University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

September 1990

Control Software of Robot Compliant Wrist System

Yangsheng Xu
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Yangsheng Xu, "Control Software of Robot Compliant Wrist System", . September 1990.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-66.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/564
For more information, please contact repository@pobox.upenn.edu.

Control Software of Robot Compliant Wrist System

Abstract

The compliant wrist combining passive compliants and sensor has been developed in GRASP laboratory. The device provides the robot system the necessary flexibility which accommodates transitions as the robot makes contact with the environment, corrects positioning error in automatic assembly, avoids high impact forces and protects the surface from damage. The device also supplies the displacement sensing of the passive compliance so that active feedback control is possible.

This report is intended to serve as a reference material to introduce the control software of the robot compliant wrist system developed and implemented in the lab. The detail discussion on system performance and parameters selection can be found in the thesis [3].

The rest of material is organized as follows.

Section 2 introduces the compliance control methods of robot manipulators. The historic development of both passive and active compliance method is discussed. The advantages and disadvantages of the methods are investigated. Based on the unsolved problems in this issue, the six-degree freedom compliant wrist is developed, and the design feature is presented.

Section 3 discusses the hybrid position/force control scheme using the sensing information from the device. The positioning error due to load or external force when robot moves in free space is compensated for, so that the effective stiffness is increased. In force control when robot is constrained by environment, the trajectory is modified by sensed force, so that the effective stiffness is decreased.

Section 4 deals with the implementation of the control scheme. Various programs have been developed to perform the hybrid control operations, such as hybrid control demonstration, surface tracking, edge tracking, insertion and pulling out, and writing operation. The programs have been successfully implemented in the experiments. Definition and selection of the parameters in the programs are discussed.

Section 5. is the source code of control scheme which has been implemented in PUMA 560 with index machine in GRASP Laboratory. The control is executed on a MicroVax I1 using the RCI primitives of RCCL.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-66.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/564

Control Software Of Robot Compliant Wrist System

MS-CIS-89-66
GRASP LAB 192

Yangsheng Xu

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

October 1989

CONTROL SOFTWARE OF ROBOT COMPLIANT WRIST SYSTEM

Yangsheng Xu

General Robotics and Active Sensory Perception Laboratory
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Contents

1. INTRODUCTION

The compliant wrist combining passive compliants and sensor has been developed in GRASP laboratory. The device provides the robot system the necessary flexibility which accommodates transitions as the robot makes contact with the environment, corrects positioning error in automatic assembly, avoids high impact forces and protects the surface from damage. The device also supplies the displacement sensing of the passive compliance so that active feedback control is possible.

This report is intented to serve as a reference material to introduce the control software of the robot compliant wrist system developed and implemented in the lab. The detail discussion on system performance and parameters selection can be found in the thesis [3].

The rest of material is organized as follows.

Section 2 introduces the compliance control methods of robot manipulators. The historic development of both passive and active compliance method is discussed. The advantages and disadvantages of the methods are investigated. Based on the unsolved problems in this issue, the six-degree freedom compliant wrist is developed, and the design feature is presented.

Section 3 discusses the hybrid position/force control scheme using the sensing information from the device. The positioning error due to load or external force when robot moves in free space is compensated for, so that the effective stiffness is increased. In force control when robot is constrained by environment, the trajectory is modified by sensed force, so that the effective stiffness is decreased.

Section 4 deals with the implementation of the control scheme. Various programs have been developed to perform the hybrid control operations, such as hybrid control demonstration, surface tracking, edge tracking, insertion and pulling out, and writing operation. The programs have been successfully implemented in the experiments. Definition and selection of the parameters in the programs are discussed.

Section 5. is the source code of control scheme which has been implemented in PUMA 560 with index machine in GRASP Laboratory. The control is executed on a MicroVax II using the RCI primitives of RCCL.

2. COMPLIANCE AND COMPLIANT WRIST

2.1 Passive and Active Compliance

Compliance is the ability of a robot manipulator to react to external forces or tactile stimuli during the motion. Compliant motion control has been considered as one of key problems of robotic manipulation since 1970. Modifying a trajectory by contact force seems easy for human being yet is very awkward for robot manipulators. That is why today's robot can only perform simple tasks, such as pick-and-place operation, spray-painting, and welding, which do not require a sophisticated compliant motion ability [13]. However, many tasks of robot manipulation require compliant motion, such as assembly operations including inserting electronic components on circuit boards, pressing bearing onto a shaft, placing armatures on motors, and surface finishing operations including grinding, deburring, and routing.

Compliance can be specified in the joint servo, or by passive compliance provided in the manipulators. The former is known as *active compliance,* while the latter is called *passive compliance.*

The passive compliance is a device or additional tool that provides a flexibility for the rigid robot and usually are attached to the robot end-effector, such as at the hand, wrist, or fingers. The advantage of introducing such a flexibility in the system is primarily from the demand of robotic operations involved in contacting the environment, especially, in assembly operation in robot manufacturing systems.

Passive compliance to provide adaptation for assembly operations has a number of advantages including:

(1) the positioning tolerances in robot operation and the geometric uncertainties in the parts are relaxed;

(2) the high forces or moments normally produced in jamming or wedging are reduced;

(3) the assembled surfaces are protected from damage, such as a scraping or galling;

(4) automatic assembly is facilitated in more operations;

(5) expensive electronics normally required in precision operations are eliminated.

Passive compliance is not only beneficial for the self-correction of positioning errors in assembly, but also for adaptation to the transient state control and force control. It has been known that the manipulator works between constrained and unconstrained modes continuously. In the unconstrained mode, position is controlled, while in constrained mode force is controlled. Between these two states, however, is a *transition.* In the transition, the force or velocity from which control is achieved may be discontinuous and the control becomes uncertain. In this case, if there is a passive compliance near the contact point of the end-effector, the kinetic energy can be absorbed and the possible high forces or moments can be avoided, and thus the discontinuity is accommodated and preformance of the entire system is smoothed [1][2][5].

Another advantage of passive compliance is that a high gain of the force control can be selected when the robot is equipped with such a device. It has been shown that the allowable force control gain is proportional to the effective stiffness of the overall system [6]. Therefore, for the system including passive compliance, the allowable force control gain is higher than that without passive compliance, which is desirable for improving sensitivity and performance of force control.

As we discussed previously, compliance may occur because the control system is programmed to react to the sensed forces. In this case, the compliance is known as active compliance. Various approaches to active compliance have been developed and involved nearly all control aspects of the robotics research. We may categorize these control schemes as two basic issues: *impedance control* and *hybrid control. Impedance control* specifies a linear relation between the force/torque and position/orientation (or velocity/angular velocity), while *hybrid control* controls position/orientation along the specified degrees of freedom and independently controls force/torque along the remaining degrees of freedom. The philosophy of the impedance control technique is based on the fact that human muscle can be viewed as a generalized spring with a controllable stiffness or damping, while the philosophy of the hybrid control technique is based on the fact that human works usually by exerting force in some joints, such as wrist, and leaving other joints free, such as shoulder for the motion. We discuss these two basic techniques respectively.

The hybrid control, or hybrid force/position control, is the one that allows force to be commanded along certain degrees of freedom and allows position to be commanded along the remaining directions. A number of approaches have been developed and main results have been contributed by several researchers: Paul (1972) [9], Paul and Shimano (1976) [8] *("Compliance and Control")* , Mason (1981) [13] *("Compliance and Force Control")* , Raibert and Craig (1981) [21] *("Hybrid Position/Force Control")* a number of papers have been published, such as [24] [25] [23] [26] [28].

In impedance control, a linear function is defined that relates the displacement or velocity variables of the end-effector to the force variables. Principal results have been obtained by Salisbury (1980) [14] [15] *("Stiffness Control")* , Whitney (1977) [12] [11] *("Damping Control")* , and Hogan (1982) [16] [17] *("Impedance Control")* published for modifying the controller, stability analysis of the system and improvement of the system behavior [18] [22] [30].

From our discussions above, a question may be raised for the active compliance techniques. Can we specify compliance of an arbitrary magnitude whether the compliance is obtained by a gain in hybrid control or a linear relation in impedance control? From either theoretical analysis [11] [18] [19] or experimental work [6] [27] [28], it is clear that for the active compliance technique, the implementations may suffer from sluggish behavior and stability problem must be taken into consideration, especially when environment is stiff.

A good hybrid or impedance control system requires either a small effective stiffness of the end-effect or a small environment stiffness. For the stiff environment, the only choice is to reduce robot effective stiffness [11]. We also can not use a smaller sample time to remedy the situation, because the sample time corresponds to the band width of the arm and its controller which is limited by the dynamic properties as arm inertia and actuator torques limits. Therefore, Whitney [11] started from a survey of current research approaches, based on an analysis of those control structures with a simple model, and suggested

only two remedies that one can consider: small arms or hands with higher bandwidth, or deliberate passive compliance installed in the arm's wrist to make the effective stiffness small, such that a stiff environment can be dealt with and a fast response of the system can be achieved.

When the robot continually works between constrained and unconstrained modes to implement position and force control, there is the transition between these two modes. Traditionally the transition is ignored with the energy of impact being absorbed in gear trains and structure. The system in the transient state has not been modeled and therefore a very conservative speeds must be employed in order to avoid damage.

Because of the reasons listed above, passive compliance is required not only for adaptation of assembly operations, but also for improving system performance of the active compliance control.

Application of passive compliance alone, however, may also present problems. A main problem is decreasing the positioning accuracy because the end-effector stiffness is reduced. A compliant system is desirable only in force control mode, while in position control case, a stiff system is required. Therefore, compensation for the position error due to presence of passive compliance is required. Passive compliance may also cause an uncertainly problem in force control because force sensor is located far away from the contact point where the force is exerted.

A natural solution is to combine sensor with the passive compliance. Only with a sensor, may we detect the deflection of the device so that the adaptation of both position control and force control modes is ideal. In this case, the active compliance methods can still be employed with consideration of passive compliance instead of the rigid end-effector.

2.2 Compliant Wrist Design

Based on the discussions above, it becoming increasingly necessary to design a device combining passive compliance and active sensing mechanism. The device must be simple and economical so that the complexity of a sensing mechanism and expensive optical transducers are avoided. The device must provide six degrees of freedom passive compliance, instead of two or five degrees of freedom, so that it can provide a spring-shock absorber analogues to accommodate the transition between force and position control modes. The compliance in and around each axis must be reasonable for most of operations. The device also provides measurement of six degrees of freedom motion of the passive compliance, so that the active control can be implemented.

Such a device has been developed in the GRASP Laboratory. In this section, we talk about the basic design feature of the device. The detail kinematic analysis, dynamic parameters, and design consideration, as well as the effect of the parameters on the system performance can be found in the thesis [3]. We have designed two prototypes of the device. The control experiment was performed with the first prototype. The kinematic design of the sensing mechanism for the second prototype is based on the kinematic sensitivity ellipsoid theory which is discussed in the thesis [3].

The device includes two plates, upper plate and lower plate. The lower plate is attached to the robot and the upper one is connected with the end-effector. The sensing mechanism installed between these two plates must be capable of measuring six DOF motions of the upper plate with respect to the lower one. We use six transducers at six joints of the mechanism. The task of the device system is to measure the joint angles and then compute the position error of the end-point of the device in Cartesian space which represents the 6 DOF deflections of the compliant wrist due to the external force. Therefore, computation from the input data is *direct kinematics*.

We at first, intented to use a parallel mechanism constructed by LVDTs as displacement sensors. However, computation of the direct kinematics is complicated for a parallel mechanism, while that of the inverse kinematics is easy. On the contrary, for a serial mechanism, computation of the direct kinematics is much easier than that of the inverse one. Additionally, for the parallel mechanism, a relatively high precision of machining and assemblying is required, thus the serial linkage is easier to fabricate than the parallel one. Therefore, we chose the serial type of mechanism. A disadvantage is the error accumulation of a serial mechanism, while a parallel mechanism compensates for the error. We, however, carefully calibrate the mechanism and potentiometers and filter the data, so the designed precision of the device is obtained.

The transformation matrix of the wrist from the lower plate to the upper plate, T_w can be formed by multiplying simple translational and rotational transformation matrices.

$$T_w = A_1 A_2 A_3 A_4 A_5 A_6 A_7$$

$$= Trans(-l_7, l_3, l_1,)\, Rot(z, \theta_1)\, Trans(-l_2, 0, 0)\, Rot(x, \theta_2)\, Trans(0, -l_3, 0)\, Rot(x, \theta_3)$$

$$Trans(l_4, -l_5, 0)\, Rot(z, \theta_4)\, Trans(0, 0, l_6)\, Rot(y, \theta_5)\, Trans(0, l_5, 0)\, Rot(z, \theta_6)$$

$$Trans(l_7, 0, l_8)$$

The parameters in the above equation can be listed in the following table.

Table 1 The lengths of the sensing mechanism (in mm)

L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8
23.0	22.0	15.0	15.0	26.0	26.0	35.0	13.0

Table 2 The initial position of the joint (in degree)

θ_1	θ_2	θ_3	θ_4	θ_5	θ_6
-90	0	0	90	-90	90

When we design a passive compliance, we must consider several facts, such as the stiffness in three directions and around three axes at the compliance center, the distance from bottom of the compliant wrist device to the compliance center (or projection), and the overall strength and load capacity which are

limited by the strength of the material.

We chose to use rubber material as the passive compliant element. The reason to choose rubber is the simplicity of installing and the significant inherent damping which is rather important for the control and device behavior from our analysis and simulation [1].

We chose a rubber structure that yields the reasonable stiffness in each direction and around each axis. The device provides similar compliance in all six generalized components, which differs from the RCC design where compliance is only presented in two to five components. In the RCC, compliance in the approaching direction (usually referred to the Z direction) is not allowed to assure position control accuracy. We, however, intent to accommodate transition and absorb the kinetic energy as the robot makes contact with environment, which requires the compliance in the Z direction. Moreover, since we provide sensing and active feedback control in position control, the positioning errors can be compensated, and thus avoiding compliance in the Z direction is unnecessary.

The stiffness in each direction must be reasonable. The stiffnesses in the lateral and torsional directions are low because the geometric tolerances is corrected usually in these directions. The stiffness in the axial direction is high so that a high load capability and low positioning error are assured.

A major difference from most RCC device is that no attempt is made to locate the center of compliance remotely, which is a source of instability as investigated in [28] and incompatible with a reasonable size as shown in [10]. Since we introduce feedback control and the effective location of the compliance center can be actively adjusted, it is unnecessary to exactly locate the compliance center at a certain point of the device.

The stiffness in each direction was measured and the results are listed in Table 3. The stiffness of the device can be represented in a form of matrix.

$$K_w = \begin{bmatrix} K_{ll} & 0 & 0 & 0 & K_{bl} & 0 \\ 0 & K_{ll} & 0 & -K_{bl} & 0 & 0 \\ 0 & 0 & K_{aa} & 0 & 0 & 0 \\ 0 & -K_{lb} & 0 & K_{bb} & 0 & 0 \\ K_{lb} & 0 & 0 & 0 & K_{bb} & 0 \\ 0 & 0 & 0 & 0 & 0 & K_{tt} \end{bmatrix}$$

K_{ll}: Lateral force/lateral displacement;
K_{aa}: Axial force/axial displacement;
K_{tt}: Torsional torque/torsional angle;
K_{bb}: Bending torque/bending angle;
K_{bl}: Bending torque/lateral displacement;
K_{lb}: Lateral force/bending angle.

Table 3 Stiffness of the passive compliance

K_{ll}	K_{aa}	K_{tt}	K_{bb}	K_{bl}	K_{lb}
(lbs/in)	(lbs/in)	(lbs-in/degree)	(lbs-in/degree)	(lbs)	(lbs/degree)
2.54	31.75	0.50	3.00	9.77	0.27
(N/m)	(N/m)	(N-m/rad)	(N-m/rad)	(N)	(N/rad)
441.00	5512.5	0.056	0.34	43.09	1.19

For the second prototype device, the sensing mechanism is designed based on the sensitivity ellipsoid theory. The design goal is to find a mechanism configuration in which the motion at the end-point is equally sensitive to the motion of each joint. In other words, given any arbitrary displacement at the end-point of mechanism, if some joints present large motions, while the others are almost stationary, the sensitivity of the instrument is poor.

We may evaluate the kinematic sensitivity of a mechanism by the procedure as follows. We partition the inverse Jacobian matrix into the rotational part and translational part. Multiplying one of the matrices by its transpose, two special matrices known as the sensitivity matrices corresponding to rotational and translational parts are formed. Then, the eigenvalue problems of these matrices are solved. The eigenvalues of these two matrices represent the kinematic sensitivity in space, and must satisfy a certain condition according to different design requirement. In our case, the isotropic sensitivity is desirable, thus each eigenvalue has to be nearly equal.

For the passive compliance of the device, we tried to make different blocks to accommodate different operations, so that any of them can be sandwiched between the upper and lower plates of the device without changing the sensing mechanism. For a block of the passive compliance, we used a partable structure assembled from several single pieces of rubber which is usually used as sandwich mounts of flex-bolt. Basically, it consists of two partions. The upper one mainly contributes to the axial stiffness, and the lower one provides the lateral and torsional stiffness. The bending stiffness is contributed by two parts.

3. HYBRID POSITION FORCE CONTROL

3.1 Position Control

"Position Control" is the control mode that robot end-effector follows a specified trajectory. Being equipped with the compliant wrist, the end-effector carrying a load or being acted on by an external force will cause inaccurate positioning since the actual stiffness of the robot system is decreased. We propose to utilize the sensed deflection of the compliant wrist and drive the robot in the opposite direction of observed deflection in order to increase the overall stiffness of the system.

Two different control schemes, control in Cartesian coordinates and control in joint coordinates, are investigated for position control. We first discuss the control scheme in Cartesian coordinates. We define the transformation from the base coordinates to the lower plate of the compliant wrist device as T_6, that from the lower plate to the upper plate of the compliant wrist as T_w, and that from the base to the upper plate of the compliant device as B which is considered as the task coordinate transformation. The kinematic relation at the initial state is

$$T_6 T_w = B \tag{1}$$

Supposing at the current state, the compliant wrist coordinate frame T_w is changed to T_w' due to a load or other external force, the task coordinate transformation is thus changed to B' and the kinematic relation becomes

$$T_6 T_w' = B' \tag{2}$$

In order that the positioning ability of the robotic system is retained, it is our aim that the robot coordinate transformation T_6 be modified to T_6' such that the task coordinate transformation B remains unchanging. Therefore, the control goal is

$$T_6' T_w' = B \tag{3}$$

Equating (1) and (3) yields

$$T_6' T_w' = T_6 T_w$$

or,

$$T_6' = T_6 T_w (T_w')^{-1} \tag{4}$$

An alternative, which we use, is joint differential control utilizing the differential displacement of the compliant wrist. There are two ways to obtain six components generalized differential displacement vector $\Delta \mathbf{X}_w$, i.e., three position displacements and three orientations (related to the initial position where deflection is zero) from the wrist sensor.

$$\Delta X_w = (\Delta x, \Delta y, \Delta z, \Delta\theta_x, \Delta\theta_y, \Delta\theta_z)^T$$

Firstly, $\Delta \mathbf{X}_w$ may be extracted from the updated transformation matrix of the compliant wrist T_w.

$$T_w = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using roll, pitch, yaw set to represent rotation [7], the transformation matrix T_w can also be written as

$$T_w(\Delta x, \Delta y, \Delta z, \Delta\theta_x, \Delta\theta_y, \Delta\theta_z) =$$

$$\begin{bmatrix} \cos\Delta\theta_z\cos\Delta\theta_y & \cos\Delta\theta_z\sin\Delta\theta_y\sin\Delta\theta_x - \sin\Delta\theta_z\cos\Delta\theta_x & \cos\Delta\theta_z\sin\Delta\theta_y\cos\Delta\theta_x + \sin\Delta\theta_z\sin\Delta\theta_x & \Delta x \\ \sin\Delta\theta_z\cos\Delta\theta_y & \sin\Delta\theta_z\sin\Delta\theta_y\sin\Delta\theta_x + \cos\Delta\theta_z\cos\Delta\theta_x & \sin\Delta\theta_z\sin\Delta\theta_y\cos\Delta\theta_x - \cos\Delta\theta_z\sin\Delta\theta_x & \Delta y \\ -\sin\Delta\theta_y & \sin\Delta\theta_y\sin\Delta\theta_x & \cos\Delta\theta_y\cos\Delta\theta_x & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Therefore,

$$\Delta\theta_y = -\sin^{-1}(n_z)$$

$$\Delta\theta_z = \sin^{-1}\left(\frac{n_y}{\cos\Delta\theta_y}\right)$$

$$\Delta\theta_x = \sin^{-1}\left(\frac{o_z}{\cos\Delta\theta_y}\right)$$

and,

$$\Delta x = p_x$$

$$\Delta y = p_y$$

$$\Delta z = p_z$$

Secondly, these six differential displacements may also be calculated from the sensing mechanism Jacobian matrix $\mathbf{J}_w$ and $\Delta\boldsymbol{\theta}_w$ which both depend upon sensing mechanism joint angles.

$$\Delta\mathbf{X}_w = \mathbf{J}_w \Delta\boldsymbol{\theta}_w \tag{5}$$

Joint differential control of the manipulator is defined as

$$\Delta\mathbf{X} = \mathbf{J}_m \Delta\boldsymbol{\theta}_m \tag{6}$$

where $\mathbf{J}_m$ and $\Delta\boldsymbol{\theta}_m$ are the Jacobian matrix and joint differential change of the manipulator respectively. Since compensation of position error is desirable in position control and the manipulator must move in the opposite direction of the wrist deflection, $\Delta\mathbf{X}$ in (6) must be the same amount of $\Delta\mathbf{X}_w$ in (5) but with the opposite sign if complete compensation is desired. Therefore,

$$\Delta\boldsymbol{\theta}_m = -\mathbf{J}_m^{-1}\Delta\mathbf{X}_w \tag{7}$$

More generally we may introduce a gain matrix K_P so that the desired joint command motion θ_{des} becomes

$$\boldsymbol{\theta}_{des} = \boldsymbol{\theta}_{traj} + \Delta\boldsymbol{\theta}_m = \boldsymbol{\theta}_{traj} - \mathbf{J}_m^{-1}K_P\Delta\mathbf{X}_w \tag{8}$$

where θ_{traj} is the desired joint angle, supplied by a trajectory generator function.

The matrix K_p in Equation (8) is actually a proportional gain of the control law. We may also extend it to a full PID feedback control for this position control problem by modifying the control algorithm (8).

$$\theta_{des} = \theta_{traj} - \mathbf{J_m}^{-1}\Delta S \tag{9}$$

$$\Delta S = K_p \Delta X_w + K_v \Delta \dot{X}_w + K_I \int \Delta X_w dt \tag{10}$$

For the detail discussions on the system analysis can be found in the thesis [3].

3.2 Force Control

When the end-effector is constrained by the environment, force control is highly desirable so that the end-effector trajectory can be modified by the contact force during operations. For the robot with the compliant wrist, the contact force can be identified by sensing information from the wrist device. We propose to utilize this sensed force to drive the manipulator in the same direction as the force, i.e., the deflection of the passive compliant mechanism such that the apparent stiffness is decreased and the desired contact force is maintained.

Joint differential control scheme is employed to force control problem. Using the notations of the last section, we can obtain the six component generalized displacement of the device $\Delta\mathbf{X}_w$ from either the wrist Jacobian matrix $\mathbf{J}_w$ and the joint displacement $\Delta\theta_w$, or the transformation matrix T_w.

The manipulator differential control scheme from Equation (6) is

$$\Delta\theta_m = \mathbf{J}_m^{-1}\Delta\mathbf{X} \tag{11}$$

Since in this method, the manipulator is driven to a certain displacement in response to the sensed deflection of the passive compliance, the control scheme actually controls the displacement, thereby indirectly controls compliance of the system, and the contact force. The desired stiffness or compliance is obtained by the ratio of the sensed force to the displacement response of the system. This displacement $\Delta\mathbf{X}$ relates to the exerted force F_w by the desired stiffness K_d

$$F_w = K_d \Delta\mathbf{X} \tag{12}$$

The measured displacement relates to the exerting force by the physical stiffness of the system which is approximately the stiffness of the passive compliance of the wrist device, K_w

$$F_w = K_w \Delta\mathbf{X}_w \tag{13}$$

Substituting yields

$$\Delta\mathbf{X} = K_F \Delta\mathbf{X}_w \tag{14}$$

where K_F is the dimensionless ratio of stiffness.

$$K_F = K_d^{-1} K_w \tag{15}$$

Substituting Equation (14) to Equation (11) results in

$$\Delta\theta_m = \mathbf{J}_m^{-1} K_F \Delta\mathbf{X}_w \tag{16}$$

The desired joint angles θ_{des} thus are

$$\theta_{des} = \theta_{curr} + \Delta\theta_m = \theta_{curr} + \mathbf{J}_m^{-1} K_F \Delta\mathbf{X}_w \tag{17}$$

where θ_{curr} is the current joint angles. Further, a full PID control can be employed as we discussed in position control case. The derivation is analogous to that in Chapter 6 and is omitted for simplicity.

Comparing the control algorithm in position control (8) with that in force control (17), it is worthwhile noting following points.

(1) The control algorithms are very similar, and both contain a dimensionless gain matrix K_P or K_F, but with a different sign in front. K_P represents the gain controlling how much of the deflection to be compensated in position control. K_F represents the gain relating the natural stiffness to the effective stiffness of the system which may be determined based on different force control tasks. If complete compensation in and around each direction is required in position control, the gain K_P is an identity matrix. If the desired compliance level is as same as the natural compliance K_w, which is mainly contributed by the passive compliant mechanism of the wrist, the gain matrix K_F is again an identity matrix.

(2) The end-effector must be driven in the opposite direction to the displacement measured in position control, but in the same direction for force control. Therefore, the updated differential displacement $\Delta\theta_m$ is negative in Equation (8) while positive in Equation (17). As a result, the overall stiffness of the system is increased in position control mode, but decreased in force control mode.

(3) The desired joint angles θ_{des} is based on the specified joint angles θ_{traj} in position control (8), but based on the current joint angles θ_{curr} in force control (17). This rationale can be explained if we consider the following case. Provided that the end-effector is in steady-state and a constant deflection (i.e, constant force) exists in the compliant wrist, the manipulator should keep moving in force control mode till the specified contact force is obtained, while it should stop if a constant compensation has been achieved in position control mode.

3.3 Hybrid Control

"Hybrid control" or *"hybrid position/force control"* is the case where the degrees of freedom of end-effector motion are partitioned into two orthogonal sets, one for position control and another for force control, by the constraints of force and position. As a robot is equipped with the compliant wrist, the desired motion has to be capable of compensating for the position error due to the passive compliance exerted by load or external forces in some degrees of freedom, and simultaneously has to be capable of responding to the sensed force in the other degrees of freedom.

It has been known that every manipulator task can be broken down into elemental components that are defined by a particular set of contacting surfaces. A generalized surface can be defined in a constraint space having six degree of freedom, with position constraints along the normal to this surface and force constraints along the tangents. These two types of constraints, force and position, partition the degrees of

freedom of possible end-effector motion into two orthogonal sets, that must be controlled according to position control scheme (8) and force control scheme (17) simultaneously. Since the desired angle θ_{des} is based on the trajectory specified angle θ_{traj} in position control (8), but based on the current angle θ_{curr} in force control (17), the hybrid control cannot be obtained simply by combining Equations (8) and (17). In what follows, we present a hybrid control scheme for a robot system with consideration of an installed passive compliance.

At first, we partition $\Delta\mathbf{X}_w$ which is the Cartesian displacement determined from the wrist sensor into two sets; $\Delta\mathbf{X}_w^F$ corresponding to the component for which force control is required, and $\Delta\mathbf{X}_w^P$ in the remaining directions in which the position control is required. For example, if force in the Z direction and torques around the X and Y directions are controlled, and the remaining directions are position controlled, we partition

$$\Delta\mathbf{X}_w = \left[\Delta x \ \Delta y \ \Delta z \ \Delta\theta_x \ \Delta\theta_y \ \Delta\theta_z\right]^T \tag{18}$$

into

$$\begin{aligned} \Delta\mathbf{X}_w^F &= \left[0\ 0\ \Delta z \ \Delta\theta_x \ \Delta\theta_y \ 0\right]^T \\ \Delta\mathbf{X}_w^P &= \left[\Delta x \ \Delta y \ 0\ 0\ 0 \ \Delta\theta_z\right]^T \end{aligned} \tag{19}$$

If the desired force $\mathbf{F}_d$ is given, the desired deformation of the wrist device $\Delta\mathbf{X}_d$ can be computed by

$$\Delta\mathbf{X}_d = K_w^{-1} F_d \tag{20}$$

where K_w is the actual stiffness of the compliant wrist device. Multiplying by a gain matrix, the desired differential motion of the end-effector corresponding to position and force control schemes can be obtained respectively.

$$\Delta\mathbf{X}_P = K_P \ \Delta\mathbf{X}_w^P \tag{21}$$

$$\Delta\mathbf{X}_F = K_F \ (\Delta\mathbf{X}_w^F - \Delta\mathbf{X}_d) \tag{22}$$

to achieve the required position and force control.

From six component differential motions $\Delta\mathbf{X}_P$ and $\Delta\mathbf{X}_F$, we can form the 4×4 differential transform matrices $T_{\Delta X_P}$ and $T_{\Delta X_F}$ respectively. When force control is considered in Cartesian space, the desired motion of the end-effector $T_{des}^{(j)}$ is based on the last desired motion $T_{des}^{(j-1)}$, then corrected by the differential motion $T_{\Delta X_F}$ representing force control scheme, where the superscript j refers to time.

$$T_{des}^{(j)} = T_{des}^{(j-1)} * \ T_{\Delta X_F} \tag{23}$$

Since we must simultaneously consider the deflection of the end-effector in the presence of passive compliance, the desired motion $T_{des}^{(j)}$ in (23) has to be modified by the differential motion $T_{\Delta X_P}$ representing the deflection of the compliant wrist in (21). The robot has to be driven in the opposite direction to compensate for the deflection $T_{\Delta X_P}$, thus the required motion $T_{req}^{(j)}$ to yield the desired motion $T_{des}^{(j)}$ is

$$T_{req}^{(j)} = T_{des}^{(j)} * \ T_{\Delta X_P}^{-1} \tag{24}$$

The resultant motion of the end-effector not only provides the desired compliance in the specified degrees of freedom, but compensates simultaneously for the deflection of the compliant wrist in other degrees of freedom.

We also can derive a hybrid control scheme in joint space. The joint differential control scheme is

$$\Delta\theta = \mathbf{J_m}^{-1}\,\Delta X \tag{25}$$

where $\mathbf{J_m}$ is the manipulator Jacobian, and $\Delta\theta$ and ΔX are the joint differential motions and the corresponding Cartesian differential motion of the end-effector. The desired joint angles of the end-effector must move in the same direction as the differential joint angles caused by $\Delta\mathbf{X}_F$, based on the current desired joint angles as in Cartesian space control (23).

$$(\theta_{des})_j = (\theta_{des})_{j-1} + \mathbf{J}_m^{-1}\,\Delta\mathbf{X}_F \tag{26}$$

Also, the end-effector motion must be modified by the differential joint angles which represents the deflection of passive compliance $\Delta\mathbf{X}_P$ in (21),

$$(\theta_{req})_j = (\theta_{des})_j - \mathbf{J}_m^{-1}\,\Delta\mathbf{X}_P \tag{27}$$

Equations (26) and (27) represent the hybrid position/force control scheme in joint space in correspondence with Equations (23) and (24) in Cartesian space.

4. PROGRAMMING AND EXPERIMENTS

Based on the control algorithm, software coding has been developed to perform various hybrid control operations and successfully implemented in experiments. In this section, we discuss the programming work and the experiments where the software is implemented. The detail discussion on the experiments, including the effect of the parameters on the system performance, can be found in the thesis [3].

Experiments with the compliant wrist have been performed on a PUMA 560. Before the experiment, all six potentiometers were adjusted in a proper range and the compliant wrist sensor was calibrated carefully. The control was executed on a MicroVAX II using the RCI primitives of RCCL [29], which allows the software to directly command robot joint angles. The software package allowed various parameters to be set, and also allows trajectory and wrist displacement data to be logged to a file for subsequent analysis.

4.1 Hybrid Control Demonstration (HYBRI)

HYBRI is a package to demonstrate the hybrid position/force control scheme with a null desired force. Any of six degrees of freedom can be assigned to perform force control or position control interactively. The gains of position and force controls and poles of filters are specified prior to operation in the file *moveh.c*.

When it is being executed, the robot stays at the original position, e.g., *"ready position"*. One may hold the end-point of the wrist by hand and move the lower plate related to the upper such that deflection of the compliant wrist is sensed. At the same time, the robot moves in a way that, in some certain degrees of freedom, it follows the force produced in the wrist. It allows one to lead the robot to move back and forth in these degrees of freedom. In other degrees of freedom, it moves in the opposite direction of the deflection to compensate for position errors. If a full position (or a full force) control is specified, the robot invariantly compensates for the position error (or follows the force error) until the error becomes zero.

The parameters that must be specified at *moveh.c* are as follows.

(1) *gain_pr:* position control gain for three rotational motions, usually can be set close to unity so that the position error can be completely compensated for. That, however, causes the less damping system;

(2) *gain_pt:* position control gain for three translational motions, can be specified as *gain_pr*. Since the measuring systems for rotational motion and translational motion are different, the gain for translational motion usually is usually set two times higher than that for rotational motion;

(3) *gain_fr:* force control gain for three rotational motions, can be set according to the desired compliance and performance. the high gain produces a high effective compliance (i.e., low effective stiffness), which makes the system less damped.

(4) *gain_ft:* force control gain for three translational motions, can be set as *gain_fr*.

(5) *fuzz_j:* "fuzzy" limit for the joint motion (i.e., potentiometer reading) of the wrist, is a measure of the hysteresis in the device. It controls the difference between the current motion and the previous motion of the joint. Namely, when the difference of the motion in the sample interval is smaller than this limit, the current motion is considered as the same as the previous one.

(6) *fuzz_t:* "fuzzy" limit for three translational motions in Cartesian space, is also a measure of the hysteresis in the device. However, it controls the absolute motion in Cartesian space. Namely, if the value of the motion is smaller than this limit, the current motion is considered as zero.

(7) *fuzz_r:* "fuzzy" limit for three rotational motions in Cartesian space. (see (6)).

(8) *s_x_t:* selection switch in X direction to signify force or position control. 1 signifies position control, and 0 signifies the force control. The default is the full position control case.

(9) *s_y_t:* selection switch in Y direction. (see (7)).

(10) *s_z_t:* selection switch in Z direction. (see (7)).

(11) *s_x_r:* selection switch around X direction. (see (7)).

(12) *s_y_r:* selection switch around Y direction. (see (7)).

(13) *s_z_r:* selection switch around Z direction. (see (7)).

(14) *filt_p_pole:* digital filter pole for position control loop, usually can be set as close to unity for good performance. That, however, makes the response slow.

(15) *filt_f_pole:* digital filter pole for force control loop, usually can be set as 0.2 ~ 0.6. The system performance is not quite sensitive to the filter pole in force control. (see the Chapter 7 in the thesis [3])

4.2 Surface Tracking (SURF)

SURF is a package to perform surface tracking operations. One may specify one degree of freedom for force control, and other five degrees of freedom for position control. The degree of freedom specified for force control does not have to be the normal direction of the surface to be tracked, but the direction at which tracking force is controlled. The control gains, desired contact force, tracking velocity, approaching velocity (the velocity prior to contact), turning force (the force that switches the controller from moving in space to tracking), and poles of filters can all be specified interactively.

Supposing the surface is nearly horizontal and force at the Z direction is controlled, robot at first moves towards the surface, upon contact with the surface, it will be switched by the turning contact force from full position control to hybrid control. Immediately after the switch, tracking is started by moving end-effector on the surface while keeping a constant contact force in the Z direction.

The parameters besides those discussed in HYBRI are listed as follows.

(1) *cf:* the desired contact force, is evaluated indirectly as the desired deformation of the wrist device. For that case that the surface is smooth, or the ideal performance is required, a high value is desirable. That, however, causes large friction, and also must be limited in the allowable range of the deformation. The maximum ranges for translational and rotational motions are $\pm 20mm$, and $\pm 20^o$.

(2) *va:* approaching velocity, is dependent on the task specification. For a large approaching velocity, a large contact force is exerted at the moment of contact. In the program, it is specified in the Z axis as approaching direction. It can also be specified in the other directions.

(3) *vt:* tracking velocity, is also dependent on the task. For the large friction surface, increasing the tracking velocity may facilitate, and thus improve the tracking performance. This is specified in the Y axis in the program. It can be also set in the other directions.

(4) *tf:* turning force, is specified according to how much force to switch the controller from the full position control to the hybrid control.

4.3 Edge Tracking (EDGE)

EDGE is a package to perform edge tracking operations. Edge tracking is one of the basic operations of robot manipulation to perform two surface grinding, sliding assembly, as well as exploratory robotics tasks. Any two of three translational components can be specified for force control and the remaining directions are position controlled. A special tool is attached to the upper plate of the compliant wrist. The feedback gains, desired contact forces in both sides of edge, filter poles, tracking and approaching velocities can be specified interactively.

The tracking process can be divided into three stages; approaching, searching, and tracking. The robot at first approaches the edge. Upon making contact with one of the two sides which forms the edge, moving in this direction stops and searching for the other side in that direction begins. When both of sides are being contacted, the end-effector starts to follow the edge along the third direction which is perpendicular to normal directions of both sides that the edge is formed, and is position controlled.

The parameters besides those talked in the previous sections are listed as follows.

(1) *vs:* searching velocity, is specified in the X axis in the program to search for the second surface of the edge.

(2) *cfs:* the desired contact force in the searching direction. In the program, that is specified in the X axis.

(3) *cfa:* the desired contact force in the approaching direction. In the program, that is specified in the Z axis.

4.4 Insertion Operation (INSER and FUZZ)

INSER is a package to perform insertion operation. The axial direction of hole must be specified for position control so that shaft can be inserted at an exact position. In other directions, force is controlled so that the shaft can be adapted to the contact force produced in operation and jamming is prevented.

The operation can be specified either inserting or pulling out. The approaching velocity, inserting velocity, pulling out velocity, feedback gains in each direction, filter poles, desired contact forces for each degree of freedom, and shaft location in the hole can be specified interactively.

FUZZ is a package to perform peg-and-hole operations with a fuzzy controller. The structure of program is based on INSER, but the fuzzy decision rule is employed to assign the gains of force control in different directions. The fuzzy control is discussed in the Chapter 10 of the thesis [3].

The parameters besides those mentioned above are listed as follows.

(1) *pull:* switch to signify inserting or pulling out operation. 0 signifies inserting, and 1 signifies pulling out.

(2) *va:* approaching velocity, is specified according to how fast you want the shaft approach to the hole. (see also the parameters in WRIT, if the hole location is initially specified).

4.5 Writing on Board (WRIT)

WRIT is a package to perform writting operations on an arbitrary surface. The control scheme is the same as that in surface tracking program SURF, but a general trajectory generation is also executed parallelly for position compensation and force control in order to accomplish the writting or painting task.

The discussion in the section of hybrid control is focused on the compliant motion control at a *certain configuration.* Since operations are fulfilled by many sequential configurations of robot manipulators, trajectory generation must be executed simultaneously with position compensation and force control.

We may specify any numbers of configurations being executed sequentially in operations. Each configuration is defined by six Cartesian displacement of the end-effector, i.e., three translations and three rotational motions with respect to the base coordinates. Supposing there are n configurations specified, and X_i and X_i^d are denoted as the current configuration and designed configuration at the sequence i (i=1,...,n), a control algorithm based on the PID law can be derived.

$$\Delta X_i = X_i^d - X_i \tag{28}$$

$$\Delta X_i = (\Delta x_{i1}, \Delta x_{i2}, ..., \Delta x_{i6})^T \tag{29}$$

$$\Delta x_i^{(k+1)} = k_{p_j} \Delta x_i^{(k)} + k_{v_j} (\Delta x_i^{(k)} - \Delta x_i^{(k-1)}) + k_{I_j} \sum_k \Delta x_i^{(k)} \tag{30}$$

where i denotes the sequential desired configuration, j denotes each component of six displacements in Cartesian space, k denotes the present time. k_{p_j}, k_{v_j}, and k_{I_j} are the proportional, derivative, and integral

gains. The motion $\Delta x_i^{(k+1)}$ is controlled within a limited range for translation and rotation respectively so that an excess velocity is prevented.

The operation can be specified as follows. The compliance at the direction of the surface normal has to be sensitively controlled so that *pen* is prevented from being broken due to an excess contact force, or being departed from the surface due to concavity of the surface. The position and orientation of the end-effector in other directions has to be controlled so that pen is kept in a proper posture. The trajectory has to be generated sequentially so that writting task can be accomplished. Based on this specification, we designed a controller which controls force in the surface normal and controls position in the remaining direction, and generates trajectory of the end-effector simultaneously. The velocity of writting operation is determined by the trajectory generator. The tolerance of positioning error, maximum velocity in each axis, desired contact force, approaching velocity, and other parameters can be specified interactively.

The parameters besides those discussed above are listed as follows.

(1) *PPgain:* proportional gain of the trajectory generation for the three translational motions. The large gain causes a fast motion, which is limited by *vlim_r*.

(2) *PRgain:* proportional gain of the trajectory generation for the three rotational motions.

(3) *DPgain:* derivative gain of the trajectory generation for the three translational motions.

(4) *DRgain:* derivative gain of the trajectory generation for the three rotational motions.

(5) *IPgain:* integral gain of the trajectory generation for the three translational motions.

(6) *IRgain:* integral gain of the trajectory generation for the three rotational motions.

(7) *vlim_t:* velocity limit for translational motion.

(8) *vlim_r:* velocity limit for rotational motion.

(9) *errtolerance:* the positioning error tolerance for the trajectory generation. Within this range, the destination is considered to have been reached.

(10) *deri_gain_ft:* derivative gain in force control loop for three translational motions.

(11) *deri_gain_fr:* derivative gain in force control loop for three rotational motions.

(12) *deri_gain_pt:* derivative gain in position control loop for three translational motions.

(13) *deri_gain_pr:* derivative gain in position control loop for three rotational motions.

5. SOURCE CODE

This is the source code of compliant wrist control designed and implemented in the GRASP Laboratory. The control is executed on a MicroVax II using RCI primitives of RCCL [29], and is implemented in PUMA 560 with Index machine. The programs under the directory */randd/rw/webster* are listed as follows.

makefile
hybri.c
moveh.c
surf.c
moves.c
edge.c
moveg.c
inser.c
movei.c
fuzz.c
movef.c
writ.c
movew.c
rw.h
recordd.c
filt.c

The user is suggested to read *hybri.c* and *moveh.c* in the beginning to learn the definitions of parameters since some similar descriptions are omitted in the other programs for simplification.

```
CFLAGS = -g -I/usr/tools/include
CLIBS   = -laxv11

OBJS1   = moveh.o hybri.o
OBJS2   = moves.o surf.o
OBJS3   = moveg.o edge.o
OBJS4   = movei.o inser.o
OBJS5   = movef.o fuzz.o
OBJS6   = movew.o writ.o
OBJS7   = mh.o test.o

REALOBJS  =
hybri: ${OBJS1} recordd.h
        rcc ${CFLAGS} -o hybri ${OBJS1} -lm REAL $(REALOBJS) ${CLIBS}
surf: ${OBJS2} recordd.h
        rcc ${CFLAGS} -o surf ${OBJS2} -lm REAL $(REALOBJS) ${CLIBS}
edge: ${OBJS3} recordd.h
        rcc ${CFLAGS} -o edge ${OBJS3} -lm REAL $(REALOBJS) ${CLIBS}
inser: ${OBJS4} recordd.h
        rcc ${CFLAGS} -o inser ${OBJS4} -lm REAL $(REALOBJS) ${CLIBS}
fuzz: ${OBJS5} recordd.h
        rcc ${CFLAGS} -o fuzz ${OBJS5} -lm REAL $(REALOBJS) ${CLIBS}
writ: ${OBJS6} recordd.h
        rcc ${CFLAGS} -o writ ${OBJS6} -lm REAL $(REALOBJS) ${CLIBS}
test: ${OBJS7} recordd.h
        rcc ${CFLAGS} -o test ${OBJS7} -lm REAL $(REALOBJS) ${CLIBS}

move.o: record.h
moved.o: recordd.h

filt:   filt.o
        cc -o filt filt.o /graspusr/pic/robot/lib/libjac.a -lm

filt.o: filt.c
        cc -c -I /graspusr/pic/robot/h filt.c
```

```
/*
 *      File:   hybri.c
 *      Remarks: Hybrid position force control for the null desired force
 *               with the compliant wrist system (ref: moveh.c)
 */

#include        <stdio.h>
#include        <rccl/rccl.h>
#include        <rccl/rci.h>
#include        <rccl/kine.h>
#include        "rw.h"
#include        "recordd.h"

#define N       6
#define dist_of_flange 55.88   /* This is the distance from the center of the
                                  last three joints of Puma 560 to the outer
                                  surface of the flange to which the compliant
                                  wrist is attached */

#define mount_dist  35.0       /* distance from the surface of the flange to
                                  the mounting plate of the compliant wrist
                                  in mm */


/*
 * the first order digital filter
 */
#define FILTER(y, u, pole)      (y=(1.0-pole)*(y*pole/(1.0-pole)+u))
/*
 * the recording function
 */
#define NCOPY(a,b)      for(i=0;i<N;i++)a[i]=b[i]

double  car_diffs[6];       /* cartesian space displacements of the
                               compliant wrist device, first three are for
                               translational displacements and the other
                               three for rotational displacements */
TRSF    mem_tw,             /* tw is the 4x4 matrix of the compliant wrist
                               kinematics */
        *tw = &mem_tw;

extern double   gain_ft,    /* force control gain for translational motion*/
                gain_fr,    /* force control gain for rotational motion*/
                gain_pt,    /* position control gain for translational motion*/
                gain_pr,    /* position control gain for rotational motion*/
                fuzz_j,     /* "fuzzy" limit for six joints of the wrist*/
                fuzz_t,     /* "fuzzy" limit for three translational cartesian
                               space motion of the wrist*/
                fuzz_r,     /* "fuzzy" limit for three rotational cartesian
                               space motion of the wrist*/
                filt_f_pole,/* digital filter pole for force control loop*/
                filt_p_pole,/* digital filter pole for position control loop*/
                s_x_t,      /* selection of position control ot force control
                               1 signifies position control, and 0 signifies
                               force control. s_x_t is that for translational
                               motion in x axis, s_y_r is that for rotational
                               motion around y axis, so as the others */
                s_y_t,
                s_z_t,
                s_x_r,
                s_y_r,
                s_z_r;
#define FUZZ            fuzz_j
#define FUZZ_CARPOS     fuzz_t
#define FUZZ_CAREUL     fuzz_r

DIFF    d_f, d_p, dvel;         /* d_f is the cartesian error updated in rw() */
                                /* representing force control, and d_p is that*/
                                /* representing position compensation. dvel is*/
                                /* the cartesian error specified by trajectory*/
                                /* generator which is updated by drive()      */

double  jang_o[RW_MAX_JOINT];  /* previous joint angles of the wrist device  */
double  rw_theta_cal[RW_MAX_JOINT],         /* pots reading corresponding to
                                               the home position angles of
                                               each joint rw_theta_bar */
        rw_theta_bar[RW_MAX_JOINT] = {      /* home position of the joint
                                               in the wrist device */
                        RW_THETA_BAR_0,
                        RW_THETA_BAR_1,
                        RW_THETA_BAR_2,
                        RW_THETA_BAR_3,
                        RW_THETA_BAR_4,
                        RW_THETA_BAR_5
        };
int     rclconst();
struct _record  rec;            /* record file variable */
int             sync,           /* user/interrupt coordination flag */
                time;           /* time maintained by interrupt function */
FILE            *recfp;         /* record file fp */

int     verbose;                /* print-out switch during operation */
double  d_angles[N];            /* desired cartesian motion with consideration
                                   of force control and trajectory
                                   specification without position
                                   compensation*/
double  r_angles[N];            /* required cartesian motion based on d_angles
                                   with consideration of position compensation
                                   */
SNCS    sncs;

/*
 * Here is where we initialize everything needed to make the robot do
 * what we want it to.  We establish the position equation that the
 * main program will continually move to.
 */
rw_init()
{
        int     i;
        void    dummy(), drive();


        setbuf(stdout, NULL);
        rw_cal();                   /* calibrate the compliant wrist */

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                jang_o[i] = 0.0;
        }

        tw = newtrans("tw",rw);  /* initialize the 4X4 matrix tw */
        rw_car_update_tw();
        printrn(tw,stdout);      /* transform print routine */
        /*
         * start the real-time process, and request arm power
         */
        RCIopen();                      /* open RCI */
        RCIcontrol(dummy, drive);       /* RCI controls arm */
        chg.power_on.com = YES;
```

```
        if ((how.state & CALIB_OK) == 0) {
                fprintf(stderr, "arm not calibrated\n");
                exit(3);
        }
}

/*
 *      Real-time drive function
 *
 *      read joint angles
 *      compute Jacobian at this point
 *      transform cartesian diff to joint space
 *      compute new joint angles
 *      output setpoint
 */
double  car_diffs[N];
double  r_angles[N];
double  d_angles[N];
void
drive()
{
        double  del_force[N], del_posn[N], del_angle_vel[N];
        short   encs[N];
        JNTS    q_f, q_p, qvel;
        int     i;
        static int      initd = 0;
        static double   del_f_smth[N];
        static double   del_p_smth[N];

        if (initd == 0) {
                enctoang(r_angles, how.pos);    /* get actual joint angles */
                for (i=0; i<N; i++) {
                        d_angles[i] = r_angles[i];
                }
                initd++;
        }
        update_sincos(&sncs, r_angles);         /* compute sin/cos */
        update_jacobian_terms(&sncs);           /* compute jacob terms */

        jacobI(&q_f, &d_f, &sncs, 0.0);         /* jacobian inverse to
                                                   transform the cartesian
                                                   error d_f to the joint
                                                   space error q_f representing
                                                   force control */
        jacobI(&q_p, &d_p, &sncs, 0.0);         /* jacobian inverse to
                                                   transform the cartesian
                                                   error d_p to the joint
                                                   space error q_p representing
                                                   position control */
        jnts_to_angle(del_force, &q_f);         /* assign the joint angles
                                                   pointed to by &q_f to an
                                                   array of double given by
                                                   del_force */
        jnts_to_angle(del_posn, &q_p);          /* assign the joint angles
                                                   pointed to by &q_p to an
                                                   array of double given by
                                                   del_posn */

        jacobI(&qvel, &dvel, &sncs, 0.0);       /* jacobian inverse to
                                                   transform the cartesian
                                                   error dvel to the joint
                                                   space error qvel
                                                   representing trajectory
                                                   generation */
        jnts_to_angle(del_angle_vel, &qvel);    /* assign the joint angles
                                                   pointed to by &qvel to an
                                                   array of double given by
                                                   del_angle_vel */

        /*
         * digital filter. del_force and del_posn are input, and
         * del_f_smth and del_p_smth are output. filter poles can be set
         * in moveh.c
         */
        for (i=0; i<N; i++) {
                FILTER(del_f_smth[i], del_force[i], filt_f_pole);
                FILTER(del_p_smth[i], del_posn[i], filt_p_pole);
                }

        /*
         * control function. The final required motion r_angles are computed
         * by force control term del_f_smth, position compensation term
         * del_p_smth, and trajectory generation term del_angle_vel.
         */
        for (i=0; i<N; i++)
                d_angles[i] += del_f_smth[i] + del_angle_vel[i];
        for (i=0; i<N; i++)
                r_angles[i] = d_angles[i] + del_p_smth[i];

        /*
         * record the current angles of robot, and cartesian error of the
         * wrist device
         */
        NCOPY(rec.r_angles, r_angles);
        NCOPY(rec.car_diffs, car_diffs);
        rec.time = time;

        angtoenc(encs, r_angles);       /* output new joint angles */
        sync++;                         /* tell user process we have data */
        time++;

        for (i=0; i<N; i++) {
                chg.motion[i].com = POS;
                chg.motion[i].value = encs[i];
        }
}


void
dummy() {}

/*
 * rw_cal reads the current pot settings to get the current joint
 * angles.  These are then subtracted from the "correct" angles to
 * get the correction angles.
 */

rw_cal()
{
        int     i,
                j;

        /*
         * Initialize the axv11 board.
         */

        if (ax_init() < 0)
        {
```

```
/*
 * Real-time drive function
 *
 *      read joint angles
 *      compute Jacobian at this point
 *      transform cartesian diff to joint space
 *      compute new joint angles
 *      output setpoint
 */
double  car_diffs[N];
double  r_angles[N];            /* robot joint angles */
double  d_angles[N];            /* wrist joint angles */
double  error;
void
drive()
{
        double  del_force[N], del_angle_vel[N];
        double  del_posn[N];
        short   encs[N];
        TRSF    T6err;
        JNTS    q_f, qvel;
        JNTS    q_p;
        static JNTS     q_j6;
        int     i;
        static int      initd = 0;
        static double   del_f_smth[N];
        static double   del_p_smth[N];
        static DIFF     T6err_integ;     /* the error transform sum value for
                                          * the integral term */
        static DIFF     T6err_deri;      /* the error transform update value
                                          * for the derivative term */
        float           yawerr, pitcherr, rollerr; /* orentational error
                                                    * for trajectory generation
                                                    */
        if (initd == 0) {
                enctoang(r_angles, how.pos);    /* get actual joint angles */
                for (i=0; i<N; i++) {
                        d_angles[i] = r_angles[i];
                }
                q_j6.conf = "blah";
                initd++;
        }

        /*
         * copy the joint angles into a JNTS
         */
        {
                real    *f = &(q_j6.th1), *g = &(jmin_c.th1);

                for (i=0; i<6; i++)
                        *f++ = r_angles[i] - *g++;
        }

        /*
         * do the forward kinematics to find current cartesian position,
         * also updates all sin/cos values in sncs.
         */
        jns_to_tr(&T6, &q_j6, &sncs);

        /*
         * trajectory generation
         *
         *      e =  T6^-1 * Pd
         *      xdot = P*e + D*de/dt + I*$ e dt
         *
         * where P. D and I are empirically determined control gains
         */
        invert(&T6err,&T6);         /* T6err = T6^-1 */
        trmultinp(&T6err,&Pd);      /* T6err = T6err * Pd */

        noatorpy(&rollerr, &pitcherr, &yawerr, &T6err);
        /* get roll, pitch, yaw from transform T6err */

        /*
         * PID controller for trajectory generation.
         */

        dvel.t.x = T6err.p.x * PPgain + (T6err.p.x - T6err_deri.t.x) * DPgain
                + T6err_integ.t.x * IPgain;
        dvel.t.y = T6err.p.y * PPgain + (T6err.p.y - T6err_deri.t.y) * DPgain
                + T6err_integ.t.y * IPgain;
        dvel.t.z = T6err.p.z * PPgain + (T6err.p.z - T6err_deri.t.z) * DPgain
                + T6err_integ.t.z * IPgain;

        dvel.r.x = yawerr * PRgain + (yawerr - T6err_deri.r.x) * DRgain +
                T6err_integ.r.x * IRgain;
        dvel.r.y = pitcherr * PRgain + (pitcherr - T6err_deri.r.y) * DRgain +
                T6err_integ.r.y * IRgain;
        dvel.r.z = rollerr * PRgain + (rollerr - T6err_deri.r.z) * DRgain +
                T6err_integ.r.z * IRgain;

        /*
         * set the max velocities for the motion along and around each axis
         */
        if( dvel.t.x <= -VLIM_T)
        dvel.t.x = -VLIM_T;
        if( dvel.t.x >= VLIM_T)
        dvel.t.x = VLIM_T;
        if( dvel.t.y <= -VLIM_T)
        dvel.t.y = -VLIM_T;
        if( dvel.t.y >= VLIM_T)
        dvel.t.y = VLIM_T;
        if( dvel.t.z <= -VLIM_T)
        dvel.t.z = -VLIM_T;
        if( dvel.t.z >= VLIM_T)
        dvel.t.z = VLIM_T;

        if( dvel.r.x <= -VLIM_R)
        dvel.r.x = -VLIM_R;
        if( dvel.r.x >= VLIM_R)
        dvel.r.x = VLIM_R;
        if( dvel.r.y <= -VLIM_R)
        dvel.r.y = -VLIM_R;
        if( dvel.r.y >= VLIM_R)
        dvel.r.y = VLIM_R;
        if( dvel.r.z <= -VLIM_R)
        dvel.r.z = -VLIM_R;
        if( dvel.r.z >= VLIM_R)
        dvel.r.z = VLIM_R;

        /*
         * update the privious error for derivative
         */
        T6err_deri.t.x = T6err.p.x;
        T6err_deri.t.y = T6err.p.y;
        T6err_deri.t.z = T6err.p.z;
        T6err_deri.r.x = yawerr;
        T6err_deri.r.y = pitcherr;
```

```
        T6err_deri.r.z = rollerr;

        /*
         * update the error integral
         */
        T6err_integ.t.x += T6err.p.x;
        T6err_integ.t.y += T6err.p.y;
        T6err_integ.t.z += T6err.p.z;
        T6err_integ.r.x += yawerr;
        T6err_integ.r.y += pitcherr;
        T6err_integ.r.z += rollerr;

        /*
         * update Pd if there are several destinations specified
         */
        error = fabs( T6err.p.x * T6err.p.x + T6err.p.y * T6err.p.y
                    + T6err.p.z * T6err.p.z);
        if ( error < errtolerance ) {
                if (++itarg < ntarg )
                        Pd = target[itarg];
        }
        update_jacobian_terms(&sncs);           /* compute jacob terms */

        jacobI(&q_f, &d_f, &sncs, 0.0);         /* transform diff in cartesian
                                                   space to joint space */
        jacobI(&q_p, &d_p, &sncs, 0.0);         /* transform diff in cartesian
                                                   space to joint space */
        jnts_to_angle(del_force, &q_f);         /* delta joint to angles */
        jnts_to_angle(del_posn, &q_p);          /* delta joint to angles */

        jacobI(&qvel, &dvel, &sncs, 0.0);       /* transform diff in cartesian
                                                   space to joint space */
        jnts_to_angle(del_angle_vel, &qvel);    /* delta joint to angles */

        for (i=0; i<N; i++) {
                FILTER(del_f_smth[i], del_force[i], filt_f_pole);
                FILTER(del_p_smth[i], del_posn[i], filt_p_pole);
        }

        for (i=0; i<N; i++)
                d_angles[i] += del_f_smth[i] + del_angle_vel[i];
        for (i=0; i<N; i++)
                r_angles[i] = d_angles[i] + del_p_smth[i];
        NCOPY(rec.r_angles, r_angles);
        NCOPY(rec.car_diffs, car_diffs);
        rec.time = time;

        angtoenc(encs, r_angles);
        sync++;                 /* tell user process we have data */
        time++;

        for (i=0; i<N; i++) {
                chg.motion[i].com = POS;
                chg.motion[i].value = encs[i];
        }
}

void
dummy() {}

/*
 * rw_cal reads the current pot settings to get the current joint
 * angles.  These are then subtracted from the "correct" angles to
 * get the correction angles.
 */

rw_cal()
{
        int     i,
                j;

        /*
         * Initialize the axv11 board.
         */

        if (ax_init() < 0)
        {
                fprintf(stderr,"Cannot initialize the axv11 board\n");
                exit(1);
        }

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                rw_theta_cal[i] = rw_ptor(i);
        }

}

/*
 * This is the routine that has to figure out how to change the postion
 * equation established in rw_init() so the robot is driven to where we
 * want it to go.
 */
rw()
{
        int     i;

        static DIFF     car_deri;
        JNTS curr_jnts;
        JNTS diff_jnts;
        if (rw_comp)
        {
                bcopy(j6, &curr_jnts, sizeof(JNTS));
                rw_car_update_tw();
                if (verbose>1)
                        printrn(tw,stdout);
                noatorpy(&car_diffs[5],&car_diffs[4],&car_diffs[3],tw);
                car_diffs[0] = tw->p.x;
                car_diffs[1] = tw->p.y;
                car_diffs[2] = tw->p.z - 62.0;

         if( car_diffs[0] <= FUZZ_CARPOS && car_diffs[0] >= -FUZZ_CARPOS )
                car_diffs[0] = 0.0;
         if( car_diffs[1] <= FUZZ_CARPOS && car_diffs[1] >= -FUZZ_CARPOS )
                car_diffs[1] = 0.0;
         if( car_diffs[2] <= 0.5 * FUZZ_CARPOS && car_diffs[2] >=
           - 0.5 * FUZZ_CARPOS )
                car_diffs[2] = 0.0;
         if( car_diffs[3] <= FUZZ_CAREUL && car_diffs[3] >= -FUZZ_CAREUL )
                car_diffs[3] = 0.0;
         if( car_diffs[4] <= FUZZ_CAREUL && car_diffs[4] >= -FUZZ_CAREUL )
                car_diffs[4] = 0.0;
         if( car_diffs[5] <= FUZZ_CAREUL && car_diffs[5] >= -FUZZ_CAREUL )
                car_diffs[5] = 0.0;
                if (verbose) {
                        for (i=0; i<N; i++)
                                printf("%10.2f ", car_diffs[i]);
                        putchar('\n');
```

```
                        printy(&T6, stdout);
                }

        /*
         * PD control for the hybrid control loop, force control in
         * the z direction, and position is controlled in the others
         */
        d.t.x = car_diffs[0] * gain_pt + car_deri.t.x * deri_gain_pt;
        d.t.y = car_diffs[1] * gain_pt + car_deri.t.y * deri_gain_pt;
        d.t.z = (car_diffs[2] + cf) * gain_ft + car_deri.t.z
                * deri_gain_ft;
        d.r.x = dtor(car_diffs[3]) * gain_pr
                + car_deri.r.x * deri_gain_pr;
        d.r.y = dtor(car_diffs[4]) * gain_pr
                + car_deri.r.y * deri_gain_pr;
        d.r.z = dtor(car_diffs[5]) * gain_pr
                + car_deri.r.z * deri_gain_pr;

        /*
         * update the previous deflection for derivative term
         */
        car_deri.t.x = car_diffs[0];
        car_deri.t.y = car_diffs[1];
        car_deri.t.z = car_diffs[2];
        car_deri.r.x = car_diffs[3];
        car_deri.r.y = car_diffs[4];
        car_deri.r.z = car_diffs[5];

        /*
         * force control mode in hybrid control
         */
        d_f.t.x = 0.0;
        d_f.t.y = 0.0;
        d_f.t.z = d.t.z;
        d_f.r.x = 0.0;
        d_f.r.y = 0.0;
        d_f.r.z = 0.0;

        /*
         * position control mode in hybrid control
         */
        d_p.t.x = -d.t.x;
        d_p.t.y = -d.t.y;
        d_p.t.z = 0.0;
        d_p.r.x = -d.r.x;
        d_p.r.y = -d.r.y;
        d_p.r.z = -d.r.z;

        if (recfp && sync) {
                fwrite(&rec, sizeof(rec), 1, recfp);
                sync = 0;
        }
        }
}

double
rw_jang(i)
int     i;
{
        double  jang,
                jang_diff,
                newdiff;

        int     s;

#ifdef notdef
        newdiff = rw_raw_diff(i);
        rw_old_diff[i] = newdiff;

#endif
        jang =  rw_raw_diff(i);

        jang_diff = (jang - jang_o[i]);
        if (jang_diff <= FUZZ && jang_diff >= -FUZZ)
        {
                        jang = jang_o[i];
        }
        jang_o[i] = jang;

        jang += rw_theta_bar[i];
        return(jang);
}

rw_car_update_tw()
{
        double  c1,
                c2,
                c3,
                c4,
                c5,
                c6,
                s1,
                s2,
                s3,
                s4,
                s5,
                s6,
                c23,
                s23,
                x1,
                x2,
                x3,
                x4,
                x5,
                x6,
                x7,
                x8,
                x9,
                x10;

        c1 = cos(rw_jang(0));
        s1 = sin(rw_jang(0));

        c2 = cos(rw_jang(1));
        s2 = sin(rw_jang(1));

        c3 = cos(rw_jang(2));
        s3 = sin(rw_jang(2));

        c4 = cos(rw_jang(3));
        s4 = sin(rw_jang(3));

        c5 = cos(rw_jang(4));
        s5 = sin(rw_jang(4));

        c6 = cos(rw_jang(5));
        s6 = sin(rw_jang(5));
```

```
        c23 = cos(rw_jang(1)+rw_jang(2));
        s23 = sin(rw_jang(1)+rw_jang(2));

        x1 = -c4*s5;
        x2 = c4*c5*c6 - s4*s6;
        x3 = c4*c5*s6 + s4*c6;
        x4 = -s4*s5;
        x5 = s4*c5*c6 + c4*s6;
        x6 = s4*c5*s6 - c4*c6;
        x7 = L8*x3 + L7*x1 - L5*s4 + L5;
        x8 = L8*s5*s6 + L7*c5 - L6;
        x9 = -L8*x6 - L7*x4 - L5*c4 - L4 + L2;
        x10 = L3*c2 + c23*x7 + s23*x8;

        tw->n.x = c1*(c23*x1 + s23*c5) - s1*x4;
        tw->o.x = c1*(c23*x2 + s23*s5*c6) - s1*x5;
        tw->a.x = c1*(c23*x3 + s23*s5*s6) - s1*x6;
        tw->p.x = c1*(L3*c2 + x7*c23 + s23*x8) + s1*x9 - L9;

        tw->n.y = s1*(c23*x1 + s23*c5) + c1*x4;
        tw->o.y = s1*(c23*x2 + s23*s5*c6) + c1*x5;
        tw->a.y = s1*(c23*x3 + s23*s5*s6) + c1*x6;
        tw->p.y = s1*(L3*c2 + c23*x7 + s23*x8) - c1*x9 + L3;

        tw->n.z = s23*x1 - c23*c5;
        tw->o.z = s23*x2 - c23*s5*c6;
        tw->a.z = s23*x3 - c23*s5*s6;
        tw->p.z = s23*x7 - c23*x8 + L3*s2 + L1;

}

rw_close()
{
        RCIrelease(1);
        RCIclose (1);
}

rccl_close()
{
}

/*
 * called on ^C to close the record file if it was open
 */
void
quit()
{
        if (recfp)
                fclose(recfp);
}
```

```
/*
 *      File:   movew.c
 *      Remarks: Writing (or drawing) operation on an unmodeled
 *               surface with the compliant wrist system (ref: writ.c)
 */

#include <stdio.h>
#include "rw.h"
double  gain_ft,
        gain_fr,
        gain_pt,
        gain_pr,
        deri_gain_ft,
        deri_gain_fr,
        deri_gain_pt,
        deri_gain_pr,
        fuzz_j,
        fuzz_t,
        fuzz_r,
        filt_f_pole,
        filt_p_pole,
        vlim_t,
        vlim_r,
        cf,
        errtolerance;

int     ntarg = 31;
int     itarg = 0;
int     rw_comp = 0;
int     verbose;
extern FILE     *recfp;
extern TRSF     Pd;
extern TRSF     target[31];
extern double   PPgain, PRgain;
extern double   DPgain, DRgain;
extern double   IPgain, IRgain;

main(ac, av)
int     ac;
char    **av;
{
        double  atof();

        gain_pr = 0.3;
        gain_pt = 0.6;
        gain_fr = 0.005;
        gain_ft = 0.02;
        deri_gain_pr = 0.0;
        deri_gain_pt = 0.0;
        deri_gain_fr = 0.0;
        deri_gain_ft = 0.0;
        fuzz_j = .01;
        fuzz_r = 0.2;
        fuzz_t = 0.2;
        filt_f_pole = 0.4;
        filt_p_pole = 0.95;
        vlim_t = 1.0;
        vlim_r = 0.2;
        cf = 0.35;
        errtolerance = 2.0;
        PPgain = 0.035;
        PRgain = 0.0001;
        DPgain = 0.005;
        DRgain = 0.0008;
        IPgain = 0.0000001;
        IRgain = 0.00000001;
        while (--ac > 0  && **++av == '-') {
                register char   *p = *av;

                while (*++p != '\0')
                        switch (*p) {
                        case 'v':
                                verbose++; break;
                        case 'P':
                                PPgain = atof(&p[1]); break;
                        case 'S':
                                PRgain = atof(&p[1]); break;
                        case 'D':
                                DPgain = atof(&p[1]); break;
                        case 'E':
                                DRgain = atof(&p[1]); break;
                        case 'I':
                                IPgain = atof(&p[1]); break;
                        case 'J':
                                IRgain = atof(&p[1]); break;
                        case 'R':
                        if ((recfp = fopen(&p[1], "w")) == NULL) {
                        fprintf(stderr,
                                "cant open file for write\n"
                                );
                                exit(3);
                                }
                                goto nextarg;
                        }
nextarg:        ;
        }
        printf("proporl gain of translation is %f\n", PPgain);
        printf("proporl gain of rotation is %f\n", PRgain);
        printf("integl gain of translation is %f\n", IPgain);
        printf("integl gain of rotation is %f\n", IRgain);
        printf("deri gain of translation is %f\n", DPgain);
        printf("deri gain of rotation is %f\n", DRgain);

        /*
         * specify trajectory sequentially. the following one is
         * for the cartoon of human's side view looking
         */
        trsl(&target[0],-614.0, 150.0, 52.5);
        rpy(&target[0], 0.0, 0.0, -180.0);
        trsl(&target[1],-614.0+23.0, 150.0+48.0, 52.5);
        rpy(&target[1], 0.0, 0.0, -180.0);
        trsl(&target[2],-614.0+19.0, 150.0+55.0, 52.5);
        rpy(&target[2], 0.0, 0.0, -180.0);
        trsl(&target[3],-614.0+47.0, 150.0+105.0, 52.5);
        rpy(&target[3], 0.0, 0.0, -180.0);
        trsl(&target[4],-614.0+7.0, 150.0+101.0, 52.5);
        rpy(&target[4], 0.0, 0.0, -180.0);
        trsl(&target[5],-614.0+5.0, 150.0+90.0, 52.5);
        rpy(&target[5], 0.0, 0.0, -180.0);
        trsl(&target[6],-614.0+9.0, 150.0+120.0, 52.5);
        rpy(&target[6], 0.0, 0.0, -180.0);
        trsl(&target[7],-614.0-22.0, 150.0+122.0, 52.5);
        rpy(&target[7], 0.0, 0.0, -180.0);
        trsl(&target[8],-614.0+12.0, 150.0+130.0, 52.5);
        rpy(&target[8], 0.0, 0.0, -180.0);
        trsl(&target[9],-614.0+6.0, 150.0+133.0, 52.5);
        rpy(&target[9], 0.0, 0.0, -180.0);
        trsl(&target[10],-614.0+10.0, 150.0+166.0, 52.5);
```

```
        rpy(&target[10], 0.0, 0.0, -180.0);
        trsl(&target[11],-614.0-60.0, 150.0+147.0, 52.5);
        rpy(&target[11], 0.0, 0.0, -180.0);
        trsl(&target[12],-614.0, 150.0+147.0, 65.0);
        rpy(&target[12], 0.0, 0.0, -180.0);
        trsl(&target[13],-614.0, 150.0+187.0, 52.5);
        rpy(&target[13], 0.0, 0.0, -180.0);
        trsl(&target[14],-614.0, 150.0+187.0+40.0, 52.5);
        rpy(&target[14], 0.0, 0.0, -180.0);
        trsl(&target[15],-614.0-20.0, 150.0+187.0+30.0, 52.5);
        rpy(&target[15], 0.0, 0.0, -180.0);
        trsl(&target[16],-614.0-20.0, 150.0+187.0+30, 65.0);
        rpy(&target[16], 0.0, 0.0, -180.0);
        trsl(&target[17],-614.0-30.0, 150.0+187.0-10.0, 52.5);
        rpy(&target[17], 0.0, 0.0, -180.0);
        trsl(&target[18],-614.0-30.0, 150.0+187.0+40.0, 52.5);
        rpy(&target[18], 0.0, 0.0, -180.0);
        trsl(&target[19],-614.0-50.0, 150.0+187.0+30.0, 52.5);
        rpy(&target[19], 0.0, 0.0, -180.0);
        trsl(&target[20],-614.0-50.0, 150.0+187.0-10.0, 52.5);
        rpy(&target[20], 0.0, 0.0, -180.0);
        trsl(&target[21],-614.0-50.0, 150.0+187.0-10.0, 65.0);
        rpy(&target[21], 0.0, 0.0, -180.0);
        trsl(&target[22],-614.0-20.0, 150.0+187.0, 52.5);
        rpy(&target[22], 0.0, 0.0, -180.0);
        trsl(&target[23],-614.0-50.0, 150.0+187.0, 52.5);
        rpy(&target[23], 0.0, 0.0, -180.0);
        trsl(&target[24],-614.0-50.0, 150.0+187.0, 65.0);
        rpy(&target[24], 0.0, 0.0, -180.0);
        trsl(&target[25],-614.0-70.0, 150.0+187.0, 52.5);
        rpy(&target[25], 0.0, 0.0, -180.0);
        trsl(&target[26],-614.0-60.0, 150.0+187.0+40.0, 52.5);
        rpy(&target[26], 0.0, 0.0, -180.0);
        trsl(&target[27],-614.0-80.0, 150.0+187.0+30.0, 52.5);
        rpy(&target[27], 0.0, 0.0, -180.0);
        trsl(&target[28],-614.0-80.0, 150.0+187.0-10.0, 52.5);
        rpy(&target[28], 0.0, 0.0, -180.0);
        trsl(&target[29],-614.0-80.0, 150.0+187.0+42.0, 52.5);
        rpy(&target[29], 0.0, 0.0, -180.0);
        trsl(&target[30],-614.0-80.0, 150.0+187.0+42.0, 65.0);
        rpy(&target[30], 0.0, 0.0, -180.0);

        Pd = target[0];

        rw_comp = 0;

        /*
         * initialize the control loop.
         */
        rw_init();

        /*
         * start the complience. watch out.
         */
        rw_comp = 1;

        for(;;)
          rw();
}
```

```
/*
 *      File:   rw.h
 *      Remarks: Kinematics and other parameters definitions of sensing
 *               mechanism of the compliant wrist
 */

#include <stdio.h>
#include <math.h>

#include <rccl/rccl.h>
#include <local/axv11.h>

#define RW_THETA_BAR_0  dtor(-90.0)
#define RW_THETA_BAR_1  dtor(0.0)
#define RW_THETA_BAR_2  dtor(0.0)
#define RW_THETA_BAR_3  dtor(90.0)
#define RW_THETA_BAR_4  dtor(-90.0)
#define RW_THETA_BAR_5  dtor(90.0)

#define RW_MAX_JOINT    AX_MAX_CHANNEL

/*
 * Joint lengths in mm
 */
#define L1      23.0
#define L2      22.0
#define L3      15.0
#define L4      15.0
#define L5      26.0
#define L6      26.0
#define L7      35.0
#define L8      13.0
#define L9      28.0

/*
 * degree to rad
 */
#define dtor(d)         ((d) * (M_PI/180.0))

/*
 * rad to degree
 */
#define rtod(r)         ((r) * (180.0/M_PI))

/*
 * voltage of pots range
 */
#define POT_VOLT_RANGE  0.5

/*
 * pots reading to voltage
 */
#define bvolt(p)        (rw_rest_volt[(p)] - (POT_VOLT_RANGE/2.0))

/*
 * pots reading to voltage
 */
#define rw_ptov(p)      ((double)ax_pot(p) * (1.25 / 4096.0))

/*
 * voltage to degree
 */
#define rw_vtod(v)      ((v) * 27.0 / 0.5)

/*
 * pots reading to degree
 */
#define rw_ptod(p)      (rw_vtod(rw_ptov(p)))

/*
 * pots reading to rad
 */
#define rw_ptor(p)      (dtor(rw_ptod(p)))

/*
 * get the difference between the current reading and the calibrated value
 */
#define rw_raw_diff(i)  ((rw_ptor(i) - rw_theta_cal[i]) )

/*
 * the joint angle actually used in the wrist kinematics
 */
double          rw_jang();

extern double   rw_theta_cal[],
                rw_theta_bar[];

extern  TRSF    *tw;
extern  TRSF    *tw_inv;
extern  TRSF    *ee;

extern  int     rw_comp;
extern  int     rw();
extern  int     rw_cal();
extern  int     rclconst();
```

```
/*
 * data saved during execution of carjd
 */
struct _record {
        int     time;
        float   r_angles[6];
        float   car_diffs[6];
};
```

```
/*
 *      File:   filt.c
 *      Remarks: Filter the required data from recorded data file
 */

#include        <stdio.h>
#include        "record.h"
#include        "rfms.h"

#define N       6

main()
{
        struct _record  rec;
        int             i;
        JOINT           J;
        TRANSFORM       t6;


        while (fread(&rec, sizeof(rec), 1, stdin)) {
                printf("%%t %d\n", rec.time);
                putrec(rec.angles, "a");
                putrec(rec.del_angles, "d");
                for (i=0; i<N; i++)
                        J.q[i] = rec.angles[i];
                jnt_to_tr(&t6, &J);
                printf("%%e %f %f %f %f %f %f\n",
                        t6.p.x, t6.p.y, t6.p.z, t6.o.x, t6.o.y, t6.o.z
                );
        }
}
putrec(v, name)
float   *v;
char    *name;
{
        int     i;

        printf("%%%s ", name);
        for (i=0; i<N; i++)
                printf("%f  ", v[i]);
                putchar('\n');
}
```

6. BIBLIOGRAPHY

[1] Y. Xu and R.P. Paul, "On position compensation and force control stability of a robot with a compliant wrist", *Proceedings of the IEEE International Conference on Robotics and Automation,* P.1173-1178, 1988.

[2] R.P. Paul, Y. Xu, and X. Yun, "Terminal link force and position control of a robot manipulator", *Proceedings of Seventh CISM and IFToMM International Symposium on Theory and Practice of Robots and Manipulators,* Udine, Italy, 1988.

[3] Y. Xu, *"Compliant wrist design and hybrid position/force control of robot manipulators",* Ph.D Dissertation, Department of Mechanical Engineering and Applied Mechanics, University of Pennsylvania, 1989.

[4] Y. Xu, R.P. Paul, and X. Yun, "Hybrid position force control in presence of passive compliance" *Fifth International Symposium of Robotics Research,* Edited by Miura and Arimoto, MIT Press, 1989

[5] R.P. Paul, "Problems and research issues associated with the hybrid control of force and displacement", *Proceedings of the IEEE International Conference on Robotics and Automation,* P.1966-1971, 1987.

[6] R.K. Roberts, R.P. Paul and B.M. Hillberry, "The effect of wrist force sensor stiffness on the control of robot manipulators", *Proceedings of the IEEE International Conference on Robotics and Automation,* P.269-274, 1985.

[7] R.P. Paul, *"Robot Manipulators: Mathematics, Programming and Control",* Cambridge, MIT press, 1981.

[8] R. Paul and B.E. Shimano, "Compliance and control", *The 1976 Joint Automatic Controls Conference,* Albuquerque, NM, P.694-699, 1976.

[9] R. Paul, *Trajectory Calculation and Servoing of a Computer Controlled Arm,* PhD thesis, Computer Science Department, Stanford University, August 1972.

[10] D.E. Whitney and J.M. Rourke, "Mechanical behavior and design equations for elastomer shear pad remote center compliance", *ASME Journal of Dynamic System, Measurement, and Control,* Vol. 108, P.223-232, 1986.

[11] D.E. Whitney, "Historical perspective and state of the art in robot force control", *Proceedings of the IEEE International Conference on Robotics and Automation,* P.262-268, 1985.

[12] D.E. Whitney, "Force feedback control of manipulator fine motions", *ASME Journal of Dynamic Systems, Measurement and Control,* P.91-97, June 1977.

[13] M. Mason, "Compliance and force control for computer controlled manipulators", in *Robot Motion Planning and Control*, M. Brady et al, ed., The MIT Press, Cambridge, MA, 1982, P.373-404. ch.5.

[14] K. Salisbury, "Active stiffness control of a manipulator in cartesian coordinates", *Proc. 19th IEEE Conference on Decision and Control*, Albuquerque, NM, December 1980, P.87-97.

[15] J.K. Salisbury, *Kinematic and Force Analysis of Articulated Hands*, PhD thesis, Stanford University, July 1982.

[16] N. Hogan and S.L. Cotter, "Cartesian impedance control of a nonlinear manipulator", *Robotics Research and Advanced Applications*, ASME Winter Annual Meeting, Phoenix, AZ, November 1982, P.121-128.

[17] N. Hogan, "Impedance control of industrial robots", *Robotics and Computer Integrated Manufacturing*, Vol.1, No.1, 1984, P.97-113.

[18] H. Kazerooni, "On the stability of the robot compliant motion control (input output approach)", *Proceedings of the IEEE International Conference on Decision and Control*, 1987.

[19] S.D. Eppinger and W.P. Seering, "On dynamic models of the robot force control", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.29-34, 1986.

[20] S.D. Eppinger and W.P. Seering, "Understanding bandwidth limitations in robot force control" *Proceedings of the IEEE International Conference on Robotics and Automation*, P.904-909, 1987.

[21] M.H. Raibert and J.J. Craig, "Hybrid position/force control of manipulators", *ASME Journal of Dynamics System, Measurement, and Control*, Vol. 102, 1981, P.126-133.

[22] D.S. Seltzer, "Compliant robot wrist sensing for precision assembly", *Robotics: Theory and Application*, P.161-168, 1986.

[23] A.K. Bejczy, T.J. Tarn, X. Yun and S. Han, "Nonlinear feedback control of puma 560 robot arm by computer", *Proceedings of the IEEE 24th Conference on Decision and Control*, P.1680-1688, 1985.

[24] O. Khatib, "A unified approach for motion and force control of robot manipulators: the operational space formulation", *Proceedings of the IEEE Journal of Robotics and Automation*, P.43-53, 1987.

[25] O. Khatib and J. Burdick, "Manipulators motion and force control", *Proceedings of the IEEE International Conference on Robotics and Automation*, 1986.

[26] C. Reboulet and A. Robert, "Hybrid control of a manipulator with an active compliant wrist", *Proceedings of the Third International Symposium on Robotics Research*, 1985.

[27] J-P. Merlet, "C-surface applied to the design of an hybrid force-position robot controller", *Proceedings of the IEEE International Conference on Robotics and Automation*, 1987.

[28] C.H. An and J.M. Hollerbach, "Dynamic stability issues in the force control of manipulator", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.890-896, 1987.

[29] V. Hayward, *RCCL User's Guide*, edited for CVaRL by John Lloyd, 1984.

[30] H. Hanafusa, K. Kobayashi and N. Terasaki, "Fine control of the object with articulated multi-finger robot hands, *1983 International Conference on Advanced Robotics*, Tokyo, Japan, P.245-252, September 1983.

```
/*
 *      File:   moveg.c
 *      Remarks: Edge tracking operation with the compliant wrist
 *               system (ref: edge.c)
 */

#include <stdio.h>
#include "rw.h"

double  gain_ft,
        gain_fr,
        gain_pt,
        gain_pr,
        fuzz_j,
        fuzz_t,
        fuzz_r,
        filt_f_pole,
        filt_p_pole,
        vs,
        vt,
        va,
        cfa,
        cfs,
        tfa,
        tfs;
int     rw_comp = 0;
int     verbose;
extern FILE     *recfp;

main(ac, av)
int     ac;
char    **av;
{
        double  atof();

        gain_pr = 0.6;
        gain_pt = 0.9;
        gain_fr = 0.005;
        gain_ft = 0.05;
        fuzz_j = .01;
        fuzz_r = 0.4;
        fuzz_t = 0.4;
        filt_f_pole = 0.3;
        filt_p_pole = 0.98;
        vs = 0.2;
        vt = 0.2;
        va = 0.6;
        cfs = 1.2;
        cfa = 1.0;
        tfa = 0.45;
        tfs = 0.65;

        while (--ac > 0  && **++av == '-') {
                register char   *p = *av;

                while (*++p != '\0')
                        switch (*p) {
                        case 'v':
                                verbose++; break;
                        case 'S':
                                vs = atof(&p[1]); break;
                        case 'T':
                                vt = atof(&p[1]); break;
                        case 'A':
                                va = atof(&p[1]); break;
                        case 's':
                                cfs = atof(&p[1]); break;
                        case 'a':
                                cfa = atof(&p[1]); break;
                        case 'm':
                                filt_f_pole = atof(&p[1]); break;
                        case 'n':
                                filt_p_pole = atof(&p[1]); break;
                        case 'R':
                        if ((recfp = fopen(&p[1], "w")) == NULL) {
                        fprintf(stderr,
                                "cant open file for write\n"
                                );
                                exit(3);
                                }
                                goto nextarg;
                        }
nextarg:        ;
        }

        printf("gain in posn control for posn is %f\n", gain_pt);
        printf("gain in posn control for rotn is %f\n", gain_pr);
        printf("gain in force control for posn is %f\n", gain_ft);
        printf("gain in force control for rotn is %f\n", gain_fr);
        printf("filt pole in posn control is %f\n", filt_p_pole);
        printf("filt pole in force control is %f\n", filt_f_pole);
        printf("fuzz level for translation is %f\n", fuzz_t);
        printf("fuzz level for rotation is %f\n", fuzz_r);
        printf("searching velocity is %f\n", vs);
        printf("tracking velocity %f\n", vt);
        printf("approaching velocity %f\n", va);
        printf("contact force in searching dir %f\n", cfs);
        printf("contact force in approaching dir %f\n", cfa);

        rw_comp = 0;

        /*
         * initialize the control loop.
         */
        rw_init();

        /*
         * start the complience. watch out.
         */
        rw_comp = 1;

        for(;;)
          rw();

}
```

```
/*
 *      File:   inser.c
 *      Remarks: Insertion or pulling operation with the compliant wrist
 *               system (ref: movei.c)
 */

/*
 * Please check the file hybri.c. Since there are detail descriptions
 * in hybri.c, those parameters appeared there are not defined here again
 * for simplicity.
 */

#include        <stdio.h>
#include        <rccl/rccl.h>
#include        <rccl/rci.h>
#include        <rccl/kine.h>
#include        "rw.h"
#include        "recordd.h"

#define N       6

#define dist_of_flange 55.88   /* 2.2 in = 55.88 mm.  This is the distance
                                  from the center of the last three joints to
                                  the outer surface of the flange */
#define mount_dist  35.0

/*
 * the first order digital filter
 */
#define FILTER(y, u, pole)      (y=(1.0-pole)*(y*pole/(1.0-pole)+u))

#define NCOPY(a,b)      for(i=0;i<N;i++)a[i]=b[i]

double  car_diffs[6];

TRSF    mem_tw,
        *tw = &mem_tw;
extern double   gain_ft,
                gain_fr,
                gain_pt,
                gain_pr,
                fuzz_j,
                fuzz_t,
                fuzz_r,
                filt_f_pole,
                filt_p_pole,
                va,             /* approaching velocity */
                tf,             /* turning force to siwtch from the full
                                 * position control to hybrid control
                                 */
                pull;           /* the switch to signify inserting or pulling
                                 * out. 0 signifies inserting, and 1 signifies
                                 * pulling out
                                 */
#define FUZZ            fuzz_j
#define FUZZ_CARPOS     fuzz_t
#define FUZZ_CAREUL     fuzz_r

DIFF    d;
DIFF    d_f;
DIFF    d_p, dvel;

double  jang_o[RW_MAX_JOINT];
double  rw_theta_cal[RW_MAX_JOINT],
        rw_theta_bar[RW_MAX_JOINT] = {
                        RW_THETA_BAR_0,
                        RW_THETA_BAR_1,
                        RW_THETA_BAR_2,
                        RW_THETA_BAR_3,
                        RW_THETA_BAR_4,
                        RW_THETA_BAR_5
        };
int     rclconst();
struct _record  rec;
int             sync,           /* user/interrupt coordination flag */
                time;           /* time maintained by interrupt function */
FILE            *recfp;         /* record file fp */

int     verbose;
double  d_angles[N];
double  r_angles[N];
SNCS    sncs;

/*
 * Here is where we initialize everything needed to make the robot do
 * what we want it to.  We establish the position equation that the
 * main program will continually move to.
 */
rw_init()
{
        int     i;
        void    dummy(), drive();


        setbuf(stdout, NULL);
        rw_cal();

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                jang_o[i] = 0.0;
        }

        tw = newtrans("tw",rw);
        rw_car_update_tw();
        printrn(tw,stdout);
        /*
         * start the real-time process, and request arm power
         */
        RCIopen();
        RCIcontrol(dummy, drive);
        chg.power_on.com = YES;
        if ((how.state & CALIB_OK) == 0) {
                fprintf(stderr, "arm not calibrated\n");
                exit(3);
        }

        /*
         * control the inserting velocity or pulling out velocity,
         */
        if (pull == 0.0) {
        dvel.t.z = va;
        }
        if (pull == 1.0) {
        dvel.t.z = -0.2*va;
        }
}

/*
```

```
 * Real-time drive function
 *
 *      read joint angles
 *      compute Jacobian at this point
 *      transform cartesian diff to joint space
 *      compute new joint angles
 *      output setpoint
 */
double  car_diffs[N];
double  r_angles[N];
double  d_angles[N];
void
drive()
{
        double  del_force[N], del_angle_vel[N];
        double  del_posn[N];
        short   encs[N];
        JNTS    q_f, qvel;
        JNTS    q_p;
        int     i;
        static int      initd = 0;
        static double   del_f_smth[N];
        static double   del_p_smth[N];

        if (initd == 0) {
                enctoang(r_angles, how.pos);    /* get actual joint angles */
                for (i=0; i<N; i++) {
                        d_angles[i] = r_angles[i];
                }
                initd++;
        }
        update_sincos(&sncs, r_angles);         /* compute sin/cos */
        update_jacobian_terms(&sncs);           /* compute jacob terms */

        /*
         * When force is higher than the specified turning force,
         * it is considered that the shaft is on the workpiece,
         * and the controller is switched to hybrid mode
         */
        if ( car_diffs[2] > tf && car_diffs[2] < -tf ) {
            dvel.t.z = 0.0;
        }

        jacobI(&q_f, &d_f, &sncs, 0.0);         /* transform diff in cartesian
                                                   space to joint space */
        jacobI(&q_p, &d_p, &sncs, 0.0);         /* transform diff in cartesian
                                                   space to joint space */
        jnts_to_angle(del_force, &q_f);         /* delta joint to angles */
        jnts_to_angle(del_posn, &q_p);          /* delta joint to angles */

        jacobI(&qvel, &dvel, &sncs, 0.0);       /* transform diff in cartesian
                                                   space to joint space */
        jnts_to_angle(del_angle_vel, &qvel);    /* delta joint to angles */

        for (i=0; i<N; i++) {
                FILTER(del_f_smth[i], del_force[i], filt_f_pole);
                FILTER(del_p_smth[i], del_posn[i], filt_p_pole);
        }

        for (i=0; i<N; i++)
                d_angles[i] += del_f_smth[i] + del_angle_vel[i];
        for (i=0; i<N; i++)
                r_angles[i] = d_angles[i] + del_p_smth[i];
        NCOPY(rec.r_angles, r_angles);
        NCOPY(rec.car_diffs, car_diffs);
        rec.time = time;

        angtoenc(encs, r_angles);
        sync++;                 /* tell user process we have data */
        time++;

        for (i=0; i<N; i++) {
                chg.motion[i].com = POS;
                chg.motion[i].value = encs[i];
        }
}


void
dummy() {}

/*
 * rw_cal reads the current pot settings to get the current joint
 * angles.  These are then subtracted from the "correct" angles to
 * get the correction angles.
 */

rw_cal()
{
        int     i,
                j;

        /*
         * Initialize the axv11 board.
         */

        if (ax_init() < 0)
        {
                fprintf(stderr,"Cannot initialize the axv11 board\n");
                exit(1);
        }

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                rw_theta_cal[i] = rw_ptor(i);
        }

}

/*
 * This is the routine that has to figure out how to change the postion
 * equation established in rw_init() so the robot is driven to where we
 * want it to go.
 */
rw()
{
        int     i;

        JNTS curr_jnts;
        JNTS diff_jnts;
        if (rw_comp)
        {
                bcopy(j6, &curr_jnts, sizeof(JNTS));
                rw_car_update_tw();
                if (verbose>1)
                        printrn(tw,stdout);
                noatorpy(&car_diffs[5],&car_diffs[4],&car_diffs[3],tw);
                car_diffs[0] = tw->p.x;
```

```
        car_diffs[1] = tw->p.y;
        car_diffs[2] = tw->p.z - 62.0;

 if( car_diffs[0] <= FUZZ_CARPOS && car_diffs[0] >= -FUZZ_CARPOS )
        car_diffs[0] = 0.0;
 if( car_diffs[1] <= FUZZ_CARPOS && car_diffs[1] >= -FUZZ_CARPOS )
        car_diffs[1] = 0.0;
 if( car_diffs[2] <= FUZZ_CARPOS && car_diffs[2] >= -FUZZ_CARPOS )
        car_diffs[2] = 0.0;
 if( car_diffs[3] <= FUZZ_CAREUL && car_diffs[3] >= -FUZZ_CAREUL )
        car_diffs[3] = 0.0;
 if( car_diffs[4] <= FUZZ_CAREUL && car_diffs[4] >= -FUZZ_CAREUL )
        car_diffs[4] = 0.0;
 if( car_diffs[5] <= FUZZ_CAREUL && car_diffs[5] >= -FUZZ_CAREUL )
        car_diffs[5] = 0.0;

if (verbose) {
        for (i=0; i<N; i++)
                printf("%10.2f ", car_diffs[i]);
        putchar('\n');
}

/*
 * The contact between the shaft and hole does not occur.
 *
if (car_diffs[2] < tf && car_diffs[2] > -tf ) {
        d.t.x = car_diffs[0] * gain_ft;
        d.t.y = car_diffs[1] * gain_ft;
        d.t.z = car_diffs[2] * gain_ft * (1.0 - pull) + car_diffs[2]
                * gain_pt * pull;
        d.r.x = dtor(car_diffs[3]) * gain_fr;
        d.r.y = dtor(car_diffs[4]) * gain_fr;
        d.r.z = dtor(car_diffs[5]) * gain_fr * 10.0;
      }

/*
 * The contact between the shaft and hole occurs, or the jamming is
 * going to happen. Increasing the correction in the lateral direction
 * is always desirable
 */
if ( car_diffs[2] > 0.5 && car_diffs[2] < -0.5 ) {
        d.t.x = (car_diffs[0] + 0.005) * gain_ft;
        d.t.y = (car_diffs[1] + 0.005) * gain_ft;
        if ( car_diffs[0] < 0.0 )
               d.t.x = (car_diffs[0] - 0.005) * gain_ft;
        if ( car_diffs[1] < 0.0 )
               d.t.y = (car_diffs[1] - 0.005) * gain_ft;
        d.t.z =  car_diffs[2] * gain_ft;
        d.r.x =  dtor(car_diffs[3]) * gain_fr * 0.05;
        d.r.y =  dtor(car_diffs[4]) * gain_fr * 0.05;
        d.r.z =  dtor(car_diffs[5]) * gain_fr;
}

/*
 * When the force between the contact surface is higher than
 * this level, it is considered that jamming occurs, thus
 * we increase large pushing force in the z direction, make
 * a large rotation around the z axis, so as to avoid jamming
 */
if ( car_diffs[0] > 0.8 && car_diffs[0] < -0.8 &&
     car_diffs[1] > 0.8 && car_diffs[1] < -0.8 ) {
d.t.x = car_diffs[0] * gain_ft;
d.t.y = car_diffs[1] * gain_ft;
d.t.z = (car_diffs[2] + 0.005) * gain_ft * (1.0 - pull) +
                car_diffs[2] * gain_pt * pull;
        d.r.x = dtor(car_diffs[3]) * gain_fr;
        d.r.y = dtor(car_diffs[4]) * gain_fr;
        d.r.z = dtor(car_diffs[5]) * gain_fr * 3.0;
        }

        /*
         * position control in z direction for pulling out case, but force
         * control in z direction for inserting case
         */
        d_f.t.x = d.t.x;
        d_f.t.y = d.t.y;
        d_f.t.z = d.t.z * (1.0 - pull);
        d_f.r.x = d.r.x;
        d_f.r.y = d.r.y;
        d_f.r.z = d.r.z;

        d_p.t.x = 0.0;
        d_p.t.y = 0.0;
        d_p.t.z = -d.t.z * pull;
        d_p.r.x = 0.0;
        d_p.r.y = dtor(30.0) * gain_pr * (1.0 - pull);/*30 degree is offset
                                                        in the ready position*/
        d_p.r.z = 0.0;

        if (recfp && sync) {
                fwrite(&rec, sizeof(rec), 1, recfp);
                sync = 0;
        }
        }
}

double
rw_jang(i)
int     i;
{
        double  jang,
                jang_diff,
                newdiff;

        int     s;

#ifdef notdef
        newdiff = rw_raw_diff(i);
        rw_old_diff[i] = newdiff;

#endif
        jang =  rw_raw_diff(i);

        jang_diff = (jang - jang_o[i]);
        if (jang_diff <= FUZZ && jang_diff >= -FUZZ)
        {
                        jang = jang_o[i];
        }
        jang_o[i] = jang;

        jang += rw_theta_bar[i];
        return(jang);
}
rw_car_update_tw()
{
        double  c1,
                c2,
                c3,
```

```
        c4,
        c5,
        c6,
        s1,
        s2,
        s3,
        s4,
        s5,
        s6,
        c23,
        s23,
        x1,
        x2,
        x3,
        x4,
        x5,
        x6,
        x7,
        x8,
        x9,
        x10;

c1 = cos(rw_jang(0));
s1 = sin(rw_jang(0));

c2 = cos(rw_jang(1));
s2 = sin(rw_jang(1));

c3 = cos(rw_jang(2));
s3 = sin(rw_jang(2));

c4 = cos(rw_jang(3));
s4 = sin(rw_jang(3));

c5 = cos(rw_jang(4));
s5 = sin(rw_jang(4));

c6 = cos(rw_jang(5));
s6 = sin(rw_jang(5));

c23 = cos(rw_jang(1)+rw_jang(2));
s23 = sin(rw_jang(1)+rw_jang(2));

x1 = -c4*s5;
x2 = c4*c5*c6 - s4*s6;
x3 = c4*c5*s6 + s4*c6;
x4 = -s4*s5;
x5 = s4*c5*c6 + c4*s6;
x6 = s4*c5*s6 - c4*c6;
x7 = L8*x3 + L7*x1 - L5*s4 + L5;
x8 = L8*s5*s6 + L7*c5 - L6;
x9 = -L8*x6 - L7*x4 - L5*c4 - L4 + L2;
x10 = L3*c2 + c23*x7 + s23*x8;

tw->n.x = c1*(c23*x1 + s23*c5) - s1*x4;
tw->o.x = c1*(c23*x2 + s23*s5*c6) - s1*x5;
tw->a.x = c1*(c23*x3 + s23*s5*s6) - s1*x6;
tw->p.x = c1*(L3*c2 + x7*c23 + s23*x8) + s1*x9 - L9;

tw->n.y = s1*(c23*x1 + s23*c5) + c1*x4;
tw->o.y = s1*(c23*x2 + s23*s5*c6) + c1*x5;
tw->a.y = s1*(c23*x3 + s23*s5*s6) + c1*x6;
tw->p.y = s1*(L3*c2 + c23*x7 + s23*x8) - c1*x9 + L3;
        tw->n.z = s23*x1 - c23*c5;
        tw->o.z = s23*x2 - c23*s5*c6;
        tw->a.z = s23*x3 - c23*s5*s6;
        tw->p.z = s23*x7 - c23*x8 + L3*s2 + L1;
}

rw_close()
{
        RCIrelease(1);
        RCIclose (1);
}
rccl_close()
{
}
/*
 * called on ^C to close the record file if it was open
 */
void
quit()
{
        if (recfp)
                fclose(recfp);
}
```

```
/*
 *      File:   movei.c
 *      Remarks: Insertion or pulling operation with the compliant wrist
 *               system (ref: inser.c)
 */

#include <stdio.h>
#include "rw.h"

double  gain_ft,
        gain_fr,
        gain_pt,
        gain_pr,
        fuzz_j,
        fuzz_t,
        fuzz_r,
        filt_f_pole,
        filt_p_pole,
        va,
        tf,
        pull;
int     rw_comp = 0;
int     verbose;
extern FILE     *recfp;

main(ac, av)
int     ac;
char    **av;
{
        double  atof();

        gain_pr = 0.2;
        gain_pt = 0.5;
        gain_fr = 0.001;
        gain_ft = 0.004;
        fuzz_j = .01;
        fuzz_r = 0.2;
        fuzz_t = 0.2;
        filt_f_pole = 0.5;
        filt_p_pole = 0.97;
        va = 0.1;
        tf = 0.4;
        pull = 0.0;

        while (--ac > 0  && **++av == '-') {
                register char   *p = *av;

                while (*++p != '\0')
                        switch (*p) {
                        case 'v':
                                verbose++; break;
                        case 'A':
                                va = atof(&p[1]); break;
                        case 'P':
                                pull = atof(&p[1]); break;
                        case 'T':
                                gain_ft = atof(&p[1]); break;
                        case 'O':
                                gain_fr = atof(&p[1]); break;
                        case 't':
                                gain_pt = atof(&p[1]); break;
                        case 'o':
                                gain_pr = atof(&p[1]); break;
                        case 'm':
                                filt_f_pole = atof(&p[1]); break;
                        case 'n':
                                filt_p_pole = atof(&p[1]); break;
                        case 'R':
                        if ((recfp = fopen(&p[1], "w")) == NULL) {
                        fprintf(stderr,
                                "cant open file for write\n"
                                );
                                exit(3);
                                }
                                goto nextarg;
                        }
nextarg:        ;
        }

        printf("gain in posn control for posn is %f\n", gain_pt);
        printf("gain in posn control for rotn is %f\n", gain_pr);
        printf("gain in force control for posn is %f\n", gain_ft);
        printf("gain in force control for rotn is %f\n", gain_fr);
        printf("filt pole in posn control is %f\n", filt_p_pole);
        printf("filt pole in force control is %f\n", filt_f_pole);
        printf("approaching velocity is %f\n", va);
        printf("inserting or pulling out %f\n", pull);

        rw_comp = 0;

        /*
         * initialize the control loop.
         */
        rw_init();

        /*
         * start the complience. watch out.
         */
        rw_comp = 1;

        for(;;)
          rw();

}
```

```
/*
 *      File:   fuzz.c
 *      Remarks: Insertion or pulling operation using the fuzzy control
 *               decision rule with the compliant wrist system (ref: movef.c)
 */

/*
 * Please check the file hybri.c. Since there are detail descriptions
 * in hybri.c, those parameters appeared there are not defined here again
 * for simplicity.
 */

#include        <stdio.h>
#include        <rccl/rccl.h>
#include        <rccl/rci.h>
#include        <rccl/kine.h>
#include        "rw.h"
#include        "recordd.h"

#define N       6

#define dist_of_flange 55.88   /* 2.2 in = 55.88 mm.  This is the distance
                                  from the center of the last three joints to
                                  the outer surface of the flange */
#define mount_dist  35.0

/*
 * the first order digital filter
 */
#define FILTER(y, u, pole)      (y=(1.0-pole)*(y*pole/(1.0-pole)+u))

#define NCOPY(a,b)      for(i=0;i<N;i++)a[i]=b[i]

double  car_diffs[6];
double  PPgain, PRgain;
double  DPgain, DRgain;
double  IPgain, IRgain;

TRSF    mem_tw,
        T6,                     /* current Puma 560 kinematic transform T6 */
        Pd,                     /* desired Puma 560 kinematic transform T6 */
        *tw = &mem_tw;
extern double   gain_ft,
                gain_fr,
                gain_pt,
                gain_pr,
                fuzz_j,
                fuzz_t,
                fuzz_r,
                filt_f_pole,
                filt_p_pole,
                va,
                pull;
#define FUZZ            fuzz_j
#define FUZZ_CARPOS     fuzz_t
#define FUZZ_CAREUL     fuzz_r

DIFF    d;
DIFF    d_f;
DIFF    d_p, dvel;

double  jang_o[RW_MAX_JOINT];
double  rw_theta_cal[RW_MAX_JOINT],
        rw_theta_bar[RW_MAX_JOINT] = {
                        RW_THETA_BAR_0,
                        RW_THETA_BAR_1,
                        RW_THETA_BAR_2,
                        RW_THETA_BAR_3,
                        RW_THETA_BAR_4,
                        RW_THETA_BAR_5
        };
int     rclconst();
struct _record  rec;
int             sync,           /* user/interrupt coordination flag */
                time;           /* time maintained by interrupt function */
FILE            *recfp;         /* record file fp */

int     verbose;
double  d_angles[N];
double  r_angles[N];
SNCS    sncs;

/*
 * Here is where we initialize everything needed to make the robot do
 * what we want it to.  We establish the position equation that the
 * main program will continually move to.
 */
rw_init()
{
        int     i;
        void    dummy(), drive();


        setbuf(stdout, NULL);
        rw_cal();

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                jang_o[i] = 0.0;
        }

        tw = newtrans("tw",rw);
        rw_car_update_tw();
        printrn(tw,stdout);
        /*
         * start the real-time process, and request arm power
         */
        RCIopen();
        RCIcontrol(dummy, drive);
        chg.power_on.com = YES;
        if ((how.state & CALIB_OK) == 0) {
                fprintf(stderr, "arm not calibrated\n");
                exit(3);
        }
}

/*
 *      Real-time drive function:
 *      read joint angles
 *      compute Jacobian at this point
 *      transform cartesian diff to joint space
 *      compute new joint angles
 *      output setpoint
 */
double  car_diffs[N];
double  r_angles[N];
double  d_angles[N];
void
```

```
drive()
{
        double  del_force[N], del_angle_vel[N];
        double  del_posn[N];
        short   encs[N];
        TRSF    T6err;          /* transform representing trajectory generation*/
        JNTS    q_f, qvel;
        JNTS    q_p;
        static  JNTS    q_j6;
        int     i;
        static int      initd = 0;
        static double   del_f_smth[N];
        static double   del_p_smth[N];
        static DIFF     T6err_integ;     /* the error transform sum value for
                                          * the intergal term */
        static DIFF     T6err_deri;      /* the error transform update value
                                          * for the derivative term */
        float           yawerr, pitcherr, rollerr; /* orentational error
                                                    * for trajectory generation
                                                    */
        if (initd == 0) {
                enctoang(r_angles, how.pos);    /* get actual joint angles */
                for (i=0; i<N; i++) {
                        d_angles[i] = r_angles[i];
                }
                q_j6.conf = "blah";
                initd++;
        }

        /*
         * copy the joint angles into a JNTS
         */
        {
                real    *f = &(q_j6.th1), *g = &(jmin_c.th1);

                for (i=0; i<6; i++)
                        *f++ = r_angles[i] - *g++;
        }

        /*
         * do the forward kinematics to find current cartesian position,
         * also updates all sin/cos values in sncs.
         */
        jns_to_tr(&T6, &q_j6, &sncs);

        /*
         * trajectory generation
         *
         *      e =  T6^-1 * Pd
         *      xdot = P*e + D*de/dt + I*$ e dt
         *
         * where P, D and I are empirically determined control gains
         */
        invert(&T6err,&T6);         /* T6err = T6^-1 */
        trmultinp(&T6err,&Pd);      /* T6err = T6err * Pd */
        noatorpy(&rollerr, &pitcherr, &yawerr, &T6err);
                        /* get roll, pitch, yaw from transform T6err */

        /*
         * PID controller for trajectory generation. The purpose here is
         * to find the location of the hole. If it is unnecessary to
         * locate the hole, "dvel" can be specified in the way in inser.c
         */
        dvel.t.x = T6err.p.x * PPgain + (T6err.p.x - T6err_deri.t.x) * DPgain
                + T6err_integ.t.x * IPgain;
        dvel.t.y = T6err.p.y * PPgain + (T6err.p.y - T6err_deri.t.y) * DPgain
                + T6err_integ.t.y * IPgain;
        dvel.t.z = T6err.p.z * 0.6 * PPgain + (T6err.p.z - T6err_deri.t.z)
                * DPgain + T6err_integ.t.z * IPgain;

        dvel.r.x = yawerr * PRgain + (yawerr - T6err_deri.r.x) * DRgain +
                T6err_integ.r.x * IRgain;
        dvel.r.y = pitcherr * PRgain + (pitcherr - T6err_deri.r.y) * DRgain +
                T6err_integ.r.y * IRgain;
        dvel.r.z = rollerr * PRgain + (rollerr - T6err_deri.r.z) * DRgain +
                T6err_integ.r.z * IRgain;

        /*
         * update the privious error for derivative
         */
        T6err_deri.t.x = T6err.p.x;
        T6err_deri.t.y = T6err.p.y;
        T6err_deri.t.z = T6err.p.z;
        T6err_deri.r.x = yawerr;
        T6err_deri.r.y = pitcherr;
        T6err_deri.r.z = rollerr;

        /*
         * update the error integral
         */
        T6err_integ.t.x += T6err.p.x;
        T6err_integ.t.y += T6err.p.y;
        T6err_integ.t.z += T6err.p.z;
        T6err_integ.r.x += yawerr;
        T6err_integ.r.y += pitcherr;
        T6err_integ.r.z += rollerr;

        /*
         * make it go fast in inserting, while go slowly in pulling out
         * where the velocity should not be related to the initial position
         */
        dvel.t.z = ( dvel.t.z + va ) * (1.0 - pull) - 0.5 * va * pull;

        /*
         * If the sensed force is higher than the turning force, it goes
         * slowly, as well as indepedently of the hole location specification
         * in case collision occurs
         */
        if ( car_diffs[2] > 0.4 ) {     /* 0.4 is the turning force, also can be
                                           specified interactively */
            dvel.t.z = 0.2 * va * (1.0 - 2.0 * pull);
        }

        update_jacobian_terms(&sncs);           /* compute jacob terms */
        jacobI(&q_f, &d_f, &sncs, 0.0);         /* transform diff in cartesian
                                                   space to joint space */
        jacobI(&q_p, &d_p, &sncs, 0.0);         /* transform diff in cartesian
                                                   space to joint space */
        jnts_to_angle(del_force, &q_f);         /* delta joint to angles */
        jnts_to_angle(del_posn, &q_p);          /* delta joint to angles */

        jacobI(&qvel, &dvel, &sncs, 0.0);       /* transform diff in cartesian
                                                   space to joint space */
        jnts_to_angle(del_angle_vel, &qvel);    /* delta joint to angles */

        for (i=0; i<N; i++) {
                FILTER(del_f_smth[i], del_force[i], filt_f_pole);
                FILTER(del_p_smth[i], del_posn[i], filt_p_pole);
```

```
        }

        for (i=0; i<N; i++)
                d_angles[i] += del_f_smth[i] + del_angle_vel[i];
        for (i=0; i<N; i++)
                r_angles[i] = d_angles[i] + del_p_smth[i];
        NCOPY(rec.r_angles, r_angles);
        NCOPY(rec.car_diffs, car_diffs);
        rec.time = time;

        angtoenc(encs, r_angles);
        sync++;                 /* tell user process we have data */
        time++;

        for (i=0; i<N; i++) {
                chg.motion[i].com = POS;
                chg.motion[i].value = encs[i];
        }
}


void
dummy() {}

/*
 * rw_cal reads the current pot settings to get the current joint
 * angles.  These are then subtracted from the "correct" angles to
 * get the correction angles.
 */

rw_cal()
{
        int     i,
                j;

        /*
         * Initialize the axv11 board.
         */

        if (ax_init() < 0)
        {
                fprintf(stderr,"Cannot initialize the axv11 board\n");
                exit(1);
        }

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                rw_theta_cal[i] = rw_ptor(i);
        }

}

/*
 * This is the routine that has to figure out how to change the postion
 * equation established in rw_init() so the robot is driven to where we
 * want it to go.
 */
rw()
{
        int     i;

        JNTS curr_jnts;
        JNTS diff_jnts;
        if (rw_comp)
{
        bcopy(j6, &curr_jnts, sizeof(JNTS));
        rw_car_update_tw();
        if (verbose>1)
                printrn(tw,stdout);
        noatorpy(&car_diffs[5],&car_diffs[4],&car_diffs[3],tw);
        car_diffs[0] = tw->p.x;
        car_diffs[1] = tw->p.y;
        car_diffs[2] = tw->p.z - 62.0;

        if( car_diffs[0] <= FUZZ_CARPOS &&
            car_diffs[0] >= -FUZZ_CARPOS )
                car_diffs[0] = 0.0;
        if( car_diffs[1] <= FUZZ_CARPOS &&
            car_diffs[1] >= -FUZZ_CARPOS )
                car_diffs[1] = 0.0;
        if( car_diffs[2] <= FUZZ_CARPOS &&
            car_diffs[2] >= -FUZZ_CARPOS )
                car_diffs[2] = 0.0;
        if( car_diffs[3] <= FUZZ_CAREUL &&
            car_diffs[3] >= -FUZZ_CAREUL )
                car_diffs[3] = 0.0;
        if( car_diffs[4] <= FUZZ_CAREUL &&
            car_diffs[4] >= -FUZZ_CAREUL )
                car_diffs[4] = 0.0;
        if( car_diffs[5] <= FUZZ_CAREUL &&
            car_diffs[5] >= -FUZZ_CAREUL )
                car_diffs[5] = 0.0;
        if (verbose) {
                for (i=0; i<N; i++)
                        printf("%10.2f ", car_diffs[i]);
                putchar('\n');
                printy(&T6, stdout);
        }

/*
 * fuzzy controller. The fuzzy levels are empirically determined.
 */
d.t.x = car_diffs[0] * gain_ft;
d.t.y = car_diffs[1] * gain_ft;
d.t.z = car_diffs[2] * gain_pt;
d.r.x = dtor(car_diffs[3]) * gain_fr;
d.r.y = dtor(car_diffs[4]) * gain_fr;
d.r.z = dtor(car_diffs[5]) * gain_fr * 40.0;

if ( car_diffs[2] > 0.6 && car_diffs[2] < -0.5 ) {
d.t.x = d.t.x * 2.0 * (1.0 + 2.0 * pull);
d.t.y = d.t.y * 2.0 * (1.0 + 2.0 * pull);
d.r.x = d.r.x * 2.0 * (1.0 + 2.0 * pull);
d.r.y = d.r.y * 2.0 * (1.0 + 2.0 * pull);
d.r.z = d.r.z * 5.0 * (1.0 + 2.0 * pull);
        if ( car_diffs[2] > 0.8 && car_diffs[2] < -0.7 ) {
        d.t.x = d.t.x * 2.0;
        d.t.y = d.t.y * 2.0;
        d.r.x = d.r.x * 2.0;
        d.r.y = d.r.y * 2.0;
        d.r.z = d.r.z * 5.0;
        }
}

/*
 * regular hybrid controller
 */
d_f.t.x = d.t.x;
```

```
        d_f.t.y = d.t.y;
        d_f.t.z = 0.0;
        d_f.r.x = d.r.x;
        d_f.r.y = d.r.y;
        d_f.r.z = d.r.z;

        d_p.t.x = 0.0;
        d_p.t.y = 0.0;
        d_p.t.z = -d.t.z;
        d_p.r.x = 0.0;
        d_p.r.y = dtor(30.0) * gain_pr * (1.0 - pull); /* 30 degree is for
                                                        * offset in ready
                                                        * position */
        d_p.r.z = 0.0;

        if (recfp && sync) {
                fwrite(&rec, sizeof(rec), 1, recfp);
                sync = 0;
        }
        }
}

double
rw_jang(i)
int     i;
{
        double  jang,
                jang_diff,
                newdiff;

        int     s;

#ifdef notdef
        newdiff = rw_raw_diff(i);
        rw_old_diff[i] = newdiff;

#endif
        jang =  rw_raw_diff(i);

        jang_diff = (jang - jang_o[i]);
        if (jang_diff <= FUZZ && jang_diff >= -FUZZ)
        {
                        jang = jang_o[i];
        }
        jang_o[i] = jang;

        jang += rw_theta_bar[i];
        return(jang);
}
rw_car_update_tw()
{
        double  c1,
                c2,
                c3,
                c4,
                c5,
                c6,
                s1,
                s2,
                s3,
                s4,
                s5,
                s6,
                c23,
                s23,
                x1,
                x2,
                x3,
                x4,
                x5,
                x6,
                x7,
                x8,
                x9,
                x10;

        c1 = cos(rw_jang(0));
        s1 = sin(rw_jang(0));

        c2 = cos(rw_jang(1));
        s2 = sin(rw_jang(1));

        c3 = cos(rw_jang(2));
        s3 = sin(rw_jang(2));

        c4 = cos(rw_jang(3));
        s4 = sin(rw_jang(3));

        c5 = cos(rw_jang(4));
        s5 = sin(rw_jang(4));

        c6 = cos(rw_jang(5));
        s6 = sin(rw_jang(5));

        c23 = cos(rw_jang(1)+rw_jang(2));
        s23 = sin(rw_jang(1)+rw_jang(2));

        x1 = -c4*s5;
        x2 = c4*c5*c6 - s4*s6;
        x3 = c4*c5*s6 + s4*c6;
        x4 = -s4*s5;
        x5 = s4*c5*c6 + c4*s6;
        x6 = s4*c5*s6 - c4*c6;
        x7 = L8*x3 + L7*x1 - L5*s4 + L5;
        x8 = L8*s5*s6 + L7*c5 - L6;
        x9 = -L8*x6 - L7*x4 - L5*c4 - L4 + L2;
        x10 = L3*c2 + c23*x7 + s23*x8;

        tw->n.x = c1*(c23*x1 + s23*c5) - s1*x4;
        tw->o.x = c1*(c23*x2 + s23*s5*c6) - s1*x5;
        tw->a.x = c1*(c23*x3 + s23*s5*s6) - s1*x6;
        tw->p.x = c1*(L3*c2 + x7*c23 + s23*x8) + s1*x9 - L9;

        tw->n.y = s1*(c23*x1 + s23*c5) + c1*x4;
        tw->o.y = s1*(c23*x2 + s23*s5*c6) + c1*x5;
        tw->a.y = s1*(c23*x3 + s23*s5*s6) + c1*x6;
        tw->p.y = s1*(L3*c2 + c23*x7 + s23*x8) - c1*x9 + L3;

        tw->n.z = s23*x1 - c23*c5;
        tw->o.z = s23*x2 - c23*s5*c6;
        tw->a.z = s23*x3 - c23*s5*s6;
        tw->p.z = s23*x7 - c23*x8 + L3*s2 + L1;

}

rw_close()
{
        RCIrelease(1);
```

```
        RCIclose (1);
}

rccl_close()
{
}

/*
 * called on ^C to close the record file if it was open
 */
void
quit()
{
        if (recfp)
                fclose(recfp);
}
```

```
/*
 *      File:   movef.c
 *      Remarks: Insertion or pulling operation using the fuzzy control
 *               decision rule with the compliant wrist system (ref: fuzz.c)
 */

#include <stdio.h>
#include "rw.h"

double  gain_ft,
        gain_fr,
        gain_pt,
        gain_pr,
        fuzz_j,
        fuzz_t,
        fuzz_r,
        filt_f_pole,
        filt_p_pole,
        va,
        pull;
int     rw_comp = 0;
int     verbose;
extern FILE     *recfp;
extern TRSF     Pd;
extern double   PPgain, PRgain;
extern double   DPgain, DRgain;
extern double   IPgain, IRgain;

main(ac, av)
int     ac;
char    **av;
{
        double  atof();

        gain_pr = 0.2;
        gain_pt = 0.5;
        gain_fr = 0.001;
        gain_ft = 0.05;
        fuzz_j = .01;
        fuzz_r = 0.3;
        fuzz_t = 0.3;
        filt_f_pole = 0.4;
        filt_p_pole = 0.97;
        PPgain = 0.005;
        PRgain = 0.001;
        DPgain = 0.001;
        DRgain = 0.0005;
        IPgain = 0.0000001;
        IRgain = 0.00000005;
        va = 0.5;
        pull = 0.0;

        while (--ac > 0  && **++av == '-') {
                register char   *p = *av;

                while (*++p != '\0')
                        switch (*p) {
                        case 'v':
                                verbose++; break;
                        case 'A':
                                va = atof(&p[1]); break;
                        case 'P':
                                pull = atof(&p[1]); break;
                        case 'R':
                                if ((recfp = fopen(&p[1], "w")) == NULL) {
                                fprintf(stderr,
                                        "cant open file for write\n"
                                        );
                                        exit(3);
                                        }
                                        goto nextarg;
                                }
nextarg:        ;
        }
        printf("proporl gain of translation is %f\n", PPgain);
        printf("proporl gain of rotation is %f\n", PRgain);
        printf("integl gain of translation is %f\n", IPgain);
        printf("integl gain of rotation is %f\n", IRgain);
        printf("deri gain of translation is %f\n", DPgain);
        printf("deri gain of rotation is %f\n", DRgain);
        printf("approaching velocity is %f\n", va);
        printf("inserting or pulling out %f\n", pull);

        /*
         * specify the hole location
         */
        rpy(&Pd, -0.017, -0.2, -179.47);
        trsl(&Pd,-638.08, 132.033, -10.954);

        rw_comp = 0;

        /*
         * initialize the control loop.
         */
        rw_init();

        /*
         * start the complience. watch out.
         */
        rw_comp = 1;

        for(;;)
          rw();

}
```

```
/*
 *      File:   writ.c
 *      Remarks: Writing (or drawing) operation on an unmodeled
 *               surface with the compliant wrist system (ref: movew.c)
 */

/*
 * Please check the file hybri.c. Since there are detail descriptions
 * in hybri.c, those parameters appeared there are not defined here again
 * for simplicity.
 */

#include        <stdio.h>
#include        <rccl/rccl.h>
#include        <rccl/rci.h>
#include        <rccl/kine.h>
#include        "rw.h"
#include        "recordd.h"

#define N       6

#define dist_of_flange 55.88   /* 2.2 in = 55.88 mm.  This is the distance
                                  from the center of the last three joints to
                                  the outer surface of the flange */
#define mount_dist  35.0

/*
 * the first order digital filter
 */
#define FILTER(y, u, pole)      (y=(1.0-pole)*(y*pole/(1.0-pole)+u))

#define NCOPY(a,b)      for(i=0;i<N;i++)a[i]=b[i]

double  car_diffs[6];
double  PPgain, PRgain;
double  DPgain, DRgain;
double  IPgain, IRgain;

TRSF    mem_tw,
        T6,             /* current Puma 560 kinematic transform T6 */
        Pd,             /* desired Puma 560 kinematic transform T6 */
        *tw = &mem_tw;
TRSF    target[31];     /* array for the specified sequential configurations*/
int     ntarg,          /* the sequential number of the specified
                         * configurations */
        itarg;
extern double   gain_ft,
                gain_fr,
                gain_pt,
                gain_pr,
                deri_gain_ft,
                deri_gain_fr,
                deri_gain_pt,
                deri_gain_pr,
                fuzz_j,
                fuzz_t,
                fuzz_r,
                filt_f_pole,
                filt_p_pole,
                vlim_t,     /* velocity limit for translational motion */
                vlim_r,     /* velocity limit for translational motion */
                cf,         /* desired contact force */
                errtolerance;/* positioning error tolerance */
#define FUZZ            fuzz_j
#define FUZZ_CARPOS     fuzz_t
#define FUZZ_CAREUL     fuzz_r
#define VLIM_T          vlim_t
#define VLIM_R          vlim_r

DIFF    d;
DIFF    d_f;
DIFF    d_p, dvel;

double  jang_o[RW_MAX_JOINT];
double  rw_theta_cal[RW_MAX_JOINT],
        rw_theta_bar[RW_MAX_JOINT] = {
                        RW_THETA_BAR_0,
                        RW_THETA_BAR_1,
                        RW_THETA_BAR_2,
                        RW_THETA_BAR_3,
                        RW_THETA_BAR_4,
                        RW_THETA_BAR_5
        };
int     rclconst();
struct _record  rec;
int             sync,           /* user/interrupt coordination flag */
                time;           /* time maintained by interrupt function */
FILE            *recfp;         /* record file fp */

int     verbose;
double  d_angles[N];
double  r_angles[N];
SNCS    sncs;

/*
 * Here is where we initialize everything needed to make the robot do
 * what we want it to.  We establish the position equation that the
 * main program will continually move to.
 */
rw_init()
{
        int     i;
        void    dummy(), drive();


        setbuf(stdout, NULL);
        rw_cal();

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                jang_o[i] = 0.0;
        }

        tw = newtrans("tw",rw);
        rw_car_update_tw();
        printrn(tw,stdout);
        /*
         * start the real-time process, and request arm power
         */
        RCIopen();
        RCIcontrol(dummy, drive);
        chg.power_on.com = YES;
        if ((how.state & CALIB_OK) == 0) {
                fprintf(stderr, "arm not calibrated\n");
                exit(3);
        }
}
```

```
                fprintf(stderr,"Cannot initialize the axv11 board\n");
                exit(1);
        }

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                rw_theta_cal[i] = rw_ptor(i);
        }

}

/*
 * This is the routine that has to figure out how to change the postion
 * equation established in rw_init() so the robot is driven to where we
 * want it to go.
 */
rw()
{
        int     i;
        JNTS curr_jnts;
        JNTS diff_jnts;
        if (rw_comp)
        {
         bcopy(j6, &curr_jnts, sizeof(JNTS));
         rw_car_update_tw();
         if (verbose>1)
                printrn(tw,stdout);
         /*
          * extract the roll-pitch-yaw angles from transform tw
          */
         noatorpy(&car_diffs[5],&car_diffs[4],&car_diffs[3],tw);

         /*
          * computer the translational error ftom tw
          */
         car_diffs[0] = tw->p.x;
         car_diffs[1] = tw->p.y;
         car_diffs[2] = tw->p.z - 62.0;

         /*
          * fuzz out the device error caused by hystreresis of the device
          */
         if( car_diffs[0] <= FUZZ_CARPOS && car_diffs[0] >= -FUZZ_CARPOS )
                car_diffs[0] = 0.0;
         if( car_diffs[1] <= FUZZ_CARPOS && car_diffs[1] >= -FUZZ_CARPOS )
                car_diffs[1] = 0.0;
         if( car_diffs[2] <= FUZZ_CARPOS && car_diffs[2] >= -FUZZ_CARPOS )
                car_diffs[2] = 0.0;
         if( car_diffs[3] <= FUZZ_CAREUL && car_diffs[3] >= -FUZZ_CAREUL )
                car_diffs[3] = 0.0;
         if( car_diffs[4] <= FUZZ_CAREUL && car_diffs[4] >= -FUZZ_CAREUL )
                car_diffs[4] = 0.0;
         if( car_diffs[5] <= FUZZ_CAREUL && car_diffs[5] >= -FUZZ_CAREUL )
                car_diffs[5] = 0.0;

         /*
          * print out the caurrent cartesian error of the wrist
          */
         if (verbose) {
                for (i=0; i<N; i++)
                        printf("%10.2f ", car_diffs[i]);
                putchar('\n');
        }
          /*
           * compute the force control term, the specified force here is
           * zero. "s" represents the selection function corresponding to
           * the force control (s=0), or posn control (s=1).
           */

         d_f.t.x = car_diffs[0] * (1.0 - s_x_t) * gain_ft;
         d_f.t.y = car_diffs[1] * (1.0 - s_y_t) * gain_ft;
         d_f.t.z = car_diffs[2] * (1.0 - s_z_t) * gain_ft;
         d_f.r.x = dtor(car_diffs[3]) * (1.0 - s_x_r) * gain_fr;
         d_f.r.y = dtor(car_diffs[4]) * (1.0 - s_y_r) * gain_fr;
         d_f.r.z = dtor(car_diffs[5]) * (1.0 - s_z_r) * gain_fr;

          /*
           * compute the position control term. The negative sign is for
           * position error compensation so that robot goes the opposite
           * direction of the sensed displacement
           */
         d_p.t.x = -car_diffs[0] * s_x_t * gain_pt;
         d_p.t.y = -car_diffs[1] * s_y_t * gain_pt;
         d_p.t.z = -car_diffs[2] * s_z_t * gain_pt;
         d_p.r.x = -dtor(car_diffs[3]) * s_x_r * gain_pr;
         d_p.r.y = -dtor(car_diffs[4]) * s_y_r * gain_pr;
         d_p.r.z = -dtor(car_diffs[5]) * s_z_r * gain_pr;

          /*
           * record function
           */
         if (recfp && sync) {
                fwrite(&rec, sizeof(rec), 1, recfp);
                sync = 0;
                }
        }
}

/*
 * This is routine to update the joint angles of the compliant wrist.
 * tw() is computed based the angles updated here. Please also read rw.h
 */
double
rw_jang(i)
int     i;
{
        double  jang,
                jang_diff,
                newdiff;

        int     s;

#ifdef notdef
        newdiff = rw_raw_diff(i);
        rw_old_diff[i] = newdiff;

#endif
        /*
         * redefine jang as the current change between the current
         * displacement and the calibrated displacement where the wrist
         * is at stationary, rw_raw_diff(i) = rw_ptor(i) - rw_theta_cal(i)
         */
        jang =  rw_raw_diff(i);

        /*
         * record the difference between the current motion and previous
         * one
```

```
         */
        jang_diff = (jang - jang_o[i]);

        /*
         * if that doesn't make big difference, do do update the current
         * motion since that is considered as the hystreresis of the device
         */
        if (jang_diff <= FUZZ && jang_diff >= -FUZZ)
        {
                        jang = jang_o[i];
        }

        jang_o[i] = jang;               /* update jang_o */

        /* the current joint angles of the device is equal to the sum
         * of the delta angles and the calibrated angles
         */
        jang += rw_theta_bar[i];

        return(jang);
}

/*
 * This is function to compted the cartesian displacement from the joint
 * displacement measured by the compliant wrist device. The output is
 * the 4X4 transform matrix tw representing the transformation from the
 * center of the lower plate to the center of the upper plate of the wrist.
 */

rw_car_update_tw()
{
        double  c1,
                c2,
                c3,
                c4,
                c5,
                c6,
                s1,
                s2,
                s3,
                s4,
                s5,
                s6,
                c23,
                s23,
                x1,
                x2,
                x3,
                x4,
                x5,
                x6,
                x7,
                x8,
                x9,
                x10;

        c1 = cos(rw_jang(0));
        s1 = sin(rw_jang(0));

        c2 = cos(rw_jang(1));
        s2 = sin(rw_jang(1));

        c3 = cos(rw_jang(2));
        s3 = sin(rw_jang(2));

        c4 = cos(rw_jang(3));
        s4 = sin(rw_jang(3));

        c5 = cos(rw_jang(4));
        s5 = sin(rw_jang(4));

        c6 = cos(rw_jang(5));
        s6 = sin(rw_jang(5));

        c23 = cos(rw_jang(1)+rw_jang(2));
        s23 = sin(rw_jang(1)+rw_jang(2));

        x1 = -c4*s5;
        x2 = c4*c5*c6 - s4*s6;
        x3 = c4*c5*s6 + s4*c6;
        x4 = -s4*s5;
        x5 = s4*c5*c6 + c4*s6;
        x6 = s4*c5*s6 - c4*c6;
        x7 = L8*x3 + L7*x1 - L5*s4 + L5;
        x8 = L8*s5*s6 + L7*c5 - L6;
        x9 = -L8*x6 - L7*x4 - L5*c4 - L4 + L2;
        x10 = L3*c2 + c23*x7 + s23*x8;

        tw->n.x = c1*(c23*x1 + s23*c5) - s1*x4;
        tw->o.x = c1*(c23*x2 + s23*s5*c6) - s1*x5;
        tw->a.x = c1*(c23*x3 + s23*s5*s6) - s1*x6;
        tw->p.x = c1*(L3*c2 + x7*c23 + s23*x8) + s1*x9 - L9;

        tw->n.y = s1*(c23*x1 + s23*c5) + c1*x4;
        tw->o.y = s1*(c23*x2 + s23*s5*c6) + c1*x5;
        tw->a.y = s1*(c23*x3 + s23*s5*s6) + c1*x6;
        tw->p.y = s1*(L3*c2 + c23*x7 + s23*x8) - c1*x9 + L3;

        tw->n.z = s23*x1 - c23*c5;
        tw->o.z = s23*x2 - c23*s5*c6;
        tw->a.z = s23*x3 - c23*s5*s6;
        tw->p.z = s23*x7 - c23*x8 + L3*s2 + L1;

}

rw_close()
{
        RCIrelease(1);
        RCIclose (1);
}

rccl_close()
{
}

/*
 * called on ^C to close the record file if it was open
 */
void
quit()
{
        if (recfp)
                fclose(recfp);
}
```

```
/*
 *      File:   moveh.c
 *      Remarks: Hybrid position force control for the null desired force
 *               with the compliant wrist system (ref: hybri.c)
 */

#include <stdio.h>
#include "rw.h"

double  gain_ft,
        gain_fr,
        gain_pt,
        gain_pr,
        fuzz_j,
        fuzz_t,
        fuzz_r,
        filt_f_pole,
        filt_p_pole,
        s_x_t,
        s_y_t,
        s_z_t,
        s_x_r,
        s_y_r,
        s_z_r;
int     rw_comp = 0;
int     verbose;
extern FILE     *recfp;

main(ac, av)
int     ac;
char    **av;
{
        double  atof();

        gain_pr = 0.6;
        gain_pt = 0.9;
        gain_fr = 0.01;
        gain_ft = 0.2;
        fuzz_j = .01;
        fuzz_r = 0.5;
        fuzz_t = 0.5;
        filt_f_pole = 0.5;
        filt_p_pole = 0.95;
        s_x_t = 1.0;
        s_y_t = 1.0;
        s_z_t = 1.0;
        s_x_r = 1.0;
        s_y_r = 1.0;
        s_z_r = 1.0;

        while (--ac > 0  && **++av == '-') {
                register char   *p = *av;

                while (*++p != '\0')
                        switch (*p) {
                        case 'v':
                                verbose++; break;
                        case 'x':
                                s_x_t = atof(&p[1]); break;
                        case 'y':
                                s_y_t = atof(&p[1]); break;
                        case 'z':
                                s_z_t = atof(&p[1]); break;
                        case 'X':
                                s_x_r = atof(&p[1]); break;
                        case 'Y':
                                s_y_r = atof(&p[1]); break;
                        case 'Z':
                                s_z_r = atof(&p[1]); break;
                        case 'r':
                                fuzz_r = atof(&p[1]); break;
                        case 't':
                                fuzz_t = atof(&p[1]); break;
                        case 'j':
                                fuzz_j = atof(&p[1]); break;
                        case 'u':
                                gain_ft = atof(&p[1]); break;
                        case 'w':
                                gain_fr = atof(&p[1]); break;
                        case 'o':
                                gain_pt = atof(&p[1]); break;
                        case 'q':
                                gain_pr = atof(&p[1]); break;
                        case 'm':
                                filt_f_pole = atof(&p[1]); break;
                        case 'n':
                                filt_p_pole = atof(&p[1]); break;
                        case 'R':
                        if ((recfp = fopen(&p[1], "w")) == NULL) {
                        fprintf(stderr,
                                "cant open file for write\n"
                                );
                                exit(3);
                                }
                                goto nextarg;
                        }
nextarg:        ;
        }

        printf("gain in posn control for posn is %f\n", gain_pt);
        printf("gain in posn control for rotn is %f\n", gain_pr);
        printf("gain in force control for posn is %f\n", gain_ft);
        printf("gain in force control for rotn is %f\n", gain_fr);
        printf("filt pole in posn control is %f\n", filt_p_pole);
        printf("filt pole in force control is %f\n", filt_f_pole);
        printf("fuzz level for translation is %f\n", fuzz_t);
        printf("fuzz level for rotation is %f\n", fuzz_r);
        printf("control mode in x tran is %f\n", s_x_t);
        printf("control mode in y tran is %f\n", s_y_t);
        printf("control mode in z tran is %f\n", s_z_t);
        printf("control mode in x rot is %f\n", s_x_r);
        printf("control mode in y rot is %f\n", s_y_r);
        printf("control mode in z rot is %f\n", s_z_r);

        rw_comp = 0;

        /*
         * initialize the control loop.
         */
        rw_init();

        /*
         * start the complience. watch out.
         */
        rw_comp = 1;

        for(;;)
          rw();
```

```
}
```

```
/*
 *      File:   surf.c
 *      Remarks: Surface tracking operation with the compliant wrist
 *               system (ref: moves.c)
 */

/*
 * Please check the file hybri.c. Since there are detail descriptions
 * in hybri.c, those parameters appeared there are not defined here again
 * for simplicity.
 */

#include        <stdio.h>
#include        <rccl/rccl.h>
#include        <rccl/rci.h>
#include        <rccl/kine.h>
#include        "rw.h"
#include        "recordd.h"

#define N       6

#define dist_of_flange 55.88   /* 2.2 in = 55.88 mm.  This is the distance
                                  from the center of the last three joints to
                                  the outer surface of the flange*/
#define mount_dist  35.0

/*
 * the first order digital filter
 */
#define FILTER(y, u, pole)      (y=(1.0-pole)*(y*pole/(1.0-pole)+u))
#define NCOPY(a,b)      for(i=0;i<N;i++)a[i]=b[i]

double  car_diffs[6];

TRSF    mem_tw,
        *tw = &mem_tw;
extern double   gain_ft,
                gain_fr,
                gain_pt,
                gain_pr,
                deri_gain_ft,   /* derivative gain for force control in trans*/
                deri_gain_fr,   /* derivative gain for force control in rotn*/
                deri_gain_pt,   /* derivative gain for posn control in trans*/
                deri_gain_pr,   /* derivative gain for posn control in rotn*/
                fuzz_j,
                fuzz_t,
                fuzz_r,
                filt_f_pole,
                filt_p_pole,
                va,             /* approaching velocity */
                vt,             /* tracking velocity */
                cf,             /* desired contact force */
                tf;             /* specified turning force */
#define FUZZ            fuzz_j
#define FUZZ_CARPOS     fuzz_t
#define FUZZ_CAREUL     fuzz_r

DIFF    d;
DIFF    d_f, d_p, dvel;

double  jang_o[RW_MAX_JOINT];
double  rw_theta_cal[RW_MAX_JOINT],
        rw_theta_bar[RW_MAX_JOINT] = {
                        RW_THETA_BAR_0,
                        RW_THETA_BAR_1,
                        RW_THETA_BAR_2,
                        RW_THETA_BAR_3,
                        RW_THETA_BAR_4,
                        RW_THETA_BAR_5
        };
int     rclconst();
struct _record  rec;
int             sync,           /* user/interrupt coordination flag */
                time;           /* time maintained by interrupt function */
FILE            *recfp;         /* record file fp */

int     verbose;
double  d_angles[N];
double  r_angles[N];
SNCS    sncs;

/*
 * Here is where we initialize everything needed to make the robot do
 * what we want it to.  We establish the position equation that the
 * main program will continually move to.
 */
rw_init()
{
        TRSF    *tw_oinv;
        int     i;
        void    dummy(), drive();

        setbuf(stdout, NULL);
        rw_cal();

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                jang_o[i] = 0.0;
        }

        tw = newtrans("tw",rw);
        rw_car_update_tw();
        printrn(tw,stdout);
        /*
         * start the real-time process, and request arm power
         */
        RCIopen();
        RCIcontrol(dummy, drive);
        chg.power_on.com = YES;
        if ((how.state & CALIB_OK) == 0) {
                fprintf(stderr, "arm not calibrated\n");
                exit(3);
        }

        /* specify approaching velocity at initial state */
        dvel.t.z = va;
}

/*
 * Real-time drive function
 *
 *      read joint angles
 *      compute Jacobian at this point
 *      transform cartesian diff to joint space
 *      compute new joint angles
 *      output setpoint
 */
double  car_diffs[N];
```

```
double  r_angles[N];
double  d_angles[N];
void
drive()
{
        double  del_force[N], del_angle_vel[N];
        double  del_posn[N];
        short   encs[N];
        JNTS    q_f, qvel;
        JNTS    q_p;
        int     i;
        static int      initd = 0;
        static double   del_f_smth[N];
        static double   del_p_smth[N];

        if (initd == 0) {
                enctoang(r_angles, how.pos);    /* get actual joint angles */
                for (i=0; i<N; i++) {
                        d_angles[i] = r_angles[i];
                }
                initd++;
        }
        update_sincos(&sncs, r_angles);         /* compute sin/cos */
        update_jacobian_terms(&sncs);           /* compute jacob terms */

        /*
         * if the sensed contact force is higher than the turning force
         * tf, the motion in Z direction stops, and tracking begins.
         */
        if ( car_diffs[4] > tf || car_diffs[4] < -tf ) {
            dvel.t.z = 0.0;
            dvel.t.y = vt;
        }
        jacobI(&q_f, &d_f, &sncs, 0.0);         /* transform d to j space */
        jacobI(&q_p, &d_p, &sncs, 0.0);         /* transform d to j space */
        jnts_to_angle(del_force, &q_f);         /* delta j to angles */
        jnts_to_angle(del_posn, &q_p);          /* delta j to angles */

        jacobI(&qvel, &dvel, &sncs, 0.0);       /* transform d to j space */
        jnts_to_angle(del_angle_vel, &qvel);    /* delta j to angles */

        for (i=0; i<N; i++) {
                FILTER(del_f_smth[i], del_force[i], filt_f_pole);
                FILTER(del_p_smth[i], del_posn[i], filt_p_pole);
        }
        for (i=0; i<N; i++)
                d_angles[i] += del_f_smth[i] + del_angle_vel[i];
        for (i=0; i<N; i++)
                r_angles[i] = d_angles[i] + del_p_smth[i];
        NCOPY(rec.r_angles, r_angles);
        NCOPY(rec.car_diffs, car_diffs);
        rec.time = time;
        angtoenc(encs, r_angles);
        sync++;                 /* tell user process we have data */
        time++;

        for (i=0; i<N; i++) {
                chg.motion[i].com = POS;
                chg.motion[i].value = encs[i];
        }
}

void
dummy() {}
```

```
/*
 * rw_cal reads the current pot settings to get the current joint
 * angles.  These are then subtracted from the "correct" angles to
 * get the correction angles.
 */

rw_cal()
{
        int     i,
                j;

        /*
         * Initialize the axvll board.
         */

        if (ax_init() < 0)
        {
                fprintf(stderr,"Cannot initialize the axvll board\n");
                exit(1);
        }

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                rw_theta_cal[i] = rw_ptor(i);
        }

}

/*
 * This is the routine that has to figure out how to change the postion
 * equation established in rw_init() so the robot is driven to where we
 * want it to go.
 */
rw()
{
        int     i;
        static DIFF     car_deri;

        JNTS curr_jnts;
        JNTS diff_jnts;
        if (rw_comp)
        {
         bcopy(j6, &curr_jnts, sizeof(JNTS));
         rw_car_update_tw();
         if (verbose>1)
         printrn(tw,stdout);
         noatorpy(&car_diffs[5],&car_diffs[4],&car_diffs[3],tw);
         car_diffs[0] = tw->p.x;
         car_diffs[1] = tw->p.y;
         car_diffs[2] = tw->p.z - 62.0;

         if( car_diffs[0] <= FUZZ_CARPOS && car_diffs[0] >= -FUZZ_CARPOS )
                car_diffs[0] = 0.0;
         if( car_diffs[1] <= FUZZ_CARPOS && car_diffs[1] >= -FUZZ_CARPOS )
                car_diffs[1] = 0.0;
         if( car_diffs[2] <= FUZZ_CARPOS && car_diffs[2] >= -FUZZ_CARPOS )
                car_diffs[2] = 0.0;
         if( car_diffs[3] <= FUZZ_CAREUL && car_diffs[3] >= -FUZZ_CAREUL )
                car_diffs[3] = 0.0;
         if( car_diffs[4] <= FUZZ_CAREUL && car_diffs[4] >= -FUZZ_CAREUL )
                car_diffs[4] = 0.0;
         if( car_diffs[5] <= FUZZ_CAREUL && car_diffs[5] >= -FUZZ_CAREUL )
                car_diffs[5] = 0.0;
```

```
        if (verbose) {
                for (i=0; i<N; i++)
                        printf("%10.2f ", car_diffs[i]);
                putchar('\n');
                }

        /*
         * PD controller
         */
        d.t.x = car_diffs[0] * gain_pt + (car_diffs[0] - car_deri.t.x)
                 * deri_gain_pt;
        d.t.y = car_diffs[1] * gain_pt + (car_diffs[1] - car_deri.t.y)
                 * deri_gain_pt;
        d.t.z = (car_diffs[2] + cf)* gain_ft + (car_diffs[2] - car_deri.t.z)
                 * deri_gain_ft;
        d.r.x = dtor(car_diffs[3]) * gain_pr +
                 (car_diffs[3] - car_deri.r.x) * deri_gain_pr;
        d.r.y = dtor(car_diffs[4]-10.0) * gain_pr +
                 (car_diffs[4] -  car_deri.r.y) * deri_gain_pr;
                 /* 10.0 degree is for offset of ready position, which
                    is also an example to have a specific initial position */
        d.r.z = dtor(car_diffs[5]) * gain_pr +
                 (car_diffs[5] - car_deri.r.z) * deri_gain_pr;

        /*
         * update the previous displacement
         */
        car_deri.t.x = car_diffs[0];
        car_deri.t.y = car_diffs[1];
        car_deri.t.z = car_diffs[2];
        car_deri.r.x = car_diffs[3];
        car_deri.r.y = car_diffs[4];
        car_deri.r.z = car_diffs[5];

         /*
          * hybrid position force controller. In Z axis force is controlled
          * while in other degrees position is controlled
          */
        d_f.t.x = 0.0;
        d_f.t.y = 0.0;
        d_f.t.z = d.t.z;
        d_f.r.x = 0.0;
        d_f.r.y = 0.0;
        d_f.r.z = 0.0;

        d_p.t.x = -d.t.x;
        d_p.t.y = -d.t.y;
        d_p.t.z = 0.0;
        d_p.r.x = -d.r.x;
        d_p.r.y = -d.r.y;
        d_p.r.z = -d.r.z;

        if (recfp && sync) {
                fwrite(&rec, sizeof(rec), 1, recfp);
                sync = 0;
        }
        }
}

double
rw_jang(i)
int     i;
{
        double  jang,
                jang_diff,
                newdiff;

        int     s;

#ifdef notdef
        newdiff = rw_raw_diff(i);
        rw_old_diff[i] = newdiff;

#endif
        jang =  rw_raw_diff(i);

        jang_diff = (jang - jang_o[i]);
        if (jang_diff <= FUZZ && jang_diff >= -FUZZ)
        {
                        jang = jang_o[i];
        }
        jang_o[i] = jang;

        jang += rw_theta_bar[i];
        return(jang);
}
rw_car_update_tw()
{
        double  c1,
                c2,
                c3,
                c4,
                c5,
                c6,
                s1,
                s2,
                s3,
                s4,
                s5,
                s6,
                c23,
                s23,
                x1,
                x2,
                x3,
                x4,
                x5,
                x6,
                x7,
                x8,
                x9,
                x10;

        c1 = cos(rw_jang(0));
        s1 = sin(rw_jang(0));

        c2 = cos(rw_jang(1));
        s2 = sin(rw_jang(1));

        c3 = cos(rw_jang(2));
        s3 = sin(rw_jang(2));

        c4 = cos(rw_jang(3));
        s4 = sin(rw_jang(3));

        c5 = cos(rw_jang(4));
        s5 = sin(rw_jang(4));
```

```
        c6 = cos(rw_jang(5));
        s6 = sin(rw_jang(5));

        c23 = cos(rw_jang(1)+rw_jang(2));
        s23 = sin(rw_jang(1)+rw_jang(2));

        x1 = -c4*s5;
        x2 = c4*c5*c6 - s4*s6;
        x3 = c4*c5*s6 + s4*c6;
        x4 = -s4*s5;
        x5 = s4*c5*c6 + c4*s6;
        x6 = s4*c5*s6 - c4*c6;
        x7 = L8*x3 + L7*x1 - L5*s4 + L5;
        x8 = L8*s5*s6 + L7*c5 - L6;
        x9 = -L8*x6 - L7*x4 - L5*c4 - L4 + L2;
        x10 = L3*c2 + c23*x7 + s23*x8;

        tw->n.x = c1*(c23*x1 + s23*c5) - s1*x4;
        tw->o.x = c1*(c23*x2 + s23*s5*c6) - s1*x5;
        tw->a.x = c1*(c23*x3 + s23*s5*s6) - s1*x6;
        tw->p.x = c1*(L3*c2 + x7*c23 + s23*x8) + s1*x9 - L9;

        tw->n.y = s1*(c23*x1 + s23*c5) + c1*x4;
        tw->o.y = s1*(c23*x2 + s23*s5*c6) + c1*x5;
        tw->a.y = s1*(c23*x3 + s23*s5*s6) + c1*x6;
        tw->p.y = s1*(L3*c2 + c23*x7 + s23*x8) - c1*x9 + L3;

        tw->n.z = s23*x1 - c23*c5;
        tw->o.z = s23*x2 - c23*s5*c6;
        tw->a.z = s23*x3 - c23*s5*s6;
        tw->p.z = s23*x7 - c23*x8 + L3*s2 + L1;

}

rw_close()
{
        RCIrelease(1);
        RCIclose (1);
}
rccl_close()
{
}

/*
 * called on ^C to close the record file if it was open
 */
void
quit()
{
        if (recfp)
                fclose(recfp);
}
```

```
/*
 *      File:   moves.c
 *      Remarks: Surface tracking operation with the compliant wrist
 *               system (ref: surf.c)
 */

#include <stdio.h>
#include "rw.h"

double  gain_ft,
        gain_fr,
        gain_pt,
        gain_pr,
        deri_gain_ft,
        deri_gain_fr,
        deri_gain_pt,
        deri_gain_pr,
        fuzz_j,
        fuzz_t,
        fuzz_r,
        filt_f_pole,
        filt_p_pole,
        vt,
        va,
        cf,
        tf;
int     rw_comp = 0;
int     verbose;
extern FILE     *recfp;

main(ac, av)
int     ac;
char    **av;
{
        double  atof();

        gain_pr = 0.6;
        gain_pt = 0.9;
        gain_fr = 0.005;
        gain_ft = 0.05;
        deri_gain_pr = -0.01;
        deri_gain_pt = -0.1;
        deri_gain_fr = 0.01;
        deri_gain_ft = 0.1;
        fuzz_j = .01;
        fuzz_r = 0.4;
        fuzz_t = 0.4;
        filt_f_pole = 0.3;
        filt_p_pole = 0.98;
        vt = 0.2;
        va = 0.8;
        cf = 0.7;
        tf = 0.5;
        while (--ac > 0  && **++av == '-') {
                register char   *p = *av;

                while (*++p != '\0')
                        switch (*p) {
                        case 'v':
                                verbose++; break;
                        case 'V':
                                vt = atof(&p[1]); break;
                        case 'A':
                                va = atof(&p[1]); break;
                        case 'c':
                                cf = atof(&p[1]); break;
                        case 'T':
                                gain_ft = atof(&p[1]); break;
                        case 'O':
                                gain_fr = atof(&p[1]); break;
                        case 't':
                                gain_pt = atof(&p[1]); break;
                        case 'o':
                                gain_pr = atof(&p[1]); break;
                        case 'm':
                                filt_f_pole = atof(&p[1]); break;
                        case 'n':
                                filt_p_pole = atof(&p[1]); break;
                        case 'R':
                        if ((recfp = fopen(&p[1], "w")) == NULL) {
                        fprintf(stderr,
                                "cant open file for write\n"
                                );
                                exit(3);
                                }
                                goto nextarg;
                        }
nextarg:        ;
        }

        printf("gain in posn control for posn is %f\n", gain_pt);
        printf("gain in posn control for rotn is %f\n", gain_pr);
        printf("gain in force control for posn is %f\n", gain_ft);
        printf("gain in force control for rotn is %f\n", gain_fr);
        printf("filt pole in posn control is %f\n", filt_p_pole);
        printf("filt pole in force control is %f\n", filt_f_pole);
        printf("tracking velocity is %f\n", vt);
        printf("approaching velocity is %f\n", va);
        printf("contact force is %f\n", cf);
        printf("turning force is %f\n", tf);

        rw_comp = 0;

        /*
         * initialize the control loop.
         */
        rw_init();


        /*
         * start the complience. watch out.
         */
        rw_comp = 1;

        for(;;)
          rw();
}
```

```
/*
 *      File:   edge.c
 *      Remarks: Edge tracking operation with the compliant wrist
 *               system (ref: moveg.c)
 */

/*
 * Please check the file hybri.c. Since there are detail descriptions
 * in hybri.c, those parameters appeared there are not defined here again
 * for simplicity.
 */

#include        <stdio.h>
#include        <rccl/rccl.h>
#include        <rccl/rci.h>
#include        <rccl/kine.h>
#include        "rw.h"
#include        "recordd.h"

#define N       6

#define dist_of_flange 55.88   /* 2.2 in = 55.88 mm.  This is the distance
                                  from the center of the last three joints to
                                  the outer surface of the flange */
#define mount_dist  35.0

/*
 * the first order digital filter
 */
#define FILTER(y, u, pole)      (y=(1.0-pole)*(y*pole/(1.0-pole)+u))
#define NCOPY(a,b)      for(i=0;i<N;i++)a[i]=b[i]

double  car_diffs[6];

TRSF    mem_tw,
        *tw = &mem_tw;
extern double   gain_ft,
                gain_fr,
                gain_pt,
                gain_pr,
                fuzz_j,
                fuzz_t,
                fuzz_r,
                filt_f_pole,
                filt_p_pole,
                va,             /* approaching velocity */
                vs,             /* searching velocity */
                vt,             /* tracking velocity */
                cfa,            /* desired contact force in the
                                   approaching direction */
                cfs,            /* desired contact force in the
                                   searching direction */
                tfa,            /* specified turning force in the
                                   approaching direction */
                tfs;            /* specified turning force in the
                                   searching direction */
#define FUZZ            fuzz_j
#define FUZZ_CARPOS     fuzz_t
#define FUZZ_CAREUL     fuzz_r

DIFF    d;
DIFF    d_f;
DIFF    d_p, dvel;

double  jang_o[RW_MAX_JOINT];
double  rw_theta_cal[RW_MAX_JOINT],
        rw_theta_bar[RW_MAX_JOINT] = {
                        RW_THETA_BAR_0,
                        RW_THETA_BAR_1,
                        RW_THETA_BAR_2,
                        RW_THETA_BAR_3,
                        RW_THETA_BAR_4,
                        RW_THETA_BAR_5
        };
int     rclconst();
struct _record  rec;
int             sync,           /* user/interrupt coordination flag */
                time;           /* time maintained by interrupt function */
FILE            *recfp;         /* record file fp */

int     verbose;
double  d_angles[N];
double  r_angles[N];
SNCS    sncs;

/*
 * Here is where we initialize everything needed to make the robot do
 * what we want it to.  We establish the position equation that the
 * main program will continually move to.
 */
rw_init()
{
        TRSF    *tw_oinv;
        int     i;
        void    dummy(), drive();


        setbuf(stdout, NULL);
        rw_cal();

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                jang_o[i] = 0.0;
        }

        tw = newtrans("tw",rw);
        rw_car_update_tw();
        printrn(tw,stdout);
        /*
         * start the real-time process, and request arm power
         */
        RCIopen();
        RCIcontrol(dummy, drive);
        chg.power_on.com = YES;
        if ((how.state & CALIB_OK) == 0) {
                fprintf(stderr, "arm not calibrated\n");
                exit(3);
        }
        /*
         * specify the approaching velocity
         */
        dvel.t.z = va;
}

/*
 * Real-time drive function
 *
 *      read joint angles
```

```
 *      compute Jacobian at this point
 *      transform cartesian diff to joint space
 *      compute new joint angles
 *      output setpoint
 */
double  car_diffs[N];
double  r_angles[N];
double  d_angles[N];
void
drive()
{
        double  del_force[N], del_angle_vel[N];
        double  del_posn[N];
        short   encs[N];
        JNTS    q_f, qvel;
        JNTS    q_p;
        int     i;
        static int      initd = 0;
        static double   del_f_smth[N];
        static double   del_p_smth[N];

        if (initd == 0) {
                enctoang(r_angles, how.pos);    /* get actual joint angles */
                for (i=0; i<N; i++) {
                        d_angles[i] = r_angles[i];
                }
                initd++;
        }
        update_sincos(&sncs, r_angles);         /* compute sin/cos */
        update_jacobian_terms(&sncs);           /* compute jacob terms */

        /* when the sensed contact force is higher than the specified
         * turning force, the motion in the approaching direction stops,
         * and the motion in the searching direction begains
         */
        if ( car_diffs[4] > tfa || car_diffs[4] < -tfa ) {
            dvel.t.z = 0.0;
            dvel.t.x = -vs;

        /* when the sensed contact force in this drection is higher than
         * the specified turning force in this direction, the motion in
         * this searching direction stops, and the motion in the
         * tracking direction begains
         */
                  if ( car_diffs[0] > tfs || car_diffs[0] < -tfs ){
                  dvel.t.x = 0.0;
                  dvel.t.y = vt;
                }
          }
        jacobI(&q_f, &d_f, &sncs, 0.0);         /* transform d to j space */
        jacobI(&q_p, &d_p, &sncs, 0.0);         /* transform d to j space */
        jnts_to_angle(del_force, &q_f);         /* delta j to angles */
        jnts_to_angle(del_posn, &q_p);          /* delta j to angles */

        jacobI(&qvel, &dvel, &sncs, 0.0);       /* transform d to j space */
        jnts_to_angle(del_angle_vel, &qvel);    /* delta j to angles */

        for (i=0; i<N; i++) {
                FILTER(del_f_smth[i], del_force[i], filt_f_pole);
                FILTER(del_p_smth[i], del_posn[i], filt_p_pole);
        }
        for (i=0; i<N; i++)
                d_angles[i] += del_f_smth[i] + del_angle_vel[i];
        for (i=0; i<N; i++)
                r_angles[i] = d_angles[i] + del_p_smth[i];
        NCOPY(rec.r_angles, r_angles);
        NCOPY(rec.car_diffs, car_diffs);
        rec.time = time;

        angtoenc(encs, r_angles);
        sync++;                 /* tell user process we have data */
        time++;

        for (i=0; i<N; i++) {
                chg.motion[i].com = POS;
                chg.motion[i].value = encs[i];
        }
}

void
dummy() {}

/*
 * rw_cal reads the current pot settings to get the current joint
 * angles.  These are then subtracted from the "correct" angles to
 * get the correction angles.
 */

rw_cal()
{
        int     i,
                j;

        /*
         * Initialize the axv11 board.
         */

        if (ax_init() < 0)
        {
                fprintf(stderr,"Cannot initialize the axv11 board\n");
                exit(1);
        }

        for (i = 0; i < RW_MAX_JOINT; ++i)
        {
                rw_theta_cal[i] = rw_ptor(i);
        }

}

/*
 * This is the routine that has to figure out how to change the postion
 * equation established in rw_init() so the robot is driven to where we
 * want it to go.
 */
rw()
{
        int     i;
        static DIFF     car_deri;

    JNTS curr_jnts;
    JNTS diff_jnts;
        if (rw_comp)
        {
                bcopy(j6, &curr_jnts, sizeof(JNTS));
                rw_car_update_tw();
                if (verbose>1)
                        printrn(tw,stdout);
```

```
                noatorpy(&car_diffs[5],&car_diffs[4],&car_diffs[3],tw);
                car_diffs[0] = tw->p.x;
                car_diffs[1] = tw->p.y;
                car_diffs[2] = tw->p.z - 62.0;

         if( car_diffs[0] <= FUZZ_CARPOS && car_diffs[0] >= -FUZZ_CARPOS )
                car_diffs[0] = 0.0;
         if( car_diffs[1] <= FUZZ_CARPOS && car_diffs[1] >= -FUZZ_CARPOS )
                car_diffs[1] = 0.0;
         if( car_diffs[2] <= FUZZ_CARPOS && car_diffs[2] >= -FUZZ_CARPOS )
                car_diffs[2] = 0.0;
         if( car_diffs[3] <= FUZZ_CAREUL && car_diffs[3] >= -FUZZ_CAREUL )
                car_diffs[3] = 0.0;
         if( car_diffs[4] <= FUZZ_CAREUL && car_diffs[4] >= -FUZZ_CAREUL )
                car_diffs[4] = 0.0;
         if( car_diffs[5] <= FUZZ_CAREUL && car_diffs[5] >= -FUZZ_CAREUL )
                car_diffs[5] = 0.0;

                if (verbose) {
                        for (i=0; i<N; i++)
                                printf("%10.2f ", car_diffs[i]);
                        putchar('\n');
                }

        /*
         * Hybrid controller specified force control in the approaching
         * and searching direction, while position control in the others
         */
        d.t.x = (car_diffs[0] - cfs) * gain_ft;
        d.t.y = car_diffs[1] * gain_pt;
        d.t.z = (car_diffs[2] + cfa) * gain_ft;
        d.r.x = dtor(car_diffs[3]) * gain_pr;
        d.r.y = dtor(car_diffs[4]-10.0) * gain_pr;
        d.r.z = dtor(car_diffs[5]) * gain_pr;

        d_f.t.x = d.t.x;
        d_f.t.y = 0.0;
        d_f.t.z = d.t.z;
        d_f.r.x = 0.0;
        d_f.r.y = 0.0;
        d_f.r.z = 0.0;

        d_p.t.x = 0.0;
        d_p.t.y = -d.t.y;
        d_p.t.z = 0.0;
        d_p.r.x = -d.r.x;
        d_p.r.y = -d.r.y;
        d_p.r.z = -d.r.z;

        if (recfp && sync) {
                fwrite(&rec, sizeof(rec), 1, recfp);
                sync = 0;
        }
        }
}

double
rw_jang(i)
int     i;
{
        double  jang,
                jang_diff,
                newdiff;
        int     s;

#ifdef notdef
        newdiff = rw_raw_diff(i);
        rw_old_diff[i] = newdiff;

#endif
        jang =  rw_raw_diff(i);
        jang_diff = (jang - jang_o[i]);
        if (jang_diff <= FUZZ && jang_diff >= -FUZZ)
        {
                        jang = jang_o[i];
        }
        jang_o[i] = jang;

        jang += rw_theta_bar[i];
        return(jang);
}
rw_car_update_tw()
{
        double  c1,
                c2,
                c3,
                c4,
                c5,
                c6,
                s1,
                s2,
                s3,
                s4,
                s5,
                s6,
                c23,
                s23,
                x1,
                x2,
                x3,
                x4,
                x5,
                x6,
                x7,
                x8,
                x9,
                x10;

        c1 = cos(rw_jang(0));
        s1 = sin(rw_jang(0));

        c2 = cos(rw_jang(1));
        s2 = sin(rw_jang(1));

        c3 = cos(rw_jang(2));
        s3 = sin(rw_jang(2));

        c4 = cos(rw_jang(3));
        s4 = sin(rw_jang(3));

        c5 = cos(rw_jang(4));
        s5 = sin(rw_jang(4));

        c6 = cos(rw_jang(5));
        s6 = sin(rw_jang(5));

        c23 = cos(rw_jang(1)+rw_jang(2));
```

```
        s23 = sin(rw_jang(1)+rw_jang(2));

        x1 = -c4*s5;
        x2 = c4*c5*c6 - s4*s6;
        x3 = c4*c5*s6 + s4*c6;
        x4 = -s4*s5;
        x5 = s4*c5*c6 + c4*s6;
        x6 = s4*c5*s6 - c4*c6;
        x7 = L8*x3 + L7*x1 - L5*s4 + L5;
        x8 = L8*s5*s6 + L7*c5 - L6;
        x9 = -L8*x6 - L7*x4 - L5*c4 - L4 + L2;
        x10 = L3*c2 + c23*x7 + s23*x8;

        tw->n.x = c1*(c23*x1 + s23*c5) - s1*x4;
        tw->o.x = c1*(c23*x2 + s23*s5*c6) - s1*x5;
        tw->a.x = c1*(c23*x3 + s23*s5*s6) - s1*x6;
        tw->p.x = c1*(L3*c2 + x7*c23 + s23*x8) + s1*x9 - L9;

        tw->n.y = s1*(c23*x1 + s23*c5) + c1*x4;
        tw->o.y = s1*(c23*x2 + s23*s5*c6) + c1*x5;
        tw->a.y = s1*(c23*x3 + s23*s5*s6) + c1*x6;
        tw->p.y = s1*(L3*c2 + c23*x7 + s23*x8) - c1*x9 + L3;

        tw->n.z = s23*x1 - c23*c5;
        tw->o.z = s23*x2 - c23*s5*c6;
        tw->a.z = s23*x3 - c23*s5*s6;
        tw->p.z = s23*x7 - c23*x8 + L3*s2 + L1;

}

rw_close()
{
        RCIrelease(1);
        RCIclose (1);
}

rccl_close()
{
}

/*
 * called on ^C to close the record file if it was open
 */
void
quit()
{
        if (recfp)
                fclose(recfp);
}
```