Copyright © 1991, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

HIERARCHICAL CONTROL IN PLANAR GRASPING AND MANIPULATION — AN EXPERIMENTAL STUDY

by

Karin Hollerbach

Memorandum No. UCB/ERL M91/113

17 December 1991

HIERARCHICAL CONTROL IN PLANAR GRASPING AND MANIPULATION — AN EXPERIMENTAL STUDY

by

Karin Hollerbach

Memorandum No. UCB/ERL M91/113

17 December 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

HIERARCHICAL CONTROL IN PLANAR GRASPING AND MANIPULATION — AN EXPERIMENTAL STUDY

by

Karin Hollerbach

Memorandum No. UCB/ERL M91/113

17 December 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Hierarchical Control in Planar Grasping and Manipulation—An Experimental Study

Karin Hollerbach
December 11, 1991

Abstract

We compare the performance of hierarchical and single-level controllers in a grasping context, and we conclude that for rapid, planar grasping motions of heavy objects the performance of a hierarchical control structure is superior to that of the two single-level controllers tested. For slow movements of lighter objects the performance of the three controllers is similar; with an increase in movement speed and object mass, however, the hierarchical structure becomes increasingly important. Although the theory discussed here applies to grasping problems of arbitrary complexity, we focus on planar, two-fingered grasping for the sake of clarity and to simplify implementation and experimental testing of the proposed control algorithms.

Contents

1 Introduction **5**

2 Hierarchical Control of Grasp Dynamics **7**
2.1 Grasp dynamics 7
2.2 Control 9
2.3 Primitives for robot control 10
2.4 Stability of Hierarchical Control 12

3 Experimental Setup **13**
3.1 Styx–Hardware 13
3.2 Control Hierarchy 15

4 Experimental Results **19**
4.1 Presentation of Results 19
4.2 Discussion of Results 29

5 Conclusion **33**

A Summary of STYX Parameters **41**

B STYX Dynamics with Rolling Contacts **45**
B.1 Forward Kinematics 45
B.2 Contact Coordinates 45
B.3 Elimination of Object Coordinate Measurements 47
B.4 Grasp Map with Rolling Contacts 47
B.5 Jacobian Matrix with Rolling Contacts 48
B.6 Dynamics 48

C STYX Source Code **49**

List of Figures

2.1 Model of a hierarchical control scheme of a human finger and thumb [7] . (Figure courtesy of D. Curtis Deno) 12

3.1 Top View of Styx 14

3.2 Hardware Supporting Styx 16

3.3 Control hierarchy for Styx, showing control of the desired and actual object trajectories, X_d and X_{act}, respectively, and the specified internal force, f_N. 17

4.1 Actual and commanded object trajectories with setpoint controller (a), joint interpolation controller (b), and hierarchical controller (c) in Cartesian space with weighted beam; commanded trajectories are shown within each graph (fast circular trajectory). 21

4.2 Deviation (in cm) from commanded trajectories of actual object trajectories with setpoint controller, joint-interpolation controller, and hierarchical controller when using weighted beam (fast circular trajectory). 22

4.3 Deviation (in cm) from commanded trajectories of actual object trajectories with setpoint controller, joint-interpolation controller, and hierarchical controller (figure-8 trajectory); experimental details are summarized in the appendix. 23

4.4 Commanded "square box" object trajectory (a), actual trajectory with setpoint controller (b), with joint interpolation controller (c), and with hierarchical controller (d); experimental details are summarized in the appendix 24

4.5 Deviation (in radians) from commanded object orientation of actual object orientation with setpoint controller, joint interpolation controller, and hierarchical controller from commanded trajectories when using weighted beam (fast circular trajectory). 25
4.6 Commanded object orientation (labeled with circles) and actual object orientation with joint interpolation controller (squares) and with hierarchical controller (triangles) in the "twisting" trajectory; experimental details are summarized in the appendix 26
4.7 Magnitude of the total force applied to the object at the fingertips (labeled with triangles) and the inertial forces (squares) as determined by the joint angle trajectory data in the the "twisting" trajectory; experimental details are summarized in the appendix 27
4.8 Object trajectory, box plot of object position error, and object orientation error as a function of time for setpoint controller (a), setpoint controller with joint interpolation (b), and hierarchical controller (c) (slow circular trajectory). 28
4.9 Effect of increased specified internal force on trajectory errors using the setpoint controller; trajectories (a) and (b) were followed with a specified internal force of 3×10^4 and 5×10^4 dyne, respectively; the plus sign ("+") marks the center of the commanded circular trajectory. 29

B.1 Definitions of angles in area of fingertip-object contact; (a) the left finger in contact with the left side of the object; (b) enlarged view of the left fingertip area—shown for clarity in a different position than that shown in (a). 46

Chapter 1

Introduction

Interest in complex, multi-fingered robotic hands has seen an increase in the last few years, as advanced designs and a rigorous theory used to describe them have been developed. Early research on multi-fingered hands tended to focus on the description of hand kinematics [13] and on the generation of stable grasps [20]. Later, researchers started to develop simple algorithms used in grasping control of single hands [4, 12, 15, 16] as well as control of associated groups of robots, cooperating to perform a single task [1, 2, 11, 19].

Advances in multi-fingered hand design are readily apparent when perusing the literature: the more well-known designs include the Utah-MIT hand [8], the Stanford/JPL hand [28], the NYU hand [5]. However, the problem of overcoming the computational burden associated with control of some of the more complicated hand designs has not been adequately addressed. In response to the computational difficulties surrounding control of complicated robotic hands we seek an approach to multi-fingered hand control that significantly reduces the computational burden placed on the controller while improving the grasping performance of the hand, and that is essentially design-independent. A beginning in the development of such an approach was made by Deno *et al.* in [7], and the experimental results presented here may be considered an implementation of the basic philosophy of hierarchical robot control as laid out there.

A motivating factor in this study of hierarchical grasping control is found in the highly effective, adaptable mammalian neuro-muscular control system and its hierarchy of spinal and cortical neural signals and control loops [14, 23]. Time delays inherent in biological motor systems indicate that control is likely to be hierarchical, occurring at many different levels

of the central nervous system. For the same reasons, communication and computation delays make hierarchical controllers an attractive method for providing high system bandwidth while coordinating many degrees of freedom. These motivations are not limited to robotics: there is a large literature concerned with the use of the related, though not directly applicable, theory of decentralized control for general dynamic systems [21, 25];

Centralized control has been defined as a case in which every sensor's output influences every actuator [25]. The study of large scale systems led to a number of results concerning weakly coupled systems, with decentralized control, and hierarchical systems, with controllers exhibiting a separation of time-scales. Graph decomposition techniques permitted the isolation of sets of states, inputs, and outputs that were weakly coupled. This decomposition simplified stability analyses and controller design. In multi-processor control systems, decentralized control is often mandated by restrictions on the communication rates between processors. Hierarchical controllers, as exemplified by Clark's HIC [3], limit inter-level communication to communication between adjacent levels of the hierarchy. HIC is an operating system intended to manage servo loops found in robot controllers in which Clark emphasizes distributed processing and interprocessor communication.

In this paper we compare the performance in a planar grasping task of a hierarchical control algorithm with that of single-level controllers containing no hierarchical structure. In doing so, we first selectively review the dynamics and control of robot systems and discuss the natural hierarchical structure that arises in biological grasping systems and may be created in robotic multi-fingered hands. We show that in the hierarchy used here, we have a control scheme that, with fast low-level controllers, induces the tracking error to converge to zero. We then proceed to describe the methods and the hardware used in the experimental comparison of the controllers' performance and subsequently present the experimental evidence demonstrating the superior performance in rapid, planar movements of heavy objects of the hierarchical controller. Finally, we discuss the merits of the hierarchical approach to the control of grasping and draw parallels between grasping in biological systems and in multi-fingered robotic hands.

Chapter 2

Hierarchical Control of Grasp Dynamics

In this section we selectively review the dynamics and control of robot systems. Following the results of [17], we show that there is a natural hierarchical structure of the control system for multi-fingered hands that mirrors the physical structure of the system.

2.1 Grasp dynamics

The dynamics for an open kinematic chain robot manipulator with joint angles $\theta \in \mathbf{R}^n$ and actuator torques $\tau \in \mathbf{R}^n$ can be derived using Lagrange's equations and written in the form

$$M(\theta)\ddot{\theta} + C(\theta,\dot{\theta})\dot{\theta} + N(\theta,\dot{\theta}) = \tau \tag{2.1}$$

where $M(\theta)$ is a positive definite inertia matrix and $C(\theta,\dot{\theta})\dot{\theta}$ is the Coriolis and centrifugal force vector. The vector $N(\theta,\dot{\theta}) \in \mathbf{R}^n$ contains all friction and gravity terms, and the vector $\tau \in \mathbf{R}^n$ represents generalized forces in the θ coordinate frame. For systems of this type, it can be shown that $\dot{M} - 2C$ is a skew symmetric matrix with proper choice of C (see [27]).

When contact constraints are added to a robot system, for example when a multi-fingered hand grasps an object, the robot dynamics can still be represented in the same form as equation (2.1). In the case of a multi-fingered hand grasping a box, we let θ be the vector of joint angles and x be the vector describing the position and orientation of the box. With these

definitions, the grasping constraint may be written as

$$J(q)\dot{\theta} = G^T(q)\dot{x}, \tag{2.2}$$

where $q = (\theta, x) \in \mathbb{R}^m \times \mathbb{R}^n$, J is the Jacobian of the finger kinematic function and G is the "grasp map" for the system. We will assume that J is bijective in some neighborhood and that G is surjective. This form of constraint can also be used to describe a wide variety of other systems, including grasping with rolling contacts, surface following and coordinated lifting. For ease of exposition, we also assume that there exists a forward kinematic function between θ and x; that is, the constraint is holonomic. A more complete derivation of grasping kinematics can be found in [18].

To include velocity constraints in the dynamics formulation, we again use Lagrange's equations. Following the approach in [18], the equations of motion for the constrained system can be written as

$$\tilde{M}(q)\ddot{x} + \tilde{C}(q,\dot{q})\dot{x} + \tilde{N}(q,\dot{q}) = F_e \tag{2.3}$$

where

$$\begin{aligned}
\tilde{M} &= M + GJ^{-T}M_\theta J^{-1}G^T \\
\tilde{C} &= C + GJ^{-T}\left(C_\theta J^{-1}G^T + M_\theta \frac{d}{dt}\left(J^{-1}G^T\right)\right) \\
\tilde{N} &= GJ^{-T}N \\
F_e &= GJ^{-T}\tau \\
M, M_\theta &= \text{inertia matrix for the object and fingers, respectively} \\
C, C_\theta &= \text{Coriolis and centrifugal terms}
\end{aligned}$$

Thus, we have an equation with a form similar to that of our "simple" robot. In the object's frame of reference, $\tilde{M}$ is the matrix of the effective mass of the object, and $\tilde{C}$ is the effective Coriolis and centrifugal matrix. These matrices include the dynamics of the fingers, which are being used to actually control the motion of the object. However, the details of the finger kinematics and dynamics are effectively hidden in the definition of $\tilde{M}$ and $\tilde{C}$. The skew symmetry of $\dot{\tilde{M}} - 2\tilde{C}$ is preserved by this transformation.

Although the grasp map G was assumed to be surjective, it need not be square. From the equations of motion (2.3), we note that if the fingertip force $J^{-T}\tau$ is in the null space of G, the net force in the object's frame of reference is zero and causes no net motion of the object. This type of force

acts against the constraint and is generally termed an *internal* or *constraint* force. We can use such an internal force to satisfy other conditions, such as keeping the contact forces inside the friction cone (to avoid slipping) or varying the load distribution of a set of manipulators rigidly grasping an object.

2.2 Control

To illustrate the control of robot systems, we look at two controllers which have appeared in the robotics literature. We start by considering systems of the form

$$M(q)\ddot{x} + C(q,\dot{q})\dot{x} + N(q,\dot{q}) = F \tag{2.4}$$

where $M(q)$ is a positive definite inertia matrix, and $C(q,\dot{q})\dot{x} \in \mathbf{R}^n$ is the Coriolis and centrifugal force vector. The vector $N(q,\dot{q}) \in \mathbf{R}^n$ contains all friction and gravity terms, and the vector $F \in \mathbf{R}^n$ represents generalized forces in the x coordinate frame.

Computed torque

The computed torque control law is a special case of the more general technique of feedback linearization. That is, through the use of nonlinear feedback, we wish to render the system dynamics linear in some appropriate set of coordinates. For a robot manipulator, given a desired trajectory, x_d, we use the control

$$F = M(q)\left(\ddot{x}_d + K_v\dot{e} + K_p e\right) + C(q,\dot{q})\dot{x} + N(q,\dot{q}) \tag{2.5}$$

where error $e = x_d - x$, and K_v and K_p are constant gain matrices. The resulting dynamics equations are linear, with exponential rate of convergence determined by K_v and K_p. Since the system is linear, we can use linear control theory to choose the gains (K_v and K_p) such that they satisfy some set of design criteria. In particular, if we choose K_v and K_p to be diagonal, we can analyze the system using the notions of system bandwidth and damping. We make use of these notions in the analysis of the experiments presented here, as we shall see later.

PD + feedforward control

PD controllers differ from computed torque controllers in that the desired stiffness (and potentially damping) of the end effector is specified, rather

than its position tracking characteristics. Typically, control laws of this form rely on the skew-symmetric property of robot dynamics, that is to say $\alpha^T \left(\dot{M} - 2C\right) \alpha = 0$ for all $\alpha \in \mathbf{R}^n$. Consider the control law

$$F = M(q)\ddot{x}_d + C(q,\dot{q})\dot{x}_d + N(q,\dot{q}) + K_v\dot{e} + K_p e \tag{2.6}$$

where both K_v and K_p are symmetric positive definite. Using a Liapunov stability argument, it can be shown that the actual trajectory of the robot converges to the desired trajectory asymptotically [9]. Extensions to the control law result in exponential rate of convergence [24, 26]. We note here that for a diagonal mass matrix, $M(q)$, the computed torque controller and the PD plus feedforward controller, when integrated into the upper level of the hierarchical structure, reduce to the same control algorithm. We make use of this fact later in determining the choice of gains by analyzing the system bandwidth and damping.

2.3 Primitives for robot control

A multi-fingered robot hand can be modeled as a set of robots that are connected to objects via a set of contact and control constraints. Given a set of robots and constraints on their interconnections, we can generate a new dynamic system and model it using the same basic principles that were used in the generation of the sub-robots. In fact, the procedure in which we use simple robots and build upon them to generate more multi-fingered robotic hands is simple enough that we can automate, to some extent, the procedure. In [6, 17], a system was proposed for building hierarchical control laws for complex, interconnected robotic systems. We review that formulation here.

The proposed procedure for generating complicated, multi-level robotic systems is based upon a set of robot primitives, consisting of objects that possess certain attributes and are connected via additional primitives. Robots, or objects, are dynamical systems that are each recursively `defined` in terms of the properties of daughter robots. The entire multi-level robot structure can be envisioned as a tree structure, in which the leaves are robots. Each robot is `defined` as an object that possesses attributes such as inertial parameters and kinematics. Robots also function as devices whose input consists of desired position and force trajectories and whose output consists, in turn, of the robots' actual position and force.

Additional primitives serve as branches to connect the various nodes of the robot tree structure. In particular, the `attach` and `control` primitives

act upon sets of robots to create new robots by linking them together. As one ascends the robot tree from the leaves to its roots, one encounters robots with progressively fewer degrees of freedom, indicating a compression of information that allows for control using increasing levels of abstraction.

The `attach` primitive reflects geometrical constraints between robots, creating new robots from sets of daughter robots, while also effecting a coordinate transformation. The `attach` primitive simultaneously expands desired positions and force trajectories as one moves out to the more distal nodes of the tree and compresses actual position and force information as one travels in a more proximal direction, thus, creating a bi-directional flow of information throughout the tree structure.

The second connecting device within the tree structure is the `control` primitive. Essentially, the purpose of the `control` primitive is to make a robot follow the position and force trajectory specified for it. The primitive applies a specified control algorithm to the desired and actual robot positions and forces and, as a result, obtains a set of expected positions and forces for the daughter robot. The daughter robot, in turn, provides robot objects located at more proximal nodes of the tree with its actual state information.

The block diagram portion of Figure 2.1 may be viewed as an example of a robot system comprised of these primitives. The robot system, in turn, is a model of the human control system depicted in the right half of the figure. Starting from the bottom: a finger and thumb are `defined`; each digit is controlled by muscle tension and stiffness; the muscles and sensory organs of each form low-level, fast spinal reflex loops that go directly from the digit to the spinal cord and back to the digit. The two digits are `attached` to form a composite hand. Further up the hierarchy, the brainstem and cerebellum help `control` and coordinate motor commands and sensory information. Finally, at the highest level, the sensory motor cortex, where sensory information is perceived and where conscious motor commands originate, the fingers are thought of as a pincer which engages in high level tasks such as picking.

In designing controllers using the primitives described above, a key issue is how to properly model a robot that has a controller attached to it. In order to allow the recursive nature of the primitives to operate, we must describe the new robot as a dynamical system with equations of motion in the form given by equation (2.3). For controllers that are very fast relative to higher levels, it is often a good approximation to model the robot as an ideal force generator, with no mass. This approximation does not imply that the robot is no longer a dynamic object, but rather that controllers at higher

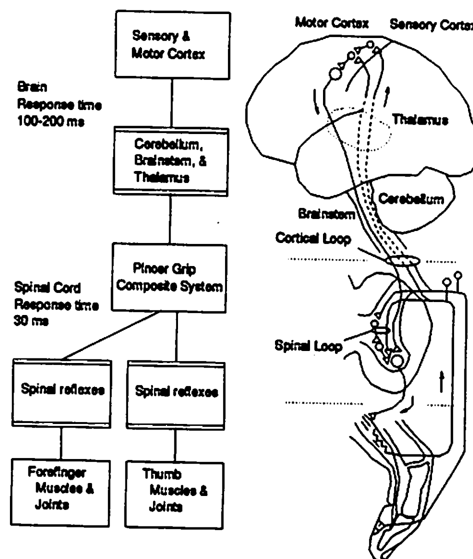

Figure 2.1: Model of a hierarchical control scheme of a human finger and thumb [7]. (Figure courtesy of D. Curtis Deno)

levels can ignore the dynamic properties of the robot, since these properties are being compensated for at a lower level.

We use the primitives formulation of the robot control structure in modeling the experimental setup used here. In particular, we implement a hierarchical control scheme in which the lower control levels, analogous to the spinal reflex model described here, operate at the finger joint levels, leaving the high level free to operate solely within the level of task coordinates.

2.4 Stability of Hierarchical Control

In order to demonstrate stability of the hierarchical control scheme proposed here, we model the scheme as a two-step hierarchy: a low level PD at the motor level and a high level PD-type control for the entire "hand". If the low level scheme were infinitely fast, then the scheme would be a "conventional PD" type that is guaranteed to be exponentially convergent, that is, the tracking error goes to zero exponentially. Standard singular perturbation arguments [10] may be used to show that the scheme is exponentially convergent, provided that the low level controller is fast enough, or, equivalently, that the sampling period is small enough.

Chapter 3

Experimental Setup

What follows is a description of the hardware and software used in the control experiments presented here. First, we describe in some detail the two-fingered hand as well as the objects that were manipulated by the hand. Next, we give an overview of the implementation of the hierarchical control structure and describe its software and hardware components.

3.1 Styx–Hardware

The control algorithms presented here have been implemented on a multi-fingered hand, known as Styx, that was designed and built to facilitate implementation and testing of control algorithms for multi-fingered hands [15]. Styx is a two-fingered, planar hand, with each finger consisting of two revolute joints and two links. The distal links are capped by small rubber cylinders that serve as fingertips and as contact "points" between the fingers and the object that is to be manipulated. A diagram of Styx is shown in Figure 3.1.

The motors used to drive Styx are direct-drive DC motors mounted at the base of each link and are driven with a pulse-width modulated 20 kHz square wave. Each motor contains a quadrature encoder used to sense joint position. The resolution for the proximal motors is 3600 counts per revolution and for the distal motors 2000 counts per revolution. Styx is connected to an IBM PC/AT running at 6 MHz with an 8087 floating point coprocessor. The motors and encoders are interfaced to the AT using a set of four HP HCTL-1000 motion control chips interfaced to the AT bus. A view of the interconnection of the hardware supporting the Styx system is shown in

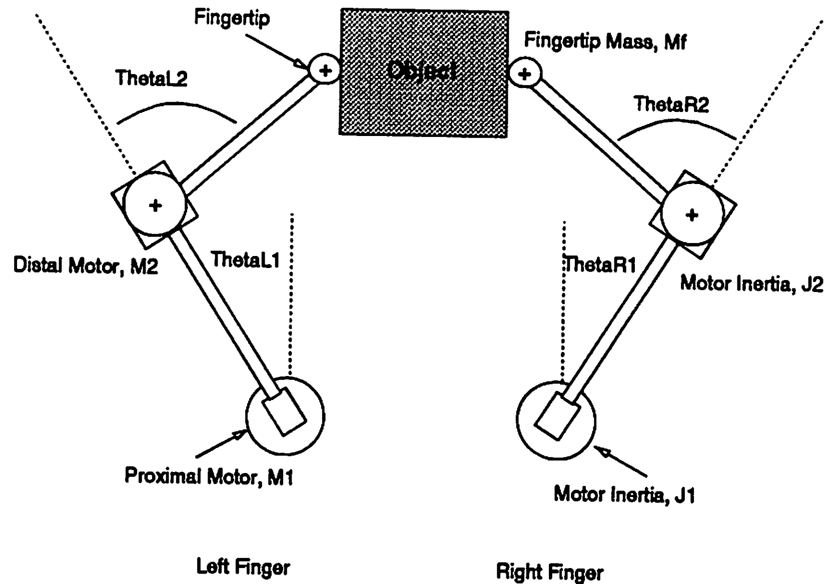

Figure 3.1: Top View of Styx

Figure 3.2.

The parameters associated with Styx kinematics and dynamics are shown in Table 3.1.

Assumptions made to simplify implementation of the control algorithms presented here include:

1. Motor dynamics can be ignored—for small velocities, the torque generated by each motor is proportional to the input pulse width.

2. Fingertips can be modeled as fixed point contacts—in order to avoid the complexity associated with implementing a model of rolling contact dynamics, the fingertips were modeled as simple point contacts. As a result, the shifting of the contact points on the object was unmodeled. However, in the commanded trajectories including zero orientation of the object, the effect of this shift was minimized.

3. The Coriolis and frictional forces are ignored—for trajectories resembling the slower movements tested here, these forces have been shown to be negligible [15]. More extensive testing regarding the relative significance of the Coriolis and frictional forces in the fast trajectories is required.

Parameter	Symbol	Value	Unit
Link Lengths	L_1, R_1	15.3	cm
	L_2	12.16	cm
	R_2	11.8	cm
Fingertip Radius	r_f	1.7	cm
Base Separation	B	20.0	cm
Link Mass	M_{L1}, M_{R1}	53	g
	M_{L2}	17	g
	M_{R2}	20	g
Distal Motor Mass	M_2	328	g
Fingertip Mass	M_f	3	g
Motor Inertia	J_1	18	g cm^2
	J_2	1.74	g cm^2

Table 3.1: Styx Parameters

3.2 Control Hierarchy

In the discussion of the tracking performance of the hierarchical structure we assumed the existence of two continuous time controllers, a high level PD plus feedforward controller in object coordinates as well as a low level PD controller in joint angle coordinates. Here we describe in more detail the implementation details of the hierarchical structure, including the points of departure of the implementation from the theory.

In order to implement the two-level hierarchical structure, we actually used the three-level structure shown in Figure 3.3. The upper two levels, consisting of a primary and a secondary control loop, are written in the C programming language, using the Microsoft 5.1 Optimizing C compiler. An assembly language scheduler controls the sample rates of the control loops. In order to more closely relate to each other the hierarchical control structures shown in Figures 3.3 and 2.1 we chose the sampling periods of the low-level and high-level control loops to be roughly equivalent to the time delays present in what are postulated in the biological system to be spinal reflex loops and cortical feedback loops, respectively.

At the top of the figure we see the highest control level, the secondary control loop running at 10 Hz. At this level we calculate the inverse kinematics of the object's desired trajectory (X_d in Figure 3.3) and perform the high level control functions when we put Styx into the hierarchical control

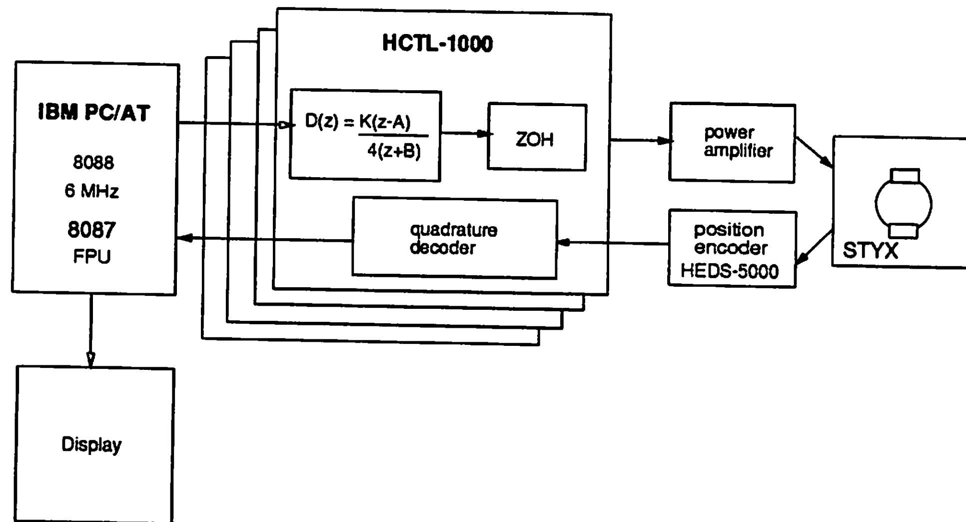

Figure 3.2: Hardware Supporting Styx

mode. The high level software can be made to implement any desired control algorithm that is to be superimposed upon the lower levels. Here we present results of the high level PD + feedforward controller in object coordinates only.

The lower level of the control hierarchy consists of the low level software block shown in Figure 3.3 in addition to the hardware block below it. The lower software level is implemented by the primary control loop. The purpose of the primary control loop is to write at a frequency of 100 Hz directly to the motion control hardware the current commanded joint angles received from the secondary loop. In addition, the low level controller is capable of joint angle interpolation, calculating a new, updated commanded joint position by adding incrementally the joint velocity multiplied by a time step based on the ratio of the primary and secondary control frequencies. This interpolation allows the low-level controller to gradually command the joint position to move from the position commanded by the high-level controller at one time step to the position commanded at the following time step.

At the lowest level of the control hierarchy exists the HP HCTL-1000, a digitally sampled, general purpose motor controller. Although we assumed in the stability analysis that the lower control level operated in continuous time, we used the HCTL-1000 in the implementation, because it had already been built into the existing hardware. In particular, the HCTL-1000

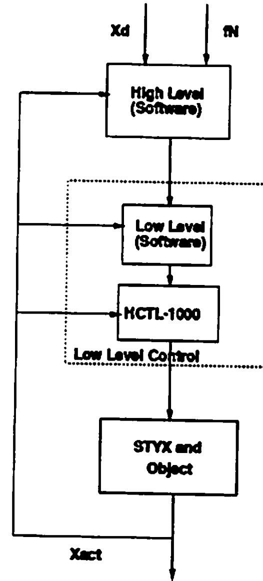

Figure 3.3: Control hierarchy for Styx, showing control of the desired and actual object trajectories, X_d and X_{act}, respectively, and the specified internal force, f_N.

was used in its position control capacity only, because the fast response relative to the software control loops allowed us to make comparisons with the human control system and its multi-level structure. In the analysis, our assumption was that the digital sampling of the controller was fast enough so that we could approximate it by a continuous time PD controller. The actual sampling frequency of the HCTL was 1.9 kHz.

Programmable variables in the HCTL's position control include the sampling time, T, as well as the parameters associated with the digital filter used to compensate for closed loop system stability:

$$D(z) = \frac{K(z - A/256)}{4(z + B/256)} \tag{3.1}$$

The position control mode performs point to point position moves. A position command is specified, which the controller compares with the actual position, calculating the position error. The full digital compensation is applied to the position error, and the calculated motor command is output until the position error changes or a new position command is given.

Since the HCTL-1000 was used in its position control mode, we did not directly send desired torque commands to the HCTL controller. Although the chip's position control is entirely adequate for specifying a desired trajectory with no high level control, a problem arose when we wished to apply a certain internal force, as, by definition, the internal force evokes no change in the system's position variables. We could not induce the motors to apply the torques required for the specified internal force by specifying the torques directly. Instead, we calculated a virtual position error which, when multiplied by the DC gain of the position controller, yielded the desired joint torques. The same difficulty arose when we wished to specify additional torques based on correctional terms calculated by the high level controller. Instead of adding in the torques directly, we calculated a "correctional" position error, by assuming that the trajectory was slow enough relative to the time constants associated with the HCTL controller that we can use the DC approximation in the torque to position conversion with some reasonable degree of accuracy.

The control system was tested with each of three different progressively more complicated control schemes, each building on the one(s) before: setpoint control without joint interpolation, setpoint control with joint interpolation, and hierarchical control (including setpoint control with joint interpolation at the lower level).

Chapter 4

Experimental Results

4.1 Presentation of Results

In the experimental results presented here, we used two different objects, each suited to meet the needs of the particular trajectory being tested. In the first set of trajectories, we used a metal bar that was loosely attached to the fingertips via pin joints. Mounted on top of the bar was a heavy mass, which the fingers were required to move. The "object" thus consisted of both the beam and the mass attached to it; the parameters associated with this object are shown in the column labeled "beam" in Table 4.1. The second object used in the experiments was a cardboard box, the dimensions of which are also shown in Table 4.1, in the column labeled "box".

The HCTL-1000 parameters used in the generation of the figures shown were the chips' default parameters, listed in Table 4.2.

The commanded trajectories for the objects' centers of mass were circular trajectories of radius 2.5 cm, centered at x = 1.3 cm, y = 21.2 cm relative to the midpoint between the two proximal motors, as shown in Figure 3.1,

	Beam	Box	
Mass	245	33	g
Moment of Inertia	1.8×10^3	1.3×10^3	g cm^2
Length	12	17	cm

Table 4.1: Object Parameters

Gain, K_d	64
Zero, A	229
Pole, B	64
Sample Freq, 1/T	1.9 kHz

Table 4.2: HCTL-1000 Parameters

with frequencies of 1.0 Hz for the "beam" and 0.25 Hz for the "box". The orientation of the objects was to remain at zero throughout each movement. The tracking performance along these trajectories was tested for consistency between trials. We determined that collecting data for a period of 20 seconds was entirely sufficient for our purposes, resulting in series of 5 trials each for the slow trajectories and 20 trials each for the fast motions. Additional trials merely served to duplicate the results.

Although we are interested primarily in grasping control rather than coordinated robot control, we used the beam in the rapid movements because of hardware limitations: In the slow movements we applied an internal force of 3×10^4 dyne. The maximum motor torque, however, was limited to the extent that we were not able to exert the large internal forces required to grasp and move a heavy object during rapid movement. Thus, we used an object that we could loosely attach to the fingertips, enabling us to use smaller internal forces (3×10^3 dyne) without losing contact with the object.

Figures 4.1-4.5 depict the performance of the three different controllers, the setpoint controller (non-hierarchical), the setpoint controller with low-level joint velocity interpolation, and the hierarchical controller. The setpoint controller consisted of a high level piece that existed solely to calculate the inverse kinematics, directing the low-level controller to a new setpoint at a frequency of 10 Hz. Position control was carried out only at the fast, low level by the HCTL-1000 and was done strictly in joint angular coordinates. The rather poor performance of this controller can be judged quickly by examining Figure 4.1(a), which shows the actual trajectory of the object's center of mass. A large overshoot, primarily in the horizontal, x, direction is evident.

The second controller that was used, the joint interpolation controller, was non-hierarchical as well, as the high level software, again, existed merely to solve the inverse kinematics, with the additional control piece, the joint velocity interpolation, added only at the low level, as described above. Al-

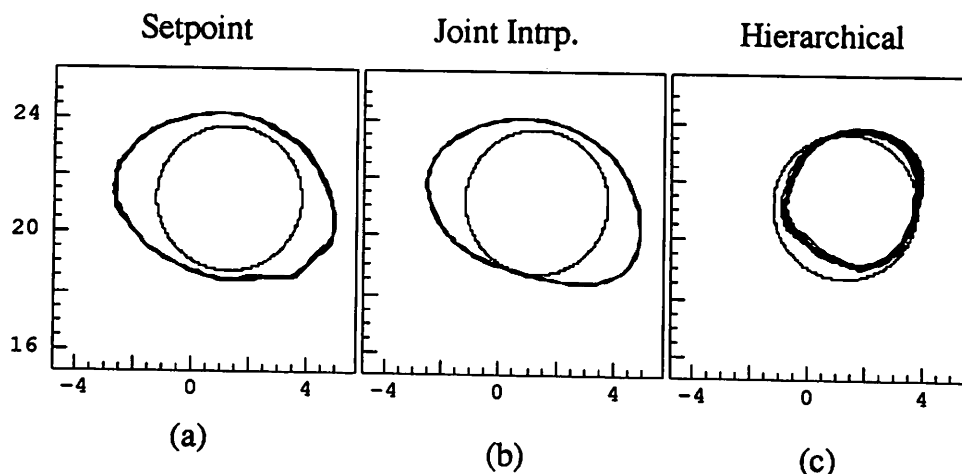

Figure 4.1: Actual and commanded object trajectories with setpoint controller (a), joint interpolation controller (b), and hierarchical controller (c) in Cartesian space with weighted beam; commanded trajectories are shown within each graph (fast circular trajectory).

though the trajectory shown in Figure 4.1(b), was found to be smoothed out more, we found the same overshooting of the goal trajectory that occurred in Figure 4.1(a). Further efforts aimed at improving overall performance by introducing more and more complicated versions of a single-level, fast controller seemed unwarranted. The problem that remained to be addressed was the object's mass, a parameter that was not and could not without undue complications be taken into account at the lower level.

A significant improvement in the trajectory tracking performance was found when a high-level controller that corrected for the object's tracking errors was superimposed upon the existing low-level control structure. By calculating the torques required to bring the object back to the desired trajectory, the high-level controller was able to compensate for and minimize the trajectory errors in the object's (Cartesian) coordinates, as shown in Figure 4.1(c). The overshoot that so grossly disfigured the trajectories of the two non-hierarchical controllers disappeared completely, resulting in a much better overall tracking performance.

In Figure 4.2 we show the calculated box position error, for the setpoint controller (a), the setpoint controller with interpolation (b), and the hierarchical controller (c) for the same trajectories that were depicted in

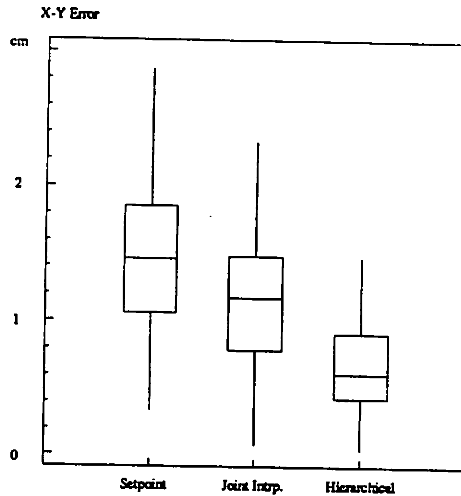

Figure 4.2: Deviation (in cm) from commanded trajectories of actual object trajectories with setpoint controller, joint-interpolation controller, and hierarchical controller when using weighted beam (fast circular trajectory).

Figure 4.1. The error is given by $\sqrt{(X_{act} - X_{des})^2 + (Y_{act} - Y_{des})^2}$ for each controller. Again, the insignificant improvement afforded by the addition of the joint interpolation to the setpoint controller, and the marked improvement of the performance of the hierarchical controller over the performance of both single-level controllers are evident.

In the figures showing data in "box-plot" form (Figures 4.2-4.8), the top and bottom of the boxes correspond to the twenty-fifth and the seventy-fifth percentiles of the given variables, while the horizontal lines through the boxes correspond to the median values of the variables. The vertical lines have ends that extend beyond the quartiles by a distance equal to one and one-half times the inter-quartile range. Approximately ninety-nine percent of normally distributed data are within the range covered by the vertical lines; outliers are identified by an asterisk, '*'.

In order to rule out the possibility that the improvement in performance of the hierarchical controller when compared with that of the single level controllers was an anomaly related to the particular circular trajectory chosen, we tested the same controllers on two additional trajectories, a figure-8, generated by specifying sinusoidal motions of different frequencies in the x and y directions, and a square "box" trajectory, generated by specifying equal amplitude, phase shifted square waves in the x and y directions. The

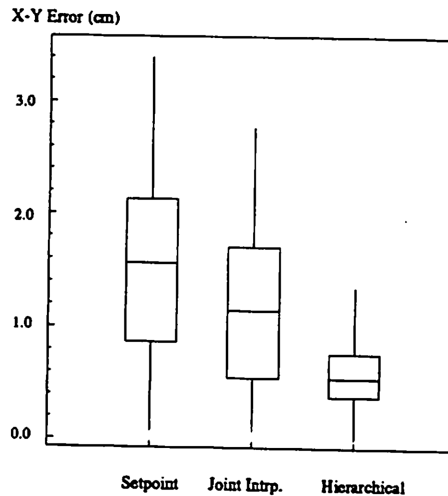

Figure 4.3: Deviation (in cm) from commanded trajectories of actual object trajectories with setpoint controller, joint-interpolation controller, and hierarchical controller (figure-8 trajectory); experimental details are summarized in the appendix.

experimental details of these trajectories are summarized in the appendix. A brief look at the box plots of the object's x-y error shown in figure 4.3 provides evidence that it is again the hierarchical controller which minimizes this error. On the other hand, with the square box trajectory we see in figure 4.4 a much less desirable level of performance in both the single level controllers and the hierarchical controller, not a surprising result, considering the infinitely fast specified trajectory at each square wave rise and fall. The relatively significant overshoot and underdamping seen in each trajectory are currently assumed to be more related to sub-optimal choices for the HCTL-1000 control parameters than to the particular control algorithm. A conclusive statement about the relative performance of single level and hierarchical controllers in this type of trajectory is, therefore, not possible at this time. However, the performance of the hierarchical controller appears to be no worse than that of the simpler algorithms.

The improvement in the object's position tracking that we found when using the hierarchical controller was, unfortunately, not mirrored in the object's orientation, as can be seen in Figure 4.5. A similar result was observed in the figure-8 trajectory described earlier (results not shown). One possible cause is a poor approximation of the moment of inertia of the object.

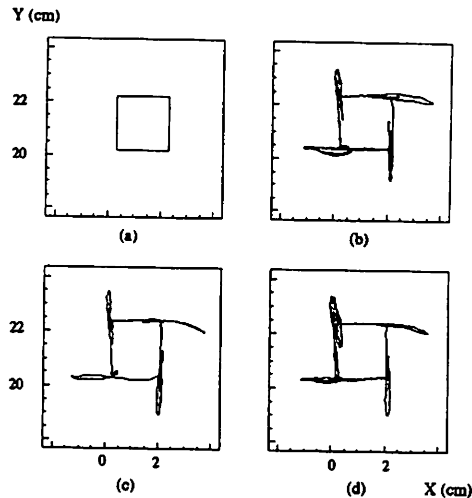

Figure 4.4: Commanded "square box" object trajectory (a), actual trajectory with setpoint controller (b), with joint interpolation controller (c), and with hierarchical controller (d); experimental details are summarized in the appendix

Due to the configuration of the system, a very slight shift in the fingertip position can cause a relatively large orientation error in the object. Thus, any error in the calculation of the object's moment of inertia can cause an error in the orientation portion of the feedback control, which, in turn, can cause a significant orientation error in the object. We expect that the same controller, when furnished with a more accurate measure of the object's moment of inertia, will show an improvement in the orientation error similar to that found in the position error. Further tests with more sophisticated measurements of the object's moment of inertia are expected to confirm this prediction.

In order to gain a better understanding of the modeling errors associated with the object's orientation we tested the three controllers on a trajectory involving only a twisting motion of the object with no translation. That is, the specified x and y trajectories of the object's center of mass were identically equal to zero, while the orientation was varied sinusoidally. The resulting trajectories for the joint interpolation single-level controller and the hierarchical controller are shown in figure 4.6. Evidence for modeling errors is readily apparent in both trajectories: when using the joint interpolation controller, the object exhibited a significant overshoot in both directions as

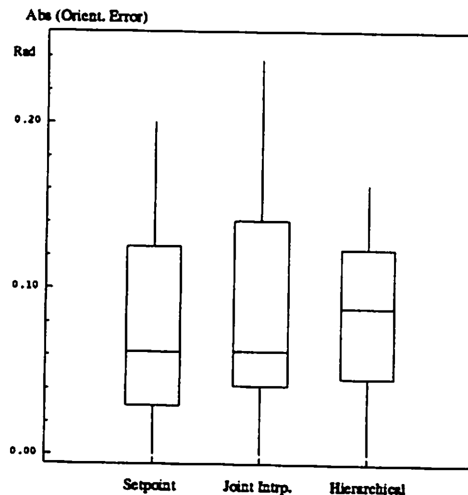

Figure 4.5: Deviation (in radians) from commanded object orientation of actual object orientation with setpoint controller, joint interpolation controller, and hierarchical controller from commanded trajectories when using weighted beam (fast circular trajectory).

well as phase shift and a periodic distortion throughout the trajectory; on the other hand, while the hierarchical controller caused the object to have a smaller phase shift and to follow a sinusoidal motion of the appropriate amplitude, the object trajectory exhibited a 0.2 radian DC phase shift. In figure 4.7 the relative magnitudes of the total applied forces and the inertial forces on the object may be observed. The data shown in the figure are calculated using data from the particular experiment using the joint interpolation controller. In this trajectory, the inertial forces only make up a small portion of the total forces applied to the object; the Coriolis/centrifugal, frictional, and other unmodeled forces are, therefore, significant and may be a large contributing factor in the orientation errors. Similar calculations (not shown) based on trajectories involving time-invariant orientations, such as the circles and figure-8 trajectories discussed above, show a much smaller, although not insignificant difference between the inertial forces and the total applied forces. The same calculations also show, however, a non-zero orientation component of the applied force. Based on these force calculations, we can conclude that both the dynamic model of the hand itself and the controllers still incorporate significant errors with respect to the object's orientation calculations, and improved versions of the models will be re-

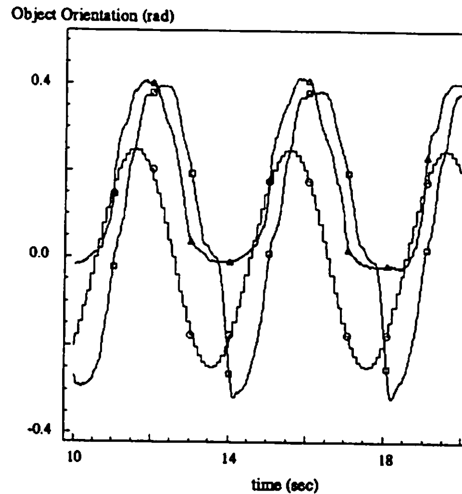

Figure 4.6: Commanded object orientation (labeled with circles) and actual object orientation with joint interpolation controller (squares) and with hierarchical controller (triangles) in the "twisting" trajectory; experimental details are summarized in the appendix

quired if a significant improvement in the tracking of the specified object orientation is desired.

As discussed above, the reasons for using the beam in the controller comparisons were to enable us to track high-speed trajectories with heavy objects without saturating the motor output. However, we wish to point out that the system is, in fact, capable of standard "grasping" manipulation. To this end, we show in Figure 4.8 the object's position, the orientation error as a function of time, and the position error when the three controllers were used to manipulate the lighter object ("box") in tracking the slow (0.25 Hz) circular trajectory. As expected, at slow speeds and when using small object masses, the differences between the single-level controllers, which did not take into account the object's parameters, and the hierarchical controller became insignificant. The position control used in the setpoint controller appeared to be entirely adequate when the object was commanded to move slowly enough for the controller to reach each specified position before receiving the next position command.

The commanded internal force exerted on the object by the hand was kept at a constant value, 3×10^4 dyne, in all of the slow trajectory experiments shown here. We currently have not measured the actual internal force

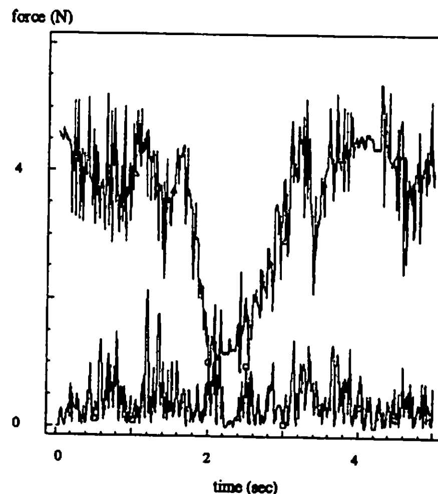

Figure 4.7: Magnitude of the total force applied to the object at the fingertips (labeled with triangles) and the inertial forces (squares) as determined by the joint angle trajectory data in the the "twisting" trajectory; experimental details are summarized in the appendix

on the object; hence, this quantity is not a controlled variable at this time. The value of the constant internal force was chosen, because it seemed an adequate compromise between having enough internal force to hold the object throughout the movement with a minimum of slipping and not having so much internal force that the grasp became unstable. This potential instability due to a large "internal force" was a real problem that needed to be addressed: the grasp and Jacobian matrices, used in the internal force calculation, were updated at the relatively slow speed of the high-level control loop. The calculated "internal" force, therefore, contained a small, at times significant portion that lay outside of the grasp matrix's null space. Similarly, the torque calculated to produce the internal force differed slightly, at times significantly, from the "true" value of the torque that would have been required to produce an internal force. A possible cause of this difference may be traced to differences between the Jacobian matrix that was calculated in the slow control loop and the "true" current value of the Jacobian. An even more significant contributing factor, perhaps, was the amplifier gain "drift" over time, which, when combined with the slight kinematic differences between the two fingers, made precise calibration of the system an extremely difficult task, and one that would need to be repeated throughout the life of

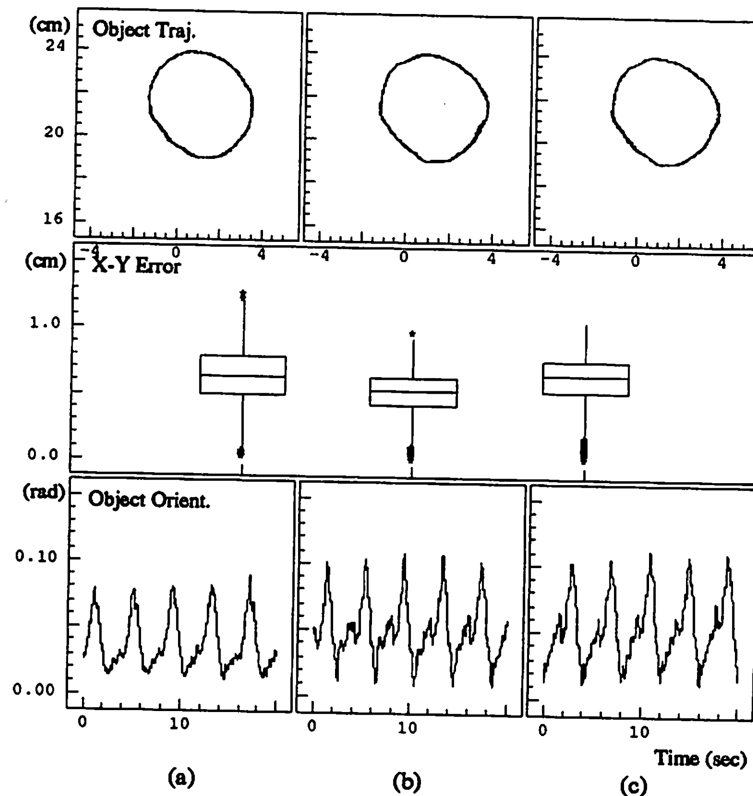

Figure 4.8: Object trajectory, box plot of object position error, and object orientation error as a function of time for setpoint controller (a), setpoint controller with joint interpolation (b), and hierarchical controller (c) (slow circular trajectory).

the robot components. We stress this difficulty in calibration, because it goes beyond the need to precisely measure the kinematic parameters associated with the robotic hand and the object. In general, the higher the desired internal force, the higher the errors associated with applying the proper torques. We found that relatively high commanded internal forces interfered significantly with proper tracking behavior, while even higher forces caused the grasp to become unstable.

A brief exploration of the effect of different levels of internal force is shown to furnish the reader with some understanding of the magnitude of this effect. The commanded trajectory was the slow circle described above. Figure 4.9 shows the increase in the tracking error accompanying an increase

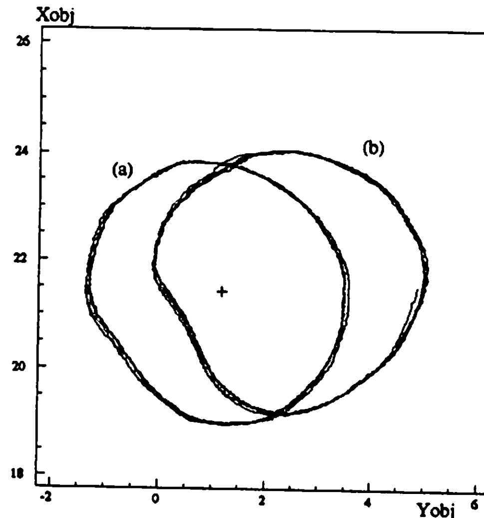

Figure 4.9: Effect of increased specified internal force on trajectory errors using the setpoint controller; trajectories (a) and (b) were followed with a specified internal force of 3×10^4 and 5×10^4 dyne, respectively; the plus sign ("+") marks the center of the commanded circular trajectory.

in the specified internal force from 3×10^4 dyne to 5×10^4 dyne. The primary cause of the shift in the trajectory appears to have been the calibration difficulty discussed above. While the figure shows the data for the non-hierarchical setpoint controller only, the results were very similar when the other controllers were tested.

4.2 Discussion of Results

The data presented here show a noticeable improvement in the performance of the hierarchical controller as compared with that of the single-level controllers when the controllers were used in fast movements of a heavy object. The most basic controller tested, the setpoint controller, was a simple, fast, single-level controller operating only on the joint position variables. The performance of this controller, even when operating at high frequencies relative to the trajectory frequencies, was clearly inadequate, demonstrating a large overshoot in the positioning of the object throughout much of the trajectory. In an attempt to improve the performance of the single-level control scheme, we introduced an additional level of complexity at the joint level by calculating the joint angular velocity and implementing an interpolation scheme

that was designed to smooth the commanded trajectory between adjacent points as specified by the high level routines calculating the inverse kinematics. Although some smoothing was observed in the object's trajectory when the joint interpolation scheme was used, there was no significant change in the tracking performance of the system. Evidently, adding some degree of complexity to the existing single-level controller was not sufficient to overcome the tracking errors of the setpoint controller. We surmise that the poor tracking performance was a result of neglecting the object's dynamics in the overall control scheme. The speed at which the single-level controllers were operating, however, placed strict limits on the amount of computation that could be completed within the control loop, thereby effectively excluding the possibility of moving the computation of the dynamics of the entire system, including both the robot hand and the object, into the lower level. We add here that, in addition to testing the fast, simple single-level controller, we implemented a more complicated single-level controller that incorporated both the object and the hand dynamics and that was run at the slow speed of the high level in the hierarchical scheme. The result, when the controller was tested in the fast movement experiments, was instability to the point where collecting data became entirely unnecessary. Clearly, a scheme making use of the best of both controller complexity and controller speed was required.

A solution to the computational problem associated with manipulating a heavy object at high speeds was the hierarchical control scheme presented here. The hierarchical scheme allowed us to incorporate in a single control scheme the advantages of fast control loops as well as the complexity required to adequately model the system dynamics. As expected, incorporating the object's dynamics in the control scheme became more important as both the object's mass and its acceleration along the trajectory increased. Our conclusion regarding the need for hierarchical control in certain movements was supported by the relative similarity in the performance of the three controllers when operating on light moving at slow speeds. With an increase in mass and movement speed, differences between the controllers became more apparent.

We exploited the system's similarity when using the computed torque and the PD plus feedforward controllers and based the particular choice of the high-level controller parameters on the second-order linear error dynamics resulting from applying a computed torque controller. This analysis enabled us to select the cutoff frequency and the damping factor for the system. In order to obtain a roll-off above the trajectory frequencies and a damping ratio of 0.5, we chose the high-level control gains to be $K_v = 2$ and $K_p = 4$,

with acceptable performance.

A subject for further inquiry is the choice of the control parameters at the low level of the hierarchy. The difficulties encountered in this choice were the limitations of the existing hardware and a lack of understanding of how and to what extent, the choices made at one level of the hierarchy were affected by the choices made at higher or lower levels. At this time, our choice of the HCTL-1000 parameters was based upon the assumption of complete separability of the two control levels, which allowed us to analyze the closed loop performance of only the low level system. The parameter values resulting from this analysis were then subjected to experimental testing to determine the validity of the assumptions, and a final choice was then made, based upon the performance of different combinations of parameters. However, we stress that no extensive testing was performed to determine the *best* choice. What is required is a more rigorous approach, including an analysis of the extent to which the adjacent levels of control interact, leading to an analytical solution and an experimental confirmation of the results.

In addition, a more detailed exploration of the problems associated with the object orientation is in order. A possible area for further research may be the effect of including rolling contact dynamics in the real time model of the hand-object system. As stated, the model implies fixed point contacts between the fingertips and the object, not necessarily a valid assumption, especially for orientation-varying trajectories. The rolling contact dynamics, however, result in extremely complicated dynamic equations, the derivation of which is sketched in the appendix. Given the existing hardware available here at this time, the computation of these complex dynamic relations is not feasible in the time allowed by the controller frequencies.

The results discussed here represent an experimental confirmation of the predicted stability of the hierarchical control structure under the conditions given. In particular, the assumptions regarding the separability of the choice of control parameters as well as the time scales of adjacent control structures have been validated experimentally for the planar, two-fingered system used here. Furthermore, the validity of the assumption that the frequencies inherent in the trajectories were low enough to merit the steady-state approximation of the HCTL-1000 gain was confirmed. What remains to be gained is a better understanding of what constitutes adequate limits on the ratios of adjacent time scales and how these limits relate to the trajectories of interest.

Chapter 5

Conclusion

Based upon the experiments performed on Styx thus far, the most effective control scheme in fast movement of heavy objects was the hierarchical control scheme. Hierarchical control schemes have the advantage of being able to run simple, lower levels at high speeds, thus rapidly correcting for tracking errors in fast movements, while running at lower speeds more complicated higher levels that improve the overall performance by incorporating system dynamics far removed from the low level actuators. Although the experimental results presented here are based solely upon work done with a simple, planar system, the advantages of using a hierarchical control scheme can easily be applied to more complicated system, as we, in the implementation of the control structure, in no way made use of simplifying assumptions based upon the simplicity of the system. In fact, the differences between the performance of the hierarchical and single-level controllers ought to become greater as the system complexity is increased and the computational complexity of the higher level dynamics becomes greater.

An example of an extraordinarily complicated grasping control system is the human motor control system and its feedback pathways from the skin surrounding and the muscles controlling the fingers to the spinal cord and to various parts of the brain. In a study of two-handed grasping control in humans Reinkensmeyer [22] suggests the use of a simple control structure that takes advantage of the spring-like properties of muscle and of the similarity between the dynamics of the single robot hand (in our case "finger") and the robot hand-object system. In the work presented here, there are no such simplifying properties of the actuators. However, by configuring the system in such a way as to resemble the relative feedback delays

of the human motor control system (20 – 30 ms for the single-reflex spinal feedback loop and up to 200 ms and more for the highest-level, voluntary control feedback loops [14, 23]), we were able to examine some connections between the two systems that merit further exploration. In both systems the structure of the lower levels enables the control algorithms at the higher levels to make simplifying assumptions about the systems that lie beneath them, thereby allowing the higher levels to control on a more abstract level the entire movement, leaving the lower levels free to implement rapidly the details of locally interfacing directly with the actuators. We hope to gain further understanding of possible and plausible control structures in both systems by exploring the parallels as well as the differences between them.

There are many areas in the design, analysis, and testing of hierarchical control algorithms pertaining to grasping that need to be investigated more fully. Lacking at this time is a rigorous theory of the interconnection of the various levels of grasping control as well as a theory of the conditions under which we can expect stability of the overall grasping control scheme. The need for more sophistication in the development of approaches to the choice of system parameters is clear. With increased sophistication, the potential for improved performance at a lower computational cost than is associated with single-level control is, in our opinion, undeniable. On the experimental side, we would like to see an implementation of other controllers, such as a computed torque controller, in the higher level of the hierarchical structure. Furthermore, extensive testing of rapid movements with high, possibly time-varying, internal forces would afford greater understanding of grasping control and might enable us to draw more conclusions about the biological as well as the roboticist's solution to the problem of controlling rapid grasping movements. In turn, the introduction of varying internal forces, especially in rapid movements, may act to compound the problem of the object slipping within the fingers' grasp and may, therefore, precipitate the implementation of models that include the dynamics of rolling contacts.

Our comparison of hierarchical and single-level control schemes has at once provided an indication of the advantages of using hierarchical control algorithms in grasping control and introduced a variety of open research questions relating to the theory and the application of hierarchical control in multi-fingered grasping situations.

Acknowledgements

Support for this research by the National Science Foundation under grant DMC-84-51129 and under grant ECS-87-19298 and by the U.S. Department of Health and Human Services under grant PHS GM07379-14 is gratefully acknowledged. Dr. D. C. Deno, of the Robotics Laboratory at the University of California at Berkeley, is thanked for the use of his figure of the proposed hierarchical control scheme in the human hand. In addition, the author would like to thank Dr. S. Lehman, of the Bioengineering Graduate Group, and Dr. S. S. Sastry, of the Department of Electrical Engineering, both at the University of California at Berkeley, for helpful comments relating to the work presented here. Finally, special thanks go to Dr. R. M. Murray for innumerable helpful discussions as well as inspiration and motivation both during his tenure as a graduate student in the Robotics Laboratory here in Berkeley and after his departure from the University of California to join the faculty in the Department of Mechanical Engineering at the California Institute of Technology.

Bibliography

[1] C. O. Alford and S. M Belyen. Coordinated control of two robot arms. In *International Conference on Robotics*, pages 468–473, 1984.

[2] S. Arimoto, F. Miyazaki, and S. Kawamura. Cooperative motion control of multiple robot arms or fingers. In *IEEE International Conference on Robotics and Automation*, pages 1407–1412, 1987.

[3] D. Clark. Hic: An operating system for hierarchies of servo loops. In *IEEE International Conference on Robotics and Automation*, pages 1004–1008, 1989.

[4] A. B. A. Cole, J. E. Hauser, and S. S. Sastry. Kinematics and control of multifingered hands with rolling contact. *IEEE Transactions on Circuits and Systems*, 34(4):398–404, 1989.

[5] J. Demmel, G. Lafferrier, J Schwartz, and M. Sharir. Theoretical and experimental studies using a multifinger planar manipulator. *IEEE International Conference on Robotics and Automation*, pages 390–395, 1988.

[6] D. C. Deno, R. M. Murray, K. S. J. Pister, and S. S. Sastry. Control primitives for robot systems. In *IEEE International Conference on Robotics and Automation*, pages 1866–1871, 1990.

[7] D. C. Deno, R. M. Murray, K. S. J. Pister, and S. S. Sastry. Primitives for robot control. In M. A. Kaashoek, J. H. van Schuppen, and A. C. N. Ran, editors, *Proceedings of the 1989 International Symposium on the Mathematical Theory of Networks and systems (MTNS-89)*, pages 13–32. Birkhäuser, 1990.

[8] S. Jacobsen, J. Wood, K. Bigger, and E. Iverson. The Utah/MIT hand: Work in progress. *International Journal of Robotics and Control*, 4(3):221–250, 1986.

[9] D. Koditschek. Natural motion for robot arms. In *IEEE Control and Decision Conference*, pages 733–735, 1984.

[10] P.V. Kokotovic, H. K. Khalil, and J. O'Reilly. *Singular Perturbation Methods in Control: Analysis and Design.* Academic Press, New York, 1986.

[11] Kader Laroussi, Hooshang Hemami, and Ralph E. Goddard. Coordination of two planar robots in lifting. *IEEE Journal on Robotics and Automation*, 4(1):77–85, 1988.

[12] Z. Li, P. Hsu, and S. Sastry. On kinematics and control of multifingered hands. In *IEEE International Conference on Robotics and Automation*, pages 384–389, 1988.

[13] M. T. Mason and Jr. J. K. Salisbury. *Robot Hands and the Mechanics of Manipulation.* The MIT Press, Cambridge, Massachusetts, 1985.

[14] T. A. McMahon. *Muscles, Reflexes, and Locomotion.* Princeton University Press, Princeton, New Jersey, 1984.

[15] R. Murray and S. S. Sastry. Control experiments in planar manipulation and grasping. *International Conference on Robotics and Automation*, 1989.

[16] R. M. Murray. Experimental results in planar grasping. Master's thesis, University of California at Berkeley, 1988.

[17] R. M. Murray, D. C. Deno, K. S. J. Pister, and S. S. Sastry. Control primitives for robot systems. *IEEE Transactions on Systems, Man and Cybernetics*, 1992.

[18] R. M. Murray and S. S. Sastry. Grasping and manipulation using multifingered robot hands. In R. W. Brockett, editor, *Robotics: Proceedings of Symposia in Applied Mathematics, Volume 41*, pages 91–128. American Mathematical Society, 1990.

[19] Y. Nakamura, K. Nagai, and T. Yoshikawa. Mechanics of coordinative manipulation of multiple robot mechanisms. In *IEEE International Conference on Robotics and Automation*, pages 991–998, 1987.

[20] Van-Duc Nguyen. Constructing force-closure grasps. *International Journal of Robotics and Control*, 7(3):3–16, 1988.

[21] Jr. N.R. Sandell, P. Varaiya, M. Athans, and M.G. Safonov. Survey of decentralized control methods for large scale systems. *IEEE Transactions on Automatic Control*, AC-23:108–128, 1978.

[22] D. J. Reinkensmeyer. Human control of a simple two-hand grasp. Master's thesis, University of California at Berkeley, May 1991.

[23] J. C. Rothwell. *Control of Human Voluntary Movement.* Aspen Publishers, Inc., 1987.

[24] N. Sadegh. *Adaptive Control of Mechanical Manipulators: Stability and Robustness Analysis.* PhD thesis, Department of Mechanical Engineering, University of California, Berkeley, California, 1987.

[25] N. R. Sandell, Jr., P. Varaiya, M. Athans, and M. G. Safonov. Survey of decentralized control methods for large scale systems. *IEEE Transactions on Automatic Control*, AC-23:108–128, 1978.

[26] J. E. Slotine and W. Li. On the adaptive control of robot manipulators. *International Journal of Robotics and Control*, 6:49–59, 1987.

[27] M. W. Spong and M. Vidyasagar. *Robot Dynamics and Control.* John Wiley and Sons, New York, 1989.

[28] S. T. Venkataraman and T. E. Djaferis. Multivariable feedback control of the JPL/Stanford hand. In *IEEE International Conference on Robotics and Automation*, pages 77–82, 1987.

Appendix A

Summary of STYX Parameters

Summarized here are the STYX parameters associated with the experiments described in the text.

Controller	K_p	K_v	K_a	K_{int}
Setpoint	0	0	0	0
Joint Interpolation	0	0	0	1
PDFF	4	2	1	1

Table A.1: Controller Gains (low level controller frequency = 100 Hz; high level controller frequency = 10 Hz) for the Single-Level and Hierarchical Controllers. Note that in the hierarchical controller, the PDFF control algorithm operates at the high level, while the joint interpolation algorithm functions at the low level, as indicated by the non-zero K_{int} shown for the PDFF controller.

Link Lengths	L_1, R_1	15.3	cm
	L_2	12.16	cm
	R_2	11.8	cm
Fingertip Radius	r_f	1.7	cm
Base Separation	B	20.0	cm
Link Mass	M_{L1}, M_{R1}	53	g
	M_{L2}	17	g
	M_{R2}	20	g
Distal Motor Mass	M_2	328	g
Fingertip Mass	M_f	3	g
Motor Inertia	J_1	18	g cm^2
	J_2	1.74	g cm^2

Table A.2: Styx Kinematic and Dynamic Parameters

Gain, K_d	64	
Zero, A	229	
Pole, B	64	
Sample Freq, 1/T	1.9	kHz

Table A.3: HCTL-1000 Parameters; the parameter values shown here are the default parameters of the chips and were used throughout the experiments described in the text.

	Beam	Box	
Mass	245	33	g
Moment of Inertia	1.8×10^3	1.3×10^3	g cm^2
Length	12	17	cm

Table A.4: Object Parameters

Trajectory	Slow Circle	Fast Circle	Figure 8	Square	Twist
Offset (cm,rad)					
x	1.3	1.3	1.3	1.3	1.3
y	21.2	21.2	21.2	21.2	21.2
phi	0.0	0.0	0.0	0.0	0.0
Magnitude (cm,rad)					
x	2.5	2.5	2.5	1.0	0.00
y	2.5	2.5	2.5	1.0	0.00
phi	0.0	0.0	0.0	0.0	0.25
Phase (deg)					
x	0	0	0	0	0
y	90	90	90	90	90
phi	0	0	0	0	0
Frequency (Hz)					
x	0.25	1.0	1.0	0.1	0.25
y	0.25	1.0	0.5	0.1	0.25
phi	0.25	1.0	0.0	0.1	0.25

Table A.5: Specified Object Trajectories; note that in all cases, except in the slow circle, the object used was the so-called "beam"

Appendix B

STYX Dynamics with Rolling Contacts

Shown here is a condensed derivation of the rolling contact dynamics. Underlying the derivation are the assumptions (1) that we have point contacts between circular fingertips and a planar object and (2) that the object is large enough to prevent each contact point from reaching the edges of the object's current contact side. In the interests of brevity, where the derivation may be broken down into two symmetric halves, only the left half is shown.

B.1 Forward Kinematics

The forward kinematics relating the left fingertip position, (x_{ftip_L}, y_{ftip_L}), to the joint angles of the left finger are

$$\begin{aligned} x_{ftip_L} &= L_1 * sin(\theta_{L1}) + L_2 * sin(\theta_{L1} + \theta_{L2}) - B/2 \\ y_{ftip_L} &= L_1 * cos(\theta_{L1}) + L_2 * sin(\theta_{L1} + \theta_{L2}) - B/2 \end{aligned} \tag{B.1}$$

where L_1 and L_2 are the proximal and distal left link lengths, respectively, B is the distance between the finger bases, and θ_{L1} and θ_{L2} are the left proximal and distal joint angles, respectively.

B.2 Contact Coordinates

The following definitions relating to the contact angles, that is, the angles defined in the area surrounding the actual contact point on the fingertip and

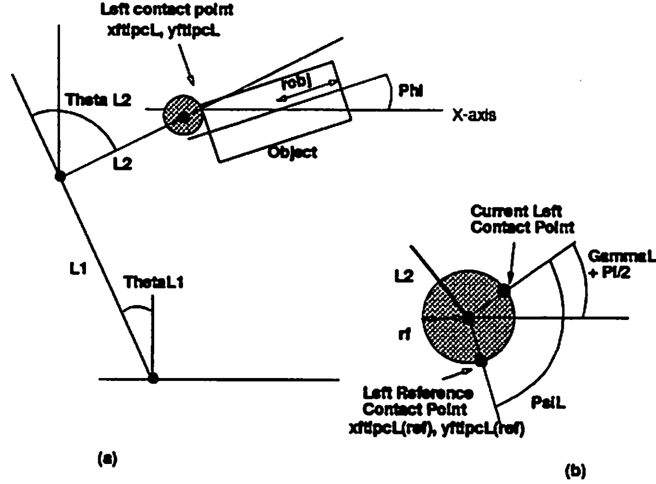

Figure B.1: Definitions of angles in area of fingertip-object contact; (a) the left finger in contact with the left side of the object; (b) enlarged view of the left fingertip area—shown for clarity in a different position than that shown in (a).

the object, are made and used below. Figure B.1 depicts the contact point area. Shown are the contact angles for the left fingertip and the left side of the object only.

$$\begin{aligned}
\delta_L &= \pi/2 - (\theta_{L2} - \theta_{L1}) \\
\psi_L &= \pi/2 - \delta_L = (\theta_{L2} - \theta_{L1}) \\
\gamma_L &= \phi + 2 * \theta_{L2} - \theta_{L1} - \theta_{L1}(ref)
\end{aligned} \tag{B.2}$$

where ϕ = the object orientation, as shown in Figure B.1, and $\theta_{L1}(ref)$ = the reference position of the left distal joint angle.

The contact positions are now expressed in terms of both the finger coordinate frame and the object coordinate frame. Accordingly, we have

$$\begin{aligned}
x_{ftip_{cL}} &= x_{ftip_L} + r_f * sin(\gamma_L) \\
y_{ftip_{cL}} &= y_{ftip_L} - r_f * cos(\gamma_L)
\end{aligned} \tag{B.3}$$

the x and y coordinates of the contact point as expressed in finger coordinates

and

$$
\begin{aligned}
x_{obj_{cL}} &= x_{obj} - r_{obj} * cos(\phi) - r_f * sin(\phi) \\
y_{obj_{cL}} &= y_{obj} - r_{obj} * cos(\phi) + r_f * sin(\phi)
\end{aligned}
\tag{B.4}
$$

the contact coordinates as expressed in object coordinates. Note that r_{obj} and r_f are defined as the object and fingertip radii (see Figure B.1).

B.3 Elimination of Object Coordinate Measurements

The difference between the left contact positions when calculated in the finger coordinates and when calculated in the object coordinates should equal zero. Thus, we can define H_L for the left finger

$$
H_L = \begin{bmatrix} x_{ftip_{cL}} - x_{obj_{cL}} \\ y_{ftip_{cL}} - y_{obj_{cL}} \end{bmatrix} = 0 \tag{B.5}
$$

and use the resulting equations to calculate the object position, x_{obj} and y_{obj}, as a function of the joint angles and the object orientation, ϕ. After following a parallel derivation at the right finger and contact location, we can also calculate the object's orientation as a function of the joint angles. We require this elimination of the object's state in real time control of the fingers, as the only information available to the controller is the state of the fingers themselves. Thus, we have

$$
\phi = tan^{-1}(a) - cos^{-1}(b) \tag{B.6}
$$

where

$$
\begin{aligned}
a &= \frac{(y_{ftip_R} - y_{ftip_L})}{(x_{ftip_R} - x_{ftip_L})} \\
b &= \frac{2 * (r_{obj} + r_f)}{[(y_{ftip_R} - y_{ftip_L})^2 + (x_{ftip_R} - x_{ftip_L})^2]^{1/2}}
\end{aligned}
\tag{B.7}
$$

B.4 Grasp Map with Rolling Contacts

The grasp map, G, is defined such that

$$
G(\Theta, X_{obj})_L = \frac{\partial H_L(\Theta, X_{obj})}{\partial X_{obj}} \tag{B.8}
$$

and

$$G(\Theta, X_{obj})_R = \frac{\partial H_R(\Theta, X_{obj})}{\partial X_{obj}} \tag{B.9}$$

where G_L and G_R represent the grasp maps of the left and right fingers, respectively, and Θ represents the vector of the proximal and distal joint angles.

By making the appropriate substitutions defined above for the object's position and orientation, we obtain left and right rolling contact grasp maps, $G_L(\Theta)$ and $G_R(\Theta)$, that are dependent only upon knowledge of the finger's joint angles.

B.5 Jacobian Matrix with Rolling Contacts

Similarly, we define the hand Jacobian matrix by calculating

$$J(\Theta, X_{obj})_L = -\frac{\partial H_L(\Theta, X_{obj})}{\partial \Theta} \tag{B.10}$$

and

$$J(\Theta, X_{obj})_R = -\frac{\partial H_R(\Theta, X_{obj})}{\partial \Theta} \tag{B.11}$$

As with the grasp map, we can then make the appropriate substitutions to obtain $J_L(\Theta)$ and $J_R(\Theta)$.

B.6 Dynamics

The grasp and the jacobian maps defined by

$$\begin{aligned} G(\Theta) &= \begin{bmatrix} G(\Theta)_R & 0 \\ 0 & G(\Theta)_L \end{bmatrix} \\ J(\Theta) &= \begin{bmatrix} J(\Theta)_R & 0 \\ 0 & J(\Theta)_L \end{bmatrix} \end{aligned} \tag{B.12}$$

may be used in the dynamic relations described in the main text, thereby including the rolling contact model in the dynamic description of the system. As noted in the text, the full rolling contact model has *not* been implemented at this time, due to the complicated nature of the grasp and jacobian maps when expressed in joint angles only.

Appendix C

STYX Source Code

What follows is a listing of the source code for the STYX program. Missing are some of the low level routines, including assembly language files containing the interrupt service handlers driving the HCTL's and some of the library functions for STYX, including primarily the machine-dependent I/O files. Several of the source files, most notably the input files, "include" files, and files defining tables of user interface variables (*.def, *.h, and *.tbl) have been consolidated in the interests of preserving space.

```
/****************************************************************** */
/*  File:    styx.c                                                 */
/*  main control program for STYX                                   */
/*                                                                  */
/* created:      RMM 16 March 1988                                  */
/* modified:     KHO 8 June 1991                                    */
/*                                                                  */
/****************************************************************** */

#include <stdio.h>                                                        10
#include <ctype.h>
#include <math.h>
#include <signal.h>
#include <process.h>
#include <styx/ddisp.h>
#include <styx/dinit.h>
#include <styx/hctl.h>
#include <styx/isr.h>
#include <styx/graf.h>
#include "styx.h"                                                         20

#include "global.h"                 /*  global variables     */
#include "stiff.h"                  /*  control module       */
#include "graph.h"                  /*  utilities            */
#include "dump.h"
#include "input.h"
#include "styx.tbl"                 /*  main level tables    */

/* Function declarations    */
DD_IDENT                                                                  30
  *menuloop();

/* Variables used in this file only       */
char
  errbuf[80];                   /* Error buffer for messages    */

/****************************************************************** */
main(argc, argv)                                                        main
     int
       argc;                                                              40
     char
       *argv[];
{
    FILE
      *fp;

    DD_IDENT
      *option,
      *menu = styx_menu,
      *control_menu = styx_menu;      register int                        50
      motor;

    signed
```

```
  pwm[HCTL_NMOTORS];

char
  filename[80];

int
  init = 1,                                                        60
  status;

/* Turn off the motors */
hctl_wrreg(0, HCTL_PWM, 0);
hctl_wrreg(1, HCTL_PWM, 0);
hctl_wrreg(2, HCTL_PWM, 0);
hctl_wrreg(3, HCTL_PWM, 0);

/* Initialize parameters (dinit)     */
printf("Reading setup data\n");                                    70
if (strcmp(argv[1], "-") != 0)
{
    char *file = argv[1];

    if (file == NULL) file = "styx.def";
    if ((fp = fopen(file, "r")) == NULL)
    {
        perror(file);
        exit(0);
    }                                                              80
}
else
{
    fp = stdin;
    printf("Enter parameters ('?' to list, '.' to exit)\n");
}
dinit(fp, inittbl);

/* Calculate lookup tables    */
printf("Calculating lookup tables\n");                             90
generate_tables();

/* Ignore ^C and math errors */
(void) signal(SIGINT, SIG_IGN);

/* 1 => Initialize the HCTL-1000 */
for (motor = 0; motor < STYX_NMOTORS; ++motor)
    hctl_wrreg(motor, HCTL_PC, 1);

/* Start the interrupt routine     */                              100
isr_set_cfreq(cfreq);
isr_set_tfreq(tfreq);
isr_set_croutine(NULL);          /* don't actually do anything yet        */
isr_set_troutine(NULL);          /* just get the intrpt handler going     */

/* Initialize the screen    */
```

```
    ddopen();

    while (1)
    {                                                                   110
        if ((option = menuloop(menu, init)) == NULL)
            /* Don't let anyone out unless they quit    */
            continue;

        /* Parse the selected option    */
        switch ((int) option->current)
        {
        case Quit:
        case 17:
            goto done;                                                  120

        case Graph:
        case 'g':
            graph();
            init = 1;
            break;

        case Capture:
            capture();
            init = 0;                                                   130
            break;

        case Dump:
            dump();
            init = 0;
            break;

        case Setpoint:
        case Circle:
        case Box:                                                       140
        case Figure8:
        case Periodic:
            input_setup((int) option->current);
            init = 0;
            break;

        case ControlRet:
            menu = control_menu;
            init = 1;
            break;                                                      150

        case Stiff:
            /* use the hierarchical/hctl position controller        */
            hctl_setup();
        case StiffRet:
            control_menu = menu = stiff_menu;
            init = 1;
            break;
```

```
        case Parm:                                                          160
            /* use input parameter menu; calculate new trajectory      */
            menu = input_menu;
            init = 1;
            break;

        case Watch:
            /* use watch menu (from table) */
            menu = menutbl[Watch];
            init = 1;
            break;                                                          170

        case InputParm:
        case 'i':
            ddread("Filename: ", filename, 80);
            if ((fp = fopen(filename, "r")) == NULL)
            {
                ddprompt("Can't open file for reading");
                init = 0;
            }
            else                                                            180
            {
                dinit(fp, inittbl);
                init = 1;
            }
            break;

        case OnOff:
        case 'o':        /* toggle the state of the controller      */
            control_on = !control_on;
            if (control_on)                                                 190
                hctl_restart();          /* restart the controller    */
            init = 0;
            break;

        case Cal:
            calibrate();
            while(kbhit())
                getch();                 /* empty keyboard buffer    */
            break;
                                                                            200
        default:
            ddprompt("Unknown option");
            init = 0;
            break;
        }       /* end of switch    */
    }           /* end of while     */

done:
    /* Clean up the display  */
    ddclose();                                                              210

    /* Turn off the interrupt routine (if on)      */
```

```
    if (isr_installed) isr_reset();

    /* 1 => Reset each HCTL-1000 */
    for (motor = 0; motor < HCTL_NMOTORS; ++motor)
        hctl_wrreg(motor, HCTL_PC, 1);

    exit(0);
}           /*  end of main()  */                                    220

/***************************************************************** */
/*  Move around in a menu and change values                         */
/***************************************************************** */
DD_IDENT *menuloop(d, init)                                     menuloop
    DD_IDENT *d;
    int init;
{
    if (init)
    {                                                                230
        /* clear the screen and initialize the menu      */
        ddcur = NULL;
        ddinit(d);
        ddcls();
        ddisp(d);
        ddselect(d);
    }

    while (1)
    {                                                                240
        DD_IDENT retopt;
        int option;

        /* update the display continuously     */
        while (!kbhit())
            ddisp(d);

        if ((option = getch()) == 0)
            /* Get extended keystroke    */
            option = -getch();                                       250

        /* clear the prompt line    */
        ddprompt("");

        switch (option)
        {
        case '\033':
            ddcls();
            return(NULL);
                                                                     260
        case -75:                   /* Left arrow    */
            ddleft(d);
            break;

        case -77:                   /* Right arrow   */
```

```
            ddright(d);
            break;

        case -72:               /* Up arrow    */
            ddup(d);                                                270
            break;

        case -80:               /* Down arrow */
            dddown(d);
            break;

        case '=':
            if (ddcur != NULL) (void) ddinput();
            ddisp(d);
            break;                                                  280

        case '\r':
            if (ddcur != NULL) return(ddcur);
            break;

        case 12:                /* ^L  */
            /* Refresh the screen   */
            ddrefresh(d);
            break;
                                                                    290
        default:
            retopt.current = (char *) option;
            return(&retopt);
#           ifdef UNUSED
            if (option > 0 && isprint(option))
                sprintf(errbuf, "Unknown option '%c'", option);
            else
                sprintf(errbuf, "Unknown keycode <%d>", option);
            ddprompt(errbuf);
            break;                                                  300
#           endif
        }           /*  end of switch   */
    }
    /*NOTREACHED */
}       /*  end of DDIDENT function */

#ifndef flag

/******************************************************************* */
/*  Write a value directly to the screen; write value to screen     */    310
/*  position indicated by offset; offset = 0--> position = row = 0,  */
/*  col = 0; offset = 2--> position = row = 0, col = 1; offset =     */
/*  odd values--> position = position indicated by offset minus 1,   */
/*  but now change attributes (e.g. color)    of that position,      */
/*  rather than writing something to the position                    */
/******************************************************************* */
flag(offset, val)
    int
```

flag

```
        offset, val;
{                                                                    320
    static long screen = 0xB8000000;

    *(char far *) (screen + offset) = val;
}
#endif

/******************************************************************* */
/*  prevent error messages from screwing up the program              */
/******************************************************************* */
matherr(){                                                           matherr
    ddprompt("math error");                                          331
}

/******************************************************************* */
```

```
/******************************************************************* */
/*  File:   stiff.c - for historical reasons ...                      */
/*  hier.c?   - hierarchical control for STYX                         */
              CT (high level) control and hctl-1000                   */
/*            position control                                        */
/*                                                                    */
/* created:   KHO 31 May 1991                                         */
/* modified:  KHO 14 September 1991                                   */
/*                                                                    */
/******************************************************************* */

#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <styx/hctl.h>
#include <styx/isr.h>
#include "styx.h"
#include "dynamics.h"
#include "hctl_loc.h"  /*  def'tns for hctl-1000 pos'tn control      */

/* Display tables   */
#include "stiff.tbl"

/*  define to move Jacobian and Internal Force calculations to HCTL
    loop; comment out the following line only to keep Jacobian and
    Internal Force calculations in the High Level loop        */
/*  #define FAST_UPDATE 1 */

/*  define motor gains   */
#define KM_MIN 21385      /*  gain for minertia motors       */
#define KM_HIT  5897      /*  gain for hitachi motors        */

extern int
    th1_2,          /*  distal joint angle for R finger; calc'd in kine.c          */
    th2_2;          /*  distal joint angle for L finger; calc'd in kine.c          */

extern float far lenR1_sin[], far lenR1_cos[],
             far lenR2_sin[], far lenR2_cos[];
extern float far lenL1_sin[], far lenL1_cos[],
             far lenL2_sin[], far lenL2_cos[];

extern double
    Force_Expec[3],
    Expec_Pos[4];           /* correcting position added to Command_Pos  */

int
    high_control(),         /*  secondary interrupt service routine        */
    hctl_control(),         /*  primary interrupt service routine          */
    high_freq,              /*  frequency of high level controller         */
    hctl_freq,              /*  frequency of interrupt service routine     */
    hctl_sample,            /*  value in HCTL_SAMPLE; set sample period  */
    NewtrajFlag,            /*  for double buffering trajectory            */
    Fake_Control,           /*  using fake test controller?                */
```

```
    control_law,          /* which high level controller?           */
    high_only;            /* HIGH_ONLY controller just turned on    */

long
    mag_force_N,          /* magnitude of grasping force (internal) */
    next_Pos[4],          /* Command_Pos at next time interval      */
    Command_Vel[4],       /* commanded finger joint velocity        */    60
    Command_Pos[4];       /* commanded finger pos, written to hctl  */

double
    ftip[4],              /* applied fingertip force                */
    Ctorque[4],           /* applied joint torques                  */
    b2r_cos_thR2,         /* B2R*cos(distal R joint angle)          */
    b2l_cos_thL2,         /* B2L*cos*distal L joint angle)          */
    last_x, last_y, last_t,  /* last actual object position         */
    IntF_Pos[4],          /* pos added to Command_Pos for int force */
    Grasp_Pseudo_Act[4][3],  /* pseudo inverse of grasp map         */    70
    force_N[4],           /* internal force, at fingertips          */
    torque_N[4],          /* torque to produce internal force       */
    Gamma[4],             /* DC gain of hctl controller             */
    R_jac_inv[2][2],      /* inverse jacobian matrix of R finger    */
    L_jac_inv[2][2],      /* inverse jacobian matrix of L finger    */
    R_jacob[2][2],        /* R finger jacobian matrix               */
    L_jacob[2][2],        /* L finger jacobian matrix               */
    Box_Act_Veloc[3],     /* actual object velocity                 */
    K_v,                  /* vel gain; detrmn how much vel interpol */
    Kv_int,               /* velocity interpolation?                */    80
    Mass_L[2][2],         /* inertia matrix of L finger             */
    Mass_R[2][2];         /* inertia matrix of R finger             */

/***************************************************************** */
/*  Initialize hctl control; function called when the user         */
/*  selects the stiff controller from the menu                     */
/***************************************************************** */
hctl_setup()                                                              hctl_setup
{
    register int                                                          90
        i;                              /* generic counter  */

    /****  Initialize Variables      *** */
    menutbl[Watch] = stiff_watch;       /* Set up menu links  */
    for (i=0; i<4; i++)
    {
        Gains[i] = 0;               /* set current gain to zero     */
        Command_Pos[i] = styx[i].reference;
    }
    /*  constant parts of 2x2, symmetric finger inertia matrices       */     100
    /*  R finger   */
    MR00 = A1R + B1R; MR01 = B1R; MR11 = A2R + B1R;
    /*  L finger   */
    ML00 = A1L + B1L; ML01 = B1L;  ML11 = A2L + B1L;

    /****  Initialize HCTL and HCTL parameters *** */
```

```
    write_hctl_pc(INIT);                    /*  initialize the hctl-1000     */

    /*  if user selects HIGH_ONLY controller, then leave in idle mode;
        do NOT switch to position control mode; want to command pw  */      110
    if(high_only)     /*  variable high_only set by user in display      */
        control_law = HIGH_ONLY;        /*  used only in this mode     */

    else              /*  do only to get into position control mode      */
    {
            /*  cosmetic ...   don't see bogus values in display      */
        Ctorque[0] = 0;          Ctorque[1] = 0;
        Ctorque[2] = 0;          Ctorque[3] = 0;

        control_law = SETPOINT;         /*  default controller          */      120

        write_gains(Gains);             /*  set control gains to zero    */
        write_command_position(Command_Pos);      /*  stretched out    */

        /*  leave the poles and zeros at default value, set by init'tn        */
        read_poles(Pole);
        read_zeros(Zero);

        hctl_sample = 64;               /*  64 = default sampling rate    */
        for (i = 0; i<4; i ++)                                                   130
        {
            hctl_wrreg(i, HCTL_STATUS, 1);   /*  set sign reversal inhibit     */
            hctl_wrreg(i, HCTL_SAMPLE, hctl_sample);  /*  set sample period   */
        }
        start_hctl_control();           /*  put hctl into control mode         */
    }                                   /*  end of if(high_only) ... else ...  */

    /****  Set up interrupt routines; do for all modes      *** */
    isr_set_cfreq(hctl_freq);
    isr_set_tfreq(high_freq);                                                    140
    isr_set_croutine(hctl_control);
    isr_set_troutine(high_control);

}                                       /*  end of hctl_setup()               */

/****************************************************************** */
/*  function to restart the controller after the                     */
/*  controller has been toggled to off and then on                   */
/*  again;  function called in styx.c                                */
/****************************************************************** */      150
hctl_restart()                                                       hctl_restart
{
    write_gains(Gains);                 /*  back to where they were           */
}

/****************************************************************** */
/*  primary (control) interrupt service routine                      */
/*  hctl-1000 position control                                       */
/*  this is the fast, low-level control loop                         */
```

```
/******************************************************************** */        160
hctl_control()                                                    hctl_control

{
    static long
        hctl_Curr_Vel[4],              /*  commanded joint velocity    */
        hctl_Curr_Pos[4];              /*  commanded joint position    */

    flag(0, 'C');                      /*  write C in row 0, col 0     */

    /*  Read the current position; use in control and data output      */    170
    hctl_actual[0] = hctl_rdpos(0, HCTL_ACTUAL);
    hctl_actual[1] = hctl_rdpos(1, HCTL_ACTUAL);
    hctl_actual[2] = hctl_rdpos(2, HCTL_ACTUAL);
    hctl_actual[3] = hctl_rdpos(3, HCTL_ACTUAL);

    /*  Read the current applied pulse-width; use in data output       */
    hctl_pwm[0] = hctl_rdreg(0, HCTL_PWM);
    hctl_pwm[1] = hctl_rdreg(1, HCTL_PWM);
    hctl_pwm[2] = hctl_rdreg(2, HCTL_PWM);
    hctl_pwm[3] = hctl_rdreg(3, HCTL_PWM);                                  180

    /*  do all control in high_control(); write to hctl there          */
    if (control_law==HIGH_ONLY)
    {
        if (control_on)    flag(1, 0x27);    /*  color row 0, col 0 green    */
        else               flag(1,0x47);     /*  color row 0, col 0 red      */
        return;
    }

    /****  controller is running       *** */                                190
    if (control_on)
    {
        flag(1, 0x27);                 /*  color row 0, col 0 green        */
        if (NewtrajFlag)               /*  high_control finished update    */
        {
            hctl_Curr_Pos[0] = Command_Pos[0];      /*  copy from          */
            hctl_Curr_Pos[1] = Command_Pos[1];      /*  intermed buffer    */
            hctl_Curr_Pos[2] = Command_Pos[2];
            hctl_Curr_Pos[3] = Command_Pos[3];
            hctl_Curr_Vel[0] = Command_Vel[0];                              200
            hctl_Curr_Vel[1] = Command_Vel[1];
            hctl_Curr_Vel[2] = Command_Vel[2];
            hctl_Curr_Vel[3] = Command_Vel[3];
            NewtrajFlag = 0;
        }                              /*  end of if(NewtrajFlag)          */

        /****  test controller; don't write to chips        *** */
        if (Fake_Control)
        {
            flag(12, 'F');             /*  put an F in row 0, col 6    */       210
            hctl_actual[0] = hctl_Curr_Pos[0];
            hctl_actual[1] = hctl_Curr_Pos[1];
```

```
        hctl_actual[2] = hctl_Curr_Pos[2];
        hctl_actual[3] = hctl_Curr_Pos[3];
    }

    /**** real controller on     *** */
    else
    {
#if defined(FAST_UPDATE)                                                        220
        int angle[4];
        double det, phi;

        /* R and L finger jacobians and their inverses      */
#       define rjac R_jacob
#       define ljac L_jacob
#       define pos hctl_actual

        /* Calculate the angle offsets for table lookups      */
        angle[0] = NUMR1*(pos[0] - styx[0].reference)/DENR1;                    230
        angle[1] = NUMR2*(pos[1] - styx[1].reference)/DENR2;
        angle[2] = NUML1*(pos[2] - styx[2].reference)/DENL1;
        angle[3] = NUML2*(pos[3] - styx[3].reference)/DENL2;

        rjac[0][1] =   lenR2_cos[angle[0] + angle[1] + TBLOFF];
        rjac[0][0] =   lenR1_cos[angle[0] + TBLOFF] + rjac[0][1];
        rjac[1][1] = -lenR2_sin[angle[0] + angle[1] + TBLOFF];
        rjac[1][0] = -lenR1_sin[angle[0] + TBLOFF] + rjac[1][1];

        ljac[0][1] =   lenL2_cos[angle[2] + angle[3] + TBLOFF];                 240
        ljac[0][0] =   lenL1_cos[angle[2] + TBLOFF] + ljac[0][1];
        ljac[1][1] = -lenL2_sin[angle[2] + angle[3] +TBLOFF];
        ljac[1][0] = -lenL1_sin[angle[2] + TBLOFF] + ljac[1][1];

        /* Calculate the inverse jacobian for the right finger      */
#       define rjac_inv R_jac_inv
        det = rjac[0][0]*rjac[1][1] - rjac[0][1]*rjac[1][0];
        rjac_inv[0][0] =   rjac[1][1]/det;     rjac_inv[1][1] =   rjac[0][0]/det;
        rjac_inv[0][1] = -rjac[0][1]/det;      rjac_inv[1][0] = -rjac[1][0]/det;
                                                                                250
        /* Calculate the jacobian for the left finger      */
#       define ljac_inv L_jac_inv
        det = ljac[0][0]*ljac[1][1] - ljac[0][1]*ljac[1][0];
        ljac_inv[0][0] =   ljac[1][1]/det;     ljac_inv[1][1] =   ljac[0][0]/det;
        ljac_inv[0][1] = -ljac[0][1]/det;      ljac_inv[1][0] = -ljac[1][0]/det;

        /* Figure out the current angle of the box      */
        /*! For now we will cheat and just assume phi == 0 ! */
        force_N[0] = -(force_N[2] = mag_force_N*cos(t));
        force_N[1] = -(force_N[3] = mag_force_N*sin(t));                        260

        /* calculate the torque required to apply internal force      */
       torque_N[0] = (rjac[0][0]*force_N[0]+rjac[1][0]*force_N[1])/KM_MIN;
       torque_N[1] = (rjac[0][1]*force_N[0]+rjac[1][1]*force_N[1])/KM_HIT;
       torque_N[2] = (ljac[0][0]*force_N[2]+ljac[1][0]*force_N[3])/KM_MIN;
```

```
            torque_N[3] = (ljac[0][1]*force_N[2]+ljac[1][1]*force_N[3])/KM_HIT;

        /*  Calculate the effect of internal force on position          */
        /*  NOTE: need negative sign on joint 1 only      */
        if (Gamma[0] != 0) hctl_Curr_Pos[0] += torque_N[0]/Gamma[0];                270
        if (Gamma[1] != 0) hctl_Curr_Pos[1] += torque_N[1]/Gamma[1];
        if (Gamma[2] != 0) hctl_Curr_Pos[2] += torque_N[2]/Gamma[2];
        if (Gamma[3] != 0) hctl_Curr_Pos[3] += torque_N[3]/Gamma[3];
#endif

            flag(12, ' ');      /*  put a space in row 0, col 6; i.e. no 'F'     */

            /*  write the commanded position to chips     */
            hctl_wrpos(0, HCTL_COMMAND, hctl_Curr_Pos[0]);
            hctl_wrpos(1, HCTL_COMMAND, hctl_Curr_Pos[1]);                          280
            hctl_wrpos(2, HCTL_COMMAND, hctl_Curr_Pos[2]);
            hctl_wrpos(3, HCTL_COMMAND, hctl_Curr_Pos[3]);
        }           /*  end of else ...     (real controller)      */

        /*  update position for next time      */
        hctl_Curr_Pos[0] += Kv_int*hctl_Curr_Vel[0];
        hctl_Curr_Pos[1] += Kv_int*hctl_Curr_Vel[1];
        hctl_Curr_Pos[2] += Kv_int*hctl_Curr_Vel[2];
        hctl_Curr_Pos[3] += Kv_int*hctl_Curr_Vel[3];
                                                                                    290
    }                   /*  end of if(control_on)            */

    /****  controller off      *** */
    else
        flag(1,0x47);   /*  color row 0, col 0 red           */

    dump_update();      /*  for saving and graphing data     */

}                       /*  end of hctl_control()            */
                                                                                    300
/******************************************************************* */
/*  secondary interrupt service routine slower, update loop; high          */
/*  level controller for hierarchical cotnrol; calcualtes and sends        */
/*  the desired joint angle location to the primary isr                    */
/******************************************************************* */
high_control()                                                          high_control
{
    register int
        i;                  /*  generic counter      */
                                                                                    310
    flag(2, 'T');           /*  T in row 0, col 1   */

    /******  Object Position      ***** */
    input_update(1);            /*  current, next desired obj pos'tn      */

#if !defined(FAST_UPDATE)
    last_x = x;      last_y = y;      last_t = t;    /* last act pos'tn     */
    pos_forward(hctl_actual);              /*  current actual obj pos'tn    */
```

```
#endif
                                                                          320
    /*  current actual box velocity, box coordinates        */
    Box_Act_Veloc[0] = (x - last_x)*high_freq;
    Box_Act_Veloc[1] = (y - last_y)*high_freq;
    Box_Act_Veloc[2] = (t - last_t)*high_freq;

    /******  Control Law  ***** */
    /*  current desired box pos -> current desired joint angles      */
    pos_inverse(pid[0].desired,pid[1].desired,pid[2].desired,Command_Pos);

#if !defined(FAST_UPDATE)                                                 330
    /*  R and L finger jacobian and jacobian inverse matrices      */
    jacobian(hctl_actual, R_jacob, L_jacob);
    jac_inverse(R_jacob, R_jac_inv);
    jac_inverse(L_jacob, L_jac_inv);
#endif

    /*  calculate DC gain of hctl controller        */
    for(i=0; i<4; i++)
        Gamma[i] = (Gains[i]*(256.0 - Zero[i]))/(4.0*(256.0 + Pole[i]));
                                                                          340
    switch (control_law)
    {
        case JOINT_INTERPOLATION:
        /*  add velocity interpolation component calc'd in joint space          */
        /*  when in this mode, the Kv_int doesn't need to explicitly be
            set = 1; Kv_int is there merely to allow the user to turn on
            joint interpolation at the low level when in hierarchical mode      */
        {
            pos_inverse(pid[0].next,pid[1].next,pid[2].next, next_Pos);
            for(i=0; i<4; i++)                                            350
            {
                Expec_Pos[i] = 0;       f         /*  no box error correction      */
                Command_Vel[i] = (next_Pos[i] - Command_Pos[i])*
                        high_freq/hctl_freq;
            }
            K_p = 0;   K_a = 0;                   /*  not required here            */
            flag(380, 'J');      flag(382, 't');      flag(384, 'I');
            break;
        }
        case SETPOINT:                                                    360
        /*  no interpolation; high control is strictly inverse kinematic        */
        {
            for(i=0; i<4; i++)
            {
                Command_Vel[i] = 0;               /*  no interpolation      */
                Expec_Pos[i] = 0;                 /*  no error correc'tn    */
            }
            K_v = 0;  K_p = 0;  K_a = 0;          /*  not required here     */
            flag(380, 'S');      flag(382, 'P');      flag(384, 't');
            break;                                                        370
        }
```

```
case OBJECT_CORRECTION:
/*  add correction for pos'tn and veloc errors in box coord's      */
/*  PD plus FeedForward high-level control                          */
{
    pos_inverse(pid[0].next,pid[1].next,pid[2].next, next_Pos);
    for(i=0; i<4; i++)
        Command_Vel[i] = (next_Pos[i] - Command_Pos[i])*
                high_freq/hctl_freq;
    expected_force();                                                     380
    exp_force_correct();
    flag(380, 'P');      flag(382, 'D');      flag(384, 'F');
    break;
}
case COMP_TORQUE:
{   /*  computed torque control              */
    /*  R finger inertia matrix definitions  */
    /*  B2R * cosine of distal joint angle   */
    b2r_cos_thR2 = B2R*lenR2_cos[th1_2 + TBLOFF]/LenR2;
    Mass_R[0][0] = MR00 + b2r_cos_thR2;                                   390
    Mass_R[0][1] = MR01 + b2r_cos_thR2*0.5;
    Mass_R[1][0] = Mass_R[0][1];          /*  symmetric matrix  */
    Mass_R[1][1] = MR11;                  /*  constant          */
    /*  L finger inertia matrix definitions  */
    /*  B2L * cosine of distal joint angle   */
    b2l_cos_thL2 = B2L*lenL2_cos[th2_2 + TBLOFF]/LenL2;
    Mass_L[0][0] = ML00 + b2l_cos_thL2;
    Mass_L[0][1] = ML01 + b2l_cos_thL2*0.5;
    Mass_L[1][0] = Mass_L[0][1];          /*  symmetric matrix  */
    Mass_L[1][1] = ML11;                  /*  constant          */          400

    pos_inverse(pid[0].next,pid[1].next,pid[2].next, next_Pos);
    for(i=0; i<4; i++)
        Command_Vel[i] = (next_Pos[i] - Command_Pos[i])*
                high_freq/hctl_freq;

    /*  K_a = 0,1 -- for testing purposes; see control.c        */
    expected_force();          /*  F = Mhand*resvd_accel(Xobj)  */
    exp_force_correct();       /*  Pos = (JacT*G++F)/hctl_gain  */
    flag(380, 'C');      flag(382, 'T');      flag(384, ' ');                 410
    break;
}
case HIGH_ONLY:
{   /*  do COMPUTED TORQUE control at HIGH LEVEL ONLY */
    if(high_only)
    {
        flag(380, 'H');      flag(382, 'C');      flag(384, 'T');

        /*  R finger inertia matrix definitions  */
        /*  B2R * cosine of distal joint angle   */                          420
        b2r_cos_thR2 = B2R*lenR2_cos[th1_2 + TBLOFF]/LenR2;
        Mass_R[0][0] = MR00 + b2r_cos_thR2;
        Mass_R[0][1] = MR01 + b2r_cos_thR2*0.5;
        Mass_R[1][0] = Mass_R[0][1];          /*  symmetric matrix  */
```

```
            Mass_R[1][1] = MR11;                /*  constant            */
            /*  L finger inertia matrix definitions      */
            /*  B2L * cosine of distal joint angle       */
            b2l_cos_thL2 = B2L*lenL2_cos[th2_2 + TBLOFF]/LenL2;
            Mass_L[0][0] = ML00 + b2l_cos_thL2;
            Mass_L[0][1] = ML01 + b2l_cos_thL2*0.5;                              430
            Mass_L[1][0] = Mass_L[0][1];        /*  symmetric matrix    */
            Mass_L[1][1] = ML11;                /*  constant            */

            /*  calculate F = Mhand*resvd_accel(Xobj) in obj coord's      */
            expected_force();

            /*  calculate applied fingertip force        */
            ftip[0] = Force_Expec[0]+Grasp_Pseudo_Act[0][2]*Force_Expec[2];
            ftip[1] = Force_Expec[1]+Grasp_Pseudo_Act[1][2]*Force_Expec[2];
            ftip[2] = Force_Expec[0]+Grasp_Pseudo_Act[2][2]*Force_Expec[2];      440
            ftip[3] = Force_Expec[1]+Grasp_Pseudo_Act[3][2]*Force_Expec[2];

            /*  calc applied tau (in pw) = Jh-Trans*ftip_force/motor gain      */
            force2torque(ftip, Ctorque);

            /*  apply the desired pulse-width to hctl       */
            write_pwm(Ctorque);
        }   /*  end of if(high_only)      */
            break;
    }       /*  end of HIGH_ONLY      */                                         450
    default:
        break;
}           /*  end of switch(control_law)              */

#if !defined(FAST_UPDATE)
    /*  calculate effect of internal force         */
    force_N[2] = mag_force_N*cos(t);          force_N[0] = -force_N[2];
    force_N[3] = mag_force_N*sin(t);          force_N[1] = -force_N[3];
    force2torque(force_N, torque_N);
#endif                                                                           460

    /******  add in correcting forces        **** */
    for(i=0; i<4; i++)
    {
#if !defined(FAST_UPDATE)
        /*  calc effect of internal force on position        */
        if (Gamma[i] == 0)        IntF_Pos[i] = 0;
        else                      IntF_Pos[i] = torque_N[i]/Gamma[i];
        if (i==1) IntF_Pos[i] = -IntF_Pos[i];
                                                                                 470
        /*  add effect of internal force on Command_Pos      */
        Command_Pos[i] += IntF_Pos[i];
#endif
        /*  add effect of box error corr'g force on Command_Pos       */
        Command_Pos[i] += Expec_Pos[i];
    }   /*  end of for()    */
```

```
    /******  Cleanup  ***** */
    /*  tell hctl_control()     to grab Command_Pos and Command_Vel */
    NewtrajFlag = 1;                                                        480

}   /*  end of high_control()    */

/****************************************************************** */
/*  command the motors to apply a specified torque;                 */
/*  motor input command is clipped at +-127 to avoid                */
/*  wrap-around errors                                              */
/****************************************************************** */
int write_pwm(ftau)                                               write_pwm
double                                                                  490
    ftau[6];
{
    int
        itau[6],
        i;

    for (i=0; i<4; i++)
    {
       if (ftau[i] > 127)              /*  limit maximum motor input  */
            ftau[i] = 127;                                              500
       else if (ftau[i] < -127)
            ftau[i] = -127;

       itau[i] = (int)ftau[i];          /*  convert dbl torque to char     */
       Torque[i] = (char)itau[i];
    }

    hctl_write_pwm_tbl(Torque);
}             /*  end of write_pwm()  function   */
                                                                        510
/****************************************************************** */
```

```
/*****************************************************************/
/*  File:   control.c                                            */
/*  routines to calculate the control algorithms for Styx; i.e.  */
/*  calculate the applied joint torque                           */
/*                                                               */
/*  created:    KHO 7 August 1991                                */
/*  modified:   KHO 14 September 1991                            */
/*                                                               */
/*****************************************************************/
                                                                        10
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include "styx.h"

/*  motor gains  */
#define KM_MIN 21385        /*  gain for minertia motors    */
#define KM_HIT  5897        /*  gain for hitachi motors     */

extern int                                                              20
    control_law;            /*  which high level controller     */

extern double
    R_jac_inv[2][2],        /*  inverse of R finger jacob       */
    L_jac_inv[2][2],        /*  inverse of L finger jacob       */
    Mass_R[2][2],           /*  inertia matrix, R finger        */
    Mass_L[2][2],           /*  inertia matrix, L finger        */
    Box_Act_Veloc[3],       /*  actual box velocity
    Grasp_Pseudo_Act[4][3], /*  pseudo inverse of grasp map     */
    Grasp_Act[3][4],        /*  actual current grasp map        */  30
    Gamma[4];               /*  DC gain of hctl controller      */

int
    K_a,                    /*  "gain"--whether to use obj mass;=0,1  */
    high_freq;              /*  frequency of secondary isr            */

double
    box_Perr[3],            /*  position error in box coord's       */
    box_Verr[3],            /*  velocity error in box coord's       */
    resv_box_accel[3],      /*  resolved box acceleration           */  40
    Expec_Pos[4],           /*  position "correction"               */
    Force_Expec[4],         /*  expected force, object coord's      */
    K_v, K_p,               /*  high level gains                    */
    R_jacob[2][2],          /*  R finger jacobian matrix            */
    L_jacob[2][2],          /*  L finger jacobian matrix            */
    Mhand[3][3],            /*  inertia of fing's plus object       */
    MJiGt[4][3],            /*  fing iner*Jac-inv*Grasp-transp      */
    JiGt[4][3];             /*  jac-inv * Grasp-transpose           */

/*****************************************************************/    50
/*  compute the extra correcting force to get box on to          */
/*  desired trajectory                                           */
/*****************************************************************/
```

```
expected_force()
{
    register int
        i,j,k;                  /*  the usual counter    */

    /*  position error in box coordinates       */
    box_Perr[0] = pid[0].desired - x;                                          60
    box_Perr[1] = pid[1].desired - y;
    box_Perr[2] = pid[2].desired - t;

    /*  velocity error in box coordinates       */
    for(i=0; i<3; i++)
        box_Verr[i] = pid[i].veloc - Box_Act_Veloc[i];

    switch (control_law)
    {
        case OBJECT_CORRECTION:                                                70
        /*  PD plus FeedForward control   */
        {
        /*  K_a = {0,1} => {without,with} box mass  */
        Force_Expec[0] = K_a*Mo*pid[0].accel+K_v*box_Verr[0]+K_p*box_Perr[0];
        Force_Expec[1] = K_a*Mo*pid[1].accel+K_v*box_Verr[1]+K_p*box_Perr[1];
        Force_Expec[2] = K_a*Io*pid[2].accel+K_v*box_Verr[2]+K_p*box_Perr[2];
        }   /*  end of OBJECT_CORRECTION */

        case HIGH_ONLY:
        case COMP_TORQUE:                                                      80
        {
            /*  "resolved" acceleration in box coordinates      */
            for(i=0; i<3; i++)
            /*  K_a for test purp's only; K_a=K_v=K_p=0-->inverse kine's   */
                resv_box_accel[i] = K_a*pid[i].accel + K_v*box_Verr[i] +
                                    K_p*box_Perr[i];
            /*  jacobian-inv*grasp-transp   */
            grasp();                    /*  calculate Grasp_Act   */
            /*  R finger part   */
            JiGt[0][0] = R_jac_inv[0][0];         JiGt[0][1] = R_jac_inv[0][1];    90
            JiGt[0][2] = Grasp_Act[2][0]*R_jac_inv[0][0] +
                         Grasp_Act[2][1]*R_jac_inv[0][1];
            JiGt[1][0] = R_jac_inv[1][0];         JiGt[1][1] = R_jac_inv[1][1];
            JiGt[1][2] = Grasp_Act[2][0]*R_jac_inv[1][0] +
                         Grasp_Act[2][1]*R_jac_inv[1][1];
            /*  L finger part   */
            JiGt[2][0] = L_jac_inv[0][0];         JiGt[2][1] = L_jac_inv[0][1];
            JiGt[2][2] = Grasp_Act[2][2]*L_jac_inv[0][0] +
                            Grasp_Act[2][3]*L_jac_inv[0][1];
            JiGt[3][0] = L_jac_inv[1][0];         JiGt[3][1] = L_jac_inv[1][1];    100
            JiGt[3][2] = Grasp_Act[2][2]*L_jac_inv[1][0] +
                            Grasp_Act[2][3]*L_jac_inv[1][1];
            /*  Mass_fing*J_inv*G_trans = 4x3 matrix   */
            /*  R finger part   */
            MJiGt[0][0] = Mass_R[0][0]*JiGt[0][0] + Mass_R[0][1]*JiGt[1][0];
            MJiGt[0][1] = Mass_R[0][0]*JiGt[0][1] + Mass_R[0][1]*JiGt[1][1];
```

```
        MJiGt[0][2] = Mass_R[0][0]*JiGt[0][2] + Mass_R[0][1]*JiGt[1][2];
        MJiGt[1][0] = Mass_R[1][0]*JiGt[0][0] + Mass_R[1][1]*JiGt[1][0];
        MJiGt[1][1] = Mass_R[1][0]*JiGt[0][1] + Mass_R[1][1]*JiGt[1][1];
        MJiGt[1][2] = Mass_R[1][0]*JiGt[0][2] + Mass_R[1][1]*JiGt[1][2];             110
        /*  L finger part    */
        MJiGt[2][0] = Mass_L[0][0]*JiGt[2][0] + Mass_L[0][1]*JiGt[3][0];
        MJiGt[2][1] = Mass_L[0][0]*JiGt[2][1] + Mass_L[0][1]*JiGt[3][1];
        MJiGt[2][2] = Mass_L[0][0]*JiGt[2][2] + Mass_L[0][1]*JiGt[3][2];
        MJiGt[3][0] = Mass_L[1][0]*JiGt[2][0] + Mass_L[1][1]*JiGt[3][0];
        MJiGt[3][1] = Mass_L[1][0]*JiGt[2][1] + Mass_L[1][1]*JiGt[3][1];
        MJiGt[3][2] = Mass_L[1][0]*JiGt[2][2] + Mass_L[1][1]*JiGt[3][2];
        /*  Mhand (3x3) = G*J-inv-trans * MJiGtr + Mobj  */
        /*        = (J-inv*G-trans)-trans * MJiGtr + Mobj  */
        for(i=0; i<3; i++)        /*  matrix multip with L mtrx trans'd    */        120
            for(j=0; j<3; j++)
            {
                Mhand[i][j] = 0;
                for(k=0; k<4; k++)
                    Mhand[i][j] += JiGt[k][i]*MJiGt[k][j];
            }   /*  end of for   */
        /*  add Mobj part; Mobj = diagonal 3x3 matrix    */
        Mhand[0][0] += Mo; Mhand[1][1] += Mo; Mhand[2][2] += Io;

        /*  calculate the force to be exerted on the object; ignore                   130
            coriolis terms; Force = Mhand(Xob) * res_accel(Xob)      */
        for(i=0; i<3; i++)
        {   /*  matrix by vector multip      */
            Force_Expec[i] = 0;
            for(j=0; j<3; j++)
                Force_Expec[i] += Mhand[i][j]*resv_box_accel[j];
        }                         /*  end of for()                  */
        break;
    }                             /*  end of COMP_TORQUE and HIGH_ONLY */
                                                                                      140
  default:                        /*  default case                  */
  {
    for(i=0; i<3; i++)
        Force_Expec[i] = 0;       /*  no high level box traj correction      */
    break;
  }                               /*  end of default          */
  }                               /*  end of switch(control_law)    */
}                                 /*  end of expected_force()       */

/****************************************************************** */                150
/*  given the extra, expected force, calculate the                  */
/*  needed to be added to Command_Pos to exert the extra            */
/*  force                                                           */
/*  pos is calculated in units of encoder counts                    */
/****************************************************************** */
exp_force_correct()                                                   exp_force_correct
{
  register int
      i;
```

```
                                                                    160
  static double
      ftip_force[4],     /*  fingertip force, as result of force     */
      tau[4];            /*  joint torque, as result of force        */

  /*  ftip_force = pseudo_inv(G)*obj_force                 */
  /*  simplify matrix multiplication, since parts of the   */
  /*  pseudo-inverse of the grasp map = 1, 0 for all       */
  /*  object orientations-->don't need to carry out mult   */
  /*  for all matrix elements                              */
  ftip_force[0] = Force_Expec[0] + Grasp_Pseudo_Act[0][2]*Force_Expec[2];   170
  ftip_force[1] = Force_Expec[1] + Grasp_Pseudo_Act[1][2]*Force_Expec[2];
  ftip_force[2] = Force_Expec[0] + Grasp_Pseudo_Act[2][2]*Force_Expec[2];
  ftip_force[3] = Force_Expec[1] + Grasp_Pseudo_Act[3][2]*Force_Expec[2];

  /*  tau (in pw) = Jh-Trans*ftip_force/motor gain    */
  force2torque(ftip_force, tau);

  /*  position = torque/DC gain of hctl_controller     */
  for(i=0; i<4; i++)
  {                                                                 180
      if (Gamma[i] == 0)          /*  Gamma = DC gain of hctl        */
          Expec_Pos[i] = 0;       /*  no compensation without gain   */
      else
          Expec_Pos[i] = tau[i]/Gamma[i];
  }                               /*  end of for()                   */

  /*  sign adjustment for joint #1; else force would LOSE grip      */
  Expec_Pos[1] = -Expec_Pos[1];

}                                 /*  end of exp_force_correct()     */   190

/****************************************************************** */
/*  function to compute the required joint torques, tau, in         */
/*  order to obtain the specified fingertip force, force            */
/****************************************************************** */
```

force2torque

```
force2torque(force, tau)
    double
        force[4],
        tau[4];                                                     200
{
    tau[0] = (R_jacob[0][0]*force[0]+R_jacob[1][0]*force[1])/KM_MIN;
    tau[1] = (R_jacob[0][1]*force[0]+R_jacob[1][1]*force[1])/KM_HIT;
    tau[2] = (L_jacob[0][0]*force[2]+L_jacob[1][0]*force[3])/KM_MIN;
    tau[3] = (L_jacob[0][1]*force[2]+L_jacob[1][1]*force[3])/KM_HIT;
}   /*  end of force2torque()      */

/****************************************************************** */
```

```
/******************************************************************* */
/*  File:    kine.c                                                  */
/*  kinematics routines for hierarchical control of STYX             */
/*                                                                   */
/*  created:     KHO 24 July 1991                                    */
/*  modified:    KHO 13 Sept 1991                                    */
/*                                                                   */
/******************************************************************* */

#include <stdio.h>                                                        10
#include <ctype.h>
#include <math.h>
#include <styx/hctl.h>
#include <styx/isr.h>
#include "styx.h"

#define POS2RAD 1.7453292e-3 /* convert count to radians: 2*PI / RES11  */

extern float far lenR1_sin[], far lenR1_cos[],
             far lenR2_sin[], far lenR2_cos[];                            20
extern float far lenL1_sin[], far lenL1_cos[],
             far lenL2_sin[], far lenL2_cos[];

int
    th1_2,         /*  distal joint angle for R finger (without SG)    */
    th2_2;         /*  distal joint angle for L finger (with SG)       */

double
    Jangle[4],                /*  joint angles                         */
    Dist,                     /*  distance between R,L fingertips      */  30
    Grasp_Pseudo_Act[4][3],   /*  pseudo inverse of grasp map          */
    Grasp_Act[3][4];          /*  grasp map at actual object pos'tn    */

/******************************************************************* */
/* Inverse kinematics:     re-entrant version                        */
/*  compute joint, angles, theta (th), from object position          */
/*  and orientation, xd, yd, and td                                  */
/*  th[0,1] = joint angles of R finger (with SG)                     */
/*  th[2,3] = joint angles of L finger (without SG)                  */
/******************************************************************* */  40
pos_inverse(xd, yd, td, count)                                   pos_inverse
    double
       xd,                    /*  object CM x coordinate     */
       yd,                    /*  object CM y coordinate,    */
       td;                    /*  object orientation, rad    */

    long
        count[4];             /* encoder counts              */

{                                                                         50
    static double             /*  temporary variables        */
        th[4],
        alpha, beta, xy2,
```

```
        x_1d, y_1d, x_2d, y_2d,
        th1_1d, th1_2d, th2_1d, th2_2d,
        temp_a, temp_b, temp_c, temp_d;

# define PI_2 (3.14159265358979/2)

    /* Calculate the end effector positions        */
    alpha = object_radius * cos(td);
    beta  = object_radius * sin(td);

    x_2d = (2*xd - alpha + BASELINE) / 2;
    y_2d = (2*yd - beta) / 2;
    x_1d = (2*xd + alpha - BASELINE) / 2;
    y_1d = (2*yd + beta) / 2;

    /*  Calculate the desired joint angles in radians              */
    /*  Choose the signs of the angles based on the usual          */
    /*  configuration to keep the joints from hyperextending       */
    /*  R finger = finger with strain gauges       */
    xy2 = x_1d*x_1d + y_1d*y_1d;
    /*! Finger 1, joint 1 encoder spins backwards => negate angle !    */
    temp_a = xy2 - LenR1*LenR1 - LenR2*LenR2;
    temp_b = 2*LenR1*LenR2;
    sqrt_87((temp_b*temp_b - temp_a*temp_a), &temp_c);
    atan2_87(temp_c, temp_a, &temp_d);
    th[1] = -temp_d;                          /*  distal joint angle        */

    temp_a = xy2 + LenR1*LenR1 - LenR2*LenR2;
    sqrt_87(xy2, &temp_d);
    temp_b = 2*LenR1*temp_d;
    sqrt_87((temp_b*temp_b - temp_a*temp_a), &temp_c);
    atan2_87(temp_c, temp_a, &temp_d);
    atan2_87(y_1d, x_1d, &temp_b);
    th[0] = (double) PI_2 - temp_b + temp_d;  /*  proximal joint angle      */

    /*  now convert the angles, th[i], from radians to encoder counts         */
    count[0] = th[0]*DENR1/(POS2RAD*NUMR1) + styx[0].reference;
    count[1] = th[1]*DENR2/(POS2RAD*NUMR2) + styx[1].reference;

    /*  L finger = finger without strain gauges      */
    xy2 = x_2d*x_2d + y_2d*y_2d;
    temp_a = xy2 - LenL1*LenL1 - LenL2*LenL2;
    temp_b = 2*LenL1*LenL2;
    sqrt_87((temp_b*temp_b - temp_a*temp_a), &temp_c);
    atan2_87(temp_c, temp_a, &temp_d);
    th[3] = temp_d;                           /*  distal joint angle        */

    temp_a = xy2 + LenL1*LenL1 - LenL2*LenL2;
    sqrt_87(xy2, &temp_d);
    temp_b = 2*LenL1*temp_d;
    sqrt_87((temp_b*temp_b - temp_a*temp_a), &temp_c);
    atan2_87(temp_c, temp_a, &temp_d);
    atan2_87(y_2d, x_2d, &temp_b);
```

```
    th[2] = (double) PI_2 - temp_b - temp_d; /* proximal joint angle     */

    count[2] = th[2]*DENL1/(POS2RAD*NUML1) + styx[2].reference;
    count[3] = th[3]*DENL2/(POS2RAD*NUML2) + styx[3].reference;                 110
}         /*  end of pos_inverse()     */

/*************************************************************** */
/*  function to compute the forward kinematics; needed to         */
/*  update the current object position/orientation for the        */
/*  graphing package found in graph.c (which plots current        */
/*  and desired object location                                   */
/*  th_count = joint angles (encoder counts)                      */
/*  obj loc    = x, y, t = global variables defined in global.h   */
/*************************************************************** */             120
pos_forward(th_count)                                                  pos_forward
    long
        th_count[4];      /*  joint angles; encoder counts     */

{
    /*  convert encoder counts to radians      */
    /*  [0,1] = R finger; [2,3] = L finger     */

    /*  R finger = finger with strain gauges        */
    th1_1 = (th_count[0] - styx[0].reference) * NUMR1/DENR1;                    130
    th1_2 = (th_count[1] - styx[1].reference) * NUMR2/DENR2;
    Jangle[0] = th1_1*POS2RAD;
    Jangle[1] = th1_2*POS2RAD;

    /*  L finger = finger without strain gauges     */
    th2_1 = (th_count[2] - styx[2].reference) * NUML1/DENL1;
    th2_2 = (th_count[3] - styx[3].reference) * NUML2/DENL2;
    Jangle[2] = th2_1*POS2RAD;
    Jangle[3] = th2_2*POS2RAD;
                                                                                140
    /* Calculate the end effector location          */
    /*  R finger = finger with strain gauges        */
    lenR1_s1 = lenR1_sin[th1_1 + TBLOFF];
    lenR1_c1 = lenR1_cos[th1_1 + TBLOFF];
    th1_12 = th1_1 + th1_2 + TBLOFF;
    lenR2_s12 = lenR2_sin[th1_12];
    lenR2_c12 = lenR2_cos[th1_12];

    /*  L finger = finger without strain gauges     */
    lenL1_s1 = lenL1_sin[th2_1 + TBLOFF];                                       150
    lenL1_c1 = lenL1_cos[th2_1 + TBLOFF];
    th2_12 = th2_1 + th2_2 + TBLOFF;
    lenL2_s12 = lenL2_sin[th2_12];
    lenL2_c12 = lenL2_cos[th2_12];

    x_1 = lenR1_s1 + lenR2_s12;      /*  x coord of R finger - B/2  */
    y_1 = lenR1_c1 + lenR2_c12;      /*  y coord of R finger        */
    x_2 = lenL1_s1 + lenL2_s12;      /*  x coord of L finger + B/2  */
    y_2 = lenL1_c1 + lenL2_c12;      /*  y coord of L finger        */
```

```
                                                                            160
    dy = y_2 - y_1;   f             /*  differ between L-R ftips    */
    dx = x_2 - x_1 + BASELINE;

    /*  distance between ftips    */
    sqrt_87((dy*dy + (dx-2*BASELINE)*(dx-2*BASELINE)), &Dist);

    x = (x_1 + x_2)/2;              /*  x coord of obj CMass  */
    y = (y_1 + y_2)/2;              /*  y coord of obj CMass  */

    atan2_87(dy, dx, &t);           /*  obj orientation       */            170
}                                   /*  end of pos_forward()  */

/**************************************************************** */
/*  function to compute the jacobian of the R and L fingers,       */
/*  given the joint angles in units of encoder counts              */
/**************************************************************** */
jacobian(pos, rjac, ljac)                                          jacobian
    double
        rjac[2][2],                 /*  jacobian for R finger   */
        ljac[2][2];                 /*  jacobian for L finger   */          180
    long
        pos[4];                     /*  raw joint angles        */
{
    long
        angle[4];                   /*  joint angles, encoder counts    */

    angle[0] = NUMR1*(pos[0] - styx[0].reference)/DENR1;
    angle[1] = NUMR2*(pos[1] - styx[1].reference)/DENR2;
    angle[2] = NUML1*(pos[2] - styx[2].reference)/DENL1;
    angle[3] = NUML2*(pos[3] - styx[3].reference)/DENL2;                    190

    rjac[0][1] =    lenR2_cos[angle[0] + angle[1] + TBLOFF];
    rjac[0][0] =    lenR1_cos[angle[0] + TBLOFF] + rjac[0][1];
    rjac[1][1] = -lenR2_sin[angle[0] + angle[1] + TBLOFF];
    rjac[1][0] = -lenR1_sin[angle[0] + TBLOFF] + rjac[1][1];

    ljac[0][1] =    lenL2_cos[angle[2] + angle[3] + TBLOFF];
    ljac[0][0] =    lenL1_cos[angle[2] + TBLOFF] + ljac[0][1];
    ljac[1][1] = -lenL2_sin[angle[2] + angle[3] +TBLOFF];
    ljac[1][0] = -lenL1_sin[angle[2] + TBLOFF] + ljac[1][1];                200
}           /*  end of jacobian()   */


/**************************************************************** */
/*  function to compute the inverse jacobian of a finger,          */
/*  given the jacobian matrix of that finger                       */
/**************************************************************** */
jac_inverse(jac, jac_inv)                                          jac_inverse
    double
        jac[2][2],          /*  jacobian matrix of finger     */                210
        jac_inv[2][2];      /*  inverse jacobian of finger    */
{
```

```
    double
        det;                /*  determinant of finger jacob    */

    det = jac[0][0]*jac[1][1] - jac[0][1]*jac[1][0];
    jac_inv[0][0] =    jac[1][1]/det;        jac_inv[1][1] =    jac[0][0]/det;
    jac_inv[0][1] = -jac[0][1]/det;          jac_inv[1][0] = -jac[1][0]/det;

}                           /*  end of jac_inverse()     */                          220

/**************************************************************** */
/*  function to compute the grasp map, given the actual object      */
/*  orientation (global variable, "t")-->      global variable      */
/*  Grasp_Act[3][4]                                                 */
/**************************************************************** */
grasp()                                                                          grasp
{
    Grasp_Act[0][0] = 1;          Grasp_Act[0][1] = 0;
    Grasp_Act[0][2] = 1;          Grasp_Act[0][3] = 0;                              230

    Grasp_Act[1][0] = 0;          Grasp_Act[1][1] = 1;
    Grasp_Act[1][2] = 0;          Grasp_Act[1][3] = 1;

    Grasp_Act[2][2] =   0.5*object_radius*sin(t);
    Grasp_Act[2][1] =   0.5*object_radius*cos(t);
    Grasp_Act[2][0] = -Grasp_Act[2][2];
    Grasp_Act[2][3] = -Grasp_Act[2][1];
}                          /*  end of grasp()   */
                                                                                    240
/**************************************************************** */
/*  function to compute the pseudo-inverse of the grasp map,        */
/*  given the actual object orientation (global variable, "t")      */
/*  --> global variable Grasp_Act[3][4]                             */
/**************************************************************** */
grasp_pseudo()                                                            grasp_pseudo
{
    Grasp_Pseudo_Act[0][0] = 0.5;        Grasp_Pseudo_Act[0][1] = 0;
    Grasp_Pseudo_Act[1][0] = 0;          Grasp_Pseudo_Act[1][1] = 0.5;
    Grasp_Pseudo_Act[2][0] = 0.5;        Grasp_Pseudo_Act[2][1] = 0;                250
    Grasp_Pseudo_Act[3][0] = 0;          Grasp_Pseudo_Act[3][1] = 0.5;

    /*  there is no factor of 0.5 in these terms, since it cancels the
        the 0.5 multiplying the object_radius, which is more like an
        "object_diameter"...     */
    Grasp_Pseudo_Act[2][2] =   sin(t)/object_radius;
    Grasp_Pseudo_Act[1][2] =   cos(t)/object_radius;
    Grasp_Pseudo_Act[0][2] = -Grasp_Pseudo_Act[2][2];
    Grasp_Pseudo_Act[3][2] = -Grasp_Pseudo_Act[1][2];
}                                                                                   260

/**************************************************************** */
```

```
/***************************************************************** */
/*  File:    input.c                                               */
/*  routines to handle input (create trajectories)                 */
/*                                                                 */
/*  created:     RMM 17 March 1988                                 */
/*  modified:    KHO 7 August 1991                                 */
/*                                                                 */
/***************************************************************** */

#include <stdio.h>                                                          10
#include <styx/ddisp.h>
#include <styx/isr.h>
#include "styx.h"
#include "input.h"

#define PI       (3.1415926535)                                             PI

extern int
    high_freq;
                                                                            20
/* Global variables    */
int
    input = Setpoint,                    /* current input type       */
    input_char = 's';                    /* current input character  */

double
    circle_radius, circle_freq,          /* circle parameters        */
    fig8_radius, fig8_freq,              /* figure8 parameters       */
    box_radius, box_freq,                /* box parameters           */
    input_off[3];                        /* center for all routines  */     30

/* Local variables   */
struct input_data {
    double out[3];                       /* desired positions        */
    double acc[3];                       /* desired acceleration     */
} far traj[1024];

char
  ityp[3];                               /* periodic input type      */
                                                                            40
int
  input_count = 0,
  input_index = 0;

double
    mag[3], phase[3], freq[3],           /* circle parameters        */
    off[3], out[3];

/* Include the option table    */
#include "input.tbl"                                                        50

/***************************************************************** */
/*  calculates current desired object location                     */
```

```
/*  rate = relative rate of the trajectory                              */
/********************************************************************* */
input_update(rate)                                                          input_update
int
  rate;
{
    register int                                                                    60
      axis,
      index;

    if (control_on && input != Setpoint)
    {
        flag(8, input_char);              /*  put input_char in row 0, col 4      */
        index = input_index;
        for (axis = 0; axis < 3; ++axis)
        {
            pid[axis].desired = traj[index].out[axis];          /* des obj loc    */          70
            pid[axis].accel = traj[index].acc[axis];            /* des obj acc    */
            if ((index+rate) >= input_count)
                pid[axis].next      = traj[0].out[axis];        /*  wraparound   */
            else
                pid[axis].next      = traj[index+rate].out[axis];      /* next loc  */
            pid[axis].veloc = (pid[axis].next - pid[axis].desired)*high_freq;
                                  /*  change veloc calc when rate != 1 ?????     */
        }
        if ((input_index += rate) >= input_count)
            input_index = 0;                                                             80
    }
    else                                  /*  if control_off or input = setpoint        */
        flag(8, ' ');                     /*  put space in row 0, col 4                 */
  }                                       /*  end of input_update()                     */

/********************************************************************* */
/* Initialize the input module                                          */
/********************************************************************* */
input_setup(type)                                                           input_setup
int type;                                                                           90
{
    int
      i;
    double
      minfreq;

    /* Put the controller is Setpoint mode while we work      */
    input = Setpoint;
    ddprompt("computing trajectory...");
                                                                                   100
    switch (type) {
    case Setpoint:
        pid[0].accel = pid[1].accel = pid[2].accel = 0;
        input_char = 's';
        break;
```

```
case Circle:
    input_char = 'c';
    circle_init(0, isr_tfreq);
    /*NOTE: should be isr_cfreq for non-hierarchical controllers!!!        */          110

    off[0] = input_off[0];                    /* offset      */
    off[1] = input_off[1];
    off[2] = input_off[2];
    mag[0] = mag[1] = circle_radius;          /* magnitude   */
    mag[2] = 0;
    phase[0] = 0;                             /* phase       */
    phase[1] = 90;
    phase[2] = 0;                             /* frequency   */
    freq[0] = freq[1] = freq[2] = circle_freq;                                          120
    ityp[0] = ityp[1] = ityp[2] = 'S';

    /* Generate all the points     */
    input_count = isr_cfreq / circle_freq;
    for (input_index = 0; input_index < input_count; ++input_index)
      {
        double pos[3], acc[3];
        int axis;

        circle(input_index, 3, off, mag, phase, freq, pos, acc);                        130
        for (axis = 0; axis < 3; ++axis)
          {
            traj[input_index].out[axis] = pos[axis];
            traj[input_index].acc[axis] = acc[axis];
          }
      }
    break;

case Box:
    input_char = 'b';                                                                   140
    box_init(0, isr_cfreq);
    off[0] = input_off[0];                    /* offset      */
    off[1] = input_off[1];
    off[2] = input_off[2];
    mag[0] = mag[1] = box_radius;             /* magnitude   */
    mag[2] = 0;
    phase[0] = 0;                             /* phase       */
    phase[1] = -90;
    phase[2] = 0;                             /* frequency   */
    freq[0] = freq[1] = box_freq;                                                       150
    freq[2] = 0;
    ityp[0] = ityp[1] = ityp[2] = 'B';

    /* Generate all the points     */
    input_count = isr_cfreq / box_freq;
    for (input_index = 0; input_index < input_count; ++input_index)
        box(input_index, 3, off, mag, phase, freq,
                traj[input_index].out);
    break;
```

```
                                                                    160
case Figure8:
    input_char = '8';
    circle_init(0, isr_cfreq);
    off[0] = input_off[0];                  /* offset      */
    off[1] = input_off[1];
    off[2] = input_off[2];
    mag[0] = fig8_radius;                   /* magnitude   */
    mag[1] = fig8_radius*2;
    mag[2] = 0;
    phase[0] = 0;                           /* phase       */       170
    phase[1] = 90;
    phase[2] = 0;
    freq[0] = fig8_freq*2;                  /* frequency   */
    freq[1] = fig8_freq;
    freq[2] = 0;
    ityp[0] = ityp[1] = ityp[2] = 'S';

    /* Generate all the points    */
    input_count = isr_cfreq / fig8_freq;
    for (input_index = 0; input_index < input_count; ++input_index)       180
    {
        double pos[3], acc[3];
        int axis;

        circle(input_index, 3, off, mag, phase, freq, pos, acc);
        for (axis = 0; axis < 3; ++axis)
        {
            traj[input_index].out[axis] = pos[axis];
            traj[input_index].acc[axis] = acc[axis];
        }                                                               190
    }
    break;

case Periodic:
    /* Generic input computation; use data from input menu    */
    input_char = 'p';
    periodic_init(0, isr_tfreq);

    /* Figure out what frequency to run    */
    for (minfreq = freq[0], i = 1; i < 3; ++i)                        200
        if (freq[i] < minfreq) minfreq = freq[i];
    if (minfreq <= 0 || (input_count = isr_cfreq / minfreq) > 1024)
      {
        ddprompt("too many points");
        return -1;
      }

    /* Generate all the points    */
    for (input_index = 0; input_index < input_count; ++input_index)
    {                                                                   210
        double pos[3], acc[3];
        int axis;
```

```
            periodic(input_index, 3, off, mag, phase, freq, pos, acc, ityp);
            for (axis = 0; axis < 3; ++axis)
            {
                traj[input_index].out[axis] = pos[axis];
                traj[input_index].acc[axis] = acc[axis];
            }
        }                                                                        220
        break;
    }
    input = type;
    input_index = 0;
    ddprompt("");
}

/****************************************************************** */
```

```
/******************************************************************* */
/*  File:    graph.c                                                 */
/*  routines to graph the actual object position and orientation    */
/*  in real time (also, the desired object trajectory for           */
/*  comparison)                                                      */
/*                                                                   */
/*  created:    RMM 22 March 1988                                    */
/*  modified:   KHO June 1991                                        */
/*                                                                   */
/******************************************************************* */   10

#include <stdio.h>
#include <styx/graf.h>
#include <styx/ddisp.h>
#include <styx/isr.h>
#include "styx.h"
#include "graph.h"

/* Graph variables  */
int g_off[9], g_color[9];                                                   20
double g_scale[9], g_data[9];

/******************************************************************* */
/* Graph the position                                                */
/*  x = object CM's x coordinate                                     */
/*  y = object CM's y coordinate                                     */
/*  t = object's orientation                                         */
/******************************************************************* */
graph()                                                                     graph
{                                                                           30
    /* Go into plotting mode  */
    ddclose();
    plot_open();
    while (!kbhit())
    {
        g_data[0] = x;             g_data[3] = pid[0].desired;
        g_data[1] = y - 21.20;     g_data[4] = pid[1].desired - 21.20;
        g_data[2] = t;             g_data[5] = pid[2].desired;
        plot(9, g_off, g_scale, g_data, g_color, isr_count/8 % 640);
    }                                                                       40
    getch();
    plot_close();
    ddopen();
}

/******************************************************************* */
/*                                                                   */
/*  Graphics routines - use GRAFIX                                   */
/*                                                                   */
/******************************************************************* */   50
static struct g_info inf;

plot_open()                                                                 plot_open
```

```
{
    g_init(2, 1);
    g_open(0);
    g_info(&inf);

    /* Set up parameters */
    /*! These should be initialized in dinit !      */                 60
    g_off[3] = g_off[6] = g_off[0];
    g_off[4] = g_off[7] = g_off[1];
    g_off[5] = g_off[8] = g_off[2];

    g_scale[3] = g_scale[0];
    g_scale[4] = g_scale[1];
    g_scale[5] = g_scale[2];

    /* Make sure the axis lines are on zero     */
    g_data[6] = g_data[7] = g_data[8] = 0;                              70

    /* Plot labels and stuff    */
    plot(9, g_off, g_scale, g_data, g_color, 0);
}

/***************************************************************** */
plot_close()                                                   plot_close
{
    g_close();
}                                                                   80

/***************************************************************** */
/*  Plot data on screen                                              */
/***************************************************************** */
plot(N, off, scale, data, color, col)                                plot
int N, off[], color[], col;
double scale[], data[];
{
    register int i;
    int start, stop;                                                90

    if (col == 0) {
        g_clearall(0);

        /* Draw some labels and stuff  */
        g_writestr(0, 30, "OBJECT POSITION", 15, -1);
        g_writech(off[0]/14 - 2, 0, 'x', 9, -1);
        g_writech(off[1]/14 - 2, 0, 'y', 9, -1);
        g_writech(off[2]/14 - 2, 0, '\351', 9, -1);
        g_writestr(24, 0, "Press any key to return", 7, -1);       100

        /* Legend  */
        g_writestr(24, 59, "zero", color[6], -1);
        g_writestr(24, 65, "desired", color[3], -1);
        g_writestr(24, 74, "actual", color[0], -1);
        return;
```

```
    }

    /* Plot the data  */
    for (i = N-1; i >= 0; --i)                                              110
        g_point(col, (int) (off[i] - scale[i]*data[i]), color[i]);
}

/******************************************************************* */
```

```
/******************************************************************** */
/*  File:    dump.c                                                  */
/*  save and write to a file the actual joint position and applied  */
/*  pulse width information as well as the desired object position  */
/*                                                                   */
/*  created:     RMM 22 March 1988                          b        */
/*  modified:    KHO 25 June 1991                                    */
/*                                                                   */
/******************************************************************** */
                                                                        10
#include <stdio.h>
#include <fcntl.h>
#include <styx/ddisp.h>
#include "styx.h"
#include "dump.h"

/* Internal variables    */
static int
  dump_on = 0,
  dump_offset,                                                          20
  dump_block;

/* Dump tables  */
#define DUMP_SIZE 0x07ff                 /* 2K (integers)  */
static int far dump_actual[DUMP_SIZE][4];
static int far dump_pwm[DUMP_SIZE][4];
static double far dump_desired[DUMP_SIZE][3];

/******************************************************************** */
/*  set up for grabbing data so that it can be saved using dump      */  30
/*  routines                                                         */
/******************************************************************** */
capture()                                                         capture
{
    dump_on = 1;
    dump_offset = 0;
    dump_block = '0';
}

/******************************************************************** */  40
/*  grab the actual data for saving                                  */
/******************************************************************** */
dump_update()                                                 dump_update
{
    register i;

    if (dump_on)
      {
        if (dump_offset % 0x0100 == 0) flag(4, dump_block++);
                                                                        50
        /* State + output  */
        for (i = 0; i < 4; ++i)
        {
```

```
            dump_actual[dump_offset][i] = hctl_actual[i];
            dump_pwm[dump_offset][i] = hctl_pwm[i];
        }

        /* Input  */
        for (i = 0; i < 3; ++i)
            dump_desired[dump_offset][i] = pid[i].desired;                    60

        if (++dump_offset >= DUMP_SIZE)
        {
            /* All done  */
            dump_on = 0;
            flag(4, ' ');
        }
    }
}
                                                                              70
/****************************************************************** */
/*  write the grabbed data to user-specified file                     */
/****************************************************************** */
dump()                                                                  dump
{
    char
      filename[80];

    register int
      i;                                                                      80

    int
      d,
      obs,
      block = '0',
      ival;

    float
      fval;
                                                                              90
    /* Get a filename  */
    ddread("Filename: ", filename, 80);

    if ((d = open(filename, O_RDWR|O_CREAT|O_TRUNC|O_BINARY)) == -1)
    {
        ddprompt("Can't open file for writing");
        return(0);
    }

    ddprompt("Dumping");                                                      100

    /* Our arrays are far pointers so we have to dump a byte at a time    */
    for (obs = 0; obs < DUMP_SIZE; ++obs) {
        if (obs % 0x0100 == 0) flag(6, block++);

        for (i = 0; i < 4; ++i)
```

```
        {
            ival = dump_actual[obs][i]; write(d, &ival, sizeof(int));
            ival = dump_pwm[obs][i]; write(d, &ival, sizeof(int));
        }                                                                  110

        for (i = 0; i < 3; ++i)
        {
            fval = dump_desired[obs][i];
            write(d, &fval, sizeof(float));
        }
    }
    (void) close(d);
    ddprompt("");
    flag(6, ' ');                                                          120
}

/****************************************************************** */
```

```
/****************************************************************** */
/*  File:    table.c                                                */
/*  generate system tables                                          */
/*                                                                  */
/*  created:    RMM 16 March 1988                                   */
/*  modified: KHO 12 August 1991                                    */
/*                                                                  */
/****************************************************************** */

#include <math.h>                                                        10
#include <styx/ddisp.h>
#include "styx.h"

/* Sin/cos tables (initialized by generate_tables()        */
float far lenR1_sin[TBLSIZ], far lenR1_cos[TBLSIZ],
      far lenR2_sin[TBLSIZ], far lenR2_cos[TBLSIZ];
float far lenL1_sin[TBLSIZ], far lenL1_cos[TBLSIZ],
      far lenL2_sin[TBLSIZ], far lenL2_cos[TBLSIZ];

#define TWOPI 6.2831852                                                  20

/****************************************************************** */
/*  generate trig function look-up tables to speed up real          */
/*  time routines                                                   */
/****************************************************************** */
generate_tables()                                          generate_tables
{
    register int
      angle;
    double                                                               30
      a, s, c;

    for (angle = 0; angle < TBLSIZ; ++angle)
    {
        a = (angle - TBLOFF) * TWOPI / TBLOFF;
        if (angle % 100 == 0) printf("\r%d %+f", angle, a);

        s = sin(a);
        /*  R finger = finger with strain gauges          */
        lenR1_sin[angle] = s * LenR1;                                    40
        lenR2_sin[angle] = s * LenR2;
        /*  L finger = finger without strain gauges       */
        lenL1_sin[angle] = s * LenL1;
        lenL2_sin[angle] = s * LenL2;

        c = cos(a);
        /*  R finger = finger with strain gauges          */
        lenR1_cos[angle] = c * LenR1;
        lenR2_cos[angle] = c * LenR2;
        /*  L finger = finger without strain gauges       */               50
        lenL1_cos[angle] = c * LenL1;
        lenL2_cos[angle] = c * LenL2;
    }
```

```
    printf("\r              \r");
}

/****************************************************************** */
```

```
/******************************************************************* */
/*  File:    calstyx.c                                                */
/*  program created to calibrate styx by driving the links to        */
/*  their calibration positions at the calibration post and          */
/*  then zeroing the position registers                              */
/*                                                                   */
/*  created:     KHO 16 June 1991                                    */
/*  modified:    KHO 8 August 1991                                   */
/*                                                                   */
/******************************************************************* */   10

#include <stdio.h>
#include <styx/hctl.h>

/******************************************************************* */
/*  function to perform Styx calibration procedure;                  */
/*  moves joints to calibration position and zeros the               */
/*  position counters                                                */
/******************************************************************* */
calibrate()                                                               calibrate
{                                                                         21
    long
        position[4];        /*  value returned by cal_read_pos()     */

/******************************************************************* */
/*  CALIBRATE FINGER WITH STRAIN GAUGES                              */
/*  = R finger                                                       */
/******************************************************************* */
    while(kbhit()) getch();                    /*  empty keyboard buffer    */
                                                                          30
    printf("Calibrate finger WITH STRAIN GAUGES\n");
    printf("Make sure ALL CABLES ARE CLEAR of finger path \n");
    printf("Done?   (y/n) \n");
    if (getch() == 'y')
        {
            hctl_wrreg(0, HCTL_PWM, -30);    /*  apply -30 pw torque  */
            hctl_wrreg(1, HCTL_PWM, -30);

            cal_read_pos(position);
            printf("%ld          %ld\n", position[0], position[1]);      40
            getch();

            hctl_wrreg(0, HCTL_ACTUAL-1, 0); /* zero the pos'tn reg's    */
            hctl_wrreg(1, HCTL_ACTUAL-1, 0);
            printf("(position registers should now be zero) \n");
            cal_read_pos(position);
            printf("%ld          %ld\n", position[0], position[1]);
            getch();
        }                        /*  end of if (ch == y)   */
                                                                          50
    /*  apply a zero torque    */
    hctl_wrreg(0, HCTL_PWM, 0);
    hctl_wrreg(1, HCTL_PWM, 0);
```

```
    while(kbhit()) getch();                     /*  empty keyboard buffer    */

    printf("Calibrate finger WITHOUT STRAIN GAUGES\n");
    printf("Make sure ALL CABLES ARE CLEAR of finger path \n");
    printf("Done?   (y/n) \n");
    if (getch() == 'y')                                                          60
        {
            hctl_wrreg(2, HCTL_PWM, 30);        /*  apply 30 pw torque   */
            hctl_wrreg(3, HCTL_PWM, -30);

            cal_read_pos(position);
            printf("%ld        %ld\n", position[2], position[3]);
            getch();

            hctl_wrreg(2, HCTL_ACTUAL-1, 0); /* zero the pos'tn reg's    */
            hctl_wrreg(3, HCTL_ACTUAL-1, 0);                                     70
            printf("(position registers should now be zero) \n");
            cal_read_pos(position);
            printf("%ld           %ld\n", position[2], position[3]);
            getch();
        }                                       /*  end of if (ch == y)  */

    /*  apply a zero torque    */
    hctl_wrreg(2, HCTL_PWM, 0);
    hctl_wrreg(3, HCTL_PWM, 0);
                                                                                 80
    while(kbhit()) getch();                     /*  empty keyboard buffer    */

}                                               /*  end of calibrate()       */

/*************************************************************** */
/*  function to continuously (until a key is pressed)            */
/*  read the joint position counters and display the             */
/*  values on the screen                                         */
/*************************************************************** */
cal_read_pos(pos_count)                                             cal_read_pos
    long                                                                         91
        pos_count[4];
{
    while (kbhit()) getch();
    while (!kbhit())
    {
        pos_count[0] = hctl_rdpos(0, HCTL_ACTUAL);
        pos_count[1] = hctl_rdpos(1, HCTL_ACTUAL);
        pos_count[2] = hctl_rdpos(2, HCTL_ACTUAL);
        pos_count[3] = hctl_rdpos(3, HCTL_ACTUAL);                               100
    }

}                                                 /*  end of cal_read_pos()      */
/*************************************************************** */
```

```
/****************************************************************** */
/*  The file all.h is the concatenation of all the *.h files in     */
/*  the Styx source code.                                           */
/****************************************************************** */
/*  dump.h                                                          */
/****************************************************************** */
extern int capture(), dump(), dump_update();

/****************************************************************** */
/*  dynamics.h                                                      */          10
/****************************************************************** */
#define Ji00  -l12_s12
#define Ji10  x_1
#define Ji20  -l22_s12
#define Ji30  x_2

#define Ji01  -l12_c12
#define Ji11  y_1
#define Ji21  -l22_c12
#define Ji31  y_2                                                               20

#define Ji02  (dy_div2 * l12_s12 - dx_div2 * l12_c12)                          Ji02
#define Ji12  (dx_div2 * y_1 - dy_div2 * x_1)                                  Ji12
#define Ji22  (dx_div2 * l22_c12 - dy_div2 * l22_s12)                          Ji22
#define Ji32  (dy_div2 * x_2 - dx_div2 * y_2)                                  Ji32

#define Jt00 mdiv2 * y_1
#define Jt10 mdiv2 * l12_c12
#define Jt20 mdiv2 * y_2
#define Jt30 mdiv2 * l22_c12                                                    30

#define Jt01  -mdiv2 * x_1
#define Jt11  -mdiv2 * l12_s12
#define Jt21  -mdiv2 * x_2
#define Jt31  -mdiv2 * l22_s12

#define Jt02  (-dyi_div_d2 * y_1 - dxi_div_d2 * x_1)                           Jt02
#define Jt12  (-dxi_div_d2 * l12_s12 - dyi_div_d2 * l12_c12)                   Jt12
#define Jt22  (dyi_div_d2 * y_2 + dxi_div_d2 * x_2)                            Jt22
#define Jt32  (dxi_div_d2 * l22_s12 + dyi_div_d2 * l22_c12)                    Jt32
                                                                                41
#define Jt03 -(dx*y_1     - dy*x_1)     * pid[3].K
#define Jt13 -(dx*l12_c12 - dy*l12_s12) * pid[3].K
#define Jt23 -(dy*x_2     - dx*y_2)     * pid[3].K
#define Jt33 -(dy*l22_s12 - dx*l22_c12) * pid[3].K

/****************************************************************** */
/*  global.h                                                        */
/****************************************************************** */
unsigned int cfreq = 100;          /*  Control frequency        */              50
unsigned int tfreq = 10;           /*  Trajector frequency      */
```

```
double LenR1, LenR2, LenL1, LenL2;      /* Styx link parameters       */
double BASELINE;                        /* distance between fingers   */
double Mo, Io;                          /* object mass/inertia        */
double object_radius;                   /* object length              */
double MIN_FACTOR, HIT_FACTOR;          /* "normalized" motor gains   */

int RES11, RES12, RES21, RES22;         /* encoder resolutions        */
int NUM11, DEN11, NUM21, DEN21;         /* encoder conversion factors */    60
int NUM12, DEN12, NUM22, DEN22;

struct  styx_parameters styx[4];        /* Styx parameter table       */
struct  pid_structure pid[4];           /* common PID parameters      */

/* Variables used by all control algorithms     */
long    hctl_actual[HCTL_NMOTORS];      /* current position           */

/*   changed hctl_pwm on 19 June 91  */
int     hctl_pwm[HCTL_NMOTORS];         /* output pulse width         */    70
int     ipwm[STYX_NMOTORS];             /* integer (2 bytes) pulse width   */
int     th1_1, th1_12, th2_1, th2_12;   /* finger angles              */

/*  changed on August 12 1991 in R/L finger clarification project...     */
double lenR1_s1, lenR1_c1, lenR2_s12, lenR2_c12;      /* R finger sin/cos  */
double lenL1_s1, lenL1_c1, lenL2_s12, lenL2_c12;      /* L finger sin/cos  */

double x_1, y_1, x_2, y_2;                      /* Cartesian positions     */
double x, y, dx, dy, t, r, d, d_squared;        /* object position         */
                                                                            80
/* Control variables    */
int     control_on = 0;                         /*  controller on          */

/* Supervisory variables    */
DD_IDENT *menutbl[NMENUS];
/**************************************************************** */
/ *  graph.h - definitions and declarations for graphing routines      */
/**************************************************************** */
extern int g_off[], g_color[];
extern double g_scale[], g_data[];                                          90

/**************************************************************** */
/*  hctl_loc.h                                                      */
/**************************************************************** */
/*  define program counter modes; numbers come from hctl-1000 specs    */
#define RESET 0
#define INIT 1
#define ALIGN 2
#define CONTROL 3
                                                                            100
/*  define flags to clear or set control mode flags          */
#define CLEAR 0

/*  define hctl control frequencies         */
```

```
#define HCTLPRIFREQ 5

/****************************************************************** */
/*  input.h                                                         */
/****************************************************************** */
extern int input, input_char;                                                110

extern double input_off[];
extern double circle_radius, circle_freq;
extern double fig8_radius, fig8_freq;
extern double box_radius, box_freq;

extern DD_IDENT input_menu[];

/****************************************************************** */
/ * stiff.h - definitions and declarations for using hctl control   */      120
/****************************************************************** */
extern long
    Command_Pos[4];

extern int
    Fake_Control,       /*  test controller; no motor output     */
    control_law,        /*  which high level control             */
    high_freq,          /*  high level control freq              */
    hctl_freq,          /*  hctl control frequency               */
    high_only,          /*  high level controller only           */      130
    Gains[4];           /*  hctl control gains                   */

extern double
    K_p,                /*  position error gain                  */
    K_v;                /*  vel intrp/vel error gain             */

extern DD_IDENT stiff_menu[];

/****************************************************************** */
/* styx.h - definitions and declarations used by all styx programs  */      140
/****************************************************************** */
/*  constant parts of inertia matrix of fingers, M(theta)      */
/*  NOTE: variable names are chosen to match those in RMM's
          MS report                                              */
/*  R finger  */
#define A1R 84565
#define A2R 1.74
#define B1R 1009
#define B2R 4152
/*  L finger  */                                                             150
#define A1L 85300
#define A2L 1.74
#define B1L 1110
#define B2L 4690

/*  make the following NUM, DEN definitions to avoid future
    confusion between R and L fingers, finger 1 and finger 2, ...  */
```

```
#define NUMR1 NUM11 /* numerator for proximal R finger conversion    */
#define NUMR2 NUM12
#define NUML1 NUM21 /* numerator for proximal L finger conversion    */      160
#define NUML2 NUM22

#define DENR1 DEN11 /* denominator for prox'l R finger conversion    */
#define DENR2 DEN12
#define DENL1 DEN21 /* denominator for prox'l L finger conversion    */
#define DENL2 DEN22

/*  control laws for high level controller        */
#define SETPOINT            0
#define JOINT_INTERPOLATION 1                                                170
#define OBJECT_CORRECTION 2
#define COMP_TORQUE         3
#define HIGH_ONLY           4

#define STYX_NMOTORS 4 /* number of motors used           */

/*! These should be dynamic so that we can change encoder resolution !    */
#define TBLSIZ 7200        /* 360 degrees in each direction     */
#define TBLOFF 3600
                                                                             180
/* Styx parameter structure   */
/* Parameter table  */
struct styx_parameters {
    double gain;            /*  gain for this motor                 */
    unsigned limit;         /*  pulse width limit for this motor    */
    unsigned offset;        /*  pulse width offset for this motor   */
    int reference;          /*  encoder reference point             */
    int res;                /*  encoder resolution (RES??)          */
    int vmax;               /*  voltage supply                      */
};                                                                           190

struct pid_structure {      /*  PID table                           */
    double K, Kz, Kp;       /*  high level control gains            */
    double last_error;      /*  values from last iteration          */
    double desired;         /*  desired obj position                */
    double accel;           /*  desired obj acceler'tn              */
    double error;           /*  current error                       */
    double output;          /*  output force                        */
    double veloc;           /*  desired object velocity             */
    double next;            /*  next desired object position        */      200
};


/* Global declarations    */
/*  constant parts of 2x2 finger inertia matrices       */
double MR00, MR01, MR11;          /*  R finger   */
double ML00, ML01, ML11;          /*  L finger   */

extern unsigned int cfreq;
extern unsigned int tfreq;                                                   210
```

```
extern double LenR1, LenR2, LenL1, LenL2, BASELINE;
extern double object_radius;
extern int RES11, RES12, RES21, RES22;
extern int NUM11, DEN11, NUM21, DEN21;
extern int NUM12, DEN12, NUM22, DEN22;
extern double Mo, Io;                      /*  obj mass and inertia      */
extern double MIN_FACTOR, HIT_FACTOR; /* for motor gains    */
extern struct styx_parameters styx[];
extern struct pid_structure pid[];

extern long    hctl_actual[];              /*  modified 25 June 1991   */
extern int     hctl_pwm[];                 /*  modified 19 June 1991   */
extern int     ipwm[];
extern int     th1_1, th1_12, th2_1, th2_12;

/*  changed 12 August 1991 -- R/L finger ...   */
extern double lenR1_s1, lenR1_c1, lenR2_s12, lenR2_c12;
extern double lenL1_s1, lenL1_c1, lenL2_s12, lenL2_c12;

/*  object    and fingertip stuff      */
extern double x_1, y_1, x_2, y_2;
extern double x, y, dx, dy, t, d, d_squared;

extern int control_on;

/* Menu options  */
enum options {
    /* Menus  */
    Watch = 256, InputParm, Parm,

    /* Actions  */
    OnOff, Help, Quit, Graph, Capture, Dump, Cal,

    /* Inputs  */
    Setpoint, Circle, Box, Demo, Figure8, FigureD, Periodic,

    /* Controllers and returns (from sub-menus)  */
    Joint, JointRet, Stiff, StiffRet,
    Full, FullRet, ControlRet, Opspace, OpspaceRet,

    None
};

#include <styx/ddisp.h>

extern DD_IDENT *menutbl[];
#define NMENUS 3

/* Redefine the interrupt routine names (Feb 91)    */
#define isr_set_cfreq               isr_pri_freq
#define isr_set_tfreq               isr_sec_freq
#define isr_set_croutine            isr_pri_routine
#define isr_set_troutine            isr_sec_routine
```

```
/* Define a macro for displaying a flag    */
#define flag(offset, val) *(char far *)((long) 0xB8000000 + offset) = val

/*************************************************************** */
```

```
/***************************************************************** */
/*  The file all.tbl is the concatenation of all the *.tbl files   */
/*  in the Styx source code.                                       */
/***************************************************************** */
/*  input.tbl                                                      */
/*  some definitions for making display tables                     */
/***************************************************************** */
#define ADDR(v) (char *)&v

char input_buffer[512];          /* Temporary buffer for ddisp   */          10
#define buffer input_buffer

/* Main menu for input.c  */
DD_IDENT input_menu[] = {
    DD_Label(0, 30, "INPUT parameters"),
    {0, 75, ADDR(isr_count), dd_integer, "%5u", buffer, 1},
    {1, 75, ADDR(isr_tack), dd_integer, "%5u", buffer+2, 1},

    DD_Label(2, 0, "----- circle -----"),
    DD_Label(3, 2, "radius:"),                                                    20
    DD_Label(4, 2, "frequency:"),
    {3, 14, ADDR(circle_radius), dd_double, "%8.2f", buffer+200, 1},
    {4, 14, ADDR(circle_freq), dd_double, "%8.2f", buffer+210, 1},

    DD_Label(2, 24, "------ box ------"),
    DD_Label(3, 26, "radius:"),
    DD_Label(4, 26, "frequency:"),
    {3, 38, ADDR(box_radius), dd_double, "%8.2f", buffer+220, 1},
    {4, 38, ADDR(box_freq), dd_double, "%8.2f", buffer+230, 1},
                                                                                  30
    DD_Label(2, 48, "---- figure8 ----"),
    DD_Label(3, 50, "radius:"),
    DD_Label(4, 50, "frequency:"),
    {3, 62, ADDR(fig8_radius), dd_double, "%8.2f", buffer+240, 1},
    {4, 62, ADDR(fig8_freq), dd_double, "%8.2f", buffer+250, 1},

    DD_Label(10, 0, "Axis    Offset Mag    Phase   Freq   Type"),
    DD_Label(11, 0, "----    ------ ---    -----   ----   ----"),

    DD_Label(12, 2, "x"),                                                         40
    {12,  8, ADDR(off[0]), dd_double, "%6.2f", buffer+10, 1},
    {12, 14, ADDR(mag[0]), dd_double, "%6.2f", buffer+20, 1},
    {12, 22, ADDR(phase[0]), dd_double, "%6.2f", buffer+30, 1},
    {12, 30, ADDR(freq[0]), dd_double, "%6.2f", buffer+40, 1},
    {12, 42, ADDR(ityp[0]), dd_char, " %c", buffer+48, 1},

    DD_Label(13, 2, "y"),
    {13,  8, ADDR(off[1]), dd_double, "%6.2f", buffer+70, 1},
    {13, 14, ADDR(mag[1]), dd_double, "%6.2f", buffer+80, 1},
    {13, 22, ADDR(phase[1]), dd_double, "%6.2f", buffer+90, 1},                   50
    {13, 30, ADDR(freq[1]), dd_double, "%6.2f", buffer+100, 1},
    {13, 42, ADDR(ityp[1]), dd_char, " %c", buffer+108, 1},
```

```
    DD_Label(14, 2, "\351"),
    {14,  8, ADDR(off[2]), dd_double, "%6.2f", buffer+130, 1},
    {14, 14, ADDR(mag[2]), dd_double, "%6.2f", buffer+140, 1},
    {14, 22, ADDR(phase[2]), dd_double, "%6.2f", buffer+150, 1},
    {14, 30, ADDR(freq[2]), dd_double, "%6.2f", buffer+160, 1},
    {14, 42, ADDR(ityp[2]), dd_char, " %c", buffer+168, 1},
                                                                          60
    DD_Label(16, 0, "Input [        ,       ,       ]"),
    DD_Option(17, 4, "setpoint", Setpoint), DD_Option(18, 4, "circle", Circle),
    DD_Option(17, 18, "figure8", Figure8), DD_Option(18, 18, "figureD", FigureD),
    DD_Option(19, 4, "box", Box), DD_Option(19, 18, "Periodic", Periodic),

    {16, 8,  ADDR(pid[0].desired), dd_double, "%5.2f", buffer+130, 1},
    {16, 15, ADDR(pid[1].desired), dd_double, "%5.2f", buffer+140, 1},
    {16, 21, ADDR(pid[2].desired), dd_double, "%5.2f", buffer+150, 1},

    DD_Option(21,  0, "ON/OFF", OnOff), DD_Option(21, 9, "RETN", ControlRet),       70
    DD_Option(21, 16, "GRAPH", Graph), DD_Option(21, 24, "WATCH", Watch),
    DD_Option(21, 32, "INPUT", InputParm), DD_Option(21, 40, "PARM", Parm),
    DD_Option(21, 47, "CAPTURE", Capture), DD_Option(21, 57, "DUMP", Dump),
    DD_Option(21, 64, "SCRIPT", None),

    DD_Label(23, 0,  "move cursor: \030\032\031\033"),
    DD_Label(23, 26, "select option: <Enter>"),
    DD_Label(23, 52, "enter value: ="),
    DD_End
};                                                                        80

/******************************************************************** */
/*  stiff.tbl                                                         */
/*  NOTE: the name stiff is a remnant of the old days of Styx and     */
/*  is no longer appropriate                                          */
/*  display tables for hierarchical controller                        */
/******************************************************************** */
#include <styx/ddisp.h>   /*  contains display functions    */

/* Some definitions for making display tables     */                      90
#define ADDR(v) (char *)&v

extern int
    control_law,        /*  high level controller chosen by user       */
    high_control(),     /*  secondary interrupt service routine        */
    hctl_control(),     /*  primary interrupt service routine          */
    high_freq,          /*  frequency of high level controller         */
    hctl_freq,          /*  frequency of interrupt service routine     */
    high_only,          /*  high level CT control only?                */
    K_a;                /*  obj_corr "gain"--use obj mass?   = 0, 1    */     100

extern long
    mag_force_N,        /*  magnitude of grasping force     */
    Command_Pos[4],     /*  desired joint angle pos'tn      */
    Command_Vel[4];     /*  desired joint ang'r velocity    */
```

```
extern double
    Dist,              /* distance between R,L ftips      */
    force_N[4],        /* internal force                  */
    torque_N[4],       /* torque for internal force       */          110
    Mhand[3][3],       /* inertia matrix of hand          */
    ftip[4],           /* applied fingertip force         */
    Ctorque[4],        /* pw written out to motors (high_only)     */
    Jangle[4],         /* joint angles                    */
    Expec_Pos[4],      /* position "correction"           */
    IntF_Pos[4],       /* pos added to C_Pos for int f    */
    K_v,               /* high level veloc error gain     */
    Kv_int,            /* add velocity interpolation      */
    K_p,               /* high level posit error gain     */
    t;                 /* object orientation              */          120

char
    hctl_title[] = "HCTL Position Controller",
    high_title[] = "High Level Controller",
    stiff_title[] = "Hierarchical Control";

int
    Pole[4],           /* HCTL control parameters         */
    Zero[4],
    Gains[4];                                                          130

static char
        buffer[512];   /* Temporary buffer for ddisp      */

/*************************************************************** */
/* Main menu                                                     */
/*************************************************************** */
DD_IDENT stiff_menu[] = {
    DD_Label(0, 24, stiff_title),
    DD_Label(2, 5, high_title),                                        140
    DD_Label(2, 43, hctl_title),
    {0, 75, ADDR(isr_count), dd_integer, "%5u", buffer, 0},
    {1, 75, ADDR(isr_tack), dd_integer, "%5u", buffer+2, 0},

    DD_Option(21,  0, "ON/OFF", OnOff), DD_Option(21, 9, "QUIT", Quit),
    DD_Option(21, 16, "GRAPH", Graph), DD_Option(21, 24, "WATCH", Watch),
    DD_Option(21, 32, "INPUT", InputParm), DD_Option(21, 40, "PARM", Parm),
    DD_Option(21, 47, "CAPTURE", Capture), DD_Option(21, 57, "DUMP", Dump),
    DD_Option(21, 64, "SCRIPT", None),
                                                                       150
    DD_Label(4, 2, "Interrupt frequency"),
    DD_Label(5, 2, "High Level"),
    {5, 14, ADDR(high_freq), dd_integer, "%3u", buffer+4, 1},
    DD_Label(5, 40, "HCTL"),
    {5, 46, ADDR(hctl_freq), dd_integer, "%3u", buffer+6, 1},

    DD_Label(5, 61, "High_CT?"),
    {5, 71, ADDR(high_only), dd_integer, "%1u", buffer+400, 1},
```

```
DD_Label(7, 2, "Gain--K_v"),                                                      160
{7, 12, ADDR(K_v), dd_double, "%2.2lf", buffer+10, 1},
DD_Label(7, 17, "K_p"),
{7, 21, ADDR(K_p), dd_double, "%2.2lf", buffer+20, 1},
DD_Label(7, 26, "K_a"),
{7, 30, ADDR(K_a), dd_integer, "%1d", buffer+30, 1},
DD_Label(8, 8, "K_int"),
{8, 14, ADDR(Kv_int), dd_double, "%2.2lf", buffer+410, 1},

DD_Label(9, 2, "High Control"),
{9, 20, ADDR(control_law), dd_integer, "%3d", buffer+34, 1},                      170

DD_Label(10, 2, "Internal Force"),
{10, 20, ADDR(mag_force_N), dd_long, "%1d    ", buffer+40, 1},

DD_Label(13, 2, "Pulse-Widths"),
{14, 2,  ADDR(Ctorque[0]), dd_double, "%3.2lf ", buffer+60, 0},
{14, 12, ADDR(Ctorque[1]), dd_double, "%3.2lf ", buffer+70, 0},
{14, 22, ADDR(Ctorque[2]), dd_double, "%3.2lf ", buffer+80, 0},
{14, 32, ADDR(Ctorque[3]), dd_double, "%3.2lf ", buffer+90, 0},
                                                                                  180
DD_Label(7, 40, "Joint      0      1      2      3"),
DD_Label(8, 40, "Gains"),
{8, 47, (char *)((0 << 8) | HCTL_GAIN), ddhctl_writeonly, "%5u", ADDR(Gains[0]), 1},
{8, 54, (char *)((1 << 8) | HCTL_GAIN), ddhctl_writeonly, "%5u", ADDR(Gains[1]), 1},
{8, 61, (char *)((2 << 8) | HCTL_GAIN), ddhctl_writeonly, "%5u", ADDR(Gains[2]), 1},
{8, 68, (char *)((3 << 8) | HCTL_GAIN), ddhctl_writeonly, "%5u", ADDR(Gains[3]), 1},

DD_Label(9, 40, "Poles"),
{9, 47, (char *)((0 << 8) | HCTL_POLE), ddhctl_writeonly, "%5u", ADDR(Pole[0]), 1},
{9, 54, (char *)((1 << 8) | HCTL_POLE), ddhctl_writeonly, "%5u", ADDR(Pole[1]), 1},   190
{9, 61, (char *)((2 << 8) | HCTL_POLE), ddhctl_writeonly, "%5u", ADDR(Pole[2]), 1},
{9, 68, (char *)((3 << 8) | HCTL_POLE), ddhctl_writeonly, "%5u", ADDR(Pole[3]), 1},

DD_Label(10, 40, "Zeros"),
{10, 47, (char *)((0 << 8) | HCTL_ZERO), ddhctl_writeonly, "%5u", ADDR(Zero[0]), 1},
{10, 54, (char *)((1 << 8) | HCTL_ZERO), ddhctl_writeonly, "%5u", ADDR(Zero[1]), 1},
{10, 61, (char *)((2 << 8) | HCTL_ZERO), ddhctl_writeonly, "%5u", ADDR(Zero[2]), 1},
{10, 68, (char *)((3 << 8) | HCTL_ZERO), ddhctl_writeonly, "%5u", ADDR(Zero[3]), 1},

DD_Label(12, 40, "Ref Pos"),                                                      200
{12, 47, ADDR(Command_Pos[0]), dd_long, "%5ld", buffer+180, 1},
{12, 54, ADDR(Command_Pos[1]), dd_long, "%5ld", buffer+184, 1},
{12, 61, ADDR(Command_Pos[2]), dd_long, "%5ld", buffer+188, 1},
{12, 68, ADDR(Command_Pos[3]), dd_long, "%5ld", buffer+192, 1},

DD_Label(13, 40, "Act Pos"),
{13, 47, ADDR(hctl_actual[0]), dd_long, "%5ld", buffer+196, 1},
{13, 54, ADDR(hctl_actual[1]), dd_long, "%5ld", buffer+200, 1},
{13, 61, ADDR(hctl_actual[2]), dd_long, "%5ld", buffer+204, 1},
{13, 68, ADDR(hctl_actual[3]), dd_long, "%5ld", buffer+208, 1},                   210

DD_Label(15, 0, "Input [      ,      ,      ]"),
```

```
    DD_Label(17, 0, "Trajectory"),
    DD_Option(18, 4, "hold", Setpoint),
    DD_Option(19, 4, "circle", Circle),

    {15, 8,   ADDR(pid[0].desired), dd_double, "%5.2f", buffer+90, 1},
    {15, 15, ADDR(pid[1].desired), dd_double, "%5.2f", buffer+100, 1},
    {15, 22, ADDR(pid[2].desired), dd_double, "%5.2f", buffer+110, 1},                220

    DD_Label(17, 40, "Control"),
    DD_Option(18, 43,  "stiff", Stiff),

    DD_Label(17, 55, "Dist"),
    {17, 61,   ADDR(Dist), dd_double, "%2.31f ", buffer+50, 0},

    DD_Label(23, 0,  "move cursor: \030\032\031\033"),
    DD_Label(23, 26, "select option: <Enter>"),
    DD_Label(23, 52, "enter value: ="),                                              230
    DD_End
};

/******************************************************************* */
/* Status menu                                                       */
/******************************************************************* */
DD_IDENT stiff_watch[] = {
    DD_Label(0, 30, "stiff status"),
    {0, 75, ADDR(isr_count), dd_integer, "%5u", buffer, 0},
                                                                                      240
    DD_Label(2,  0, "Control frequency:"),
    {2, 20, ADDR(hctl_freq), dd_integer, "%3u", buffer+6, 0},

    DD_Label(4,  0, "Joint angles:"),
    {4, 16, ADDR(hctl_actual[0]), dd_long, "%5ld", buffer+156, 0},
    {4, 21, ADDR(hctl_actual[1]), dd_long, "%5ld", buffer+160, 0},
    {4, 26, ADDR(hctl_actual[2]), dd_long, "%5ld", buffer+164, 0},
    {4, 31, ADDR(hctl_actual[3]), dd_long, "%5ld", buffer+168, 0},

    DD_Option(21,  0, "ON/OFF", OnOff), DD_Option(21, 9, "RETN", StiffRet),          250
    DD_Option(21, 16, "GRAPH", Graph), DD_Option(21, 24, "WATCH", Watch),
    DD_Option(21, 32, "INPUT", InputParm), DD_Option(21, 40, "PARM", Parm),
    DD_Option(21, 47, "CAPTURE", Capture), DD_Option(21, 57, "DUMP", Dump),
    DD_Option(21, 64, "SCRIPT", None),

    DD_Label(23, 0,  "move cursor: \030\032\031\033"),
    DD_Label(23, 26, "select option: <Enter>"),
    DD_Label(23, 52, "enter value: ="),
    DD_End
};                                                                                    260

/******************************************************************* */
/*  styx.tbl                                                         */
/******************************************************************* */
/* Initialization table      */
```

```
D_IDENT inittbl[] = {
    /* Styx parameters  */
    D_Double(BASELINE),
    D_Double(LenR1),  D_Double(LenR2),  D_Double(LenL1),  D_Double(LenL2),
    D_Integer(RES11), D_Integer(RES12), D_Integer(RES21), D_Integer(RES22),          270
    D_Integer(NUM11), D_Integer(DEN11), D_Integer(NUM21), D_Integer(DEN21),
    D_Integer(NUM12), D_Integer(DEN12), D_Integer(NUM22), D_Integer(DEN22),

    D_Double(styx[0].gain), D_Integer(styx[0].limit),
    D_Integer(styx[0].reference), D_Integer(styx[0].offset),
    D_Integer(styx[0].res), D_Integer(styx[0].vmax),

    D_Double(styx[1].gain), D_Integer(styx[1].limit),
    D_Integer(styx[1].reference), D_Integer(styx[1].offset),
    D_Integer(styx[1].res), D_Integer(styx[1].vmax),                                  280

    D_Double(styx[2].gain), D_Integer(styx[2].limit),
    D_Integer(styx[2].reference), D_Integer(styx[2].offset),
    D_Integer(styx[2].res), D_Integer(styx[2].vmax),

    D_Double(styx[3].gain), D_Integer(styx[3].limit),
    D_Integer(styx[3].reference), D_Integer(styx[3].offset),
    D_Integer(styx[3].res), D_Integer(styx[3].vmax),

    /* Default location    */                                                         290
    D_Double(pid[0].desired), D_Double(pid[1].desired),
    D_Double(pid[2].desired), D_Double(pid[3].desired),

    /* Graph parameters (graph.c)  */
    D_Integer(g_off[0]),        D_Integer(g_off[1]),        D_Integer(g_off[2]),
    D_Double(g_scale[0]),       D_Double(g_scale[1]),       D_Double(g_scale[2]),
    D_Integer(g_color[0]), D_Integer(g_color[1]), D_Integer(g_color[2]),
    D_Integer(g_color[3]), D_Integer(g_color[4]), D_Integer(g_color[5]),
    D_Integer(g_color[6]), D_Integer(g_color[7]), D_Integer(g_color[8]),
                                                                                      300
    /* Input parameters (input.c)   */
    D_Double(input_off[0]), D_Double(input_off[1]), D_Double(input_off[2]),
    D_Double(circle_radius), D_Double(circle_freq),
    D_Double(fig8_radius), D_Double(fig8_freq),
    D_Double(box_radius), D_Double(box_freq),

    /* Object parameters (object.c)    */
    D_Double(Mo), D_Double(Io), D_Double(object_radius),

    /*  Motor parameters  */                                                          310
    D_Double(MIN_FACTOR), D_Double(HIT_FACTOR),

    /* hctl-1000 parameters  */
    D_Integer(high_only),            /*  high level CT control only     */
    D_Integer(Fake_Control),         /*  don't write to chips           */
    D_Integer(control_law),          /*  which high level controller?   */
    D_Integer(hctl_freq),       D_Integer(high_freq),
    D_Double(K_v),              D_Double(K_p),
```

```
    D_End                                                                      320
};

/* Main menu  */
DD_IDENT styx_menu[] = {
    DD_Label(0, 30, "*** STYX ***"),
    DD_Label(2, 0, "Select option:"),
    DD_Option(4, 5, "cal", Cal),        DD_Label(4, 18, "- calibrate STYX"),
    DD_Option(5, 5, "stiff", Stiff),       DD_Label(5, 18, "- stiffness control law"),
    DD_Option(6, 5, "quit", Quit),        DD_Label(6, 18, "- exit demonstration program"),
                                                                               330
    DD_Label(15, 0,  "move cursor: \030\032\031\033"),
    DD_Label(15, 26, "select option: <Enter>"),
    DD_Label(15, 52, "enter value: ="),
    DD_End
};

/****************************************************************** */

                                                                               340
```

```
##################################################
The file all.def is the concatenation of all the *.def files in the
Styx source code.

##################################################
# beam.def file for object = "beam"
Mo              245     # object mass in g
Io              3.0e3   # g-cm^2
object_radius   12      # obj "length"; not really a "radius"
                                                                    10
##################################################
# kodak.def file for object = inside half of kodak disk box
# object = "box" in papers
Mo              33        # object mass in g
Io              1.36e3    # (Mo/12)*(len*len + wid*wid); len = 14.20 cm
object_radius   17.13     # box "width"; not really a "radius"

##################################################
# fakeoff.def file for fake controller OFF
Fake_Control 0                                                      20

##################################################
# fakeon.def file for fake controller ON
Fake_Control 1

##################################################
# graph.def file for initialization of graphics plot
g_off[0]      74                # Axis offsets
g_off[1]      180
g_off[2]      286                                                   30
g_scale[0]    8                 # X, Y axis scaling
g_scale[1]    8
g_scale[2]    100               # Theta axis scaling

g_color[0]    15
g_color[1]    15
g_color[2]    15
g_color[3]    12
g_color[4]    12
g_color[5]    12                                                    40
g_color[6]    7
g_color[7]    7
g_color[8]    7

##################################################
# input.def for default input parameters
# offsets for object CM position/orientation
input_off[0]    1.30          # x offset from midpt between fingers; cm
input_off[1]    21.20         # y offset from fingertip bases; cm
input_off[2]    0             # orientation offset of object        50

# object trajectory defaults
circle_radius   2.5           # cm
```

```
circle_freq         0.25          # Hz

##################################################
# stiff.def file for default parameters for hierarchical control
# note that the name "stiff" is a remnant of the old version of
# Styx and has no real meaning currently
hctl_freq        100        # control frequency of hctl position control          60
high_freq         10        # control frequency of high level controller

K_v                0        # no joint velocity interpolation/vel err gain
K_p                0        # no position error gain

Fake_Control       0        # not in fake control mode
control_law        0        # computed torque
high_only          0        # start with hierarchical (or low level) control,
                            # not the test case of high level only
                                                                                  70
##################################################
# file styx.def
# link lengths
LenR1 15.3      # Length of prox'l link of R finger (cm)                      finger
LenR2 11.8      # Length of distal link of R finger - to outer screw hole
LenL1 15.3      # Length of prox'l link of L finger (cm)                      finger
LenL2 12.16     # Length of distal link of L finger – to outer screw hole

BASELINE 20.0          # Baseline distance (cm)                             distance
                                                                                  80
# Number of encoder counts per revolution
RES11 3600             # Encoder resolution, link 1, finger 1
RES12 2000             # Encoder resolution, link 2, finger 1
RES21 3600             # Encoder resolution, link 1, finger 2
RES22 2000             # Encoder resolution, link 2, finger 2

# Set up encoders for right–handed coordinate system
# Normalize encoders to proximal link encoders (counts/revolution)          encoders
NUM11 1                # R finger
DEN11 1                                                                           90
NUM12 –9
DEN12 5
NUM21 1                # L finger
DEN21 1
NUM22 9
DEN22 5

# Encoder offset counts; reference position = fingers stretched out
# NOTE!!! these numbers must be the same as those found in the SST
#         batch.cmd file computing the forward kinematics!                       100
# R finger
styx[0].reference        400
styx[1].reference        962
# L finger
styx[2].reference        –387
```

```
styx[3].reference       970

# Encoder limits, offsets and gains
styx[0].gain            5.225e-5
styx[0].limit           100
styx[0].offset          0
styx[1].gain            -2.3856e-4
styx[1].limit           100
styx[1].offset          0
styx[2].gain            5.225e-5
styx[2].limit           100
styx[2].offset          0
styx[3].gain            2.3856e-4
styx[3].limit           100
styx[3].offset          0

# default desired position (if no input is used)
pid[0].desired          0
pid[1].desired          18
pid[2].desired          0
pid[3].desired          13.97     # For joint only
```

110

120

position

```
/*
 *
 * periodic.c — generate a periodic pattern in N dimensions
 *
 * Richard M. Murray
 * October 19, 1987
 *
 */

#include <stdio.h>                                                    10
#include <math.h>

#define TWOPI      (3.1415926535*2)                                   TWOPI
#define DEG_TO_RAD (3.1415926535/180)                                 DEG_TO_RAD

static unsigned int Tstart;         /* Starting time    */
static unsigned int clock;          /* Clock frequecy (hertz)    */

periodic_init(T, C)                                                   periodic_init
unsigned int T, C;                                                    20
{
    Tstart = T;
    clock = C;
}

periodic(T, N, off, mag, phase, freq, pos, acc, type)                 periodic
unsigned int T, N;
double off[], mag[], phase[], freq[];
double pos[], acc[];
char type[];                                                          30
{
    register int i;
    double t, s, w;

    /* Figure out the current time    */
    t = ((double) (T - Tstart) / (double) clock) * TWOPI;

    for (i = 0; i < N; ++i) {
        switch (type[i]) {
        /* Figure out the waveform based on the type    */            40
        case 'S':       /* sinusoid    */
            s = sin(freq[i]*t + phase[i]*DEG_TO_RAD);
            break;
        case 'B':       /* box    */
            s = sin(freq[i]*t + phase[i]*DEG_TO_RAD) > 0 ? 1 : -1;
            break;
        case 'T':       /* triangle - not implemented    */
        default:
            s = 0;
            break;                                                    50
        }
        pos[i] = off[i] + mag[i] * s;
```

```
        w = freq[i] * TWOPI;            /* omega - angular frequency */
        acc[i] = -mag[i] * w*w * s;
    }
}
```

```
/*
 *
 * circle.c - generate a circular pattern in N dimensions
 *
 * Richard M. Murray
 * October 19, 1987
 *
  */

#include <stdio.h>                                                    10
#include <math.h>

#define TWOPI      (3.1415926535*2)                                 TWOPI
#define DEG_TO_RAD (3.1415926535/180)                          DEG_TO_RAD

static unsigned int Tstart;            /* Starting time   */
static unsigned int clock;             /* Clock frequecy (hertz)    */

circle_init(T, C)                                             circle_init
unsigned int T, C;                                                    20
{
    Tstart = T;
    clock = C;
}

circle(T, N, off, mag, phase, freq, pos, acc)                      circle
unsigned int T, N;
double off[], mag[], phase[], freq[];
double pos[], acc[];
{                                                                     30
    register int i;
    double t, s, w;

    /* Figure out the current time    */
    t = ((double) (T - Tstart) / (double) clock) * TWOPI;

    for (i = 0; i < N; ++i) {
        s = sin(freq[i]*t + phase[i]*DEG_TO_RAD);
        pos[i] = off[i] + mag[i] * s;
                                                                      40
        w = freq[i] * TWOPI;           /* omega - angular frequency  */
        acc[i] = -mag[i] * w*w * s;
    }
}
```

```
/*
 *
 * circle.c — generate a circular pattern in N dimensions
 *
 * Richard M. Murray
 * October 19, 1987
 *
 */

#include <stdio.h>                                                   10
#include <math.h>

#define TWOPI       (3.1415926535*2)                                 TWOPI
#define DEG_TO_RAD  (3.1415926535/180)                               DEG_TO_RAD

static unsigned int Tstart;          /* Starting time */
static unsigned int clock;           /* Clock frequecy (hertz) */

box_init(T, C)                                                       box_init
unsigned int T, C;                                                   20
{
    Tstart = T;
    clock = C;
}

box(T, N, off, mag, phase, freq, out)                                box
unsigned int T, N;
double off[], mag[], phase[], freq[], out[];
{
    register int i;                                                  30
    double t;
    int sign;

    /* Figure out the current time */
    t = ((double) (T - Tstart) / (double) clock) * 6.1428;

    for (i = 0; i < N; ++i) {
        sign = sin(freq[i]*t + phase[i]*DEG_TO_RAD) > 0 ? 1 : -1;
        out[i] = off[i] + mag[i] * sign;
    }                                                                40
}
```

```
/*
 * read.c - read parameters from a file
 *
 * Richard M. Murray
 * September 1, 1987
 *
 */

#include <stdio.h>
#include <ctype.h>                                                    10
#include <math.h>
#include "dinit.h"

int dinit(fp, list)                                                dinit
FILE *fp;
D_IDENT *list;
{
    int n = 0, count, ch;
    char name[32], value[80];
    enum state_type {Name, Value, Comment} state;                     20
    int namef, valuef, exitf = 0;
    FILE *include_fp;

    while (!exitf) {
        state = Name;
        namef = valuef = 0;
        count = 0;

        /* Read everything a character at a time    */
        for (;;) {                                                    30
            if ((ch = fgetc(fp)) == EOF) {
                exitf = 1;
                break;
            }

            /* Check for comments or end of line first    */
            if (ch == '\n') break;
            if (ch == '#') state = Comment;

            switch (state) {                                          40
            case Comment:       /* Ignore everything until \n    */
                break;

            case Name:          /* Store the name in the buffer    */
                if (count == 0 && isspace(ch))
                    /* Skip leading spaces    */
                    break;
                else if (isspace(ch)) {
                    /* Terminator - store the name and switch states    */
                    state = Value;                                    50
                    count = 0;
                } else {
                    name[count++] = ch;
```

```
                name[count] = '\0';
                namef = count;
            }
            break;

        case Value:
            if (count == 0 && isspace(ch))
                /* Skip leading spaces   */                                 60
                break;
            else {
                value[count++] = ch;
                value[count] = '\0';
                valuef = count;
            }
            break;
        }
    }                                                                       70

    /* Truncate trailing spaces from value (if any)      */
    if (valuef)
        for (; valuef > 0; --valuef)
            if (isspace(value[valuef-1]))
                value[valuef-1] = '\0';
            else
                break;

                                                                            80
    /* Check to make sure we have a name and a state   */
    if (namef) {
        if (!valuef) {
            /* Only a name was present   */
            /* Check for special characters    */
            switch (*name) {
            case '?':       /* List all valid identifier and types        */
                list_matches(name+1, list);
                break;
                                                                            90
            case '.':       /* Exit dinit   */
                exitf = 1;
                break;

            case '<':       /* Include a file    */
                include(name+1, list);
                break;

            default:
                fprintf(stderr, "Incomplete line, %s\n", name);             100
                break;
            }

        } else {
            /* Name and value present   */
            /* Check for special names   */
```

```
            switch (*name) {
            case '?':       /* List identifier and type     */
                list_matches(value, list);
                break;                                                      110

            case '<':       /* Include a file    */
                include(value, list);
                break;

            default:        /* Normal value  */
                /* Look for the name in the list    */
                for (count = 0; list[count].name != NULL; ++count)
                    if (strcmp(list[count].name, name) == 0)
                        break;                                              120

                if (list[count].name == NULL) {
                    fprintf(stderr,
                            "Identifier not found, %s\n", name);
                } else {
                    /* Parse the buffer  */
                    scanv(value, list[count].type, list[count].addr);
                    /* c86_sscanf(value, list[count].fmt, list[count].addr);        */
                    ++n;
                    break;                                                  130
                }
            }
        }
    }
}
    return(n);
}

/* Read in the value of an element  */
scanv(value, type, addr)                                                scanv
char *value;                                                              141
int type;
char *addr;
{
    switch (type) {
    case Integer:
        *(int *) addr = atoi(value);
        break;
    case Long:
        *(long *) addr = atoi(value);                                       150
        break;
    case Byte:
        *(char *) addr = atoi(value);
        break;
    case Float:
        *(float *) addr = atof(value);
        break;
    case String:
        strcpy(addr, value);
```

```
        break;                                                     160
    case Double:
        *(double *) addr = atof(value);
        break;
    }
}

/* Print the value of element   */
fprintv(fp, type, addr)                                            fprintv
FILE *fp;
enum ident_type type;                                              170
char *addr;
{
    switch (type) {
    case Byte:
        fprintf(fp, "%d", *((unsigned char *)addr));
        break;
    case Integer:
        fprintf(fp, "%d", *((int *)addr));
        break;
    case Long:                                                     180
        fprintf(fp, "%ld", *((long *)addr));
        break;
    case Float:
        fprintf(fp, "%g", *((float *)addr));
        break;
    case String:
        fprintf(fp, "%s", addr);
        break;
    case Double:
        fprintf(fp, "%g", *((double *)addr));                      190
        break;
    }
}

/* Check for a match (for listings)    */
match(string, expr)                                                match
char *string;
char *expr;
{
    return(*expr == '\0' || strncmp(string, expr, strlen(expr)) == 0);   200
}

/* List values   */
list_matches(s, list)                                              list_matches
char *s;
D_IDENT *list;
{
    int count;

    /* Look for match in the list    */                            210
    for (count = 0; list[count].name != NULL; ++count)
```

```
        if (match(list[count].name, s)) {
            printf("%s\t%s\t", list[count].name,
                   list[count].fmt);
            fprintv(stdout, list[count].type, list[count].addr);
            putchar('\n');
        }
}

include(s, list)                                            include
char *s;                                                        221
D_IDENT *list;
{
    FILE *fp;

    if ((fp = fopen(s, "r")) == NULL)
        perror(s);
    else {
        dinit(fp, list);
        fclose(fp);                                             230
    }
}

/* Dump a table to disk for later reading    */
didump(fp, list)                                            didump
FILE *fp;
D_IDENT *list;
{
    int count;
                                                                240
    /* Look for match in the list    */
    for (count = 0; list[count].name != NULL; ++count) {
        fprintf(fp, "%s\t", list[count].name);
        fprintv(fp, list[count].type, list[count].addr);
        fputc('\n', fp);
    }
}
```

```
/*
 * dinit.h – Definitions and structures for dynamic initialization package
 *
 * Richard M. Murray
 * September 1, 1987
 *
  */

/* Structure to store a dynamic identifier     */
typedef struct identifier {                                                        10
    char *name;                  /* Parameter name  */
    char *fmt;                   /* Format string  */
    char *addr;                  /* Data address  */
    enum ident_type {
        Unknown = 0, Float, Integer, Long, String, Double, Byte
    } type;                      /* Identifier type    */
} D_IDENT;

/* Define a few macros for easy entry  */
#ifdef MSDOS                                                                       20
    /* Use ANSI C operators  */
#   define D_Byte(N)      {"" #N, "%i", (char *)(&N), Byte}                    D_Byte
#   define D_Long(N)      {"" #N, "%li", (char *)(&N), Long}                   D_Long
#   define D_Integer(N) {"" #N, "%i", (char *)(&N), Integer}                   D_Integer
#   define D_Float(N)     {"" #N, "%f", (char *)(&N), Float}                   D_Float
#   define D_Double(N)    {"" #N, "%lf", (char *)(&N), Double}                 D_Double
#   define D_String(N)    {"" #N, "%s", (char *)(N), String}                   D_String
#   define D_End              {NULL, NULL, NULL, Unknown}
#else
#   define D_Integer(N) {"N", "%i", (char *)(&N), Integer}                     D_Integer
#   define D_Long(N)      {"N", "%li", (char *)(&N), Long}                     D_Long
#   define D_Float(N)     {"N", "%f", (char *)(&N), Float}                     D_Float
#   define D_Double(N)    {"N", "%lf", (char *)(&N), Double}                   D_Double
#   define D_String(N)    {"N", "%s", (char *)(N), String}                     D_String
#   define D_End              {NULL, NULL, NULL, Unknown}                      35
#endif
```

```
/*
 * ddisp.c - dynamic display package
 *
 * Richard M. Murray
 * November 4, 1987
 *
 */

#include <stdio.h>
#include "ddisp.h"                                                  10
#include "termio.h"

/* Initialize the screen (and a list)        */
ddinit(disp)                                                    ddinit
DD_IDENT disp[];
{
    int entry;

    for (entry = 0; disp[entry].value != NULL; ++entry) {
        disp[entry].initialized = 0;                                20
        disp[entry].reverse = Normal;
    }
}

ddisp(disp)                                                      ddisp
DD_IDENT disp[];
{
    int row, col;
    int entry;
                                                                    30
    for (entry = 0; disp[entry].value != NULL; ++entry)
        (*disp[entry].function)(Update, disp+entry);
}

ddrefresh(disp)                                              ddrefresh
DD_IDENT *disp;
{
    ddcls();
    ddinit(disp);
    ddisp(disp);                                                    40
    if (ddcur != NULL) ddselect(ddcur);
}

/*
 * Item selection
 *
 * ddselect                Select any item
 * ddnext                  Select the next item
 * ddprev                  Select the previous item
 *                                                                  50
 */
```

```
DD_IDENT *ddcur = NULL;       /* Currently selected item    */

ddselect(dd)                                                          ddselect
DD_IDENT *dd;
{
    /* Redisplay the old cursor position      */
    if (ddcur != NULL) {
        ddcur->reverse = Normal;                                           60
        (*ddcur->function)(Refresh, ddcur);
    }

    /* Set up the new cursor position - find first selectable option        */
    for (ddcur = dd;  ddcur->value != NULL; ++ddcur)
        if (ddcur->selectable) break;

    if (ddcur->value != NULL) {
        ddcur->reverse = Reverse;
        (*ddcur->function)(Refresh, ddcur);                                70
        return(1);
    }

    ddcur = NULL;
    return(0);
}

int ddnext(tbl)                                                         ddnext
DD_IDENT *tbl;
{                                                                          80
    DD_IDENT *dd = ddcur + 1;

    if (ddcur == NULL) return(0);

    while (dd->value != NULL) {
        if (dd->selectable) {
            ddselect(dd);
            return(1);
        }
        ++dd;                                                              90
    }
    return(0);
}

int ddprev(tbl)                                                         ddprev
DD_IDENT *tbl;
{
    DD_IDENT *dd = ddcur;

    if (ddcur == NULL) return(0);                                         100

    /* Make sure we aren't at the start of the table      */
    if (dd-- == tbl) return(0);
```

```
    while (1) {
        if (dd->selectable) {
            ddselect(dd);
            return(1);
        }
                                                                                110
        /* Termination check */
        if (dd-- == tbl) break;
    }
    return(0);
}

/* Move cursor up */
int ddup(tbl)                                                                   ddup
DD_IDENT *tbl;
{                                                                               120
    DD_IDENT *dd = ddcur, *best_dd;
    int row, col, best;

    if (ddcur == NULL) return(0);

    /* Look for something on the previous row */
    for (row = ddcur->row-1, col = ddcur->col, best = 80; row >= 0; --row) {
        /* Search the entire table */
        for (dd = tbl; dd->value != NULL; ++dd) {
            if (dd->row == row && dd->selectable &&                             130
                abs(dd->col - col) < best) {
                best = abs(dd->col - col);
                best_dd = dd;
            }
        }
        if (best != 80) break;
    }
    if (row < 0) return(0);
    ddselect(best_dd);
    return(1);                                                                  140
}

/* Move cursor right */
int ddright(tbl)                                                                ddright
DD_IDENT *tbl;
{
    DD_IDENT *dd = ddcur, *best_dd;
    int row, col, best;

    if (ddcur == NULL) return(0);                                               150

    /* Look for something on the this row */
    row = ddcur->row;
    col = ddcur->col;
    best = 80;

    /* Search the entire table */
```

```
    for (dd = tbl; dd->value != NULL; ++dd) {
        if (dd->row == row && dd->selectable &&
            dd->col > col && dd->col - col < best) {                160
            best = dd->col - col;
            best_dd = dd;
        }
    }

    if (best != 80) {
        ddselect(best_dd);
        return(1);
    }
    return(0);                                                      170
}

/* Move cursor down */
int dddown(tbl)                                                     dddown
DD_IDENT *tbl;
{
    DD_IDENT *dd = ddcur, *best_dd;
    int row, col, best;

    if (ddcur == NULL) return(0);                                   180

    /* Look for something on the previous row */
    for (row = ddcur->row+1, col = ddcur->col, best = 80; row < 25; ++row) {
        /* Search the entire table */
        for (dd = tbl; dd->value != NULL; ++dd) {
            if (dd->row == row && dd->selectable &&
                abs(dd->col - col) < best) {
                best = abs(dd->col - col);
                best_dd = dd;
            }                                                       190
        }
        if (best != 80) break;
    }
    if (row >= 25) return(0);
    ddselect(best_dd);
    return(1);
}

/* Move cursor left */
int ddleft(tbl)                                                     ddleft
DD_IDENT *tbl;                                                      201
{
    DD_IDENT *dd = ddcur, *best_dd;
    int row, col, best;

    if (ddcur == NULL) return(0);

    /* Look for something on the this row */
    row = ddcur->row;
    col = ddcur->col;                                               210
```

```
    best = 80;

    /* Search the entire table    */
    for (dd = tbl; dd->value != NULL; ++dd) {
        if (dd->row == row && dd->selectable &&
            dd->col < col && col - dd->col < best) {
            best = col - dd->col;
            best_dd = dd;
        }
    }                                                                    220

    if (best != 80) {
        ddselect(best_dd);
        return(1);
    }
    return(0);
}

/*
 * Input functions                                                       230
 *
 * ddinput                  Input a new value
 * ddread                   Read a string from the console
 *
 */
```

ddinput

```
ddinput()
{
    if (ddcur != NULL) (*ddcur->function)(Input, ddcur);
}                                                                        240
```

ddread

```
ddread(prompt, address, length)
char *prompt, *address;
int length;
{
    int ch;

    ddprompt(prompt);

#if CURSOR                                                               250
    TTcurson();
#endif

    while (length > 1) {
        *address = ch = TTgetc();
        if (ch == '\n' || ch == '\r')
            break;
        ++address;
        TTputc(ch);  TTflush();
        --length;                                                        260
    }
    *address = '\0';
```

```
#if CURSOR
    TTcursoff();
#endif

    TTmove(term.t_nrow, 0);   TTeeol();
}
                                                                        270
/* Print a string on the prompt line   */
ddprompt(s)                                                        ddprompt
char *s;
{
    /* Move the the bottom line of the display and clear it      */
    TTmove(term.t_nrow, 0);   TTeeol();
    while (*s) TTputc(*(s++));
    TTflush();
}
                                                                        280
/*
 * Predefined display functins
 *
 * dd_integer
 * dd_long
 *
 */

/* Check and display integers   */
dd_integer(action, dd)                                           dd_integer
DD_ACTION action;                                                       291
DD_IDENT *dd;
{
    char ibuf[8];
    int *value = (int *)dd->value, *current = (int *)dd->current;

    switch (action) {
    case Update:
        if (dd->current != NULL && (!dd->initialized || *value != *current)) {
            sprintf(ibuf, dd->format, *current = *value);               300
            ddputs(dd, ibuf);
            dd->initialized = 1;
        }
        break;

    case Refresh:
        sprintf(ibuf, dd->format, *current = *value);
        ddputs(dd, ibuf);
        dd->initialized =. 1;
        break;                                                          310

    case Input:
        ddread("Integer: ", ibuf, 8);
        sscanf(ibuf, "%d", value);
        break;
    }
```

```
}

/* Check and display long integers    */
dd_long(action, dd)                                                    dd_long
DD_ACTION action;                                                          321
DD_IDENT *dd;
{
    char ibuf[8];
    long *value = (long *)dd->value, *current = (long *)dd->current;

    switch (action) {
    case Update:
        if (dd->current != NULL && (!dd->initialized || *value != *current)) {
            sprintf(ibuf, dd->format, *current = *value);                  330
            ddputs(dd, ibuf);
            dd->initialized = 1;
        }
        break;

    case Refresh:
        sprintf(ibuf, dd->format, *current = *value);
        ddputs(dd, ibuf);
        dd->initialized = 1;
        break;                                                             340

    case Input:
        ddread("Integer: ", ibuf, 8);
        sscanf(ibuf, "%ld", value);
        break;
    }
}

/* Check and display chars   */
dd_char(action, dd)                                                    dd_char
DD_ACTION action;                                                          351
DD_IDENT *dd;
{
    char ibuf[8];
    char *value = (char *)dd->value, *current = (char *)dd->current;
    char ctmp;

    switch (action) {
    case Update:
        if (dd->current != NULL && (!dd->initialized || *value != *current)) {   360
            sprintf(ibuf, dd->format, *current = *value);
            ddputs(dd, ibuf);
            dd->initialized = 1;
        }
        break;

    case Refresh:
        sprintf(ibuf, dd->format, *current = *value);
        ddputs(dd, ibuf);
```

```
        dd->initialized = 1;                                          370
        break;

    case Input:
        ddread("Character: ", ibuf, 8);
        sscanf(ibuf, "%c", &ctmp);
        *value = ctmp;
        break;
    }
}
                                                                      380
/* Check and display bytes  */
dd_byte(action, dd)                                               dd_byte
DD_ACTION action;
DD_IDENT *dd;
{
    char ibuf[8];
    char *value = (char *)dd->value, *current = (char *)dd->current;
    int itmp;

    switch (action) {                                                 390
    case Update:
        if (dd->current != NULL && (!dd->initialized || *value != *current)) {
            sprintf(ibuf, dd->format, *current = *value);
            ddputs(dd, ibuf);
            dd->initialized = 1;
        }
        break;

    case Refresh:
        sprintf(ibuf, dd->format, *current = *value);                 400
        ddputs(dd, ibuf);
        dd->initialized = 1;
        break;

    case Input:
        ddread("Byte: ", ibuf, 8);
        sscanf(ibuf, "%d", &itmp);
        *value = itmp;
        break;
    }                                                                 410
}

/* Check and display unsigned bytes  */
dd_ubyte(action, dd)                                              dd_ubyte
DD_ACTION action;
DD_IDENT *dd;
{
    char ibuf[8];
    unsigned char *value = (unsigned char *)dd->value;
    unsigned char *current = (unsigned char *)dd->current;            420
    int itmp;
```

```
    switch (action) {
    case Update:
        if (dd->current != NULL && (!dd->initialized || *value != *current)) {
            sprintf(ibuf, dd->format, *current = *value);
            ddputs(dd, ibuf);
            dd->initialized = 1;
        }
        break;                                                          430

    case Refresh:
        sprintf(ibuf, dd->format, *current = *value);
        ddputs(dd, ibuf);
        dd->initialized = 1;
        break;

    case Input:
        ddread("Byte: ", ibuf, 8);
        sscanf(ibuf, "%d", &itmp);                                      440
        *value = itmp;
        break;
    }
}

/* Check and display floats (single precision)        */
dd_float(action, dd)                                                    dd_float
DD_ACTION action;
DD_IDENT *dd;
{                                                                       450
    char ibuf[8];
    float *value = (float *)dd->value, *current = (float *)dd->current;

    switch (action) {
    case Update:
        if (dd->current != NULL && (!dd->initialized || *value != *current)) {
            sprintf(ibuf, dd->format, *current = *value);
            ddputs(dd, ibuf);
            dd->initialized = 1;
        }                                                               460
        break;

    case Refresh:
        sprintf(ibuf, dd->format, *current = *value);
        ddputs(dd, ibuf);
        dd->initialized = 1;
        break;
    }
}
                                                                        470
/* Check and display doubles (double precision)       */
dd_double(action, dd)                                                   dd_double
DD_ACTION action;
DD_IDENT *dd;
{
```

```
    char ibuf[32];
    double *value = (double *)dd->value, *current = (double *)dd->current;
    double dtmp;

    switch (action) {                                                   480
    case Update:
        if (dd->current != NULL && (!dd->initialized || *value != *current)) {
            c86_sprintf(ibuf, dd->format, *current = *value);
            /* c86_dtos(*current = *value, ibuf, 2, 'f');       */
            ddputs(dd, ibuf);
            dd->initialized = 1;
        }
        break;

    case Refresh:                                                       490
        c86_sprintf(ibuf, dd->format, *current = *value);
        ddputs(dd, ibuf);
        dd->initialized = 1;
        break;

    case Input:
        ddread("Float: ", ibuf, 16);
        sscanf(ibuf, "%lf", &dtmp);
        *value = dtmp;
        break;                                                          500
    }
}
```

dd_label

```
dd_label(action, dd)
DD_ACTION action;
DD_IDENT *dd;
{
    switch (action) {
    case Update:
        if (!dd->initialized) {                                         510
            ddputs(dd, dd->value);
            dd->initialized = 1;
        }
        break;

    case Refresh:
        ddputs(dd, dd->value);
        dd->initialized = 1;
        break;
    }                                                                   520
}

/*
 * Output functions
 *
 * ddopen                   Open terminal for ddisp
 * ddputs                   Output a string at the item location
 * ddcls                    Clear the screen
```

```
 * ddclose                    Close the terminal
 *                                                              530
  */

/* Open the terminal in raw mode  */
ddopen()                                                        ddopen
{
    TTopen();
    TTkopen();
    TTmove(0, 0);
    TTeeop();
#if CURSOR                                                      540
    TTcursoff();
#endif
}

/* Output a string  */
ddputs(dd, s)                                                   ddputs
DD_IDENT *dd;
char *s;
{
    TTmove(dd->row, dd->col);                                   550
    if (dd->reverse) TTrev(1);
    while (*s) TTputc(*(s++));
    if (dd->reverse) TTrev(0);
    TTflush();
}

/* Clear the entire string    */
ddcls()                                                         ddcls
{
    TTmove(0, 0);                                               560
    TTeeop();
}

/* Close the terminal (cooked mode)  */
ddclose()                                                       ddclose
{
#if CURSOR
    TTcurson();
#endif
    TTmove(term.t_nrow, 0);                                     570
    TTkclose();
    TTclose();
}
```

```
/*
 *
 * ddisp.h -- DDisp structures and definitions
 *
 * Richard M. Murray
 * November 4, 1987
 *
 */

#ifndef DDISP                                                                          10
#define DDISP

/* Display entry structure */
typedef struct display_entry {
    int row, col;
    char *value;
    int (*function)();
    char *format;
    char *current;
    int selectable;              /* Allow entry to be selected */                     20

    /* Parameters not normally initialized */
    unsigned initialized: 1;
    unsigned reverse: 1;
} DD_IDENT;

/* DDisp actons */
typedef enum {
    Update, Refresh, Input
} DD_ACTION;                                                                           30

#define DD_Label(r, c, s)        {r, c, s, dd_label, NULL, NULL, 0}                    DD_Label
#define DD_Option(r, c, s, l)    {r, c, s, dd_label, NULL, (char *) l, 1}              DD_Option
#define DD_End                   {0, 0, NULL, NULL, NULL, NULL, 0}

/* DDisp variables */
extern DD_IDENT *ddcur;

/* DDisp functions */
extern int dd_integer(), dd_byte(), dd_float(), dd_label();                            40
extern int dd_double(), dd_ubyte(), dd_long(), dd_char();

/* Screen attributes */
#define Normal 0x0
#define Reverse 0x1

#endif
```