To the Graduate Council:

I am submitting herewith a thesis written by Melissa R. Wilson entitled "A Multi-Robot Programming and Control System." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Suzanne Lenhart

Suzanne Lenhart, Major Professor

We have read this thesis
and recommend its acceptance:

D J Smyth

Michael G Thomason

Accepted for the Council:

Lewminkel

Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Melissa R. Wilson

Date Feb. 27, 1987

A MULTI-ROBOT PROGRAMMING AND CONTROL SYSTEM

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Melissa R. Wilson

March 1987

In memory of my mother,

Mary Eleanor Piper Rogers.

ACKNOWLEDGMENTS

I would like to thank my advisor, Richard Blake, for his guidance and for the opportunity to work in robotics. With equal appreciation, I wish to thank Suzanne Lenhart for her continued support and constructive comments. I want to thank the members of my committee, David Straight and Michael Thomason. Also, I wish to thank Eric Grannum for his interest and comments.

My friend, Christine Deane, has offered friendship and support without which this thesis would never have been accomplished. And, of course, my daughter Maggie has been so patience throughout this experience. Also, Suzie Witenbarger was extremely helpful with the mysteries of LaTeX.

ABSTRACT

The goal of this thesis was to simultaneously control in real-time two robot arms sharing a common workspace. To support this goal, it was first necessary to develop a robot programming and control system. Thus, many aspects of robotics were explored including robot programming, kinematics, path control, collision detection, and collision avoidance. The goal was partially achieved in that the arms share a common workspace without colliding but the path computation does not run in an acceptable time frame for real-time usage.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Overview

The goal of this thesis was to develop a robot control system in which two robots share a common workspace. The problem of developing a robot control system necessarily includes many fields of robotics research. Each of these fields has generated much interest and work in its own right. The major areas of research may be categorized as follows:

1. Robot programming languages. This includes the development of a high level language which allows the robot programmer to write robot programs off-line using a familiar reference point such as the Cartesian coordinate system rather than the joint coordinate system. The joint coordinate system is that space described by the joint angles which are those angles formed at the link junctures.

2. Kinematics and Dynamics. This includes the study of force and motion with respect to robot manipulators. Kinematics, or motion, addresses the relationship between the Cartesian coordinate system and the joint coordinate system. Dynamics includes the development of force equations for the robot control system.

3. Path control. Path control is the process of giving the robot a path, or trajectory, to follow and keeping the end effector as close as possible to the given path.

4. Path Planning and Collision Avoidance. This involves the generation of paths which meet prespecified criteria while avoiding collisions with other robots and with objects in the work environment. There are several different methods of path specification.

5. Sensory Feedback. This is the task of making the robot responsive to its environment through various sensory devices. Although frequently treated separately, vision and pattern recognition may be included in this category.

Because of limitations of the robots used, some aspects normally included in a robot control system were not included in the control system written for this thesis. The succeeding chapters address those fields of robotics research that were explored in this thesis. Chapter 2 describes environment and task specification in the robot control system which I developed. These facilities allow for the development of simplistic robot programs. Chapter 3 deals with the kinematics problem as it relates to the robots used in this control system. The kinematic problem can be separated into two problems: (1) the direct (or forward) transformation which transforms joint coordinates to position and orientation of the end-effector in the base coordinate system and (2) the inverse (or reverse) transformation which transforms from position and orientation of the end-effector to joint coordinates. Chapter 4 discusses the robot joint controller portion of the robot control system. This controller drives and monitors the robots providing path control for multiple robot arms. Chapter 5 describes the collision avoidance algorithm developed and implemented in this control system. The goal of the algorithm is to have both arms reach their respective destinations without collision. Chapter 6 discusses results, experiments, and conclusions. Several suggestions for future research are proposed.

Hardware

The hardware configuration consists of two Mentor robot arms built by Cybernetics Applications, Inc. of Britain (Figure 1.1). The robot arms were interfaced to an IBM PC/AT. Each robot arm has five revolute joints labeled as axes in Figure 1.2. Each joint of the robot arm may be addressed independently. This gives each robot arm five degrees-of-freedom. Axis 3 and Axis 4 are combined to produce the pitch and twist of the wrist (Figure 1.3). Each arm has a sixth axis, Axis 5, which controls the open/close of the end effector (or gripper).

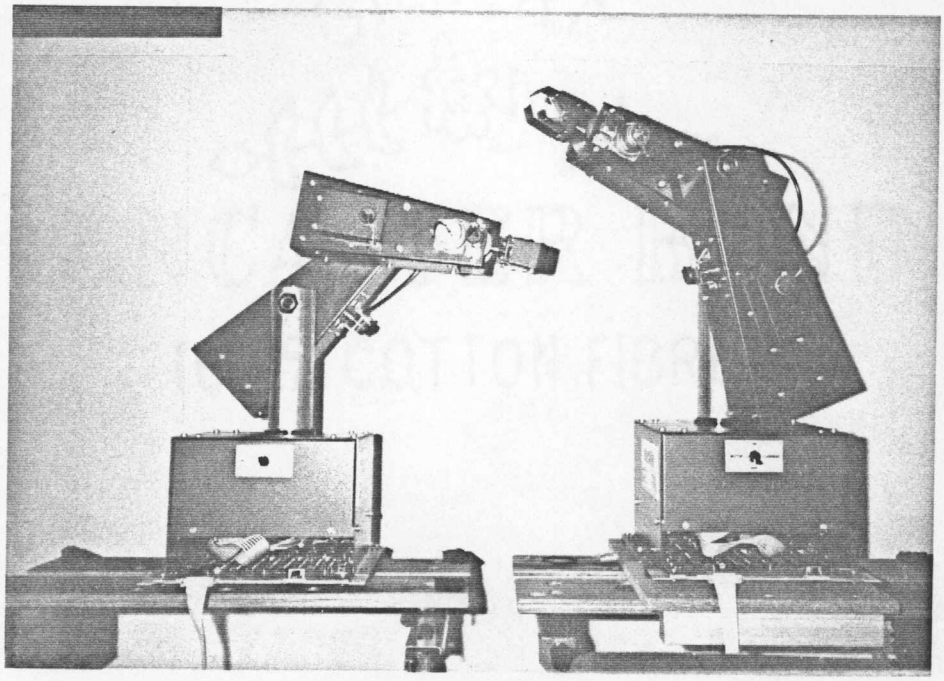


Figure 1.1: Mentor Robot Arms

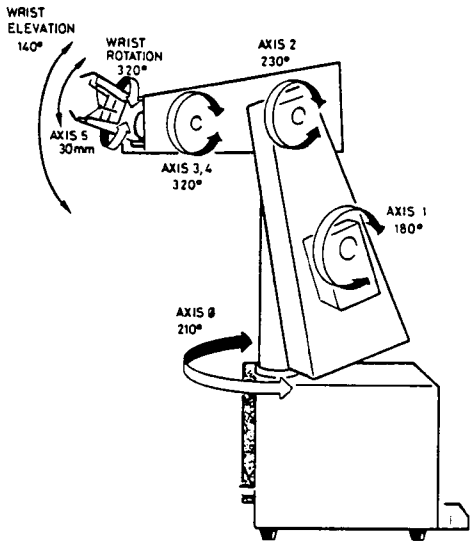
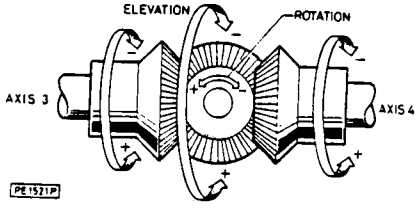


Figure 1.2: Axes of Mentor Robot
(From Mentor User's Guide)



AXIS 3	AXIS 4	E	R
0	−1	$-\frac{1}{2}$	$+\frac{1}{2}$
0	+1	$+\frac{1}{2}$	$-\frac{1}{2}$
−1	0	$-\frac{1}{2}$	$-\frac{1}{2}$
−1	−1	−1	0
−1	+1	0	−1
+1	0	$+\frac{1}{2}$	$+\frac{1}{2}$
+1	−1	0	+1
+1	+1	+1	0

Figure 1.3: Axes 3 & 4 of Mentor Robot
(From Mentor User's Guide)

The robot arms were placed side-by-side to achieve a common workspace (Figure 1.4). They were placed in such a way as to have the common workspace beyond the robot bases. This was done so that the base of the robot and Axis 0 would not interfere with a task. It was within this common workspace that collisions between the robots could occur, although with the introduction of objects being carried by the arm this common area may need to be extended.

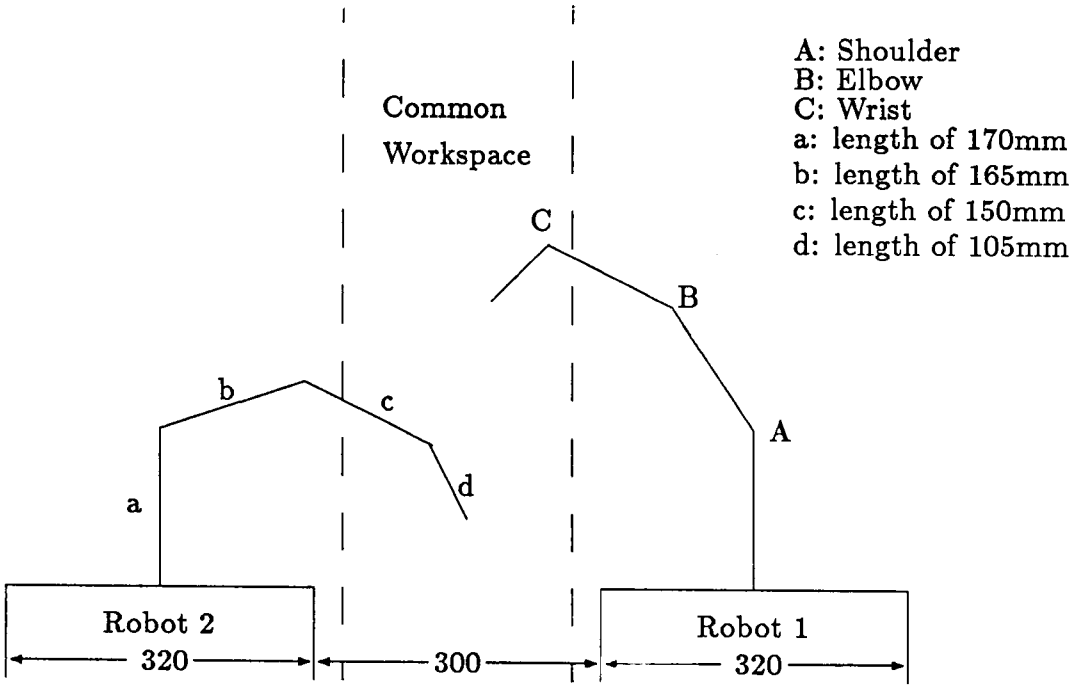


Figure 1.4: Common Workspace

Since the Mentor robot arm was developed for educational purposes, it is lacking some desirable capabilities. The Mentor does not have interrupt capabilities, does not have sensors of any kind, and does not allow for variations in the speed of movement. A detailed description of the Mentor robot arm is contained in Appendix A.

Software

The Mentor robot arm came with a primitive control system written in BASIC. This primitive system was much too limited to allow for a real study of problems in robotics. Thus, for this thesis, I wrote a robot program development

and control system using Turbo Pascal and Microsoft 8086 assembler. It is this control system which is described throughout this thesis. The control system gives the robot programmer the convenience of using Cartesian coordinates in robot program specification and allows control of more than one robot simultaneously. Robot programs are referred to as "tasks" throughout this thesis.

In keeping with currently accepted design principles, a hierarchical approach was used in the software design of my control system (Figure 1.5). This approach allows the modularization of the different functions of the robot program development and control system. The robot joint controller portion of my control system was written in Microsoft 8086 Assembler. It controls the actual sending of raw joint values to the robot arms and monitors the progress of the arms' movement. The robot program development and high-level control portion was written in Turbo Pascal. The major functions provided were (1) entry and viewing of robot tasks, (2) running a single task on a single arm, and (3) running two tasks on two arms simultaneously. The high-level control program and the robot joint controller communicate through a robot control block which is further described in Chapter 2. The robot-dependent routines were well isolated making changes in robot characteristics relatively easy.

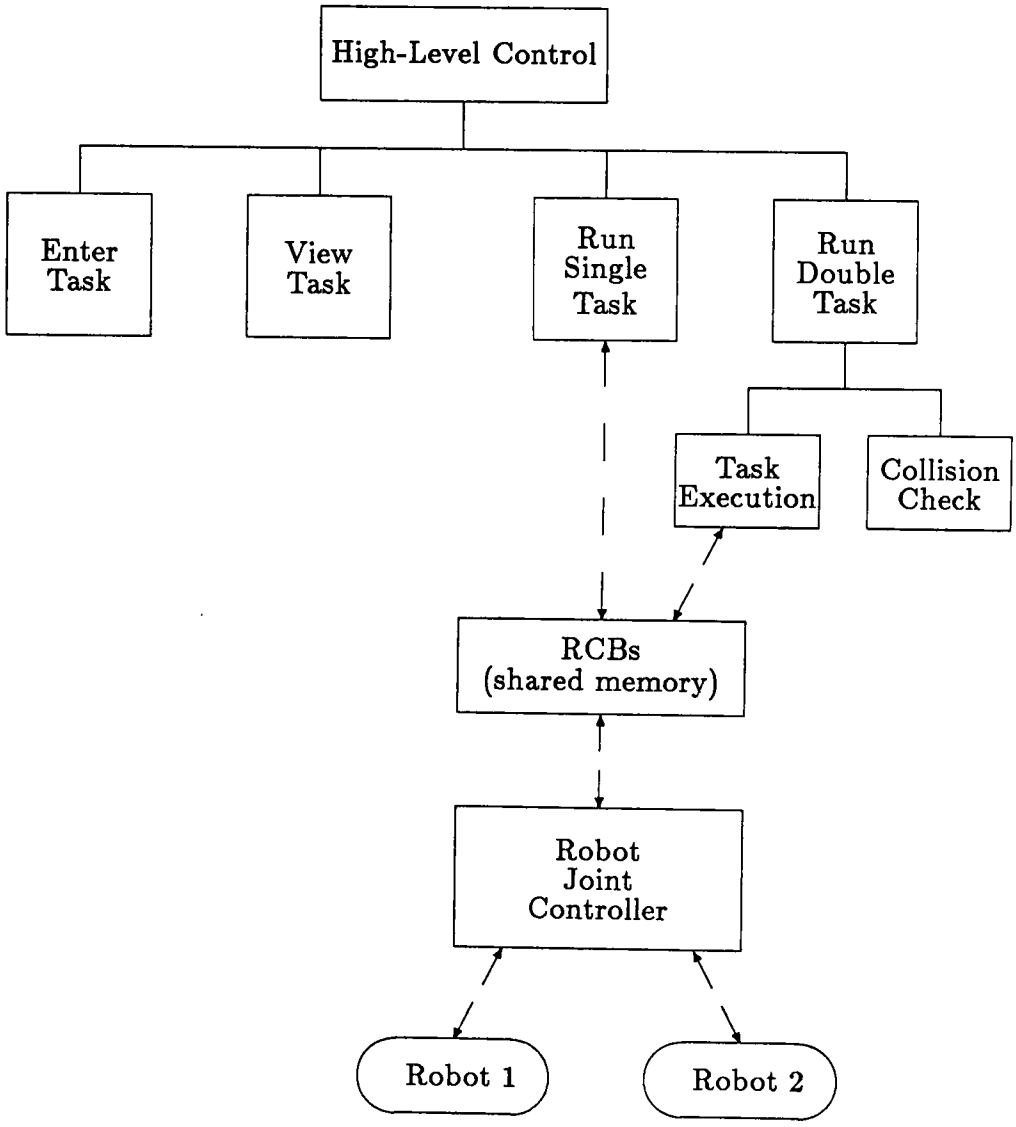


Figure 1.5: Hierarchical Software Chart

CHAPTER 2

ROBOT ENVIRONMENT AND TASK SPECIFICATION

Overview

Two necessary features of a robot control system are (1) the ability to describe the environment in which the robots work and (2) the ability to describe and execute a task to be done in this environment. This chapter discusses the facilities developed for environment specification, for task specification, and for task execution in the programming development and high-level control portion of the system. It also discusses the support facilities embedded in Turbo Pascal written for the above functions. Low-level control of the robots is described in Chapter 4.

Support Facilities

Data Types

Three new data types were used to provide descriptions in 3-space:

1. VECTOR: describes a point in 3-space using Cartesian coordinates.

2. REGION: describes a rectangular volume in space by specifying the eight vertices of the volume.

3. COLL_REGION: a collection of regions describing a robot arm (ie—the upper arm, the forearm, and the end effector each describe a region).

Functions

Several functions were developed for this system to be used in conjunction with the functions provided by Turbo PASCAL. These new functions are described below.

Transformation functions. These functions are useful in the forward and inverse transformations discussed in Chapter 3.

1. TRANS_ORIGIN: translates coordinate frame to a new origin.
2. REVER_ORIGIN: translates coordinate frame back to base origin.
3. X_ROT: rotates a point about the base x-axis.
4. Y_ROT: rotates a point about the base y-axis.
5. Z_ROT: rotates a point about the base z-axis.

Collision avoidance functions. These functions are useful in the collision avoidance software.

1. XY_INTERSECT: determines the intersection of two lines in the base xy-plane.
2. YZ_INTERSECT: determines the intersection of two lines in the base yz-plane.
3. XZ_INTERSECT: determines the intersection of two lines in the base xz-plane.

4. COLL_ZONE: determines if a robot arm enters the collision zone.

5. ADJUST_REGION: adjusts the base coordinate system of a second robot to the base system of first robot.

Environment Specification

To provide a flexible mechanism for describing the robot work environment three parameters were defined and two new data structures called descriptors were created.

Parameters

The parameters allow for changing the work environment specification and are defined at compile time.

1. NO_OF_ROBOTS: the number of robots in the system. Each robot is assigned a sequentially determined number.

2. NO_OF_OBJECTS: the number of objects in the workspace.

3. NO_OF_REGIONS: the breakdown of the robot arm into regions to be considered in collision avoidance.

Descriptors

Two new data structures provide a mechanism for describing a robot arm or an external object.

1. Robot descriptor: This data structure is used to identify and characterize a robot. It includes an ID number, the axis centers, the axis lengths, the limits of axis movement, the range of axis movement, the number of joint values to cover the axis range, a fudge factor which allows the start of axis movement at a predetermined point, and the regions defined by the arm as collision regions.

2. Object descriptor: This data structure has an object ID number and a region defining the volume of the object as a rectangle.

Robot Task Specification

Robot programming is the definition of a task to be done by a robot manipulator. Traditionally, this job has been done on-line by physically moving the robot manipulator through the desired sequence of steps either manually or by using some device designed for this purpose [15]. However, it is more efficient, less costly, and less dangerous to define a task off-line with the use of computers and robot programming languages.

The first robot programming languages were stand-alone systems designed for a specific robot or at best for a specific vendor's robots. Recently, there has been a growing interest in embedding robot programming in existing general-purpose, high-level languages. Blume and Jakob [3] have developed such a system with Pascal, named PASRO, and Paul [18] has described a method of embedding transformations in Pascal. This enables the language designer to take advantage of high-level contructs such as flexible data structures and sophisticated control structures.

However, it was not the intent of this thesis to develop a new robot programming language. Therefore, only the capability of defining a series of robot moves was provided. These moves, referred to here as steps, represent a task to be accomplished by the robot arm. A task then is a point-to-point (PTP) description of moves in which the robot programmer has no control over the path between two points. Path interpolation and path planning are discussed in Chapter 5.

The only step type provided is a move with or without a delay after the move is completed. Other actions, such as conditional execution, normally found in a programming language were not included. Conditional execution relies on sensory feedback which is unavailable on the Mentor. Also, gripper control is primitive. Currently, to open or close the gripper it is necessary to completely specify the current position of the robot arm changing only the gripper position. This method of gripper control is awkward and time-consuming.

For each step of the robot task, the robot programmer describes the position of the end effector in Cartesian coordinates and the wrist orientation of pitch and twist in sexagesimal degrees. The use of Cartesian coordinates and sexagesimal degrees is desirable as it is difficult to visualize the end effector's position when described in joint coordinates or joint values. (However, radians are used internally for computation and joint coordinates.)

Wrist orientation limits the number of solutions to the inverse transformation. To provide a unique solution, an arbitrary choice of the "elbow up" configuration is used. Each robot task is necessarily robot specific as the transformation is done at task definition time rather than run time. This ensures that all points in a robot task are reachable by the selected robot arm. The inverse transformation of the step specifications to the joint coordinates is discussed in Chapter 3.

Each step of a robot task is defined off-line by the robot programmer. The

inverse transformation is computed and joint coordinates are converted to joint values. All this information is stored in a step descriptor.

- Step descriptor: The step descriptor contains a robot ID number, a step number, a step status word, the position vector, the wrist orientation, the gripper position, the joint coordinates, the joint values, and the step delay time.

Robot Task Execution

The user may select to run either a single task on one robot or to run two tasks on two robots. Tasks are run by sending the destination joint values to the robot joint controller. Communication between the Pascal task execution procedure and the robot joint controller is achieved through a shared memory region. This region is further described in Chapter 4.

Dual tasks require more attention due to the collision avoidance problem. This is discussed in Chapter 5. Although destination points are computed at the time of task definition, collision checking is deferred to run time. This allows for the selection of tasks at run time. In other words, tasks are defined in terms of a single robot. But at run time any two tasks may be run simultaneously.

CHAPTER 3

ROBOT KINEMATICS

Overview

One of the oldest areas of research in robotics is that of motion, or kinematics. It is difficult for humans to visualize the position of a robot arm when expressed as joint coordinates. For this reason it is desirable to express the path of a robot arm in the Cartesian coordinate system. The base of the arm is the origin of the Cartesian coordinate system and is referred to as the base coordinate system. Coordinate transformations provide the mechanism for translating between the joint and base coordinate systems.

The problem of calculating Cartesian coordinates from joint coordinates is referred to as the kinematic problem, or forward transformation. The reverse problem of transforming from Cartesian coordinates to joint coordinates is the inverse kinematic problem, or inverse transformation. The forward transformation is straight-forward and always has a unique solution. The inverse transformation is the more interesting and the more difficult partly due to the facts that there may be be multiple solutions and that a solution may not be obtainable through normal means. For example, the inverse transformation involves the arctan function which is undefined at certain angles and the method fails to give a solution. These points of failure are called points of singularity.

Much research has been done on analytical solutions [5, 11, 14, 15, 18, 21] to the inverse kinematic problem in hopes of developing a general solution applicable to most classes of articulated robots. However, a geometrical solution was chosen

for this implementation. The geometrical approach provides insight into the physical movement of the robot arm that an analytical solution does not. It also provides a unique solution given some design criteria and the points of singularity are more readily identified. Singularities are discussed in greater detail in Sadre, Smith, and Cartwright [21].

The following sections describe the forward and inverse transformations used in my control system. These transformations are specific to the Mentor robot arm. This discussion refers to angle θ_0 through angle θ_4 corresponding to the five joint angles of the robot arm (Figure 3.1). It should be noted that in the actual software and in the Mentor User's Guide the angles are labeled Axis 0 to Axis 4.

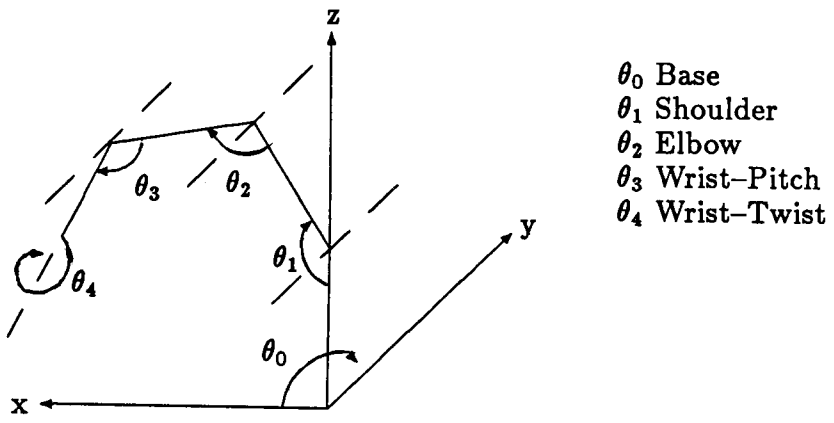


Figure 3.1: Joint Angles θ_0, θ_1, θ_2, θ_3, and θ_4

Forward Geometric Transformation

The forward transformation is straight-forward and always offers a unique solution. This transformation is useful in the collision detection software. The

forward geometric transformation for the Mentor robot arm is achieved by applying the rotations and translations of each axis to it's "home" position. The "home" position is when all joint angles are zero. Angles θ_1 through θ_4 are applied first and angle θ_0 is applied last. On the Mentor robot the shoulder, elbow, and wrist always lie in a common plane which is called the robot plane. By having the robot plane coincide with the xz-plane, the initial rotations are simplified. Once positioned in the robot plane the arm can be moved to its correct position about Axis 0.

Before the transformation can occur, the joint coordinates must be determined. A distinction must be made between the joint values understood by the robot and the joint coordinates. A joint value is an 8 bit data word which is used to drive the robot arm. This allows each joint 256 unique positions within its specified range. A joint coordinate, on the other hand, is the angle formed at the juncture of two links. The conversion between joint values and joint coordinates is:

$$\theta_i = v_i * (\frac{range_i}{values_i})$$

where

θ_i — joint coordinate for joint i

v_i — current joint value for joint i corresponding to θ_i

$range_i$ — range of joint i in angles

$values_i$ — maximum joint value for joint i

See Appendix A for more details.

Once the joint coordinates are calculated from the joint values it is possible to proceed with the transformation. Through a series of rotations and translations, the end effector point is moved from its "home" position (Figure 3.2) to the position defined by the joint coordinates. The algorithm used is as follows:

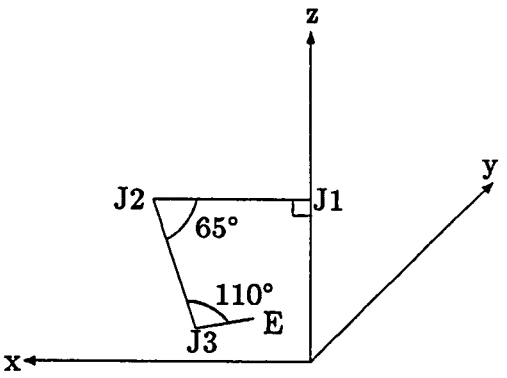


Figure 3.2: "Home" Position

1. Translate origin to J1 and rotate the Shoulder, Elbow, and Wrist θ_1 about the y-axis.

2. Translate origin to new J2 and rotate Elbow and Wrist θ_2 about the y-axis.

3. Translate origin to new J3 and rotate Wrist θ_3 about the y-axis.

4. Rotate θ_4 about the axis defined by the vector through J3 and E.

5. Translate back to base origin and rotate Base θ_0 about the z-axis.

Although, the Denavit-Hartenberg matrix representation is the preferred means of representation by engineers, this system takes advantage of the flexible Pascal data types and produces a clear, straight-forward algorithm.

Inverse Geometric Transformation

Given the position (*px*, *py*, *pz*) and orientation (*pitch*, *twist*) of the end effector, the inverse transformation calculates the joint coordinates defined by

angles θ_0 through θ_4. The transformation described here is specific to the Mentor robot arm which allows only five degrees-of-freedom. (The Mentor does not have roll in the wrist axis). This means that the end effector always lies in the robot plane. The robot plane is the plane in which the shoulder, elbow, and wrist axis lie. The robot plane coincides with the xz-plane of the base coordinate system until θ_0 is applied. This fact is used to simplify the transformations.

Another design problem was how to reach quadrants III and IV since the base rotation is limited to 180° in this system. This is accomplished by first positioning the shoulder in a position diametrically opposed to the third or fourth quadrant and then flipping the shoulder and elbow axis to the negative y quadrants. This means that when axis 1 is flipped, the computed angles become the exterior angles in our drawings, rather than the interior (see Figure 3.3).

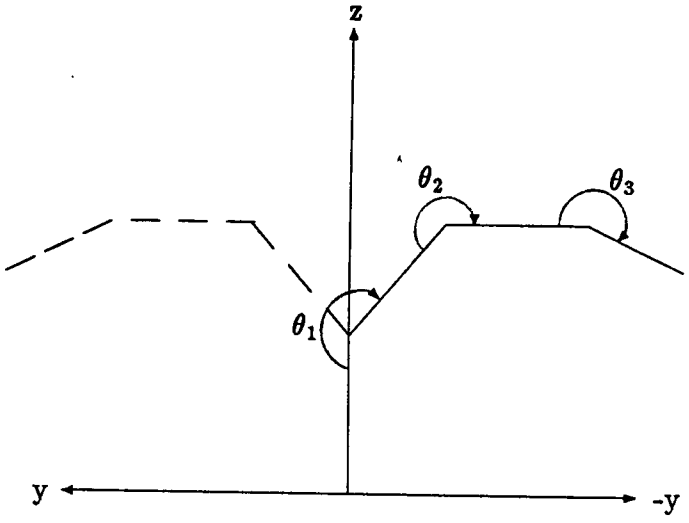


Figure 3.3: Angles in negative y quadrants

Calculation of Angle θ_0 (Base)

When $px = 0$, a point of singularity occurs. There is no unique solution in this situation, so θ_0 is arbitrarily set to 90°. Otherwise, Angle θ_0 is calculated in one of two ways (Figure 3.4):

Quadrant I or III: $\theta_0 = \arctan(\frac{py}{px})$

Quadrant II or IV: $\theta_0 = \arctan(\frac{py}{px}) + \pi$

The second case is necessary as the Mentor base rotates from 0° to 180° and the arctan function returns a value from −90° to +90°. Because the third and fourth quadrants are reached by flipping the robot arm at the shouder, θ_0 always lies in the first or second quadrant.

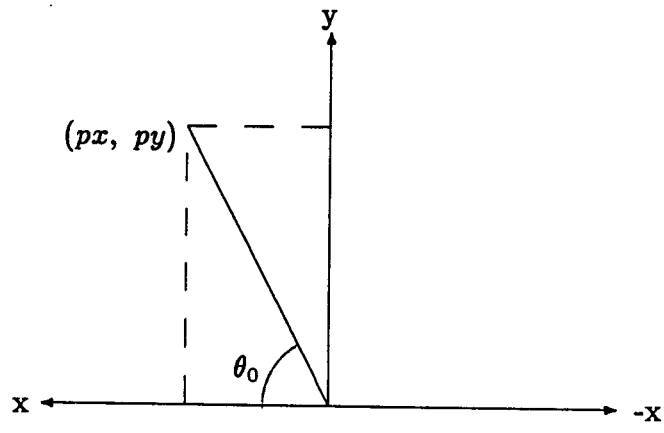


Figure 3.4: Calculation of Angle θ_0 at Base

Calculation of Angle θ_1 (Shoulder)

If the requested position lies close to the z-axis, a point of singularity occurs. In this case, angle θ_1 is set to 180°. Otherwise, θ_1 is computed in two parts: θ_{11} and θ_{12} (Figure 3.5). Applying the formula for half-angles to the triangle ABE:

$$\theta_{11} = 2 * \arctan(\frac{r}{s - lgthAE})$$

where a, b, and c are the lengths of the sides of the triangle ABE and

$$s = \frac{1}{2} * (a + b + c)$$

and

$$r = \sqrt{\frac{(s-a) * (s-b) * (s-c)}{s}}$$

Note that angle θ_{11} is derived from axis lengths using the formula for half angles of triangles. This formula always involves acute angles and thus the arctan is never negative.

Angle θ_{12} is computed simularly using triangle BCE. These two angles are then combined as follows:

Quadrant I or II: $\theta_1 = \theta_{11} + \theta_{12}$

Quadrant III or IV: $\theta_1 = 2 * \pi - (\theta_{11} + \theta_{12})$

The second case flips the shoulder axis to the negative quadrants.

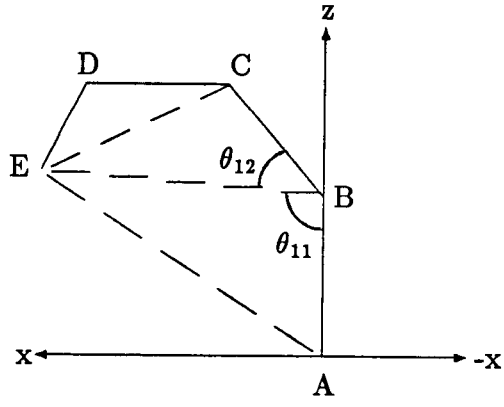


Figure 3.5: Calculation of Angle θ_1

Calculation of Angle θ_2 (Elbow)

The rotation about the elbow is similar to the computation for θ_1. If the requested position lies close to the z-axis, a point of singularity occurs and angle θ_2 is set to 180°. Otherwise, angle θ_2 is found using two partial angles θ_{21} and θ_{22} (Figure 3.6). There are three cases: (1) end effector is below the wrist, (2) end effector is above the wrist, and (3) end effector is in-line with the wrist.

Below: $\theta_2 = \theta_{21} + \theta_{22}$

Above: $\theta_2 = \theta_{21} - \theta_{22}$

In-line: $\theta_2 = \theta_{21}$

Once again the arm must be flipped to the other side when the position is in quadrants III or IV:

$$\tilde{\theta}_2 = 2 * \pi - \theta_2$$

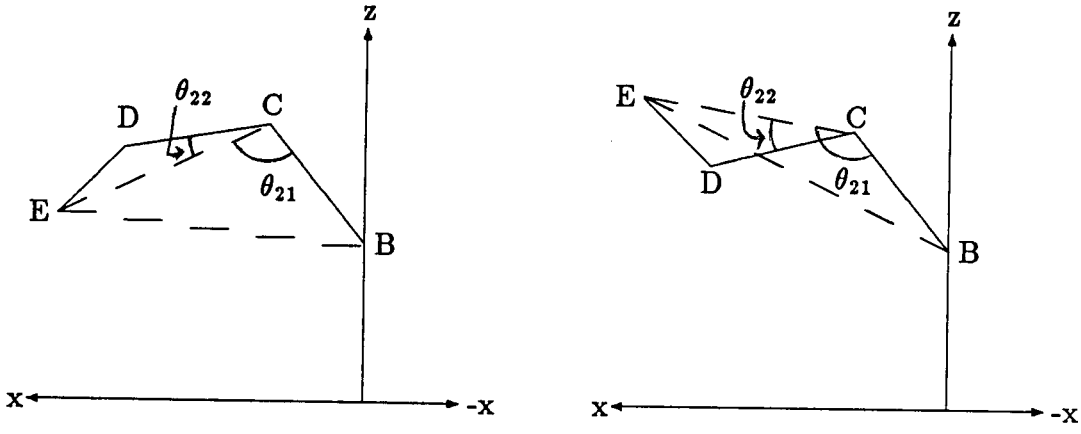


Figure 3.6: Calculation of Angle θ_2

Calculation of Angle θ_3 (Wrist—Pitch)

Angle θ_3 is computed directly from the orientation pitch angle. This angle is given by the robot programmer when defining a step. There are two cases to consider (Figure 3.1 on page 15 and Figure 3.3 on page 18):

Quadrant I or II: $\theta_3 = pitch$

Quadrant III or IV: $\theta_3 = 2 * \pi - pitch$

Calculation of Angle θ_4 (Wrist—Twist)

Angle θ_4 is taken directly from the orientation twist. This angle is also given by the robot programmer at step definition time.

$$\theta_4 = twist$$

Conversion of Joint Coordinates to Joint Values

In converting from joint coordinates to joint values, the joint coordinates θ_1, θ_2, and θ_3 must first be modified to accommodate the robot limitations. Since the joints cannot move a through a complete 180° arc, the limits of their movement must be subtracted from the joint coordinates. For instance, Axis 1 has a range of 180° beginning at a 90° angle. So, 90° is subtracted from θ_1. After this adjustment, joint coordinates θ_0 through θ_4 are converted to joint values v_0 through v_4 as follows:

$$v_i = ROUND\ (\theta_i * \frac{values_i}{range_i})$$

where

v_i — current joint value for joint i corresponding to θ_i

θ_i — joint coordinate for joint i

$values_i$ — maximum joint value for joint i

$range_i$ — range of joint i in angles

This solves the problem for v_0, v_1, and v_2, but v_3 and v_4 cannot be used directly for pitch and twist due to the Mentor robot design. Pitch and twist are obtained by rotating Axis 3 and Axis 4 together. Pitch is obtained by rotating Axis 3 and Axis 4 in the same direction. Twist is obtained by rotating Axis 3 and Axis 4 in opposite directions. To compensate, half the robot values available are allocated to pitch and half of the robot values are allocated to twist. This design leads to limitations on the pitch and twist available computationally. The final joint values for pitch and twist, $\tilde{v}_3$ and $\tilde{v}_4$, are computed as follows:

$$\tilde{v}_3 = v_3 + v_4$$

$$\tilde{v}_4 = v_3 - v_4$$

Using the above geometrical approach we can determine if a point is reachable by looking at the resulting joint value. If the joint value falls outside the valid range of joint values (0-255), then the position is not reachable by the robot arm and is rejected by the robot programming system.

CHAPTER 4

ROBOT JOINT CONTROLLER

Overview

The lowest level of control in my robot control system, software-wise, is the robot joint controller. The joint controller is concerned with moving the individual joints of the robots to their destination setpoints. Each joint is driven independently without respect to the movement of the other joints. There is no concept of a manipulator, of wrist position and orientation, or even of joint coordinates at this level. There are only joint values.

The robot joint controller was written in Microsoft 8086 assembler and was installed as an interrupt handler. Since the robot itself does not generate interrupts, the joint controller is driven by the PC-DOS User Clock Interrupt. Thus, the controller is initiated at discrete time intervals to monitor joint movement. It does this without interfering with the higher level components of the robot control system.

Environment and Robot Specification

As with the higher level control some parameters were provided for environment and robot specification. These are defined in the ROBOTDEF.ASM file. For environment, a parameter specifies the number of robots. Theoretically, this controller allows for any number of robots. But, in practice, it is limited by memory size and run speed. It must be able to complete one monitoring cycle

in an acceptable time frame for real-time use. For robot specification, we need the number of axes and the offsets from the robot base address of the axes data words, of the multiplexor, and of the adc converter.

Also described in ROBOTDEF.ASM is the robot control block (RCB). Mapped into a shared memory region is a RCB for each robot. The shared memory region, and the RCBs in particular, is used to communicate with the PASCAL task execution procedure. Each RCB contains a robot ID number, the base address in memory of the robot, a status word, the destination setpoints expressed as joint values, and the last read position of the joints, also expressed as joint values.

Communication Mechanisms

Communication with the robots is done by offsets from the base address. The robot base address defines a region in which the robot writes and reads data words containing joint values. Assembly level functions are provided which (1) read the current joint values of the robot, (2) write the destination setpoints, or (3) write intermediate setpoints to the robot.

The joint controller communicates with the PASCAL task execution procedure through the RCB destination setpoints and the status word. The status word contains four flags: READY, ERROR, BUSY, and SETPOINT. The first two are for communication with the task execution procedure and the last two are for internal use by the joint controller.

Step Execution

A task in this system is of the PTP (point-to-point) type. It is defined as a series of destination points. Each destination point is transformed into a set of joint values, or setpoints, as described in Chapter 3. These setpoints are passed to the joint controller by the task execution procedure. Pauses at the destination point are achieved through the task execution procedure, not the robot joint controller.

When all status flags are clear, the task execution procedure may initiate a move by putting the destination setpoints into the robot's RCB and setting the READY flag. This may be done for more than one robot thus running the robots simultaneously. The task execution procedure then waits for the READY flag(s) to be cleared. All READY flags must be cleared (and no ERROR flags set) before the next step can be initiated.

When the joint controller detects the READY flag is set, it sends the new setpoints to the robot and sets the BUSY flag indicating that a move has been initiated. This is done for all robots which have the READY flag set. Every time the controller is inititated by the USER CLOCK INTERRUPT it monitors the BUSY robots for movement. Four situations may arise here and are described below:

1. If the arm is moving (that is, at least one joint is still moving), the controller just goes on to the next robot or exits if all robots have been serviced.

2. If the arm has stopped (that is, all joints have stopped moving) and the joints are within an acceptable range of the setpoints, all status flags are cleared.

3. If the arm has stopped and the joints are not within an acceptable range of the setpoints, the joint controller computes a new setpoint using the original setpoint and the current position of the arm. The new setpoint is then sent to the robot arm. This attempt to get closer to the original setpoint is done only once.

4. If the arm has stopped outside an acceptable range of the setpoints and boosting of the setpoints as described in the previous step has been attempted, all action is aborted and all robots are stopped where they are and their READY flags are cleared. The ERROR flag is set in the RCB of the offending robot.

Meanwhile, the PASCAL task execution procedure is monitoring the READY flags of the robots' RCBs. When it detects that all the READY flags have been cleared, it can proceed. First the ERROR flag of all robots is checked. If there has been an error, the operator is notified and action requested. If no error has occurred, or if ignored by the operator, the task execution procedure delays if a pause is indicated in the step descriptor. It then inititates the next step of the task.

Accuracy

Since there are no external devices to measure the wrist position and orientation we are forced to rely upon the joint values. Due to load, nonlinearity, and other factors, the position actually achieved may only be within some neighborhood of the desired position. For a further discussion on the problems of accuracy see Lumelsky [15].

The joint controller is unaware of dual or single task execution. It merely moves the joints to the desired setpoints. It is up to the task execution procedure to coordinate steps between robots. Since this is not the focus of this thesis, all steps are paused the same length of time. Thus, for two different robot tasks the steps are already synchronized.

CHAPTER 5

MULTI-ROBOT SYSTEMS AND COLLISION AVOIDANCE

Multi-Robot Systems

Integration of multiple robots into one system creates new problems for robotics researchers. The problems have an inherent hierarchical structure which is useful in the development of a multi-robot system. At the highest level are problems of scheduling and task allocation [16]. These problems are beyond the scope of this thesis.

At the next lower level is the need to coordinate the movements of two or more robots. Coordinated movement may viewed from three different perspectives [7]. First as non-interfering independent tasks running simultaneously. Second as cooperating tasks on separate robots, such as passing an object. Third as a single task requiring more than one robot, such as in lifting an object using two grippers. In this case, the two arms must maintain a relative position to one another. This case has been studied by Alford and Belyeu [1]. This thesis considers only the first case of non-interfering tasks.

The lowest level of my multi-robot control system is the robot joint controller which is discussed in Chapter 4.

Multiple Tasks on Separate Robots

One major problem of multiple tasks in a multi-robot system is that of collision avoidance. It is assumed that the robots in such a system are set up to

share some workspace called the common workspace (Figure 1.4 on page 5). It is within the confines of the common workspace that collisions may occur.

Several different approaches have been taken to this problem. Gouzenes [9] treats space and workareas as resources which may be allocated and deallocated. This is a common approach since it works and it avoids the complexities of optimization and the time-consumption of extensive searches. In the more complex camp, is Freund's nonlinear control method [7] which provides a greater accuracy in position determination. To avoid collisions, Freund computes a new path for one of the robots. This computation involves both searching and optimization.

The approach taken for this thesis is of the more complex nature and does allow both arms to enter the shared workspace at the same time. However, in this thesis, one of the robots will wait for the other robot to pass rather than compute new trajectories as a means of avoiding collisions.

Furthermore, a distinction can be made between static and dynamic collision avoidance [16]. Static collision avoidance refers to a single moving object avoiding static objects such as conveyor belts and worktables. Dynamic collision avoidance refers to simultaneous moving objects avoiding one another. This thesis addresses problems of dynamic collision avoidance.

Briefly, the method to avoid collisions described here involves approximating the robot paths and comparing the paths on a point-by-point basis for collisions. If there is a potential collision, one arm is held back to allow the other to pass.

Collision Detection

Collision detection is that aspect of collision avoidance which identifies potential collisions. One method commonly used is to sweep out a volume in space

which describes the robot's movement between two points [4, 6]. This method ignores the time function of such a movement and may consequently show a collision when actually there is none. Another method described by [4] uses abstract spaces to detect collisions. Cameron [4] discusses some of the advantages and disadvantages of several collision detection algorithms.

This thesis takes into account the time function by looking at the robot's position at discrete intervals along it's trajectory. There are two phases of collision detection in my control system. One happens before the tasks are run. The other occurs in real-time between each step of the robot tasks.

Phase 1 Collision Detection

Before any attempt is made to run two simultaneous tasks it is necessary to determine as far as possible that they will not interfere with one another. To do this, the destination setpoints of the end-effectors are compared at each step. The steps of all tasks in this thesis have the same fixed delay time and are therefore automatically synchronized. This was done to narrow the scope of the project. We will call the setpoint of each step the destination point of the robot move. If the destination points of the corresponding steps are the same then a collision is unavoidable since both arms are trying to occupy the same space at the same time. Note that this check does not guarantee the simultaneous runability of the two tasks, but does detect some cases where simultaneously running the tasks is not possible.

If the tasks pass the first phase of collision detection, then the tasks are initiated and phase 2 takes over.

Phase 2 Collision Detection

Phase 2 looks at a task on a step-by-step basis. The problem is to determine if the two arms will collide with one another when moving along their paths to their next destination points. The method described here involves approximating the path trajectory and comparing the robot positions at discrete time intervals.

Path approximation. The tasks of this system are of the point-to-point (PTP) type. That is, each joint moves directly to its destination and we assume all joints move at the same speed. This means some joints may reach their destinations before others and the path described is unknown to the robot programmer.

However, it is still possible to approximate the path by computing where each joint would be after a discrete time interval. In particular, it takes a joint on the Mentor about ten seconds to move its full range, that is from joint value 0 to joint value 255. As a rough approximation this is about twenty joint values per second. This is the time interval chosen for dividing the movement into substeps. Each substep being twenty joint values closer to its destination. Thus, the maximum number of substeps is thirteen. Of course, at each substep position the joint values of the individual joints must be converted to joint coordinates.

After the substep positions of each joint of each robot are computed, the substeps from the first robot are compared to the corresponding substeps of the second robot. Note that in this method, if the number of substeps is too small (ie—the time interval is too long), it is possible to fail to detect potential collisions. On the other hand, if there are too many substeps the extra work generated can be time-consuming. (Cameron [4] used a similar approach in his path planning algorithm.)

Comparison Algorithm. During phase 2 collision detection we are concerned with the whole robot arm. Thus, the comparison technique used must be more sophisticated than that used in phase 1. In my algorithm, each robot arm is divided into three major regions, corresponding to the shoulder (Axis 1), the forearm (Axis 2), and the end-effector. Each region is rectangular in shape and is described by eight vertices. Initially the three regions of each robot are set to their "home" positions (Figure 1.4 on page 5).

To compare substeps each of a robot's region vertices are first rotated through the joint coordinates using the forward transformations discussed in Chapter 3. The origin of Robot 2's coordinate system is translated to the origin of Robot 1's coordinate system giving the coordinates of both robots' regions a common origin. Once the approximate position of the two arms is obtained, each region is compared to all the regions of the other robot looking for overlapping regions which would indicate a collision.

The comparison between regions is done by projections of each region in 3-space onto the orthogonal planes defined by the three axes of the base coordinate system. That is, each of the regions is projected onto the xy-, yz-, and xz-planes. The projections from Robot 1's regions are compared to the projections of Robot 2's regions. If there is a collision in 3-space, there will be an overlap of projections in each of the three planes. If there is no overlap of projections in any one of the orthogonal planes, there is no collision in 3-space.

Looking for overlaps of projections in the xy-plane, yz-plane, and xz-plane is a time-consuming process. To improve the speed, a collision zone is defined which coincides with the common workspace of the robots. The above comparison of substep projections is done only if any of the robots' regions are wholly or partially in the collision zone. Note that it is possible for some part of the robot

arm to be in the collision zone even though the end-effector is outside the collision zone.

Path Modification

The computation of path trajectory and the comparison of substeps is straight-forward and fast. The real problems arise in trying to determine which arm to delay and when to delay in such a way that allows both arms to reach their destinations.

Originally, at each substep where a collision was imminent, we first tried holding Robot 2 back at a previous substep and letting Robot 1 pass. If this failed, we held Robot 1 back and let Robot 2 pass. This method allowed for the possiblity of two or more potential collisions in any one step and for alternating right-of-way priorty between the robot arms. However, because of the vast number of possible combinations the amount of time needed to compute a possible path was unusable for real-time usage. (Remember, not only are there substeps but there are also "hold" steps. That is, one of the robots may not be moving during one or more of the substeps.)

The scheme was modified slightly resulting in some loss of flexiblity. The modification was to allocate right-of-way priority to one of the robots throughout the entire step. If this failed, the right-of-way priority was given to the other robot. If both failed to find a solution, an error occurred. This can happen when the destination points are too close to one another.

Although the modified method is considerably faster than the first approach it still requires several minutes (as opposed to tens of minutes) to compute the new path and still is not useful for real-time usage.

CHAPTER 6

RESULTS AND CONCLUSIONS

This thesis describes the integration of several problems in robotics research. These topics have been explored in varying degrees with an emphasis on collision detection and avoidance.

Results

The robot joint controller was developed first. This was done to verify the modifications to the second robot interface board which altered the base address of the second robot. Robot motions were initially entered directly as joint values to demonstrate the correctness of the robot joint controller and to verify the running of multiple robots. Because the robots move slowly, it was necessary to increase the time interval between the running of the joint controller.

The Turbo Pascal high-level control was developed next. Entry of several robot tasks was made for the individual robots. The tasks were first run alone to demonstrate the correctness of the inverse coordinate transformations and the passing of steps to the robot joint controller. At this point in development, I was able to specify a task in Cartesian coordinates and to run that task. One minor problem was finding an object the robot could handle. Because of gripper design, the object could not be heavy. The lack of sensors required the object to be soft as the robot does not know when it has successfully gripped an object. The best object I found was a Nerf ball. With the Nerf ball, it did not matter at what angle the ball was picked up or how hard the ball was squeezed. In my

test, Robot 1 picked up the ball, moved it to the common workspace, released the ball, and moved out of the way. As a separate task, Robot 2 picked up the ball and moved it out of the common workspace.

After successfully running single tasks, several experiments were made running two tasks simultaneously on two robots. Without any heuristic aids in path planning, when a collision was detected the time to compute a path was unpredictable and frequently exhorbitant. The first heuristic method used allowed for alternating right-of-way priority between the robots during any one step. The method was still excessively time-consuming. The addition of an 80287 math coprocessor chip cut computation time in half and the heuristic method was modified to allow only one robot the right-of-way during any one step. These modifications further improved the speed of the method although it was still of questionable value in real-time.

To demonstrate the collision detection algorithm, several dual tasks were run. Figures 6.1 through 6.3 illustrate these tasks. In these figures, A_1 and B_1 refer to steps 1 and 2 of Robot 1. A_2 and B_2 refer to steps 1 and 2 of Robot 2.

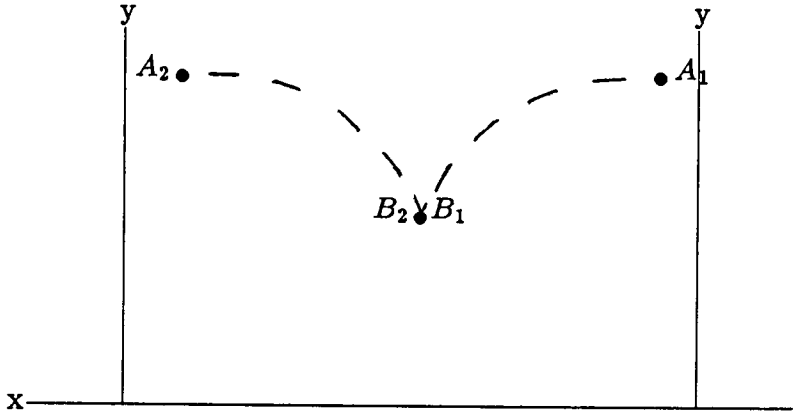


Figure 6.1: Tasks Failing in Phase 1 Collision Detection

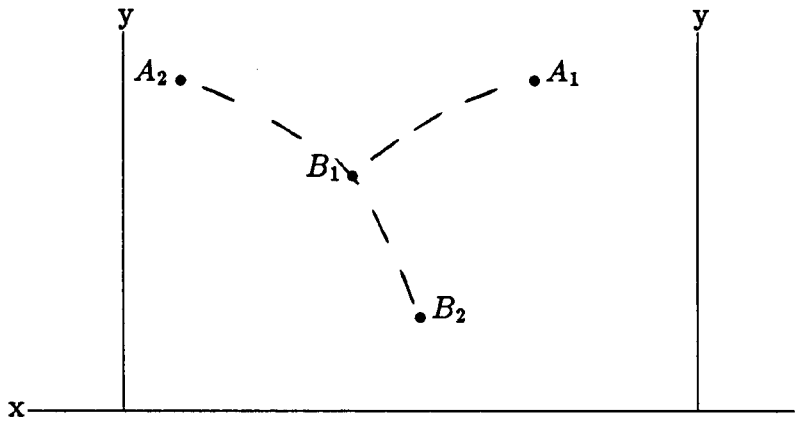


Figure 6.2: Tasks Failing in Phase 2 Collision Detection

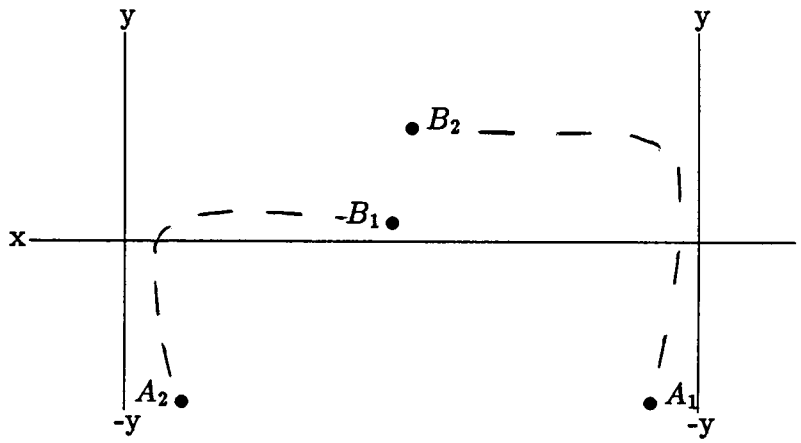


Figure 6.3: Tasks not failing Collision Detection

The first set of two tasks (one for each robot) required that the end-effectors of Robot 1 and Robot 2 be in the same position during the same step (Figure 6.1). This task failed during Phase 1 collision detection and was never run. It is easy to see it is an impossible combination of tasks.

To demonstrate Phase 2 collision detection, a second set of two tasks was

developed (Figure 6.2). In this dual task run, task 1 was subdivided into four substeps and task 2 was subdivided into seven substeps. A collision was detected when Robot 1 reach substep 4 and Robot 2 reached substep 5. The path modification algorithm was able to determine that by holding Robot 1 at substep 2 until Robot 2 had reached substep 6 both robots were able to reach their destinations. A variation of this test was run switching the roles of the two robots. In this case, Robot 2 was successfully held back while Robot 1 passed.

The most interesting test was one which showed a move from the negative y quadrants to the positive y quadrants (Figure 6.3). Intuitively, those unfamiliar with the particular robot would expect a collision to occur somewhere midway through the move. However, in this case, the variability of trajectories was illustrated. Because the robot shoulder and elbow were flipped to reach the negative y quadrants, the arms move in an unexpected fashion. Midway through the move both arms were in a nearly vertical position rather than in the collision zone. There was no collision in this test.

Conclusions

This thesis has demonstrated the feasibility of robotics research even in a limited environment and it has illustrated the extensive base for robotics research. Obviously, there are some interesting topics such as sensory feedback which cannot be explored on the Mentor robots. However, in each of the areas described in the previous chapters, there is additional work which can be done on the Mentor robots and each area of robotics offers interesting and timely research problems. In particular, the area of multiple robots and of multiple moving objects in an environment is still a relatively young field of research. Some possible topics are:

1. A full-fledged robot programming language could be developed including features such as conditional actions and non-movement actions such as might be used by the gripper. A language which addresses the specific problems of multi-robot task definition would be an interesting problem. This kind of language would be useful in describing tasks which require the use of two, or more, robots—such as lifting an object with two robots.

2. The method of entry of robot "tasks" could be extended to include entry by joint coordinates or by the simulator. Entry by relative amounts (UP 10mm, LEFT 30 degrees) or by high-level commands (MOVE-TO-POSITION-A) would be useful.

3. In the area of kinematics, it would be interesting to apply one or more of the mathematical methods to the transformation problem. However, it would be difficult to include the force equations without some modification.

4. The robot joint controller would benefit from a structure like the motion queue described by Hayward and Paul. A motion queue would enhance the smoothness of robot operation and address the step synchronization problem.

5. Another important area of concern is the treatment of objects. Although the software currently has descriptors for describing objects and regions reserved for their use, objects were not included in the collision avoidance algorithm. Problems of object orientation and rotation, and of object pickup points must first be addressed.

With this list of suggestions, it is possible to see the richness of robotics as a field of research and exploration.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Alford, C.O. and S.M. Belyeu, "Coordinated Control of Two Robot Arms," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 468–473, 1984.

[2] Angell, Ian, *A Practical Introduction to Computer Graphics*, Halsted Press, 1981.

[3] Blume, C. and W. Jakob, *PASRO: Pascal for Robots*, Springer-Verlag, 1985.

[4] Cameron, S., "A Study of the Clash Detection Problem in Robotics," *Proceedings 1985 IEEE International Conference on Robotics and Automation*, pp. 488–493, 1985.

[5] Ersu, E. and D. Nungesser, "A Numerical Solution of the General Kinematic Problem," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 162–168, 1984.

[6] Faverjon, B., "Obstacle Avoidance Using an Octree in the Configuration Space of a Manipulator," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 504–512, 1984.

[7] Freund, E., "On the Design of Multi-Robot Systems," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 477–491, 1984.

[8] Freund, E. and H. Hoyer, "Pathfinding in Multi-Robot Systems: Solution and Applications," *Proceedings 1986 IEEE International Conference on Robotics and Automation*, vol. 1, pp. 103–111, 1986.

[9] Gouzenes, L., "Collision Avoidance for Robots in an Experimental Flexible Assembly Cell," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 474–476, 1984.

[10] Hayward, V. and R.P. Paul, "Introduction to RCCL: A Robot Control 'C' Library," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 293–297, 1984.

[11] Lee, C.S.G., "Robot Arm Kinematics, Dynamics, and Control," *Computer*, vol. 15, no. 12, pp. 62–80, December, 1982.

[12] Lee, C.S.G. and M. Ziegler, "A Geometric Approach in Solving the Inverse Kinematics of PUMA Robots," *13th International Symposium on Industrial Robots and Robots 7*, vol. 2, pp. 16-1–16-18, 1983.

[13] Lee, C.S.G., "On the Control of Robot Manipulator," *Proc. of SPIE, Robotics and Robot Sensing Systems*, vol. 442, pp. 58–83, 1983.

[14] Litvin, F.L. and V. Parenti Castelli, "Robot's Manipulators: Simulation and Identification of Configurations, Execution of Prescribed Trajectories," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 34–44, 1984.

[15] Lumelsky, Vladimir, "Control of Robot Motion," *Proc. of SPIE, Robotics and Robot Sensing Systems*, vol. 442, pp. 84–96, 1983.

[16] Maimon, Oded Z., "A Multi-Robot Control Experimental System with Random Parts Arrival," *Proceedings 1985 IEEE International Conference on Robotics and Automation*, pp. 895–899, 1985.

[17] McInnis, B. and C. Liu, "Coordinate Frames, Transformations, and Inverse Functions for Joint Variables in Robotics," *Proceedings 1986 IEEE International Conference on Robotics and Automation*, vol. 3, pp. 1853–1858, 1986.

[18] Paul, R., *Robots Manipulators: Mathematics, Programming, and Control*, MIT Press, 1981.

[19] Roach, J. and M. Boaz, "Coordinating the Motions of Robot Arms in A Common Workspace," *Proceedings 1985 IEEE International Conference on Robotics and Automation*, pp. 494–499, 1985.

[20] Ruoff, C., "Comments on Robot Control Systems," *Robotics Research - 1st International Symposium*, pp. 941–948, 1983.

[21] Sadre, A., R. Smith and W. Cartwright, "Coordinate Transformations for Two Industrial Robots," *IEEE Proc. of Intl Conf. on Robotics*, pp. 45–61, 1984.

[22] Shimano, B.E., C.C. Geschke and C.H. Spalding III, "VAL-II: A New Robot Control System for Automatic Manufacturing," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 278–292, 1984.

[23] Shin, K.G. and S.B. Malin, "A Hierarchical System Structure for Coordinated Control of Industrial Manipulators," *Proceedings 1984 IEEE International Conference on Robotics*, pp. 609–620, 1984.

[24] Wong, E.K. and K.S. Fu, "A Hierarchical-Orthogonal-Space Approach to Collision-Free Path Planning," *Proceedings 1985 IEEE International Conference on Robotics and Automation*, pp. 506–511, 1985.

APPENDICES

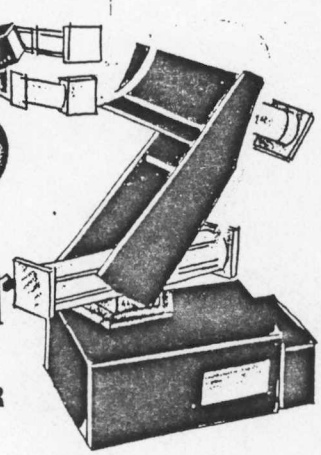
APPENDIX A

HARDWARE DESCRIPTION

RICHARD B. H. BECKER — SYSTEM DESIGN AND MECHANICAL ENGINEERING.
TIM ORR — COMPUTER INTERFACE AND CONTROL ELECTRONICS.

ROBOTS

neptune and mentor



PART FOUR

THIS SERIES is concerned in part four with the control electronics and mechanical construction of the MENTOR robot. Like the NEPTUNEs, MENTOR interfaces directly to any one of three popular home computers and provides the facility for control either from the keyboard, or by making use of the "learning arm".

PRINCIPLES OF OPERATION

MENTOR is a 6-axis servo controlled electric robot (Fig. 4.1). Whilst the NEPTUNEs are designed for industrial use, MENTOR is primarily intended for educational purposes but as can be seen from the specification (Table 1) it too has the repeatability for many commercial applications, where the load is light. Each of the axes is powered by a small d.c. servo motor with integral gearbox with a large reduction ratio.

Axis Ø is the centre column which rotates in a nylon bearing in the top plate of the base of the robot and is powered by the motor through an additional pair of gears (Fig. 4.2). The radial position of the column is sensed by a conductive polymer potentiometer fitted to the underside of it. The column is hollow to enable the cables to pass down it to the computer interface board.

Axis 1 is the lower section of the arm which rotates about an axle fitted to the centre column (Fig. 4.3). Again a pair of gears is used to transfer torque from the motor to the axle and a potentiometer provides the feedback information. The axle is hollow for the cables to pass through it.

Axis 2, the fore arm is driven in the same manner as axis 1. A large piece of steel at the back end of the lower arm section counterbalances the weight of this section of the arm making the robot's position stable when there is no power applied to it;

Table 1. MENTOR SPECIFICATION

AXIS Ø (centre column)	Angular movement 210°. Axle centre 170mm above top of base.
AXIS 1 (shoulder)	Angular movement 180°. Arm length between axle centres 165mm.
AXIS 2 (elbow)	Angular movement 230°. Arm length between axle centres 150mm.
AXIS 3* (left wrist axle)	Angular movement 320°.
AXIS 4* (right wrist axle)	Angular movement 320°.
WRIST PITCH	Angular movement 140°.
WRIST ROTATION	Angular movement 320°.
AXIS 5 (gripper)	Jaw opening 30mm. Jaw pressure 10 newton. Distance from end of jaws and axis 3 and 4 axles 105mm.
REPEATABILITY	2mm.
LIFTING CAPACITY	300gm.
REACH (from axis 1 axle centre)	420mm.
BASE DIMENSIONS	Overall width 320mm. Overall depth 270mm. Overall height 189mm.
CONTROL SYSTEM	All axes servo controlled with servoing performed by the control electronics. Position defined by 8 data bits giving angular resolution of 0·4%.
COMPUTER INTERFACE	Parallel. Robot addressed as if part of computer memory. Connects to expansion port (1MHz bus on BBC).
SOFTWARE	Accepts commands in BASIC or machine code.
SOFTWARE PROVIDED	Extensive package of BASIC programs including direct control from computer keyboard, control by simulator, sequence storing, replay, sequence editing, sequence storage on disc or tape, multi-speed control and graphical illustration of robot dynamics.

*AXIS 3, 4 movements are combined to provide wrist pitch and wrist rotation.

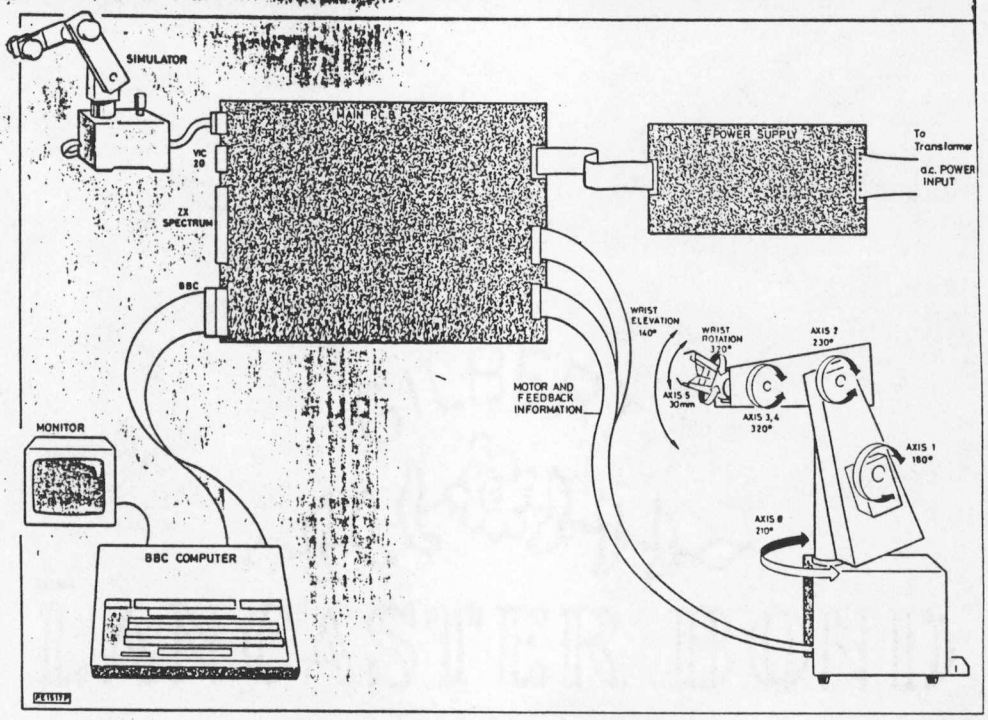
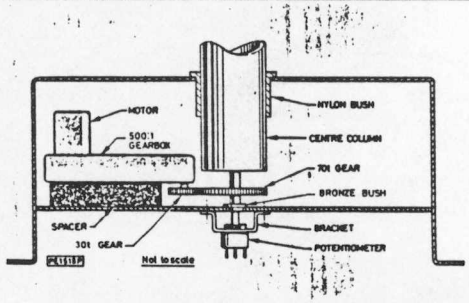
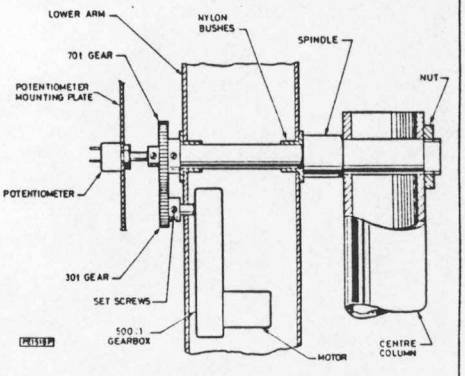


Fig. 4.1. Exploded view of the complete MENTOR system

therefore the drive motor does not have to be continuously on to keep the arm in position. There are also counterbalance weights on the fore arm to balance out the wrist motors.

Axes 3 and 4 are the wrist drive motors. The movements are combined to provide wrist elevation and wrist rotation by means of a set of bevel gears. When axes 3 and 4 advance together the





Axis drive systems: Fig. 4.2, above, shows the system used to drive axis 0. Fig. 4.3, opposite, shows the drive system for axis 1. See also Fig. 4.1, and the text

vrist rises. When the axes move in equal and opposite directions he elevation remains constant but the wrist rotates, as shown in ?igs. 4.4 and 4.5.

Axis 5 is the gripper which is also servo controlled. To keep lown the weight at the front end of the fore arm the drive motor or this is fitted to the lower arm with the power applied to the aws via a flexible cable consisting of a nylon-coated multi-:trand steel wire inside a spiral wound PVC-coated conduit. This s similar to the brake cable of a bicycle. Torsion springs open he jaws when tension is released on the cable. Being servo con-rolled it is possible to program different degrees of jaw closure. .f the programmed position is for a gap between the jaws of less han the size of the object to be handled then the grip will depend on how much further the servo system is trying to move the aws.

THE MENTOR CONTROL SYSTEM

The MENTOR, like the NEPTUNE, can be connected to one of three popular computers, the Commodore VIC20, the Sin-:lair ZX Spectrum and the BBC. When the computer sends the robot new data it responds by moving to a position where the feedback information from the potentiometers exactly matches the information from a DAC (digital-to-analog converter) con-rolled by the computer. This process is called servoing. Each of the 6 axes of movement can be defined with 8-bit (1 part in 256 or 0·4%) resolution.

ROBOT CONTROL

To move each of the axes the computer sends (WRITEs) 6 bytes of data to the robot, one to each axis. The servoing is per-formed by dedicated hardware on the computer interface board of the robot. In a typical movement of the robot, the computer sends 6 new bytes of data to the robot. The computer is then writing to what it "thinks" is just another area of its memory. The interface board then stores these 6 bytes of information and converts them to 6 position control voltages. The analog elec-tronics on the interface board then compares these voltages with the feedback voltages. The difference between these two voltages is delivered to high power operational amplifiers which drive the motors in the direction which reduces the difference to zero. This is the servoing process.

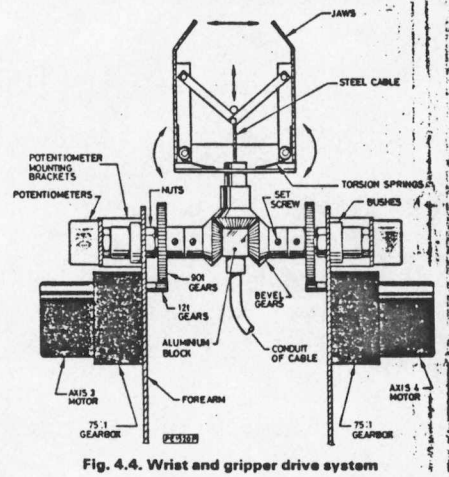
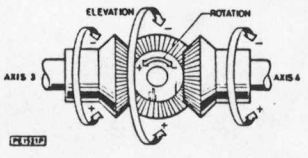


Fig. 4.4. Wrist and gripper drive system



AXIS 3	AXIS 4	E	R
0	−1	−½	+½
0	+1	+½	−½
−1	0	−½	−½
−1	−1	−1	0
−1	+1	0	−1
+1	0	+½	+½
+1	−1	0	+1
+1	+1	+1	0

Fig. 4.5. Wrist operation showing effect of one-bit change on axes 3 and 4

FEEDBACK

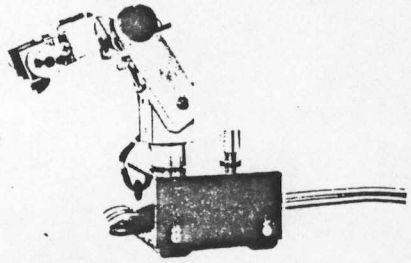
The computer is not involved in the servoing, it is merely generating data and relying on the robot to follow the instruc-tions correctly. However the computer can perform a READ of any or all of the 6 feedback voltages and can thus be aware of all the movements of the arm. This process can be used to delay sending the next set of co-ordinates until the previous set of co-ordinates have been reached or for varying the speed of the robot during playback of a sequence. It is also easy to produce a graphical display of the movements and observe the effect of varying loads and of alterations to the components determining the characteristics of the servo system.

The computer can also READ the position of the simulator, which is a small hand operated model of the robot which can be plugged into the learn axis input CN400. The computer READS the position of the simulator and then WRITES this data into the robot. The arm follows the real-time motion of the simulator. The movements are stored in the memory of the computer and can be transferred to tape or disc for later use.

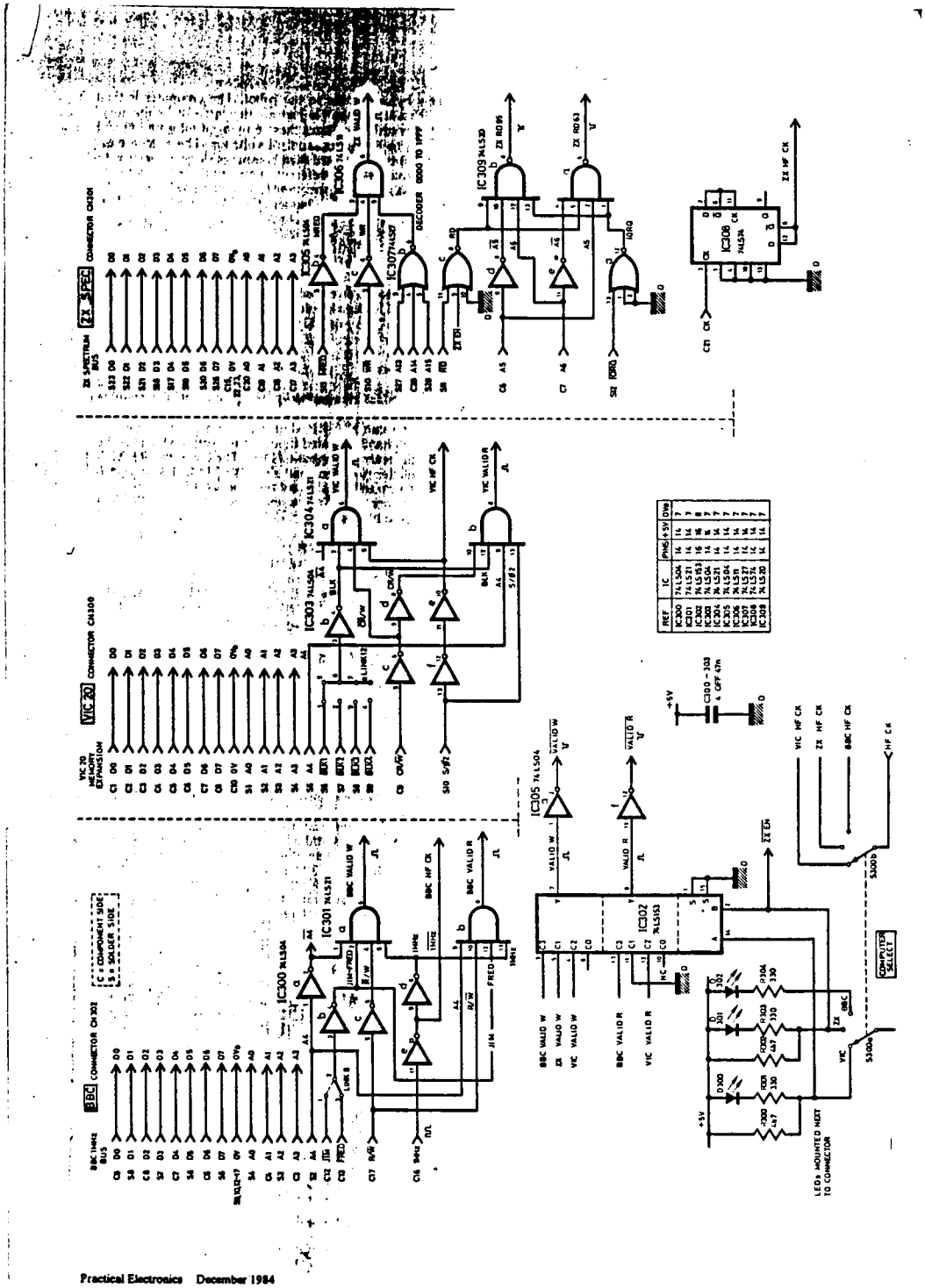
COMPUTER INTERFACES

All three computers make available various signals for exter-nal use. See Fig. 4.6. These include the data bus DØ to D7, some of the lower address lines, READ and WRITE signals, the system clock and on the VIC20 and BBC a memory block decode. The BBC and VIC20 both use the 6502 microprocessor, so have relatively similar decoding electronics. The Spectrum uses a Z80. This together with some unusual practices in the Spectrum design makes interfacing somewhat different.

All the data and address lines are hard wired together so if two computers were simultaneously connected a bus clash would occur. However there is no reason why two or more robots should not be daisy-chained together either to perform the same task or completely different tasks if a different address is used. For example, using a BBC Computer one robot could be addressed with the link-selected "FRED" decode (FCØØ-FCFF) whilst the other is addressed with the link-selected "JIM" decode (FEØØ-FEFF). All three interfaces generate a VALID WRITE signal. The Spectrum WRITES to the block ØØØØ-1FFF which is where the ROM resides and as ROMs are not normally writ-ten to no conflict occurs. The other computers WRITE to non-existent areas of RAM. IC302 is a multiplexer which selects which interface will generate the READ and WRITE signals. The Spectrum interface generates two READ signals (ZXRD63 and ZXRD95). These are not passed through the multiplexer but are used later on in the electronics to perform direct READs.



The MENTOR simulator (learning arm)



WRITING

Data is sent to the robot as 6 bytes, each with a different address. See Fig. 4.7. Six latches are used to store this data (IC104, 107, 109, 112, 114, 117). These are individually clocked with the address decodes from IC102, which is driven by the lowest three address lines. A typical WRITE is as follows. In one computer instruction cycle, the data is set up on the data bus, a VALID WRITE signal is generated and the LSBs of the address bus are set to the correct axis address. When the VALID WRITE signal goes high the data is written into the latch corresponding to that axis. The other axes are then written to similarly. The data held in the latches is converted into position control voltages by the DACs (IC105, 108, 110, 113, 115, 118). All the data passes through IC100 which is a 74LS245 bus transceiver. When the DIRection signal is low, data passes from B to A (WRITE mode). When the DIR signal is high, data passes in the other direction from A to B (READ mode).

READING

There are two ADCs in the system, IC400 and 402 as shown in Fig. 4.8. These are 8-bit devices with integral 8-way analog multiplexers and integral 3-bit address latches. IC400 is used to look at the feedback voltages VFBØ to VFB5 and the other looks at the learn axis (simulator) signals VLØ to VL5. The input voltages to the analog multiplexer must not exceed the ADC

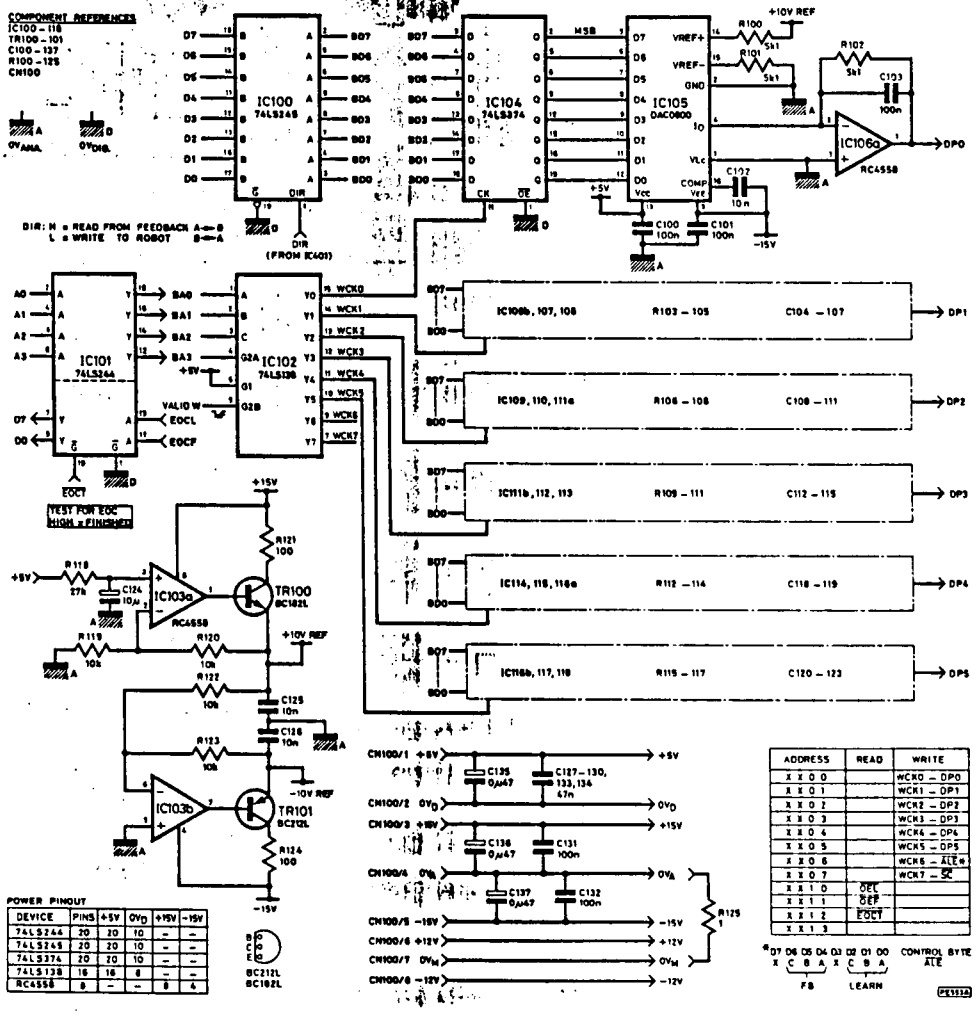


Fig. 4.7. The electronics concerned with a "Write" operation

supply rails. To prevent this from occurring diode and transistor clamps are used wherever higher voltages could possibly arise.

A READ is performed as follows. The axis channel is selected by writing a 3-bit code (BD0,1,2, and BD4,5,6) into the ADCs. The ALE (Address Latch Enable) signal latches these codes into the ADC internal registers. These codes select the multiplexer position. A start conversion signal is then generated. The ADC takes about 100 microseconds to perform the conversion. A test can be made for the end of conversion (EOC). By generating the EOCT signal both EOCs can be tested as both ADCs convert at the same time. A high indicates end of conversion (see IC101). When the conversion is complete then one or both of the ADCs can be READ. The ADCs have tri-state outputs which are enabled by the OEF and OEL signals (active high). The READ is then performed by generating a VALID READ signal and an address code to select one of the two ADCs. This enables the tri-state output of the selected ADC and also generates a DIR signal which sends data out from the interface board onto the computer's data bus. The computer can then READ the axis data.

NEXT MONTH: MENTOR construction

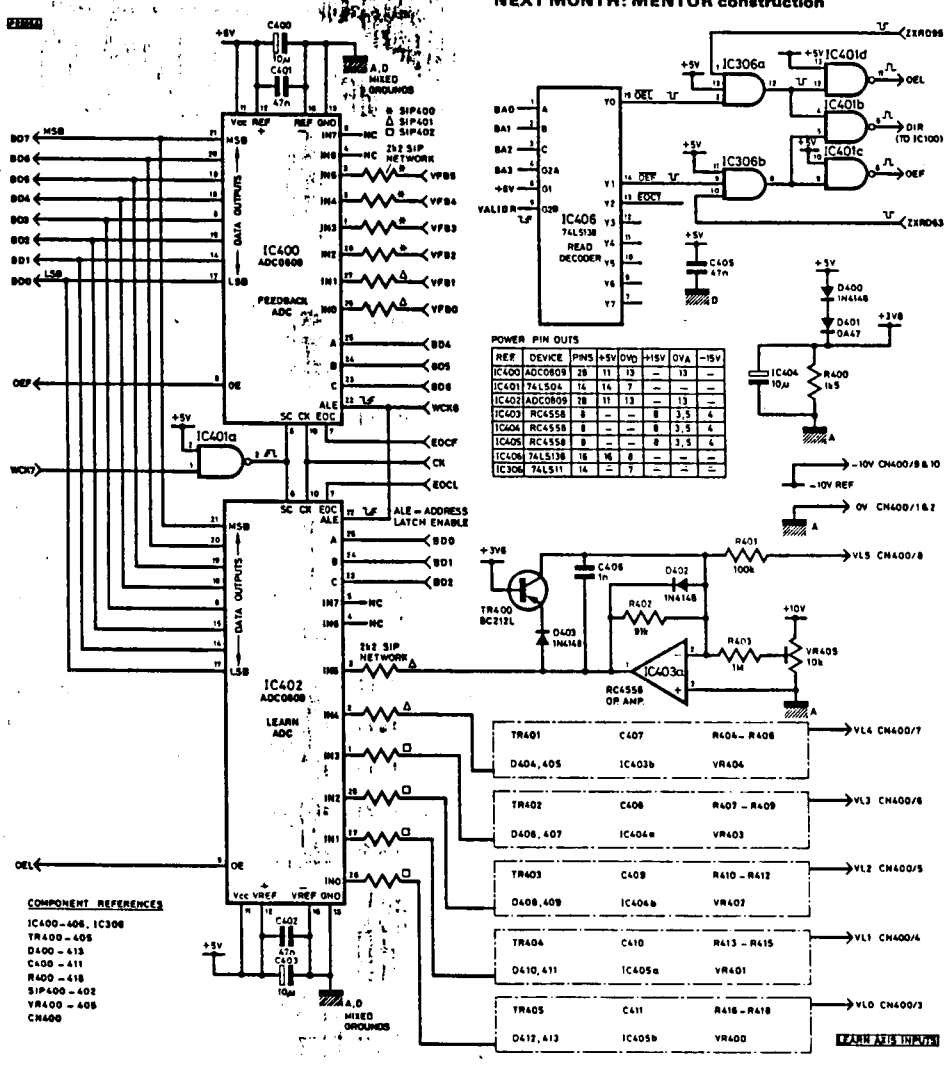


Fig. 4.8. Analog-to-digital conversion for a "Read" operation

APPENDIX B

ROBOT JOINT CONTROLLER SOFTWARE

```
;
; rcb.asm
;
; last revision date: 6-23-86
;
; MASM version of share memory handler
;
; This handler is a resident task that installs the buffer space to be
; shared between cooperating processes. The address of the start of the
; buffer is not fixed and can be determined by making software interrupt
; share_mem=80h.  (page 4.8 of ZDS programmer utilities.)
;
;
include robotdef.asm                    ; include robot control definitions
;
include \supp\msdos.def                 ; include DOS function call
;                                                           definitions
;
;
CSEG SEGMENT
          assume      cs:cseg,ds:cseg       ;defaults for MASM
          org         100h                  ;standard for COM files
start:    jmp         initialise            ;start here so that entry
                                            ; point is easy to remember
;
;reserve some space for old vector
;
old_share_mem_vec           dw   0          ; reserve 2 words for old
                            dw   0          ; vector value
;
; allocate the robot control blocks (rcb)
;  initializing the robot_id and robot base address
;
robot1    robot_control_block     <1,280h,0>
robot2    robot_control_block     <2,300h,0>
;
handler   proc   far                    ; returns address of shared buffer
          assume    cs:cseg,ds:cseg
          sti                           ; reenable interrupts

          mov  bx,cs                    ; return address of segment in es
          mov  es,bx                    ;
          mov  bx,offset robot1         ; return address of rcb's in bx

          iret                          ; end of interrupt routine
handler   endp                          ; end proc handler
```

```
initialise:     mov  bx,cs              ; dseg=cseg
                mov  ds,bx
;
; set up necessary vectors etc
;
;    save old interrupt vector
;
                mov  al,USER_RCB_INTR       ; location of new interrupt
                mov  ah,DOSF_INTVEC         ; funtion to get vector
                int  DOSI_FUNC              ; make DOS fn call

                mov  old_share_mem_vec,bx  ;old vec returned in ES:BX
                mov  old_share_mem_vec[2],es ;saved to workspace
;
                cli                        ;inhibit interrupts since
                                           ;the vector is being modified
;
;    set up new interrupt vector to this routine
;
                mov  bx,cs                  ; DOS fn wants vec in DS:DX
                mov  ds,bx
                mov  dx,offset handler
                mov  al,USER_RCB_INTR       ; interrupt for shared memory
                mov  ah,DOSF_SIVEC          ; set new interrupt vector
                int  DOSI_FUNC              ; new vector is now set up
;
                sti                         ;reenable interrupts
;
;   now create the resident handler
;
                mov  bx,cs                 ;tell DOS the first free position
                mov  ds,bx                 ; put seg:ofs in DS:DX
                mov  dx,offset last        ; could use initialise instead
                mov  ah,DOSF_KEEPPRC       ; terminate, but stay resident
                int  DOSI_FUNC             ; good-bye
last:           nop                        ; dummy instruction
;
CSEG ENDS
                END  start                    ;defining start address
;
```

```
;
; inirobot.asm
;
; last revision date: 9-02-86
;
; This routine initializes the rcb for each robot arm. It is intended
; to be run before starting the joint motion controller to insure the
; arms are in a known state.
;
; NOTE: the joint motion controller (driver.asm) actually puts the
; robot into its initial position. This routine only sets up the
; NEXT_ARM position in the rcb.
;
.xlist
include \supp\msdos.def           ; include the interrupt definitions
include \languages\assembler\macro.lib  ; include macro definitions
.list
;
include robotdef.asm          ; include the robot control definitions


CSEG SEGMENT
     assume    cs:cseg,ds:cseg
     org       100h

initialize     proc far
     assume    cs:cseg,ds:cseg
     mov     bx,cs                ;load segment registers
     mov     ds,bx                ; ds = cs
     jmp     start

init_table:
     db        12                 ; axis 0 setpoint (to put on the x-axis)
     db        0                  ; axis 1 setpoint
     db        mid_range          ; axis 2 setpoint
     db        mid_range          ; axis 3 setpoint
     db        mid_range          ; axis 4 setpoint
     db        25                 ; axis 5 setpoint (gripper)
```

```
start:
  ; get first rcb offset into base register

     int  USER_RCB_INTR          ; returns shared memory segment
                                 ; address in es and the first rcb
                                 ; offset in bx

  ; for all robots initialize the robot arm positions

     mov  cx,no_robots

robotloop:
     push cx                             ; save robot count
     mov  cx,no_axis                     ; number of axis to initialize
     lea  di,es:[bx].robot_next_arm      ; initialize next arm position
     mov  si,offset init_table           ; load from here
     rep  movsb                          ; load robot_next_arm positions

     setbit  es:[bx].robot_status,RCB_READY_FLAG
                                         ; data in next_arm is good

     pop  cx                             ; restore robot counter
     add  bx,rcb_size                    ; point to next rcb
     loop robotloop

     mov  ah,DOSF_EXIT
     int  DOSI_FUNC                      ; DOS function call to exit

initialize endp

CSEG  ENDS
        END initialize
```

```
;
; driver.asm
; revision date: 9-30-86
;      allow more leeway with grippers; plus/minus five
;
; revision date: 7-31-86
;      fix setpoints when out of range
;      slow it down, give robots time to move
;
; This is the joint motion controller. It initiates and monitors
; movement of the individual robot joints. It communicates with
; the PASCAL robot control system through a shared memory mechanism
; called a robot control block (RCB).
;
.xlist
include \supp\msdos.def           ; include the interrupt definitions
include \supp\z150rom.def         ; & clock interrupt definition
include \language\assembler\macro.lib   ; include macro definitions
.list
;
include robotdef.asm          ; include the robot control definitions
;
; define local flags
;
driver_busy     EQU       1
out_of_range    EQU       2
move_abort      EQU       4

CSEG SEGMENT
         assume     cs:cseg,ds:cseg
         org        100h
begin:   jmp        initialize      ; install the driver to be
                                     ;               interrupt driven


; set aside data region

old_tick_vec  dw    0              ; reserve 2 words for old vector
              dw    0
local_flags   db    0              ; flags for internal handler use
local_count   db    1              ; execution handler only every
                                     ;                          other time.
```

```
;
; robotdef.asm (6-23)
;    This file contains all the definitions associated with the
;    robots for the joint motion controller (driver.asm).
;
; define the user interrupt to reach this routine
;
USER_RCB_INTR       equ   80h           ;user interrupt for address of rcb's
;
; layout of the robot interface mechanism
;
;  base address + 0h: axis 0
;               + 1h: axis 1
;               + 2h: axis 2
;               + 3h: axis 3
;               + 4h: axis 4
;               + 5h: axis 5
;               + 6h: mpx port
;               + 7h: adc start
;               +10h: adc simulator result
;               +11h: adc robot arm result
;
;
; define the robot control block (rcb) for use by ASM programs
;
robot_control_block struc
  robot_id            db    ?             ;unique robot id - numeric
  robot_base          dw    ?             ;port address for robot
  robot_status        db    ?             ;status flags  - defined below
  robot_last_arm      db    6 DUP (?)     ;last joint positions read (6 axis)
  robot_next_arm      db    6 DUP (?)     ;joint setpoints / next position
  robot_simu_arm      db    6 DUP (?)     ;joint readings from simulator arm
robot_control_block ends
;
; define the flag bits of the robot status byte
;
RCB_READY_FLAG      EQU   1             ;next move is in next_arm field
RCB_BUSY_FLAG       EQU   2             ;a move is in progress
RCB_SETPOINT_FLAG   EQU   4             ;setpoint has been boosted
RCB_ERROR_FLAG      EQU   8             ;an error has been detected
;
; other parameters for robot control
;
rcb_size            equ   22            ; size of robot control block
no_robots           equ   2             ; number of robots in system
no_axis             equ   6             ; number of axis per robot
mid_range           equ   128           ; mid point positions for joints
multiplexor_offset  equ   6             ; offsets from robot base address
adc_start_offset    equ   7
adc_sim_result_offset equ 10h
adc_arm_result_offset equ 11h
;
; end of robotdef.asm
```

```
;*************************************************************************
;
; DRIVER ROUTINE
;
;   This procedure does the following:
;    1. sets busy bit to allow only one copy to run at a time
;    2. gets address of shared memory containing the robot control
;       blocks
;    3. For each robot,
;       starts a new move or monitors the current move
;    4. exits til next clock tick
;
;*************************************************************************

DRIVER   proc far
         assume    cs:cseg,ds:cseg

  ; save all registers

         push    ds
         push    es
         push    ss
         push    ax
         push    bx
         push    cx
         push    dx
         push    di
         push    si

  ; and proceed

         mov     bx,cs                ;load segment registers
         mov     ds,bx                ; ds = cs

  ; don't run every time, give robot time to move

         dec     local_count
         cmp     local_count,0
         jnz     driver_exit

         mov     local_count,10    ; set up delay

  ; if driver is already busy, just quit

         testbit  local_flags,driver_busy
         bset     driver_exit

         setbit   local_flags,driver_busy
                                   ; if not busy before, we are now

  ; get first rcb offset into base register

         int  USER_RCB_INTR        ; returns shared memory segment
                                   ; address in es and the first rcb
                                   ; offset in bx
```

```
  ; for all robots, check the robot arm positions

        mov  cx,NO_ROBOTS

ROBOT_LOOP:

    ; if not READY, then this robot is not moving
        testbit es:[bx].robot_status,rcb_ready_flag
        bclr    next_robot

    ; if not BUSY, then get this robot going
        testbit es:[bx].robot_status,rcb_busy_flag
        bclr    start

    ; if READY & BUSY, then see if movement is happening
        call    monitor_move
        jmp     next_robot

START:
        call    start_move              ; start next robot move

NEXT_ROBOT:
        add     bx,rcb_size             ; point to next rcb
        loop    robot_loop

        clrbit  local_flags,driver_busy  ;finished with this pass

DRIVER_EXIT:

    ; restore registers

        pop   si
        pop   di
        pop   dx
        pop   cx
        pop   bx
        pop   ax
        pop   ss
        pop   es
        pop   ds

        iret                            ; wait til next clock tick

DRIVER endp
```

```
;*************************************************************************
;
; START_MOVE
;   This procedure first saves the current position of the robot
;   joints and then sends the new setpoints to the robot joints.
;
;*************************************************************************

START_MOVE proc

        push    cx          ; save robot counter
        push    di
        push    si

  ; set up new move

        setbit  es:[bx].robot_status,RCB_BUSY_FLAG

  ; get the starting position of the robot arm
        call    read_robot_arm    ; current arm position

  ; and set last = starting position
        call    update_last

  ; output new arm positions to the robot
        call    write_robot_arm_next

  ; restore registers used
        pop     si
        pop     di
        pop     cx

        ret

START_MOVE endp
```

```
;*********************************************************************
;
; MONITOR MOVE
;   This procedure checks the progress of one robot arm. There are
;   four possible cases as follows:
;     1. CASE 1 - movement is complete
;                 (current = next or within one)
;        ACTION - clear READY, BUSY, and SETPOINT flags
;     2. CASE 2 - movement is not complete and arm is still moving,
;                 (at least one axis reading has changed by more than
;                 one since the last reading).
;        ACTION - no action, wait for next time
;     3. CASE 3 - movement is not complete and arm is not moving and
;                 setpoint has not been altered.
;        ACTION - boost setpoint by half the difference between
;                 current and next.
;     4. CASE 4 - movement is not complete, arm is not moving,
;                 setpoint has been altered, and arm is still not
;                 within tolerance
;        ACTION - set error flag, stop all robots
;
;*********************************************************************

MONITOR_MOVE proc
   ; save registers used
        push    cx
        push    di
        push    si

  ; see if movement is happening

        call    read_robot_arm      ; find out where the arm is now

;CASE 1
  ; if all axis have reached setpoint within one,
  ;    clear status and return

        lea     di,es:[bx].robot_next_arm     ; offset within rcb
        lea     si,current_arm                ; temporary buffer

        call    check_variance        ; between current position
                                      ;                 and setpoint

        testbit local_flags,out_of_range
        bset    case_2

  ; move is complete, clear the status flags

        clrbit  es:[bx].robot_status,RCB_SETPOINT_FLAG
        clrbit  es:[bx].robot_status,RCB_BUSY_FLAG
        clrbit  es:[bx].robot_status,RCB_READY_FLAG
        call    update_last
        jmp     monitor_return
```

```
CASE_2:
    ; axis movement is not complete,
    ;  and at least one axis is still moving
    ;  (last is different from current by more than one),
    ;  just wait some more

        lea     di,es:[bx].robot_last_arm
        lea     si,current_arm

        call    check_variance      ; between last reading & current
        testbit local_flags,out_of_range
        bclr    CASE_3              ; branch if no movement detected

        call    update_last             ; set last reading = current
        jmp     monitor_return


CASE_3:
    ; movement is not complete, yet no movement is detected
    ; try boosting setpoint (this is done only once)

        testbit es:[bx].robot_status,RCB_SETPOINT_FLAG
        bset    TROUBLE                         ; branch if already tried
                                                ; boosting the setpoint

        setbit  es:[bx].robot_status,RCB_SETPOINT_FLAG
        call    update_last                     ; be sure and keep
                                                ;         current reading

    ; compute setpoint boost value, first find difference between
    ; where joint is and where it should be. Boost by half this
    ; value.  Note: if the difference is zero or one, there is no
    ; change in the setpoint.

        lea     di,es:[bx].robot_next_arm
        lea     si,current_arm
        mov     cx,no_axis

compute_new_setpoints:
        mov     ax,0                ; use fullword since rv's
        mov     dx,0                ;    are positive numbers
        mov     al,es:[di]          ; get setpoint
        mov     dl,[si]             ; get current reading
        sub     ax,dx               ; subtract current reading
        sar     ax,1                ; divide by two, giving boost amt
        mov     dl,es:[di]          ; get setpoint again
        add     ax,dx               ;             and add to boost amt

  ; make sure new setpoints are within valid range

        cmp     ax,255                      ; if new setpoint >255
        jle     cmplow                      ; then new setpoint = 255
        mov     ax,255
cmplow:
        cmp     ax,0                        ; if new setpoint < 0
        jge     set_okay                    ; then new setpoint = 0
        mov     ax,0
```

```
set_okay:
        mov     [si],al                         ; put new setpoints into
                                                ;       current positions
        inc     di                              ; point to next axis
        inc     si                              ;
        loop    compute_new_setpoints           ; do next axis

        call    write_robot_arm_current         ; send new joint values
                                                ;               to robot
        jmp     monitor_return                  ; and wait some more

TROUBLE:
  ; trouble in ROBOT CITY
        setbit  es:[bx].robot_status,RCB_ERROR_FLAG
        call    ABORT                           ;STOPS ALL ROBOTS
        jmp     monitor_return                  ; rest of robots will be
                                                ; scanned, but no action
                                                ; taken because READY
                                                ; flags have been cleared

monitor_return:
  ; restore registers used
        pop     si
        pop     di
        pop     cx
        ret

MONITOR_MOVE endp




;*************************************************************************
;
; CHECK_VARIANCE
;
; This routine compares two series of axis readings to see if they
; are within a valid range of one another (plus/minus one, plus/minus
; five for the gripper). If any of the axis readings are out of range
; the out_of_range flag is set in the local flags field.
;
; The values of si and di are destoyed.
;
;*************************************************************************

CHECK_VARIANCE proc
        push    cx
        push    ax
        push    dx

        clrbit  local_flags,out_of_range ; assume all is okay
        mov     cx,NO_AXIS              ; number of axes
                                        ;       to be checked
        dec     cx                      ; do gripper separately
```

```
check_next:

    ; see if equal or within one point of setpoint

        mov     ax,0                ; clear out so can use whole word
        mov     dx,0
        mov     al,es:[di]          ; get setpoint
        mov     dl,[si]
        sub     ax,dx               ; subtract current position

        cmp     ax,1
        jle     c1                  ; branch if within range

        setbit  local_flags,out_of_range

c1:     cmp     ax,-1
        jge     c2

        setbit  local_flags,out_of_range

c2:     inc     si
        inc     di

        loop    check_next          ; must have been within one of
                                    ; the setpoint, continue with
                                    ; rest of axes

    ; see if gripper equal or within five points of setpoint

        mov     ax,0                ; clear out so can use whole word
        mov     dx,0
        mov     al,es:[di]          ; get setpoint
        mov     dl,[si]
        sub     ax,dx               ; subtract current position

        cmp     ax,5
        jle     c3                  ; branch if within range

        setbit  local_flags,out_of_range

c3:     cmp     ax,-5
        jge     c4

        setbit  local_flags,out_of_range

c4:
    ; restore registers
        pop     dx
        pop     ax
        pop     cx
        ret

CHECK_VARIANCE endp
```

```
;*************************************************************************
;
; UPDATE_LAST
;
; This routine updates the rcb last reading with current_arm.
;
;*************************************************************************

UPDATE_LAST proc

  ; set last = current
        mov     cx,no_axis
        lea     di,es:[bx].robot_last_arm
        lea     si,current_arm
        rep movsb
        ret

UPDATE_LAST endp




;*************************************************************************
;
; ABORT routine
;       - Stops movement of all arms
;       - Clears the READY, BUSY, and SETPOINT flags in all RCB's
;
;*************************************************************************

ABORT   proc
  ; save registers used
        push    cx

  ; stop all robots
        int     USER_RCB_INTR           ; point to first RCB again
        mov     cx,no_robots

abort_loop:
  ; set all robots to current position (stopping all movement)
        call    read_robot_arm
        call    write_robot_arm_current
        clrbit  es:[bx].robot_status,RCB_READY_FLAG
        clrbit  es:[bx].robot_status,RCB_BUSY_FLAG
        clrbit  es:[bx].robot_status,RCB_SETPOINT_FLAG
        add     bx,rcb_size                 ; point to next rcb
        loop    abort_loop

  ; restore registers used
        pop     cx
        ret
ABORT   endp
```

```
include PRIMITIVES.ASM
;
;*****************************************************************
;
; INITIALIZE routine
;   this routine sets up the driver initially by replacing any
;   old user clock interrupt handler with this handler to monitor
;   the robot arms
;
;*****************************************************************

INITIALIZE:
        mov  bx,cs          ; ds = cs
        mov  ds,bx

  ; set up interrupt handler for user clock interrupt

        mov  al,USER_CLOCK_INTR
        mov  ah,DOSF_INTVEC       ; get old vector
        int  DOSI_FUNC            ; DOS function call
        mov  old_tick_vec,bx      ; save old vector
        mov  old_tick_vec[2],es

        cli                       ; inhibit interrupts a moment
        mov  bx,cs
        mov  ds,bx                ; DOS expects vector in DS:DX
        mov  dx,offset driver
        mov  al,USER_CLOCK_INTR
        mov  ah,DOSF_SIVEC        ; set new clock vector
        int  DOSI_FUNC            ; DOS function call

        sti                       ; should be able to deal with
                                  ;                    interrupts now

        mov  bx,cs                ; ds=cs
        mov  ds,bx
        mov  dx,offset initialize ; don't need initialize anymore
        int  DOSI_TERMR           ; terminate, but stay resident

CSEG  ENDS
        END begin
```

```
;
; PRIMITIVES.ASM
;
; last revision date: 7-23-86
;
; this file contains the following robot primitives
;   READ_ROBOT_ARM
;   READ_SIMULATOR
;   WRITE_ROBOT_ARM_CURRENT
;   WRITE_ROBOT_ARM_NEXT
;   ADC_WAIT
;
;
; define local variables

axis_count     db    0
current_arm    db    6 DUP (?)     ; temporary buffer for axis positions
```

```
;*************************************************************************
;
; READ_ROBOT_ARM
;   reads the current position of each axis into a temporary buffer
;   called current_arm.
;
;*************************************************************************

READ_ROBOT_ARM proc

  ; save registers used
        push    cx
        push    dx
        push    ax
        push    di

  ; read each axis
        mov     axis_count,0            ; axis address on robot
        mov     cx,no_axis              ; number of axes to be read
        mov     di,0                    ; axis count

read_next_robot_axis:
        mov     dx,es:[bx].robot_base   ; get base address of robot
        add     dx,multiplexor_offset
        mov     al,axis_count           ; set up axis to read
        out     dx,al                   ; send axis number to
                                        ;               multiplexor
        inc     dx                      ; adc conversion port is
                                        ;     in the next position
        mov     al,0
        out     dx,al                   ; start conversion
        call    adc_wait

        mov     dx,es:[bx].robot_base   ; get base address again
        add     dx,adc_arm_result_offset
        in      al,dx                   ; put axis position into al
                                        ;   register and save in
        mov     current_arm[di],al      ;   temporary buffer
        inc     di                      ; point to next offset in
                                        ;   the buffer
        add     axis_count,16           ; next axis address
        loop    read_next_robot_axis

  ; restore registers used
        pop     di
        pop     ax
        pop     dx
        pop     cx

        ret

READ_ROBOT_ARM endp
```

```
;*************************************************************************
;
; READ_SIMULATOR
;   reads the position of the simulator into rcb (in shared memory)
;   location robot_next_arm.
;
;*************************************************************************

READ_SIMULATOR proc

  ; save registers used
        push    cx
        push    dx
        push    ax
        push    di

  ; read each axis
        mov       di,0            ; axis counter
        mov       cx,no_axis

read_next_sim_axis:
        mov       dx,es:[bx].robot_base
        add       dx,multiplexor_offset
        mov       ax,di           ; set up axis to read
        out       dx,al
        inc       dx              ; adc conversion port is
                                  ;                 in next position
        mov       al,0
        out       dx,al           ; start conversion
        call      adc_wait

        mov       dx,es:[bx].robot_base
        add       dx,adc_sim_result_offset
        in        al,dx
        mov       es:[bx+di].robot_next_arm,al      ; save results
        inc       di
        loop      read_next_sim_axis

  ; restore registers used
        pop     di
        pop     ax
        pop     dx
        pop     cx

        ret

READ_SIMULATOR endp
```

```
;*************************************************************************
;
; WRITE_ROBOT_ARM_NEXT
;   sends to the robot the robot positions in the rcb (in shared
;   memory) location robot_next_arm.
;
;*************************************************************************

WRITE_ROBOT_ARM_NEXT proc

  ; save registers used
        push    cx
        push    dx
        push    ax
        push    si

  ; write out each axis
        mov     cx,no_axis
        mov     si,0                            ; axis count
        mov     dx,es:[bx].robot_base           ; get robot base

write_next_robot_axis:
        mov     al,es:[bx+si].robot_next_arm ; get value to output
        out     dx,al                           ; and send it
        inc     dx                      ; next axis is next port number
        inc     si                              ; next axis number
        loop    write_next_robot_axis

 ; restore registers used
        pop     si
        pop     ax
        pop     dx
        pop     cx

        ret

WRITE_ROBOT_ARM_NEXT endp
```

```
;*************************************************************************
;
; WRITE_ROBOT_ARM_CURRENT
;   sends to the robot the positions in the temporary buffer
;   current_arm.
;
;*************************************************************************

WRITE_ROBOT_ARM_CURRENT proc

  ; save registers used
        push    cx
        push    dx
        push    ax
        push    si

  ; write out each axis
      . mov       cx,no_axis
        mov       si,0                  ; axis count
        mov       dx,es:[bx].robot_base

write_curr_robot_axis:
        mov       al,current_arm[si]    ; get next value to output
        out       dx,al                 ; and send it
        inc       dx                   ; next axis is next port number
        inc       si                    ; next axis number
        loop      write_curr_robot_axis

  ; restore registers used
        pop     si
        pop     ax
        pop     dx
        pop     cx

        ret

WRITE_ROBOT_ARM_CURRENT endp
```

```
;*************************************************************************
;
; ADC_WAIT
;   wait about 100 microseconds for conversion to complete
;
;*************************************************************************

ADC_WAIT proc
         push      cx
         mov       cx,25           ; adc conversion takes
                                   ;              about 100 microsecs
adc_wait_loop:
         nop                       ; note - each nop uses 3 cycles
         loop      adc_wait_loop   ;      - each loop takes 17 cycles
         pop       cx
         ret
ADC_WAIT endp
```

```
;
; MACRO.LIB
;
;  A LIBRARY OF USEFUL MACRO ROUTINES INCLUDING:
;     SETBIT  - set bit
;     CLRBIT  - clear bit
;     TESTBIT - test bit
;     BCLR    - branch if bit clear
;     BSET    - branch if bit set
;
;
; THE FOLLOWING ARE BIT MANIPULATION MACROS
;
SETBIT    MACRO     FIELD,BIT
          OR        FIELD,BIT       ;SET BIT
          ENDM

CLRBIT    MACRO     FIELD,BIT
          OR        FIELD,BIT       ;SET THE BIT IN THE FIELD
          XOR       FIELD,BIT       ;NOW, KEEP ALL THE OTHERS
          ENDM

TESTBIT   MACRO     FIELD,BIT
          TEST      FIELD,BIT
          ENDM

BCLR      MACRO     LABEL
          JZ        LABEL
          ENDM

BSET      MACRO     LABEL
          JNZ       LABEL
          ENDM
;
; END OF BIT MANIPULATION MACROS
;
```

APPENDIX C

ROBOT PROGRAMMING AND CONTROL SOFTWARE

```
{ ROBOTDEF.PAS }

{ This file contains the environment parameters and descriptor types    }
{ useful to the programming and control software. It also contains the }
{ constants necessary to reference the RCB fields.                      }

Const
     NumberOfRobots   = 2;
     NumberOfObjects  = 2;
     NumberOfRegions  = 4;          (* one per major axis, & one for object *)
     NumberOfCenters  = 5;

     USER_RCB_INTR    = $80;        (* interrupt to get rcb addresses *)
     RCB_LENGTH       = 22;         (* length of a robot control block *)
     STATUS           = 3;          (* status offset in rcb *)
     NEXT             = $A;         (* next setpoints offset in rcb *)
     READY            = 1;          (* READY bit in rcb status field *)
     ERROR            = 8;          (* ERROR bit in rcb status field *)

     APPROACH         = 1;          (* FLAG definitions for step descriptor *)
     PICKUP           = 2;
     HOLD             = 3;
     RELEASE          = 4;

Type
   regpack =
         record
            AX,BX,CX,DX,BP,SI,DI,DS,ES,flags: Integer;
         end;

   vector =                         (* describe a point in space *)
         record
            x, y, z :Real
         end;

   region = array[1..8] of vector; (* 8 points defining a region in space *)

   collision_region = array[1..NumberOfRegions] of region;

   step_descriptor =                (* one step in a robot program or task *)
         record
            robot_id:   integer;    (* identify robot descriptor used *)
            step_id:    integer;    (* identify step number *)
            step_status: integer;
            step_object: integer;   (* identify object associated robot *)
            pos_vector: vector;     (* compute joint_angles from this *)
            pitch,                  (* rotation about y-axis in degrees *)
            twist,                  (* rotation about z-axis in degrees *)
            gripper:    real;       (* open/close of gripper in mm *)
            joint_angle: array[0..3] of real;       (* angles in radians *)
            joint_value: array[0..5] of integer;    (* robot joint values *)
            step_delay:  integer;   (* seconds to stay in this step *)
         end;

   substeps = array [0..13] of step_descriptor;
```

```
   robot_descriptor =                (* one per robot *)
         record
            robot_id:     integer;  (* identify robot *)
            axis_center:  array[0..5] of vector;
            axis_length:  array[0..5] of real;
            axis_limit:   array[0..5] of real;
            axis_range:   array[0..5] of real;
            axis_rvs:     array[0..5] of integer;
            axis_fudge:   array[0..5] of integer;
            axis_region:  collision_region;
         end;

   object_descriptor =               (* one per object *)
         record
            object_id: integer;     (* number objects in environment *)
            object_region: region; (* describe object *)
         end;

   filename = string[12];

   lines = array[1..12, 1..2] of integer;     (* 12 lines per region *)
                                              (* each defined by 2 endpoints *)

Const
   Endpoint: lines = ((1,2), (2,3), (3,4), (4,1),
                      (5,6), (6,7), (7,8), (8,5),
                      (1,5), (2,6), (3,7), (4,8));
```

```
{ MENU.PAS }
{
  This file contains the main menu of the programming and control
  software. Most of the major procedures are defined in independent
  files which are included here. Because includes may not be nested,
  all includes must be done at this level. Also, the robot and
  object descriptors are allocated and initialized here before
  the menu is displayed.
}

Program menu;
{$I ROBOTDEF.PAS}                    (* robot data types *)

Var
 (* variables intended for local usage *)
   R,                                 (* robot counter *)
   O,                                 (* object counter *)
   selection     :integer;            (* menu selection number *)

 (* variables intended for global usage *)
   robot         : array[1..NumberOfRobots]  of robot_descriptor;
   object        : array[1..NumberOfObjects] of object_descriptor;
   RobotTask     : array[1..NumberOfRobots]  of file of step_descriptor;

(* include pascal procedures as required *)
(* note that includes may NOT be nested - what a pain *)

{$I MATHFUNC.PAS}              (* low-level math functions and procedures *)
{$I MISCFUNC.PAS}              (* low-level collision avoidance functions *)
{$I TRANSFORM.PAS}             (* convert pos & orient to joint coordinates *)
{$I ENTERTASK.PAS}             (* enter a robot task *)
{$I VIEWTASK.PAS}              (* view  a robot task *)
{$I SINGLETASK.PAS}            (* run single robot task *)
{$I ROTREG.PAS}                (* rotate cart coordinates to step position *)
{$I OVERLAP.PAS}               (* check region projections for overlap *)
{$I CHKTASK.PAS}               (* compare robot tasks for runability *)
{$I SUBSTEPS.PAS}              (* computes substeps for each task *)
{$I RUNDUAL.PAS}               (* build substeps & run dual tasks *)
{$I DUALDRIVER.PAS}            (* check and initiate run of dual tasks *)
{$I INITROBOT.PAS}             (* fill in static values in robot descriptor *)
{$I INITOBJECT.PAS}            (* fill in object descriptions *)
```

```
Begin (* menu *)
      For R := 1 to NumberOfRobots do
      begin
           Init_robot_descriptor(robot[R], R);
      end;

      For O := 1 to NumberOfObjects do
      begin
           Init_object_descriptor(object[O], O);
      end;

      selection := -1;
      While (selection <> 0) do
      begin
           Writeln;
           Writeln ('0. Exit');
           Writeln ('1. Enter Robot Task');
           Writeln ('2. View  Robot Task');
           Writeln ('3. Run Single Robot Task');
           Writeln ('4. Run Dual Robot Tasks');
           Writeln;
           Write ('Enter selection number: ');
           Readln (Selection);
           Case Selection of
                1: enter_task;
               2: view_task;
               3: single_task;
               4: dual_driver;
              else ;
          end;
     end; (* while *)
end. (* menu *)
```

```
{ INITROBOT.PAS }
{
 This procedure initializes the robot characteristics including:
  1. robot id number
  2. joint centers, lengths, ranges, joint values, and limits
  3. collision region vertices for each major arm link and for
     one possible object.
}
{ last revision: 10-02-86 }

procedure init_robot_descriptor(Var robot_description: robot_descriptor;
                                    robot_number: integer);
begin
   with robot_description do
   begin
      robot_id := robot_number;

   (* define axis centers as points in space *)
      with axis_center[0] do
      begin
         x := 0; y := 0; z := 0;          (* origin *)
      end;
      with axis_center[1] do              (* also fixed *)
      begin
         x := 0; y := 0; z := 170;
      end;
      with axis_center[2] do              (* starting point, all axes at 0 *)
      begin
         x := 165; y := 0; z := 170;
      end;
      with axis_center[3] do              (* starting point, all axes at 0 *)
      begin
         x := 90; y := 0; z := 40;
      end;
      with axis_center[4] do              (* unused *)
      begin
         x := 90; y := 0; z := 40;
      end;
      with axis_center[5] do              (* used for object translation *)
      begin
         x := -13; y := 0; z := 56;
      end;

   (* length from center to center *)
      axis_length[0] := 170;             (* in millimeters *)
      axis_length[1] := 165;
      axis_length[2] := 150;
      axis_length[3] := 105;
      axis_length[4] := 0;               (* unused *)
      axis_length[5] := 0;               (* unused *)

   (* range of movement of each axis, in radians *)
      axis_range[0] := 3.66;              (* 210 degrees *)
      axis_range[1] := 3.14;              (* 180 degrees *)
      axis_range[2] := 4.19;              (* 240 degrees (230 in manuals) *)
      axis_range[3] := 2.44;              (* 140 degrees *)
      axis_range[4] := 2.44;              (* 140 degrees *)
      axis_range[5] := 30;               (* this one is millimeters *)
```

```
(* robot values allowed for above ranges *)
   axis_rvs[0] := 255;
   axis_rvs[1] := 255;
   axis_rvs[2] := 255;
   axis_rvs[3] := 127;
   axis_rvs[4] := 127;
   if robot_id = 1 then axis_rvs[5] := 55;
   if robot_id = 2 then axis_rvs[5] := 90;

(* angle of axis relative to previous axis when rv = 0 *)
   axis_limit[0] := 0;              (* in radians *)
   axis_limit[1] := 1.57;           (*  90 degrees *)
   axis_limit[2] := 1.05;           (*  60 degrees, (65 in manuals) *)
   axis_limit[3] := 1.92;           (*  110 degrees *)
   axis_limit[4] := 1.92;           (*  same as axis 3 *)
   axis_limit[5] := 0;              (* unused *)

(* fudge factor so robots start at x=0 location *)
   axis_fudge[0] := 12;             (* x = 0 when rv = 12 *)
   axis_fudge[1] := 0;
   axis_fudge[2] := 0;
   axis_fudge[3] := 64;             (* axis 3 & 4 use rv range 64-194 *)
   axis_fudge[4] := 64;             (* the rest allows for twist *)
   if robot_id = 1 then axis_fudge[5] := 90;
   if robot_id = 2 then axis_fudge[5] := 10;

(* vertices of collision regions when rv = 0 *)
(* shoulder *)
   axis_region[1,1].x := -175;
   axis_region[1,1].y :=  -50;
   axis_region[1,1].z :=   90;
   axis_region[1,2].x :=  200;
   axis_region[1,2].y :=  -50;
   axis_region[1,2].z :=  130;
   axis_region[1,3].x :=  200;
   axis_region[1,3].y :=  -50;
   axis_region[1,3].z :=  210;
   axis_region[1,4].x := -175;
   axis_region[1,4].y :=  -50;
   axis_region[1,4].z :=  250;
   axis_region[1,5].x := -175;
   axis_region[1,5].y := -180;
   axis_region[1,5].z :=   90;
   axis_region[1,6].x :=  200;
   axis_region[1,6].y := -180;
   axis_region[1,6].z :=  130;
   axis_region[1,7].x :=  200;
   axis_region[1,7].y := -180;
   axis_region[1,7].z :=  210;
   axis_region[1,8].x := -175;
   axis_region[1,8].y := -180;
   axis_region[1,8].z :=  250;
```

```
    (* forearm *)
       axis_region[2,1].x :=  168;
       axis_region[2,1].y := -100;
       axis_region[2,1].z :=  306;
       axis_region[2,2].x :=   41;
       axis_region[2,2].y := -100;
       axis_region[2,2].z :=   45;
       axis_region[2,3].x :=  120;
       axis_region[2,3].y := -100;
       axis_region[2,3].z :=    0;
       axis_region[2,4].x :=  281;
       axis_region[2,4].y := -100;
       axis_region[2,4].z :=  242;
       axis_region[2,5].x :=  168;
       axis_region[2,5].y :=  100;
       axis_region[2,5].z :=  306;
       axis_region[2,6].x :=   41;
       axis_region[2,6].y :=  100;
       axis_region[2,6].z :=   45;
       axis_region[2,7].x :=  120;
       axis_region[2,7].y :=  100;
       axis_region[2,7].z :=    0;
       axis_region[2,8].x :=  281;
       axis_region[2,8].y :=  100;
       axis_region[2,8].z :=  242;

    (* end-effector *)
       axis_region[3,1].x :=  78;
       axis_region[3,1].y := -50;
       axis_region[3,1].z :=  87;
       axis_region[3,2].x := -21;
       axis_region[3,2].y := -50;
       axis_region[3,2].z := 103;
       axis_region[3,3].x := -35;
       axis_region[3,3].y := -50;
       axis_region[3,3].z :=  14;
       axis_region[3,4].x :=  64;
       axis_region[3,4].y := -50;
       axis_region[3,4].z := -01;
       axis_region[3,5].x :=  78;
       axis_region[3,5].y :=  50;
       axis_region[3,5].z :=  87;
       axis_region[3,6].x := -21;
       axis_region[3,6].y :=  50;
       axis_region[3,6].z := 103;
       axis_region[3,7].x := -35;
       axis_region[3,7].y :=  50;
       axis_region[3,7].z :=  14;
       axis_region[3,8].x :=  64;
       axis_region[3,8].y :=  50;
       axis_region[3,8].z := -01;

    (* possible object *)
       zero_region(axis_region[4]);      (* reserved for objects *)

    end;
end;
```

```
{ INITOBJECT.PAS }
{
 This procedure initialized object id numbers and the vertices
 defining their shape.
}
{ last revision: 10-01-86 }

procedure init_object_descriptor(Var object_description: object_descriptor;
                                 object_number: integer);
begin
   with object_description do
   begin
      object_id := object_number;

   (* vertices of object regions when at rest *)
     If object_id = 1 then            (* Nerf Ball *)
     begin
      object_region[1].x := -50;
      object_region[1].y := -50;
      object_region[1].z :=  20;
      object_region[2].x := -50;
      object_region[2].y := -50;
      object_region[2].z := -80;
      object_region[3].x :=  50;
      object_region[3].y := -50;
      object_region[3].z := -80;
      object_region[4].x :=  50;
      object_region[4].y := -50;
      object_region[4].z :=  20;
      object_region[5].x := -50;
      object_region[5].y :=  50;
      object_region[5].z :=  20;
      object_region[6].x := -50;
      object_region[6].y :=  50;
      object_region[6].z := -80;
      object_region[7].x :=  50;
      object_region[7].y :=  50;
      object_region[7].z := -80;
      object_region[8].x :=  50;
      object_region[8].y :=  50;
      object_region[8].z :=  20;
     end;
```

```
      If object_id = 2 then              (* protractor *)
      begin
       object_region[1].x :=  -85;
       object_region[1].y :=  -10;
       object_region[1].z :=   20;
       object_region[2].x :=  -85;
       object_region[2].y :=  -10;
       object_region[2].z :=  -80;
       object_region[3].x :=   85;
       object_region[3].y :=  -10;
       object_region[3].z :=  -80;
       object_region[4].x :=   85;
       object_region[4].y :=  -10;
       object_region[4].z :=   20;
       object_region[5].x :=  -85;
       object_region[5].y :=   10;
       object_region[5].z :=   20;
       object_region[6].x :=  -85;
       object_region[6].y :=   10;
       object_region[6].z :=  -80;
       object_region[7].x :=   85;
       object_region[7].y :=   10;
       object_region[7].z :=  -80;
       object_region[8].x :=   85;
       object_region[8].y :=   10;
       object_region[8].z :=   20;
      end;
   end;
end;
```

```
{ ENTERTASK.PAS }
{
 This procedure allows the operator to enter in a robot task
 using cartesian coordinates. The position and orientation of
 the end-effector are given for each step of the task.
}
{ last revision: 9-18-86 }

procedure enter_task;

Var
   taskname       :filename;
   RobotTask      :file of step_descriptor;
   next_step      :step_descriptor;       (* step being entered in *)
   RobotNo,                          (* which robot we're working with *)
   I              :integer;          (* current task step number *)
   step_valid     :boolean;          (* return from transform routine *)
   continue       :char;

begin (* enter task *)
     continue := 'y';
     next_step.step_id := 1;

     Write ('For which robot is the task intended (1 or 2): ');
     Readln (RobotNo);
     next_step.robot_id := RobotNo;

     Writeln;
     Write ('Enter task filename: ');
     Readln (taskname);
     Assign (RobotTask, taskname);
     Rewrite (RobotTask);

     While ((continue = 'y') or (continue = 'Y')) do     (* still more steps *)
     begin
        Writeln;
        Writeln ('Robot ', RobotNo:1, '  Step ', next_step.step_id:3);
        Writeln ('Enter position coordinates in millimeters:');

        get_step(next_step);
        Coord_Transform(robot[RobotNo], next_step, step_valid);
                                        (* fill in radians and robot values *)
        If (step_valid) then
          begin
             Write (RobotTask, next_step);
             next_step.step_id := next_step.step_id + 1;
          end
        else Writeln ('Robot cannot reach specified position');

        Writeln;
        Write ('Do you wish to continue: ');
        readln (continue);
     end; (* while *)
     Close (RobotTask);
end;
```

```
{ TRANSFORM.PAS }
{ last revision date: 9-03-86

 This routine does the inverse coordinate transformations for the robot
 control system. It transforms Cartesian coordinates specifying position
 and angles specifying orientation to robot joint coordinates and then
 to robot joint values.

 Arguments:
 1. robot descriptor of the robot for which this sequence is being developed.
 2. step descriptor which contains the degree of close for the robot tool,
    the angle of the robot tool relative to the last robot arm, and the
    position of the end of the robot tool relative to the base in cartesian
    coordinates.
 3. validity field in which this routines passes back to the calling routine
    the validity of the requested move. IE, whether the robot can reach
    the requested position or not.

 The following are defined in the step descriptor and are supplied by the user:
     pos_vector - the cartesian coordinate position of the end effector
     pitch    - up/down position of the gripper tool
     twist    - degree of rotation of the tool itself
     gripper - millimeters open of the gripper

 We want to find the following:
     tilt     - angle at joint D (that is, between the tool and the last arm)
     Axis0 angle - the angle at point A
     Axis1 angle - the angle at point B
     Axis2 angle - the angle at point C

 Also, convert all angles to radians
       and then to robot values
}

procedure coord_transform( robot_description: robot_descriptor;
                    Var step_description: step_descriptor;
                    Var validity: boolean);
var
   Tilt,                                (* angle at joint D in degrees *)
   LgthAE,                              (* distance between point A and point E *
   LgthBE,                              (* distance between point B and point E *
   LgthCE:        Real;                 (* distance between point C and point E *
   temp1,
   temp2:         Real;                 (* for use in half angles function *)
   partial1,                            (* part of an angle *)
   partial2:      Real;
   rv_twist,                            (* twist in robot values *)
   I:             Integer;

function s(var a,b,c:real): real;     (* one-half the perimeter of a triangle *
begin
     s := (a + b + c)/2;
end;

function r(var s,a,b,c:real): real;     (* used to simplify further expressions
)
begin
     r := sqrt( Abs(((s-a)*(s-b)*(s-c))/s) );
end;
```

```
(*************************************************************************)
begin (* transform *)

  validity := TRUE;                    (* assume this step is okay *)

  (* first compute angles (in radians) *)

  With robot_description, step_description do
  begin

  (* compute needed distances *)

     tilt   := pitch + 110;
     lgthAE := dist_points(axis_center[0], pos_vector);
     lgthBE := dist_points(axis_center[1], pos_vector);
     lgthCE := dist_cos(axis_length[2], axis_length[3], tilt);

  (*************************************************************)
  (* compute axis 0 angle *)

     with pos_vector do
     if x <> 0 then
        begin
          joint_angle[0] := Arctan (y/x);
          if (joint_angle[0] < 0) or ((y = 0) and (x < 0))
          then                  (* Quadrants II & IV *)
             joint_angle[0] := joint_angle[0] + PI;
        end
     else
          joint_angle[0] := PI/2;

  (*************************************************************)
  (* compute axis 1 angle *)

     temp1 := s(axis_length[0], lgthBE, lgthAE);
     temp2 := r(temp1, axis_length[0], lgthBE, lgthAE);

     If abs(lgthAE-temp1) < 0.001  then     (* special case - 180 degrees *)
         partial1 := PI
     else
         partial1 := 2 * arctan ( temp2/(temp1-lgthAE));

     temp1 := s(axis_length[1], lgthCE, lgthBE);
     temp2 := r(temp1, axis_length[1], lgthCE, lgthBE);

     partial2 := 2*arctan( temp2/(temp1-lgthCE) );

     joint_angle[1] := partial1 + partial2;

    (* If Quadrants III & IV, flip arm over to other side *)
     If pos_vector.y < 0 then joint_angle[1] := 2*PI - joint_angle[1];

     Joint_angle[1] := Joint_angle[1] - Axis_limit[1];
```

```
    (*************************************************************)
    (* compute axis 2 angle *)

       If abs(lgthBE-temp1) < 0.001 then      (* special 180 degrees case *)
          partial1 := PI
       else
          partial1 := 2 * arctan ( temp2/(temp1-lgthBE));

       temp1 := s(axis_length[2], axis_length[3], lgthCE);
       temp2 := r(temp1, axis_length[2], axis_length[3], lgthCE);

       partial2 := 2*arctan( temp2/(temp1-axis_length[3]) );

       If Tilt > 180 then joint_angle[2] := partial1 - partial2;
       If Tilt < 180 then joint_angle[2] := partial1 + partial2;
       If Tilt = 180 then joint_angle[2] := partial1;

      (* If Quadrants III & IV, flip arm over *)
       If pos_vector.y < 0 then joint_angle[2] := 2*PI - joint_angle[2];

       Joint_angle[2] := Joint_angle[2] - axis_limit[2];

    (*************************************************************)
    (* convert rest of angles to radians *)

      (* If Quadrants III & IV, pitch from below *)
       If pos_vector.y < 0
       then joint_angle[3] := deg_to_rad(140-pitch)
       else joint_angle[3] := deg_to_rad(pitch);

       joint_angle[4] := joint_angle[3];
       joint_angle[5] := gripper;             (* no angles here *)

    (*************************************************************)
    (* Change all angles to robot joint values and take in twist *)
      (* Note:                                                     *)
      (* pitch is obtained by rotating 3 and 4 in the same direction  *)
      (* twist is obtained by rotating 3 and 4 in opposite directions *)

       for I := 0 to 5 do
       begin
          joint_value[I] :=
            round (joint_angle[I] * (axis_rvs[I]/axis_range[I])) + axis_fudge[I];
       end;

       (* compute in twist *)

       rv_twist := 2 * round( deg_to_rad(twist) * (axis_rvs[4]/5.58));

       joint_value[3] := joint_value[3] + rv_twist;
       joint_value[4] := joint_value[4] - rv_twist;

       for I := 0 to 5 do
       begin
          If (joint_value[I] < 0) or (joint_value[I] > 255)
          then validity := FALSE;
       end;

    end; (* with step_description do *)

end;
```

```
{ VIEWTASK.PAS }
{
 This procedure allows the operator to view a task which he
 has already entered. It displays one step at a time to the
 operator including the computed joint coordinates and joint
 values.
}
{ revision: 10-02-86
     use print_step procedure }

procedure view_task;
Var
   RobotTask      :file of step_descriptor;
   taskname       :filename;
   next_step      :step_descriptor;

begin (* view task *)
     write ('Enter task filename: ');
     Readln (taskname);
     Assign (RobotTask, taskname);
     Reset (RobotTask);              (* ready for read *)

     While (not EOF(RobotTask)) do
     begin
        Read (RobotTask, next_step);
        print_step(next_step);
     end;
     Close(RobotTask);
end;
```

```
{ SINGLETASK.PAS }
{
 This procedure allows the operator to run a single task on
 one robot. In this case, there is no collision detection or
 collision avoidance attempted.
{ last revision: 9-05-86 }

procedure single_task;
Var
   taskname       :filename;
   RobotTask      :file of step_descriptor;
   next_step      :step_descriptor;
   R,I,step       :integer;
   rcb_regs       :regpack;
   letter         :char;
   continue       :Boolean;

(*************************************************************************)

begin (* single task *)

  (* get task to run *)

   write ('Enter Robot Number: ');
   readln (R);
   write ('Enter task filename for Robot ', R:1, ': ');
   Readln (taskname);
   Assign (RobotTask, taskname);
   Reset (RobotTask);              (* ready for read *)

  (* get rcb address: rcb_regs.es points to segment *)
  (*                  rcb_regs.bx is offset of first rcb *)

   intr(USER_RCB_INTR,rcb_regs);

   rcb_regs.bx := rcb_regs.bx + (R-1)*RCB_LENGTH;    (* find appropriate rcb *)

   continue := TRUE;
```

```
  (*************************************************************)
  (* output task to robot *)

   While ((not EOF(RobotTask)) and (continue)) do
   begin
   (* read next steps for each robot *)
      Read (RobotTask, next_step);

   (* put next steps into rcb's *)
      With next_step do
      begin
        (* fill in rcb.next fields with new setpoints *)
           For I := 0 to 5 do
           begin
              mem[rcb_regs.es:(rcb_regs.bx+next+i)] := joint_value[i];
              writeln ('joint value: ', mem[rcb_regs.es:(rcb_regs.bx+next+i)]);
           end; (* for *)
      end;

   (* set READY bit in RCB *)
      mem[rcb_regs.es:(rcb_regs.bx+status)] :=
                              mem[rcb_regs.es:(rcb_regs.bx+status)] or READY;

   (* wait delay time in step *)

      delay(next_step.step_delay * 1000);

   (* wait until robot has completed move *)
      While ((mem[rcb_regs.es;(rcb_regs.bx+status)] and READY) = READY) do;
                                                  (* do nothing just wait *)
   (* and check for any ERRORS *)
      If (mem[rcb_regs.es:(rcb_regs.bx+status)] and ERROR) = ERROR then
      begin
         writeln;
         writeln ('ROBOT ERROR in Robot ', R:1);
         writeln;
      (* clear ERROR bit in RCB *)
         mem[rcb_regs.es:(rcb_regs.bx+status)] :=
                              mem[rcb_regs.es:(rcb_regs.bx+status)] xor ERROR;

         write ('Do you wish to continue? ');
         readln (letter);
         If (letter <> 'y') and (letter <> 'Y')
         then continue := FALSE;
      end;
   end;

   close (RobotTask);
end;
```

```
{ DUALDRIVER.PAS }
{
 This is the control procedure for running two tasks simultaneously
 on different robots. Before attempting to run the tasks it does a
 quick check to make sure that the destination points of each step
 of the two robots do not interfer with one another.
}

procedure dual_driver;
Label
   driver_return;
Var
   taskname       :filename;
   R              :integer;
   dual_okay      :boolean;


begin (* dual driver *)

  (* get tasks to run *)
    For R := 1 to NumberOfRobots do
    begin
       write ('Enter task filename for Robot ', R:1, ': ');
       Readln (taskname);
       Assign (RobotTask[R], taskname);
    end;

  (* check final steps against each other *)
    check_tasks (dual_okay);
    if dual_okay
    then run_dual_tasks
    else writeln ('tasks contain colliding destinations ');

driver_return:
    For R := 1 to NumberOfRobots do
       Close (RobotTask[R]);
end;
```

```
{ CHKTASK.PAS }
{
  This routine checks the destination position of each robot arm at
  each step. This ensures that it is possible to run the tasks simul-
  taneously before building a series of moves that quarantees no
  collisions will occur. It assumes the robot arms are not presently
  in a collision state.
}
(*********************************************************************)
Procedure check_tasks (Var checks_okay: boolean);
Var
   task_step      :array [1..NumberOfRobots] of step_descriptor;
   rotated_region :array [1..NumberOfRobots] of collision_region;
   R              :integer;
   ALL_FILES_EOF  :Boolean;

begin (* check tasks *)
   checks_okay := TRUE;              (* assume all okay *)

   ALL_FILES_EOF := TRUE;            (* assume done *)
   For R := 1 to NumberOfRobots do
   begin
      Reset(RobotTask[R]);
      ALL_FILES_EOF := ALL_FILES_EOF and EOF(RobotTask[R]);
   end;

 (* now go for it *)
   While (not(ALL_FILES_EOF)) do
   begin
      For R := 1 to NumberOfRobots do
      begin
         If not(EOF(RobotTask[R])) then
         begin
            Read (RobotTask[R], task_step[R]);
          (* if associated objects, bring their descriptions in *)
            with task_step[R] do
            if (step_object <> 0) and
               ((step_status = HOLD) or (step_status = PICKUP))
            then Robot[R].axis_region[4] := object[step_object].object_region
            else zero_region (Robot[R].axis_region[4]);
         end;
      end;
```

```
      For R := 1 to NumberOfRobots-1 do     (* work with robot pairs *)
      begin
        (* rotate regions to reflect step position *)
          rotate_regions (Robot[R],   task_step[R],   rotated_region[R]);
          rotate_regions (Robot[R+1], task_step[R+1], rotated_region[R+1]);
          adjust_regions (rotated_region[R+1], rotated_region[R+1]);

        (* now check for possible collision *)
          If (collision_zone(rotated_region[R]) and
              collision_zone(rotated_region[R+1]) and
              regions_overlap(rotated_region[R], rotated_region[R+1])) then
          begin
             (* report the error, set failure flag *)
                writeln;
                writeln ('Robot ',R:1, ' - Robot ', R+1:1,
                              ' Collision in Step ',task_step[R].step_id:2);
                checks_okay := FALSE;
          end;
      end;

      ALL_FILES_EOF := TRUE;
      For R := 1 to NumberOfRobots do
      begin
         ALL_FILES_EOF := ALL_FILES_EOF and EOF(RobotTask[R]);
      end;
   end; (* while *)
end;
```

```
{ RUNDUAL.PAS }
{
 This procedure runs two task simultaneously. Each pair of steps (one
 step for each robot) is checked for collisions. If a collision seems
 likely, steps are taken to avoid the collision by delaying one of the
 robot arms.
}

(*************************************************************************)
procedure run_dual_tasks;
Label
   step_again;
Var
   R                  :integer;
   task_step          :array [1..NumberOfRobots] of step_descriptor;

   substep_table      :array [1..NumberOfRobots] of substeps;
   NumberOfSubsteps   :array [1..NumberOfRobots] of integer;

   run_table          :array [1..NumberOfRobots, 0..26] of integer;
   NumberOfRunSteps   :integer;
   red_table          :array [1..NumberOfRobots, 0..26] of integer;
   NumberOfRedSteps   :integer;

   step_valid,
   ALL_FILES_EOF,
   task_okay,
   step_failed,                          (* build step flag *)
   step_okay          :boolean;          (* run step flag *)
   dummy              :char;


(*************************************************************************)
procedure build_step;
Var
   R,I        :integer;
   hold_robot,
   failed_substep_number  :integer;
   failed_substep       :array [1..2] of integer;
   hold_flag            :boolean;
```

```
procedure check_path(I1, I2, this_substep_number: integer;
                     Var this_substep_fails: boolean);
Label
   CHECK_PATH_RETURN;
Var
   next_substep_failed : boolean;
   rotated_regions : array [1..2] of collision_region;


begin   (* check path - works with pairs of robots *)

   this_substep_fails := FALSE;                (* assume step is okay *)

 (* if step complete, quit *)
   If ((I1 > NumberOfSubsteps[R]) and (I2 > NumberOfSubsteps[R+1])) then
   begin
      NumberOfRunSteps := this_substep_number - 1;
      goto CHECK_PATH_RETURN;
   end;

 (* if one robot is done, keep it on last substep *)
   If (I1 > NumberOfSubsteps[R])    then I1 := I1 - 1;
   If (I2 > NumberOfSubsteps[R+1])  then I2 := I2 - 1;

   (* rotate regions to reflect step position *)
      rotate_regions (Robot[R],   substep_table[R,I1],   rotated_regions[1]);
      rotate_regions (Robot[R+1], substep_table[R+1,I2], rotated_regions[2]);
      adjust_regions (rotated_regions[2], rotated_regions[2]);

   (* rotate any object held by this robot *)

   (* now check for possible collision *)
      If (collision_zone(rotated_regions[1]) and
          collision_zone(rotated_regions[2]) and
          regions_overlap(rotated_regions[1], rotated_regions[2])) then
         begin
            failed_substep_number := this_substep_number;
            failed_substep[1] := I1;
            failed_substep[2] := I2;
            this_substep_fails := TRUE;
            goto CHECK_PATH_RETURN;
         end;

   (* if get to here, then no overlap occurs *)

      run_table[R,   this_substep_number] := I1;
      run_table[R+1, this_substep_number] := I2;
```

```
      If this_substep_number >= failed_substep_number
      then  check_path (I1+1, I2+1, this_substep_number+1,
                                                  next_substep_failed)
      else  (* keep holding *)
         begin
            if hold_robot = 2 then
               begin
                  if (I1 = NumberOfSubsteps[R])
                  then step_failed := TRUE      (* whole thing has failed *)
                  else check_path (I1+1, I2, this_substep_number+1,
                                                  next_substep_failed)
               end
            else  (* hold_robot = 1 *)
               begin
                  if (I2 = NumberOfSubsteps[R+1])
                  then step_failed := TRUE      (* whole thing has failed *)
                  else check_path (I1, I2+1, this_substep_number+1,
                                                  next_substep_failed)
               end
         end;

    (* If reached end of the line, all substeps fail *)
      If step_failed then goto CHECK_PATH_RETURN;

    (* if no success, try holding back one of the robots *)

      If (next_substep_failed) and
        (failed_substep[2] > I2) and (failed_substep[1] > I1) and
        (I1 < NumberOfSubsteps[R]) and (I2 < NumberOfSubsteps[R+1]) then
        begin
           if (hold_robot = 2) then
              check_path (I1+1, I2, this_substep_number+1, next_substep_failed
           if (hold_robot = 1) then
              check_path (I1, I2+1, this_substep_number+1, next_substep_failed
        end;

      If (next_substep_failed) then this_substep_fails := TRUE;


CHECK_PATH_RETURN:
end;  (* check path *)
```

```
(*****************************************************************)
begin (* build step *)
 (* get next step in task for each robot *)
   For R := 1 to NumberOfRobots do
   begin
      substep_table[R,0] := task_step[R];
      If not(EOF(RobotTask[R])) then
      begin
         Read (RobotTask[R], task_step[R]);
         with task_step[R] do
         if (step_object <> 0) and
            ((step_status = HOLD) or (step_status = PICKUP))
         then Robot[R].axis_region[4] := object[step_object].object_region
         else zero_region (robot[R].axis_region[4]);
      end;
   end;

 (* determine substeps and path for each robot *)
   For R := 1 to NumberOfRobots-1 do
   begin
      compute_substeps(Robot[R], task_step[R], substep_table[R],
                                                    NumberOfSubsteps[R])

      compute_substeps(Robot[R+1], task_step[R+1], substep_table[R+1],
                                                  NumberOfSubsteps[R+1]);

    (* check substeps for possible collisions *)

      step_failed := FALSE;
      hold_robot := 2;                    (* give right-of-way to first robot *
      failed_substep_number := 0;
      check_path (0, 0, 0, step_failed);

      If (step_failed) then               (* try with other robot *)
      begin
         step_failed := FALSE;
         hold_robot := 1;        (* try with second robot having right-of-way *
         failed_substep_number := 0;
         check_path (0, 0, 0, step_failed);
      end;

      If not(step_failed) then
      begin
        (* reduce run table to critical steps *)
         hold_flag := FALSE;
         NumberOfRedSteps := 1;
         For I := 0 to NumberofRunSteps-1 do
         begin
            If (run_table[1,I+1] > run_table[1,I]) and
               (run_table[2,I+1] > run_table[2,I]) and  hold_flag  then
               begin
                  hold_flag := FALSE;
                  red_table[1,NumberOfRedSteps] := run_table[1,I];
                  red_table[2,NumberOfRedSteps] := run_table[2,I];
                  NumberOfRedSteps := NumberOfRedSteps + 1;
               end;

            If (run_table[1,I+1] > run_table[1,I]) and
               (run_table[2,I+1] = run_table[2,I])
            then hold_flag := TRUE;

            If (run_table[1,I+1] = run_table[1,I]) and
               (run_table[2,I+1] > run_table[2,I])
            then hold_flag := TRUE;
         end; (* for *)
```

```
            red_table[1,NumberOfRedSteps] := run_table[1,I+1];
            red_table[2,NumberOfRedSteps] := run_table[2,I+1];
         end

         else
            begin
               writeln;
               write ('Cannot compute non-conflicting moves in step ',
                                                   task_step[R].step_id:5);
            end;

      end; (* for each robot pair *)
   end;    (* build step *)
```

```
(*************************************************************************)
procedure run_step (Var step_okay: boolean);
Var
   R,I,substep    :integer;
   rcb_regs       :regpack;
   letter         :char;


begin (* run steps for dual tasks *)

  (* get rcb address: rcb_resg.ex points to segment *)
  (*                  rcb_regs.bx is offset of first rcb *)

   intr(USER_RCB_INTR,rcb_regs);
   rcb_regs.dx := rcb_regs.bx;     (* save address of first rcb *)

  (* output substeps to robots *)
   substep := 1;
   While (substep <= NumberOfRedSteps) and step_okay do
   begin
      rcb_regs.bx := rcb_regs.dx;        (* restore pointer to first rcb *)

   (* put next substeps into rcb's *)
      For R := 1 to NumberOfRobots do
      begin
         With substep_table[R, red_table[R, substep]] do
         begin
             writeln;
           (* fill in rcb.next fields with new setpoints *)
             For I := 0 to 5 do
             begin
                mem[rcb_regs.es:(rcb_regs.bx+next+i)] := joint_value[i];
                writeln ('joint value: ', mem[rcb_regs.es:(rcb_regs.bx+next+i)
;
             end; (* for *)
         end;
         rcb_regs.bx := rcb_regs.bx + RCB_LENGTH;    (* point to next rcb *)
      end;

   (* set all READY bits *)
      rcb_regs.bx := rcb_regs.dx;       (* point to first rcb again *)

      For R := 1 to NumberOfRobots do
      begin
         (* set READY bit in RCB *)
           mem[rcb_regs.es:(rcb_regs.bx+status)] :=
                           mem[rcb_regs.es:(rcb_regs.bx+status)] or READY;
           rcb_regs.bx := rcb_regs.bx + RCB_LENGTH;    (* point to next rcb *)
      end;

      rcb_regs.bx := rcb_regs.dx;       (* point to first rcb again *)

   (* then wait until all robots have completed their submoves *)
      For R := 1 to NumberOfRobots do
      begin
         While ((mem[rcb_regs.es:(rcb_regs.bx+status)] and READY) = READY) do;
                                                  (* do nothing just wait *)
         rcb_regs.bx := rcb_regs.bx + RCB_LENGTH;
      end;
```

```
      rcb_regs.bx := rcb_regs.dx;       (* point to first rcb again *)

   (* and check for any ERRORS *)
      For R := 1 to NumberOfRobots do
      begin
         If (mem[rcb_regs.es:(rcb_regs.bx+status)] and ERROR) = ERROR then
         begin
            writeln;
            writeln ('JOINT ERROR in ROBOT ', R:1);
         (* clear ERROR bit in RCB *)
            mem[rcb_regs.es:(rcb_regs.bx+status)] :=
                           mem[rcb_regs.es:(rcb_regs.bx+status)] xor ERROR;
            writeln ('Do you wish to continue');
            readln (letter);
            If (letter <> 'y') and (letter <> 'Y')
            then step_okay := FALSE;
         end;
         rcb_regs.bx := rcb_regs.bx + RCB_LENGTH;
      end; (* error check *)

   (* point to next set of substeps *)
      substep := substep + 1;
   end; (* while more steps *)

   (* and stay in this step the requested number of seconds *)
      delay (task_step[R].step_delay * 1000);
end; (* run step *)
```

```
(*************************************************************************)
begin (* run dual tasks *)

    task_okay := TRUE;              (* things always start out well *)

  (* do set up stuff *)
    For R := 1 to NumberOfRobots do
    begin   (* get starting point of each robot into initial position *)
step_again:
       writeln;
       writeln ('enter starting point for Robot ', R:1);
       get_step(task_step[R]);
       coord_transform (robot[R], task_step[R], step_valid);
       if not(step_valid) then
       begin
          writeln ('Invalid step - please try again');
          goto step_again;
       end;
    end;

   ALL_FILES_EOF := TRUE;                (* assume still steps in each file *)
   For R := 1 to NumberOfRobots do
   begin
      Reset(RobotTask[R]);               (* point to beginning of files *)
      ALL_FILES_EOF := ALL_FILES_EOF and EOF(RobotTask[R]);
   end;

 (* now go for it *)
   While not(ALL_FILES_EOF) and task_okay do
   begin

     (* check substeps against each other *)
        build_step;

        if (step_failed) then
          begin
             writeln;
             writeln ('FAIL TO COMPUTE STEP');
             task_okay := FALSE;
          end
        else run_step (task_okay);

      ALL_FILES_EOF := TRUE;               (* assume still steps in each file *)
      For R := 1 to NumberOfRobots do
      begin
         ALL_FILES_EOF := ALL_FILES_EOF and EOF(RobotTask[R]);
      end;
   end; (* while *)

   if not(task_okay) then
   begin
      writeln;
      writeln ('TASK FAILED');
      writeln;
   end;
end;  (* run dual tasks *)
```

```
{ SUBSTEPS.PAS }
{
 This procedure interpolates the substeps for each task.
}

procedure compute_substeps (robot_description: robot_descriptor;
                            new_step: step_descriptor;
                            Var table: substeps;
                            Var step_count: integer);
Var
   Axis            :integer;
   rv_diff         :array[0..5] of integer;
   complete        :boolean;

begin (* compute substeps *)
  (* compute differences between last and next axis positions *)
   For Axis := 0 to 5 do
   begin
      rv_diff[Axis] := new_step.joint_value[Axis] - table[0].joint_value[Axis];
   end;

  (* now fill in substep table *)
   complete := FALSE;
   step_count := 0;

   With robot_description do
   begin
      While not(complete) and (step_count < 13) do
      begin
         complete := TRUE;
         table[step_count+1].step_id := step_count+1;

         For Axis := 0 to 5 do
         begin
            case rv_diff[Axis] of

               0: begin (* do nothing, but this case must be here *)
                     table[step_count+1].joint_value[Axis] :=
                                       table[step_count].joint_value[Axis];
                  end;

               21..255:
                  begin
                     table[step_count+1].joint_value[Axis] :=
                                    table[step_count].joint_value[Axis] + 20;
                     rv_diff[Axis] := rv_diff[Axis] - 20;
                     complete := FALSE;
                  end;

               -20..20:
                  begin
                     table[step_count+1].joint_value[Axis] :=
                          table[step_count].joint_value[Axis] + rv_diff[Axis];
                     rv_diff[Axis] := 0;
                  end;

               -255..-21:
                  begin
                     table[step_count+1].joint_value[Axis] :=
                                    table[step_count].joint_value[Axis] - 20;
                     rv_diff[Axis] := rv_diff[Axis] + 20;
                     complete := FALSE;
                  end
```

```
            end; (* case *)


            table[step_count+1].joint_angle[Axis] :=
               (table[step_count+1].joint_value[Axis] - axis_fudge[Axis]) *
                                          (axis_range[Axis]/axis_rvs[Axis]);
         end; (* for each axis *)

         step_count := step_count+1;
      end; (* while not complete *)
   end; (* with *)
end;    (* compute substeps *)
```

```
{ ROTREG.PAS }
{
 This procedure rotates the various arm regions from their home
 positions to the position and orientation in the step descriptor.
}

Procedure rotate_regions (robot_description: robot_descriptor;
                          step_description: step_descriptor;
                          Var rotated_region: collision_region);
Var
   C, I  : integer;
   rot_axis_center :array[1..NumberOfCenters] of vector;

begin (* rotate region *)
  With robot_description, step_description do
  begin
    For C := 1 to NumberOfRegions-1 do (* Y-axis rotation on axis regions *)
    For I := 1 to 8 do                 (* for each vertex *)
    begin
      TRANS_ORIGIN (axis_center[1],  axis_region[C,I],    rotated_region[C,I]);
      Y_ROT        (-joint_angle[1], rotated_region[C,I], rotated_region[C,I]);
      REVER_ORIGIN (axis_center[1],  rotated_region[C,I], rotated_region[C,I]);
    end;

    For C := 2 to NumberOfCenters do    (* rotate centers for next step *)
    begin
      TRANS_ORIGIN (axis_center[1],  axis_center[C],     rot_axis_center[C]);
      Y_ROT        (-joint_angle[1], rot_axis_center[C], rot_axis_center[C]);
      REVER_ORIGIN (axis_center[1],  rot_axis_center[C], rot_axis_center[C]);
    end;

    For C := 2 to NumberOfRegions-1 do          (* second Y-axis rotation *)
    For I := 1 to 8 do                          (* for each vertex *)
    begin
      TRANS_ORIGIN
                (rot_axis_center[2], rotated_region[C,I], rotated_region[C,I]);
      Y_ROT     (-joint_angle[2],    rotated_region[C,I], rotated_region[C,I]);
      REVER_ORIGIN
                (rot_axis_center[2], rotated_region[C,I], rotated_region[C,I]);
    end;

    For C := 3 to NumberOfCenters do        (* rotate rest of axis centers *)
    begin
      TRANS_ORIGIN(rot_axis_center[2], rot_axis_center[C], rot_axis_center[C]);
      Y_ROT       (-joint_angle[2],    rot_axis_center[C], rot_axis_center[C]);
      REVER_ORIGIN(rot_axis_center[2], rot_axis_center[C], rot_axis_center[C]);
    end;

    For C := 3 to NumberOfRegions-1 do          (* last Y-axis rotation *)
    For I := 1 to 8 do                          (* for each vertex *)
    begin
      TRANS_ORIGIN
                (rot_axis_center[3], rotated_region[C,I], rotated_region[C,I]);
      Y_ROT     (-joint_angle[3],    rotated_region[C,I], rotated_region[C,I]);
      REVER_ORIGIN
                (rot_axis_center[3], rotated_region[C,I], rotated_region[C,I]
    end;
```

```
    For C := 5 to NumberOfCenters do          (* rotate rest of axis centers *)
    begin
      TRANS_ORIGIN
                  (rot_axis_center[3], rot_axis_center[C], rot_axis_center[C]);
      Y_ROT       (-joint_angle[3],    rot_axis_center[C], rot_axis_center[C]);
      REVER_ORIGIN(rot_axis_center[3], rot_axis_center[C], rot_axis_center[C]);
    end;

    (* special case, now move object to position of end effector *)
    trans_region (rot_axis_center[5], axis_region[4], rotated_region[4]);

    For C := 1 to NumberOfRegions do          (* for all regions the Z-rotation *
    For I := 1 to 8 do
    begin
      Z_ROT (joint_angle[0], rotated_region[C,I], rotated_region[C,I]);
    end;
  end;
end;
```

```
{ OVERLAP.PAS }
{
 This routine compares the regions of one arm with those of another
 arm for possible collisions. Any overlap of collision regions indicates
 a potential collision. There must be overlap in all three orthogonal
 planes for collision to occur. Thus if there is no overlap in any one
 of the planes, there is no need to check the other projections.

 This routine could be speeded up by comparing only those regions which
 actually enter the collision zone or by finding the intercept in 3D
 (probably preferable).
}

function regions_overlap (regions_A, regions_B: collision_region): boolean;
Label
   check_xy_projection,
   check_xz_projection,
   check_yz_projection,
   check_overlap;

Var
   RA, RB,                          (* count thru regions *)
   LA, LB          :integer;        (* count thru lines defining region *)
   dummy_point     : vector;        (* we don't actually need the intercept *)
   line_intersection,               (* do two lines intersect *)
   segment_intersection,            (* do two line segments intersect *)
   xy_overlap,
   xz_overlap,
   yz_overlap      :boolean;

(*************************************************************************)

begin (* check projections for overlap *)
  (* for starters, assume everything is okay *)
   xy_overlap := FALSE;
   xz_overlap := FALSE;
   yz_overlap := FALSE;

check_xy_projection:
   For RA := NumberOfRegions downto 1 do    (* compare first set of regions *)
   For RB := NumberOfRegions downto 1 do    (*    to second set of regions *)
      For LA := 1 to 12 do
      For LB := 1 to 12 do
      begin
         xy_intersect
               (regions_A[RA, endpoint[LA, 1]], regions_A[RA, endpoint[LA, 2]],
                regions_B[RB, endpoint[LB, 1]], regions_B[RB, endpoint[LB, 2]],
                dummy_point, line_intersection, segment_intersection);
         IF (line_intersection and segment_intersection) then
         begin
            xy_overlap := TRUE;
            goto check_xz_projection;
         end;
      end;
```

```
check_xz_projection:
 If xy_overlap then
 begin
   For RA := NumberOfRegions downto 1 do  (* compare first set of regions *)
   For RB := NumberOfRegions downto 1 do  (*   to second set of regions *)
      For LA := 1 to 12 do
      For LB := 1 to 12 do
      begin
         xz_intersect
               (regions_A[RA, endpoint[LA, 1]], regions_A[RA, endpoint[LA, 2]],
                regions_B[RB, endpoint[LB, 1]], regions_B[RB, endpoint[LB, 2]],
                dummy_point, line_intersection, segment_intersection);
         IF (line_intersection and segment_intersection) then
         begin
            xz_overlap := TRUE;
            goto check_yz_projection;
        end;
      end;
 end;

check_yz_projection:
 If xy_overlap and xz_overlap then
 begin
   For RA := NumberOfRegions downto 1 do   (* compare first set of regions *)
   For RB := NumberOfRegions downto 1 do   (*   to second set of regions *)
      For LA := 1 to 12 do
      For LB := 1 to 12 do
      begin
         yz_intersect
               (regions_A[RA, endpoint[LA, 1]], regions_A[RA, endpoint[LA, 2]],
                regions_B[RB, endpoint[LB, 1]], regions_B[RB, endpoint[LB, 2]],
                dummy_point, line_intersection, segment_intersection);
         IF (line_intersection and segment_intersection) then
         begin
            yz_overlap := TRUE;
            goto check_overlap;
         end;
      end;
 end;

check_overlap:
   If xy_overlap and xz_overlap and yz_overlap
   then regions_overlap := TRUE
   else regions_overlap := FALSE;
end;
```

```
{
; MATHFUNC.PAS
;  last revision: 9-11-86
;
; This file contains low-level functions for the coordinate
; transformation and procedures for the collision detection software.
;
; The functions are:
;
;    DEG_TO_RAD  - convert an angle expressed in degrees to radians
;    RAD_TO_DEG  - convert an angle expressed in radians to degrees
;    DIST_POINTS - compute the distance between two points
;    DIST_COS - compute the distance of third line given two lines and the
;               included angle.
;
; The procedures are:
;
;    TRANS_ORIGIN - translates point to a new coordinate system
;    REVER_ORIGIN - reverses back to original coordinate system
;    X_ROT - rotate point about x-axis
;    Y_ROT - rotate point about y-axis
;    Z_ROT - rotate point about z-axis
;    XY_INTERSECT - returns point where two lines intersect in the xy plane
;    XZ_INTERSECT - returns point where two lines intersect in the xz plane
;    YZ_INTERSECT - returns point where two lines intersect in the yz plane
;    point_betw_two_points - TRUE/FALSE function
;
}

(*************************************************************************)
function DEG_TO_RAD(degrees: real): real;
begin
     DEG_TO_RAD := degrees * (PI/180);
end;

(*************************************************************************)
function RAD_TO_DEG(radians: real): real;
begin
     RAD_TO_DEG := radians * (180/PI);
end;

(*************************************************************************)
function DIST_POINTS(pt1, pt2: vector): real;
begin
     DIST_POINTS := sqrt(sqr(pt1.x-pt2.x) + sqr(pt1.y-pt2.y) + sqr(pt1.z-pt2.z
;
end;

(*************************************************************************)
function DIST_COS(lgth1, lgth2, degrees: real): real;
begin
  DIST_COS := sqrt(sqr(lgth1) + sqr(lgth2)
                              - 2 * lgth1 * lgth2 * cos(DEG_TO_RAD(degrees)));
end;
```

```
(*********************************************************************)
procedure TRANS_ORIGIN (new_origin, orig_point: vector; Var new_point: vector);
begin
  new_point.x := orig_point.x - new_origin.x;
  new_point.y := orig_point.y - new_origin.y;
  new_point.z := orig_point.z - new_origin.z;
end;

(*********************************************************************)
procedure REVER_ORIGIN (new_origin, orig_point: vector; Var new_point: vector);
begin
  new_point.x := orig_point.x + new_origin.x;
  new_point.y := orig_point.y + new_origin.y;
  new_point.z := orig_point.z + new_origin.z;
end;

(*********************************************************************)
procedure X_ROT (alpha: real; orig_point: vector;  Var new_point: vector);
(* note: alpha must be in radians *)
begin
  new_point.x := orig_point.x;
  new_point.y := (orig_point.y * COS(alpha)) - (orig_point.z * SIN(alpha));
  new_point.z := (orig_point.y * SIN(alpha)) + (orig_point.z * COS(alpha));
end;

(*********************************************************************)
procedure Y_ROT (alpha: real; orig_point: vector;  Var new_point: vector);
(* note: alpha must be in radians *)
begin
  new_point.x := (orig_point.x * COS(alpha)) + (orig_point.z * SIN(alpha));
  new_point.y := orig_point.y;
  new_point.z := (orig_point.z * COS(alpha)) - (orig_point.x * SIN(alpha));
end;

(*********************************************************************)
procedure Z_ROT (alpha: real; orig_point: vector;  Var new_point: vector);
(* note: alpha must be in radians *)
begin
  new_point.x := (orig_point.x * COS(alpha)) - (orig_point.y * SIN(alpha));
  new_point.y := (orig_point.x * SIN(alpha)) + (orig_point.y * COS(alpha));
  new_point.z := orig_point.z;
end;

(*********************************************************************)
function point_betw_two_points (coord1, coord2, point: integer): boolean;
begin
   If coord2 > coord1 then
      begin
         if (point >= coord1) and (point <= coord2)
         then point_betw_two_points := TRUE
         else point_betw_two_points := FALSE
      end
   else
      begin
         if (point >= coord2) and (point <= coord1)
         then point_betw_two_points := TRUE
         else point_betw_two_points := FALSE;
      end;
end;
```

```
(*********************************************************************)
procedure xy_intersect (pt1, pt2, pt3, pt4: vector;
                        Var intercept: vector;
                        Var intersection: boolean;
                        Var on_line_segments: boolean);

(* where pt1, pt2 define line1 and pt3, pt4 define line 2 *)
(* returns point of intersection between line1 and line 2 *)
(* also, returns Boolean value indicating whether the point *)
(* of intersection lies on the line segments defined by the *)
(* four endpoints *)

Var
   delta, mu : real;

begin
   intersection := FALSE;
   on_line_segments := FALSE;

   delta := (pt1.x-pt2.x) * (pt4.y-pt3.y) - (pt1.y - pt2.y) * (pt4.x-pt3.x);
   If abs(delta) > 0.001
   then
   begin
      intersection := TRUE;
      mu :=
          ((pt4.y-pt3.y)*(pt4.x-pt2.x) - (pt4.x-pt3.x)*(pt4.y-pt2.y)) / delta;
      intercept.x := mu * pt1.x + (1.0-mu)*pt2.x;
      intercept.y := mu * pt1.y + (1.0-mu)*pt2.y;
      intercept.z := 0;

      if (abs(intercept.x) < 32767) and (abs(intercept.y) < 32767) then
        if
          (point_betw_two_points
                        (round(pt1.x), round(pt2.x), round(intercept.x)) and
           point_betw_two_points
                        (round(pt1.y), round(pt2.y), round(intercept.y)) and
           point_betw_two_points
                        (round(pt3.x), round(pt4.x), round(intercept.x)) and
           point_betw_two_points
                        (round(pt3.y), round(pt4.y), round(intercept.y)))
        then
           on_line_segments := TRUE;

   end;
end;
```

```
(*********************************************************************)
procedure xz_intersect (pt1, pt2, pt3, pt4: vector;
                        Var intercept: vector;
                        Var intersection: boolean;
                        Var on_line_segments: boolean);

(* where pt1, pt2 define line1 and pt3, pt4 define line 2 *)
(* do lines 1 and 2 intersect in the xz plane, if so where *)

Var
   delta, mu : real;

begin
   intersection := FALSE;
   on_line_segments := FALSE;

   delta := (pt1.x-pt2.x) * (pt4.z-pt3.z) - (pt1.z - pt2.z) * (pt4.x-pt3.x);
   If abs(delta) > 0.001 then
   begin
      intersection := TRUE;
      mu :=
          ((pt4.z-pt3.z)*(pt4.x-pt2.x) - (pt4.x-pt3.x)*(pt4.z-pt2.z)) / delta;
      intercept.x := mu * pt1.x + (1.0-mu)*pt2.x;
      intercept.y := 0;
      intercept.z := mu * pt1.z + (1.0-mu)*pt2.z;
      if (abs(intercept.x) < 32767) and (abs(intercept.z) < 32767) then
        if
          (point_betw_two_points
                          (round(pt1.x), round(pt2.x), round(intercept.x)) and
           point_betw_two_points
                          (round(pt1.z), round(pt2.z), round(intercept.z)) and
           point_betw_two_points
                          (round(pt3.x), round(pt4.x), round(intercept.x)) and
           point_betw_two_points
                          (round(pt3.z), round(pt4.z), round(intercept.z)))
        then
          on_line_segments := TRUE;
   end;
end;
```

```
(*********************************************************************)
procedure yz_intersect (pt1, pt2, pt3, pt4: vector;
                        Var intercept: vector;
                        Var intersection: boolean;
                        Var on_line_segments: boolean);

(* where pt1, pt2 define line1 and pt3, pt4 define line 2 *)
(* do lines 1 and 2 intersect in the yz plane, if so where *)

Var
   delta, mu : real;

begin
   intersection := FALSE;
   on_line_segments := FALSE;

   delta := (pt1.y-pt2.y) * (pt4.z-pt3.z) - (pt1.z - pt2.z) * (pt4.y-pt3.y);
   If abs(delta) > 0.001 then
   begin
      intersection := TRUE;
      mu :=
          ((pt4.z-pt3.z)*(pt4.y-pt2.y) - (pt4.y-pt3.y)*(pt4.z-pt2.z)) / delta;
      intercept.x := 0;
      intercept.y := mu * pt1.y + (1.0-mu)*pt2.y;
      intercept.z := mu * pt1.z + (1.0-mu)*pt2.z;

      if (abs(intercept.y) < 32767) and (abs(intercept.z) < 32767) then
        if
          (point_betw_two_points
                        (round(pt1.y), round(pt2.y), round(intercept.y)) and
           point_betw_two_points
                        (round(pt1.z), round(pt2.z), round(intercept.z)) and
           point_betw_two_points
                        (round(pt3.y), round(pt4.y), round(intercept.y)) and
           point_betw_two_points
                        (round(pt3.z), round(pt4.z), round(intercept.z)))
        then
          on_line_segments := TRUE;
   end;
end;

(*********************************************************************)
```

```
{ MISCFUNC.PAS }

(*********************************************************************)
procedure get_step(Var step: step_descriptor);
Var
   dummy: char;
begin
   with step do
   begin
      write ('Enter position of robot tool (x, y, z): ');
      readln (pos_vector.x, pos_vector.y, pos_vector.z);
      write ('Enter pitch of robot tool (0 to 140):   ');
      readln (pitch);
      write ('Enter twist of robot tool (0 to 90):    ');
      readln (twist);
      write ('Enter gripper position (0 to 30 millimeters): ');
      readln (gripper);
      write ('Object associated with step (0 if none): ');
      readln (step_object);
      if step_object <> 0 then
        begin
          write ('Status of object (APPROACH, PICKUP, HOLD, RELEASE): ');
          readln (dummy);
          if dummy = 'A' then step_status := APPROACH;
          if dummy = 'P' then step_status := PICKUP;
          if dummy = 'H' then step_status := HOLD;
          if dummy = 'R' then step_status := RELEASE;
        end
      else step_status := 0;
      step_delay := 3;
   end;
end;

(*********************************************************************)
procedure zero_region (Var zregion: region);
Var
   I: integer;
begin
   For I := 1 to 8 do
   begin
      zregion[I].x := 0;
      zregion[I].y := 0;
      zregion[I].z := 0;
   end;
end;

(*********************************************************************)
function collision_zone(regions: collision_region): Boolean;
Var
   C, I :integer;
begin
     collision_zone := FALSE;
     begin
        For C := 1 to NumberOfRegions do
        For I := 1 to 8 do
          If ((regions[C,I].x > 165.0) and (regions[C,I].x < 455.0))
          then collision_zone := TRUE;
     end;
end;
```

```
(*************************************************************************)
procedure trans_region (new_origin: vector; orig_region: region;
                        Var new_region: region);
Var
   I: integer;
begin
   For I := 1 to 8 do
      REVER_ORIGIN (new_origin, orig_region[I], new_region[I]);
end;

(*************************************************************************)
procedure adjust_vector(point: vector; Var newpoint: vector);
begin
     newpoint.x := point.x + 620;
     newpoint.y := point.y;
     newpoint.z := point.z;
end;

(*************************************************************************)
procedure adjust_regions (region: collision_region;
                        Var newregion: collision_region);
Var
   C,I :integer;
begin
     For C := 1 to NumberOfRegions do
     For I := 1 to 8 do
        adjust_vector(region[C,I], newregion[C,I]);
end;

(*************************************************************************)
procedure print_point(point: vector);
begin
   with point do
   begin
      writeln (x:8:2, y:8:2, z:8:2);
   end;
end;

(*************************************************************************)
procedure print_region (single: region);
Var
   I: integer;
begin
   writeln;
   For I := 1 to 8 do
   begin
      writeln (single[I].x:10:0, single[I].y:4:0, single[I].z:4:0);
   end;
end;

(*************************************************************************)
procedure print_regions (regions: collision_region);
Var
   C:     integer;
   dummy: char;
begin
     For C := 1 to NumberOfRegions do
     begin
        print_region (regions[C]);
     end;
      writeln ('Hit RETURN to continue');
      readln (dummy);
end;
```

```
(*********************************************************************)
procedure print_step(step: step_descriptor);
Var
   dummy :char;
begin
   With step do
      begin
         Writeln (' Robot ', robot_id:1, '  Step ', step_id:3);
         with pos_vector do
         begin
            writeln ('   position vector: ', x:6:2,y:7:2,z:7:2);
         end;
         writeln ('   tilt, twist, & gripper ',pitch:6:2,twist:7:2,gripper:7:2
         writeln;
         if step_object <> 0 then
         begin
            writeln ('   Object:           ', step_object:4);
            writeln ('   Status of Object: ', step_status:4);
            writeln;
         end;
         writeln ('   angles in radians:      ');
         writeln ('      axis 0: ', joint_angle[0]:6:2 );
         writeln ('      axis 1: ', joint_angle[1]:6:2 );
         writeln ('      axis 2: ', joint_angle[2]:6:2 );
         writeln ('      axis 3: ', joint_angle[3]:6:2 );
         writeln ('      axis 4: ', joint_angle[4]:6:2 );
         writeln ('      axis 5: ', joint_angle[5]:6:2 );
         writeln;
         writeln ('   robot values: ');
         writeln ('      axis 0: ', joint_value[0]:4);
         writeln ('      axis 1: ', joint_value[1]:4);
         writeln ('      axis 2: ', joint_value[2]:4);
         writeln ('      axis 3: ', joint_value[3]:4);
         writeln ('      axis 4: ', joint_value[4]:4);
         writeln ('      axis 5: ', joint_value[5]:4);
         writeln;
      end;
      writeln ('Hit RETURN for next step');
      readln (dummy);
end;
```

VITA

Melissa Rogers Wilson was born on February 12, 1952 in Indianapolis, Indiana. She attended elementary school in Brooklyn, Indiana and in eighth grade, spent one year with her family in Bloomington, Indiana. In 1965, her family moved to Charleston, Illinois where she graduated from Charleston High School in 1969.

She attended Eastern Illinois University for two years between which she spent a year in Oakland, California. She also spent one year in Champaign, Illinois where she attended the University of Illinois. In 1973, she moved to Knoxville, Tennessee where she attended the University of Tennessee. She received her Bachelor of Arts degree in Computer Science from the University of Tennessee in 1976.

After receiving her undergraduate degree, the author worked for two years at Administrative Data Systems with the University of Tennessee. She then worked for Computer Concepts Corporation and Tech-Con, Inc. in Knoxville, Tennessee and for ORFMA in Oak Ridge, Tennessee. These companies were primarily concerned with supplying computer control systems to manufacturing firms.

In 1984, she entered the Computer Science graduate program at the University of Tennessee. During her graduate studies, she held a Graduate Teaching Assistantship for two years. Ms. Wilson was awarded her Master of Science degree in March 1987.