex Robotics Systems". *Proceedings of the IEEE Conference on Robotics*. 1984. pp. 342 - 348

[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).

[Cincinnati 80] Cincinnati Milacron. *"Operation Manual for Cincinnati Milacron T3 Industrial Robot"* Publication No. 1-1R-79149. Cincinnati. 1980.

[Cole and Sundman 85] Clement Cole and John Sundman. "Unix in Real Time". *Unix Review*. November 1985. pp 61 - 67. 101 - 103.

[CVaRL 85] Computer Vision and Robotics Laboratory Progress Report. Technical Report no TR-85-10R Dept of Electrical Engineering. McGill University. Montreal. Canada. October 1985

[Dario, et al. 83] P Dario. C Domenici. R Bardelli. D De Rossi. P C Pinotti. "Piezoelectric Polymers New Sensor Materials for Robotic Applications". *Proceedings. 13th International Symposium on Industrial Robots and Robotics*. Chicago. Illinois. April 17-21. 1983. Volume 2. pp 14 34 - 14.49

[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

[Dupont 84] Yves Dupont. "Intuitive Robot Language. Version 3 2 (A Revised Manual)". Computer Vision and Robotics Laboratory. Dept of Electrical Engineering. McGill University. Montréal. Canada. August. 1984

[Evans, et al. 76] R C Evans. et al . "Software System for a Computer Controlled Manipulator". *IBM Research Report RC 6210*. IBM T J Watson Research Center. Yorktown Heights. N Y . September 1976

[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

[Freedman, et al. 85] Paul Freedman. Gregory Carayannis. David Gauthier. and Alfred Malowany. "A Session Layer Design for a Distributed Robotics Environment".

Proceedings, COMPINT 85 - Computer Aided Technologies, Montreal, Canada, September 8 - 12, 1985, pp 459 - 465

[Grossman,77] D D Grossman, "Programming of a Computer Controlled Industrial Manipulator by Guiding Through the Motions", *IBM Research Report RC 6393*, IBM T J Watson Research Center, Yorktown Heights, N Y , March 1977

[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 86] Vincent Hayward, Private Communication, Computer Vision and Robotics Laboratory, Dept of Electrical Engineering, McGill University, Montréal, Canada, January 1986

[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

[Hinnant 84] David F Hinnant, "Benchmarking Unix Systems", *Byte Magazine*, August 1984, pp 132 - 135, 400 - 409.

[Isbister 84] D J Isbister, "A Robotic System for Modification of Printed Circuit Boards", *Canadian Conference on Industrial Computer Systems*, Ottawa, Ontario, May 22 - 24, 1984, p 83-1

[Intel 84] Intel Corporation, *iRMX86 Introduction and Operator's Reference Manual* Order number 146194-001, Santa Clara, California 95051, 1984

[Keizer 85] Ed Keizer, "ack - Amsterdam Compiler Kit", Wiskundig Seminarium, Vrije Universiteit, Amsterdam, Nederlands, *Unix manual entry* (dated 1983 - 1985), Available from Unipress Software 2025 Lincoln Highway, N J 08817

[Kernighan 84] Brian Kernighan "The Unix System and Software Reusability", *IEEE Transactions on Software Engineering*, September 1984, pp 513 - 518 (Vol SE-10, No 5)

[Kernighan and Ritchie 78] Brian K Kernighan and Dennis Ritchie, *The C Programming Language* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1978

[Lee 82] C S G Lee, "Robot Arm Kinematics, Dynamics, and Control" *Computer*, December 1982, pp 62 - 80 (Vol 15, No 12)

[Lavin and Lieberman 82] M A Lavin and L Lieberman, "AML/V, An Industrial Machine Vision Programming System", *The International Journal of Robotics Research*, Fall 1982, pp 42 - 56 (Vol 1 No 3)

[Lieberman and Wesley 77] L Lieberman and M Wesley. "AUTOPASS. An Automated Programming System for Computer Controlled Mechanical Assembly" *IBM Journal of Research and Development*. July 1977

[Lloyd 85] John Lloyd. "Implementation of a Robot Control Development Environment" (M Eng Thesis) Dept of Electrical Engineering. McGill University. Montreal. Canada. (submitted December 1985)

[Michaud 85] Christian Michaud. "Mroutines c Using the Microbo Robot With Style". *Technical Report TR-85-3R*. Computer Vision and Robotics Laboratory. Dept of Electrical Engineering. McGill University. Montréal. Québec. Canada. January 1985

[Michaud, et al. 85] C Michaud. A Malowany. M Levine. "Electronic Assembly by Robots". Proceedings of Graphics Interface '85. Montréal. Québec. Canada pp 391-397. May 27-31. 1985

[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

[Numerix 85] Advertisement for array processors. back cover of IEEE Computer. December. 1985 ISSN 0018-9162. (Vol 18. No 12)

[Paul 77] Richard P Paul. "WAVE - A Model Based Language for Manipulator Control". *The Industrial Robot*. March 1977. pp 10 - 17 (Vol 4 No 1)

[Paul 81] Richard P Paul. *Robot Manipulators Mathematics. Programming. and Control* MIT Press Cambridge. Mass . 1981

[Popplestone, et al. 78] R Popplestone. A Ambler. and I Velos. "RAPT A Language for Describing Assemblies". *The Industrial Robot*. September 1978 pp 131 - 137

[Schwan, et al. 85] Karşten 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

[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

[Studenny and Bélanger 84] J Studenny, P Bélanger "Robot Manipulator Control by Acceleration Feedback", Proceedings of the 23rd IEEE Conference on Decision and Control, Las Vegas, Nevada December 12-14, pp 1070-1072

[Takase 81] K Takase, R Paul and E Berg, "A Structured Approach to Robot Programming and Teaching", *Transactions on Systems, Man and Machine Cybernetics*, April 1981 pp 274 - 289

[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)

[Thompson 78] K Thompson, "Unix Implementation", *Bell System Technical Journal*, July/August 1978, pp 1931 - 1946 (Vol 57, No. 6)

[Unimation 82] Unimation Inc., *User's Guide to VAL The Unimation Robot Programming and Control System* Shelter Rock Lane, Danbury, Conn 06810, March 1982

[Unimation 83] Unimation Inc., *User's Guide to VAL-II. The Unimation Robot Programming and Control System*, Shelter Rock Lane, Danbury, Conn 06810, 1983.

[Volz, et al. 84] Richard Volz, Trevor Mudge, and David Gal, "Using Ada as a Programming Language for Robot-Based Manufacturing Cells", *IEEE Trans on Systems, Man, and Cybernetics* November/December 1984 (Vol SMC-14, No 6)

[Wang 74] S Wang, "ALFA (A Language for Automation)" *Proceedings, Milwaukee Symposium on Automatic Control*, Milwaukee, Wisconsin, 1974, pp 235 - 239

Appendix A. Creating and Running Microbo RCCL and RTC Programs

This appendix is intended as brief, informal guide for those wishing to create and execute RCCL or RTC programs for the Microbo Robot

We presuppose that the reader has a working knowledge of the C programming language. For more complete information on the user interface to the iRMX-86 operating system, the reader should refer to "iRMX-86 Introduction and Operator's Reference Manual" Intel publication number 146194-001 For more information about the various compilers linkers, editors, etc the reader is referred to the appropriate Intel manuals

The hardware referred to in this Appendix consists of the Microbo Ecureuil robot and its RCU control unit, connected via a bus interface to an Intel System 286/310 running the iRMX-86 operating system The System 310 has, in addition to the standard 286-10 CPU card, an additional processor in the form of an iSBC 86/30 In addition to the standard 5 1/4 inch floppy and 20Mbyte winchester disk, there is an iSBC 208 floppy controller connected to an 8-inch floppy disk drive The latter is used for booting the 86/30 CPU

A.1 Notation

This font shows material typed by you or the system. The symbol "^" is used to denote the CTRL key Terminate each command input with the RETURN key.

A.2 Starting Up The RMX System

1 Connect a terminal to the secondary RS232 port (J21), and then turn the power on

2 As soon as you start to see a few "*"s , type U. This will trigger a self-check As soon as you see the system prompt "INPUT I" type ^C

3 Move the terminal to the primary port (J20) (unless there is another terminal available) and wait for the RMX system to boot up You must type in the date and time (after the 'DATE." and "TIME· "prompts) so that all your work with files will be properly timestamped Here is the date format dd month year, for example 4 dec 85

4 Get out the 8 inch floppy disk marked "86/30 boot disk" from the plastic BASF storage case in the tool cupboard (The cupboard should normally be locked, so you might

need to find someone with a key)

5 Turn on the upper 8 inch external disk drive (which will actually turn on the lower one too), and then insert the boot disk into the lower drive

6 Move the terminal back to the secondary port, and type `b :af0:8630` Now wait for a system message like `xxxx = robot task token` Remove the boot disk and then power off the disk drives

7 Disconnect the terminal, and connect the Microbo to the same secondary port of the RMX system, through an RS-232 inverter

8 Power up the Microbo from the front panel of the robot controller

9 Type `calib` to calibrate the Microbo Note that the "calibration" position is *not* the rccl "park" position

A.3 How To Transfer Files To/From The RMX System

WARNING The RMX system only allocates 14 characters for a filename plus its extension. Take note of your naming convention, since `foo c` (5 characters) becomes `foo.obj` (7 characters) when compiled ("translated", in Intelspeak)

1 Set up the UNIX aliases for `rmxcom`, `rmxput`, and `rmxget` in your `.cshrc` file

2. Disconnect the primary terminal (@J20) and connect the RMX system to the TTY02 port on CURLY (which has no getty) You need to use an INVERTING cable to do this

3 Type `rmxcom` to establish communication with the RMX system You can check the link by typing, say, `dir` for a look at the directory

4 To SEND a file `foo.bar` to the RMX system

(a) Type `rmxput`

(b) Type `foo bar` after the prompt

(c) Type `RETURN` after the next prompt, unless you want to use a different filename on the RMX system The file will now be echoed at your terminal as the downloading takes place

5 To GET a file `foo bar` from the RMX system, just use `rmxget` and follow the prompts as for sending a file (see previous step).

A.4 Some Notes About The RMX System Editor "Aedit"

WARNING This editor, unlike EMACS, is riddled with "modes" It is menu-driven, and really not too hard to learn with a command line at the bottom of the screen

1 Start up the editor by typing `aedit filename` The default "macro file" (aedit mac) is currently a copy of "tvi924 mac", which means that the editor works for the Televideo 924 To use it with a Televideo 950, type `aedit filename mr(tv950.mac)`

2 Your very first command ought to be ^G to properly initialize the editor for the TVI 924 or 950 terminal types

3 Thanks to this author, the editor now resembles our ol' EMACS (albeit in a terribly superficial way!) Here are a few commands to get you started

^G = set tabs, indent, viewline, etc

^C = cancel the current command

^F = forward char

^B = backward char

^Xf = Jump forward word

^Xb = Jump backward word

^N = next line

^P = previous line

^D = delete char

^K = delete to end of line

^Y = undelete (a la yank from kill buffer)

^A = go to beginning of line

^E = end of line

^V = scroll 1 screen down

^X = macro execute

4 And here are a few true blue editor commands

i = enter "insert" mode, to add stuff after deletions

ESC = return to the command mode

q = enter "quit" mode From here, type u to update your file on disk, l to look at (visit) a new file, a to leave the editor.

5 For more details (denials?), see the Intelspeak "AEDIT" manual.

A.5 How To Backup Your Files On Floppy Disk

WARNING A 5 1/4 inch floppy diskette only holds 360 Kbytes Before you begin, delete all unnecessary files such as BAK, LST, MP1, OBJ, and ESPECIALLY the executable images

1 Acquire (beg, borrow) a 5 1/4 inch soft-sectored unformated double-sided double-density 48 tpi floppy diskette

2 Insert the diskette into the drive on the RMX system

3 Type `attachdevice wmfdx0 as .fd0:` If this fails, pull out the diskette, re-insert it, and try again.

4 Type `format fd0.` to format the diskette This will take a few moments

5 Type `backup /foo/bar over .fd0:` to save all files and directories with the root `/foo/bar`

6 When the system asks the question about "Mount Backup Volume" (Intelspeak) just type y (you agree)

7 When you get the "Backup Complete" message, type `detachdevice fd0:` and remove the diskette

A.6 How To Compile A Program Called foo.c

1 First prepare a cc csd file Note that you must use the Intel "LARGE" compilation model, and that the C interface libraries used in the link step are currently only available for this option The standard submit file looks like this

```
c86 %0.c large verbose include(/inc/) include (/rcc1/h/) to %0 obj
```

2 To actually compile ("translate", in Intelspeak) the file type `submit cc(foo)` This will create a file called `foo obj` in your local directory (Unfortunately, RMX does not provide something like the UNIX `makefile`, so it's up to YOU to keep track of your own changes!

A.7 How To Link All Your Programs And Create An Executable RCCL Image

WARNING Linking under RMX is a slow slow process. Be patient!

1 First prepare a file to direct the linking called, say, `lnk.csd` If your programs were called `foo1 obj`, `foo2 obj`, and `foo3 obj`, and you wanted the executable image to be called `foo`, then your file would look like this

```
link 86 &
foo1 obj, &
foo2.obj, &
foo3 obj, &
/lib/cc86/lqmain obj, &
/rccl/lib/rccl.lib, &
/rccl/lib/rtc lib, &
/rccl/lib/microbo.lib, &
/rccl/lib/math.lib, &
/lib/cc86/ios.lib, &
/lib/cc86/nucleus lib, &
/lib/cc86/lclib lib, &
/rmx86/lib/large.lib, &
/rmx86/lib/rpifl.lib, &
/rmx86/lib/epifl.lib, &
/lib/ndp87/8087 lib &
to foo &
map bind segsize(stack(+3000h), memory(8000h)) &
mempool(8000h,0FF000h)
```

Note that the order of items in the link list is important; if this is violated, `unresolved variables` errors will result

2 To actually perform the link step, type `submit lnk` (assuming that your link submit file is called `lnk csd`)

A.8 How To Run An Executable Image Called `foo`

1 Bring the panic button of the microbo near the terminal

2 Power on the microbo by pressing the green ("I") button on the front of the orange RCU controller The green light should come on

3 Type `calib` You ought to see the yellow lamp on the RCU controller box light up, and the program will echo `calibration initiated`, and then `calibration successful`. If there is a timeout problem, just try typing `calib` again. If this persists, check all hoses, electrical paraphernalia, etc which might be hindering the movement of some of the robot joints If the yellow light does not come on, make sure that the Microbo's serial line is connected to the RMX box via the inverter.

4 Type `foo` (and cross your fingers!)

A.9 Notes on Configuring the iRMX-86 Operating System for RTC

The iRMX-86 operating system must be reconfigured in order to support the RTC system. The Intel Interactive Configuration Utility (ICU) uses *definition* files to store system configurations; the appropriate files are `/rccl/def/rmx286.def` and `/rccl/def/8630.def`. The first describes the configuration for the 286/10 processor, and the second is for the 86/30 processor. The user is referred to the "iRMX-86 Installation and Configuration Guide", Intel publication number 146548-001, for information on the ICU.

The 286/10 configuration is different from the standard configuration in that memory between 3000:0 and 3100:0 is reserved for an I/O user job, called `createtask`. [This job consists of a single task that is used indirectly to create the high-priority RTC interrupt task. The task waits for an object to be catalogued in the object directory of the root job under the name "task". It takes this to be a data structure (segment) containing information about an RMX task, and creates the task.]

The 86/30 configuration is a simple system consisting of just the Nucleus and a single user job. This user job contains the robot communication task. The appropriate source files are in the `/rccl/mic/` directory, the communication task is `/rcc/mic/robot task.c`.

Appendix B. Microbo Joint Controller Command Protocol

B.1 General

The reader should note that there were several mistakes in the documentation supplied by the vendor The information presented here has all been verified experimentally However, there is much that remains obscure about the exact operation of the joint controllers

The (up to 8) joint processors communicate with the master processor via three eight-bit registers. These are memory-mapped onto the C1 bus The base address of the registers is jumper-selectable on each joint processor card The manufacturer's settings have not been changed, and are currently set so that joint 0 base address is 5800H, joint 1 is 5802H, etc

- the *data* register is read/write, located at the base address,
- the *command* register is write only, located at the base address +1
- the *status* register is read only, also located at base address+1

It should be noted the command and status registers are in fact the same. We maintain the distinction simply to remain consistent with the documentation provided by the vendor Thus the communication registers for each joint take up 2 consecutive locations in the address space of the bus, and the 8 joints then take up 16 consecutive locations

These addresses are mapped into the I/O space of the System 310 via the Multibus Adapter Card. The mapping is also switch selectable, but is also currently set to 5800H. Thus reading and writing the registers of the various joint controllers may be done using "`IN reg, 58xxH`" or "`OUT 58xxH, reg`" if writing in Intel ASM86 assembler language, for example.

B.2 Communication Protocol

All communication between the master and the joint micros must observe the handshaking defined by two bits in the status register

bit 0: OBF (output buffer full) indicates that the joint micro's output buffer is full and is ready to be read by the master.

bit 1 IBE (input buffer full) indicates that the joint micro's input buffer is full and may *not* be written by the master (slave busy)

Communication consists of the master sending a byte to the command register, sometimes followed by reading or writing one or more bytes in the data register The handshaking must be observed for every command send and every data byte read or written.

Note that the joint micros are interrupted by commands, they do not return to the servo control function until a command has been completely executed It is therefore important not to interrupt the joint micros too often (i e more than every 3 msec) and to execute the communication as quickly as possible

Upon initialization of the joint controllers, all *motor parameters*, including those associated with velocity and acceleration, are set to the vendor's default values However, the actual velocity and acceleration of robot motions are contolled via another set of commands which interact with certain Microbot registers directly. The velocity increments are 1/32 radians/second: the default value is 32, hence 1 rad/sec. The acceleration increments are 1/4 radians/second/second, the default value is 1, hence 1/4 rad/sec/sec.

B.3 Command Summary

```
COMMAND = 00H                    GET JOINT STATUS
read DATA  bit 0  ok- in position or within zone of tolerance
           bit 1  initialized
           bit 2  fault during motion or initialization
           bit 5  phase error (simultaneous transition on both encoders)
           bit 6  encoder error
           bit 7  user-specified tolerance mode ''on''
COMMAND = 02H                    GET CURRENT TARGET POSITION
read    DATA   low byte of word containing current target position
read    DATA   high byte of word
COMMAND = 04H                    GET CURRENT POSITION
read    DATA   low byte of word containing current position
read    DATA   high byte of word
COMMAND = 06H                    GET CURRENT ACCELERATION
read    DATA   byte containing acceleration low byte
read    DATA   byte containing acceleration high byte
COMMAND = 08H                    GET CURRENT VELCOCITY
read    DATA   byte containing velocity low byte
read    DATA   byte containing velocity high byte
COMMAND = 0AH                    GET CURRENT TOLERANCE
read    DATA   byte containing tolerance low byte
read    DATA   byte containing tolerance high byte
COMMAND = 0CH                    SET TARGET POSITION
write   DATA   low byte of word containing target position
write   DATA   high byte of word containing target position
COMMAND = 0EH                    SET MOTOR PARAMETERS
write   DATA   byte 1            proportional gain value
write   DATA   byte 2            initialization speed setting
```

```
write    DATA    byte 3          sensor error tolerance
write    DATA    byte 4          size of static regulation zone
write    DATA    byte 5          maximum speed/256
write    DATA    byte 6          dynamic integration constant
write    DATA    byte 7          static integration constant
write    DATA    byte 8          static speed setting
write    DATA    byte 9          motor resistance/thermal capacity
write    DATA    byte 10         1.0/thermal resistance
write    DATA    byte 11         maximum temperature
write    DATA    byte 12         gravity compensation parameter
write    DATA    byte 13         inertial parameter
write    DATA    byte 14         viscous friction parameter
write    DATA    byte 15         dry friction parameter
write    DATA    byte 16         maximum acceleration/4
COMMAND = 10H                    SET MAXIMUM ACCELERATION
write    DATA    low byte of acceleration value
write    DATA    high byte of acceleration value
COMMAND = 12H                    SET MAXIMUM SPEED
write    DATA    low byte of speed value
write    DATA    high byte of speed value
COMMAND = 14H                    SET TOLERANCE VALUE
write    DATA    maximum difference between target position and
                 current position within which the ''ok'' bit will be
                 set to ''1'', low order byte
write    DATA    high order byte
COMMAND = 16H                    SET TOLERANCE MODE ON
                 tolerance will now be whatever is specified by the
                 user via command 14H (see above)
COMMAND = 18H                    SET TOLERANCE MODE OFF
                 tolerance will be +/- 1 encoder value
COMMAND = 1AH                    INITIATE STAND-BY MODE
                  The joint micro halts after this command is executed.
                  this is the default state after the system is powered
                  on.  To exit from this state, the joint micro must
                  either receive an initialization command, or if
                  initialized, receive a position command.
COMMAND = 1CH                    INITIALIZE
write    DATA    low byte of ''initialize'' position
write    DATA    high byte of ''initialize'' position
                 The joint micro will move the joint to the mechanical
                 initial position, then assign to this position the
                 ''initialize'' value sent via the data register.
COMMAND = 1EH                    STOP (SET TARGET TO CURRENT POSITION)
COMMAND = 22H                    SET POINTER FOR ACCESS TO JOINT MICRO RAM
write    DATA    low byte of an address in the ram of the joint micro.
write    DATA    high byte of the address.  The default value of this
                 pointer is the location of the software version
                 number.
COMMAND = 24H                    WRITE BYTE IN JOINT MICRO RAM
read     DATA    byte will contain the value of the byte pointed to by
                 the pointer set up by command 22H. (see above)
COMMAND = 26H                    READ BYTE IN JOINT MICRO RAM
write    DATA    byte should contain a value to be written to the joint
                 micro's ram in the location pointed to by the pointer
                 set up by command 22H.  (see above)
COMMAND = 28H                    CHANGE WORK AREA
                 requests a change of work area
COMMAND = 2AH                    READ SPEED FACTOR
read     DATA    byte will contain the current speed factor.
```

```
COMMAND = 2CH                    WRITE SPEED FACTOR
write   DATA    byte should contain the desired speed factor.
COMMAND = 2EH                    READ WORD IN JOINT MICRO RAM
read    DATA    byte will contain the value of the low byte of the word
                pointed to by the pointer set up by command 22H.
                (see above)
read    DATA    byte will contain the value of the high byte of the word
                pointed to by the pointer set up by command 22H.
COMMAND = 30H                    WRITE WORD IN JOINT MICRO RAM
write   DATA    byte should contain a value to be written to the joint
                micro's ram in the location pointed to by the pointer
                set up by command 22H.  (see above)
write   DATA    byte should contain a value to be written to the next
                location
```

B.4 Writing Programs to Talk to the Joint Controllers

The recommended way to do this is to use the RTC system, however, if the chg structure does not provide the support required, the following C-callable functions are available, they will run on any 86-family Multibus master if the Microbo bus adapter card is plugged into the bus

These functions are in the RMX library "/rccl/lib/microbo.lib", the error definition macros are in "/rccl/h/microbo.h" The "udi.h" header is the standard include file for RMX C programs, it defines things like byte and word

To send a command byte to a Microbo joint controller

Usage:

```
#include <udi.h>
int put_cmd( joint, command );
int joint;
byte command;
```

Inputs.

`int joint` The joint number (1 is the first joint).

`int command` The command byte.

Outputs function returns 0 if everything ok, otherwise:

L_TIMEOUT_CMD: timed out attempting to send command to joint.

L_INVAL_JNT_CMD: invalid joint number.

To send a data byte to a Microbo joint controller

Usage.

```
#include <udi.h>
int put_dat( joint, data );
int joint;
byte data;
```

Inputs.

int joint. The joint number (1 is the first joint).

int data. The data byte to send.

Outputs. function returns 0 if everything ok. otherwise

L_TIMEOUT_PUT timed out attempting to send data to joint.

To read a data byte from a Microbo joint controller

Usage:

```
#include <udi.h>
int get_dat( joint, data );
int joint;
byte *data;
```

Inputs:

int joint The joint number (1 is the first joint).

byte *data Pointer to a one byte buffer to put data in.

Outputs: function returns 0 if everything ok. otherwise.

L_TIMEOUT_GET timed out waiting for data from joint.

Example

The following example illustrates the usage of the above commands.

```
/*----------McGill-Computer-Vision-and-Robotics-Laboratory----------
*
*   NAME:       set_acc.c
*
*   FUNCTION:   procedure to send an acceleration to a microbo joint
*               controller.
*               returns 0 if no errors; otherwise passes errors from
*               put_cmd() and put_dat();
*
*   USAGE:      int set_acc( joint, acc );
*               int joint;
*               word acc;
*
*   AUTHOR:     don kossman
*
*   DATE:       25 aug 85
*
*---------McGill-Computer-Vision-and-Robotics-Laboratory---------*/
#include <udi.h>
#include "microbo.h"
int set_acc(jnt, acc)
int jnt;
word acc;
 {
   byte *acc_lo,
        *acc_hi;
   int  sts;
   acc_lo = &acc;
   acc_hi = acc_lo+1;
   if (sts = put_cmd(jnt, REG_SET_ACC)) /* the "acceleration" cmd */
    { return( sts );
    }
   sts = put_dat(jnt, *acc_lo);         /* send acceleration byte 1 */
   sts |= put_dat(jnt, *acc_hi);        /* send acceleration byte 2 */
   return( sts );                       /* completion status code */
}
```