INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313:761-4700 800:521-0600
Design, analysis, implementation, and control of a mobile robotic testbed for telepresence

Adnan, Sarmad, Ph.D.

Rice University, 1992
RICE UNIVERSITY

DESIGN, ANALYSIS, IMPLEMENTATION, AND CONTROL OF A MOBILE ROBOTIC TESTBED FOR TELEPRESENCE

by

SARMAD ADNAN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE

John B. Cheatham
Dr. John B. Cheatham, Director, Professor of Mechanical Engineering

Ian Walker
Dr. Ian B. Walker, Assistant Professor of Electrical and Computer Engineering

Angelo Miele
Dr. Angelo Miele, Professor of Mechanical Engineering

Thomas A. Krouskop
Dr. Thomas A. Krouskop, Adjunct Professor of Mechanical Engineering

Houston, Texas

March, 1992
ABSTRACT

DESIGN, ANALYSIS, IMPLEMENTATION, AND CONTROL OF A MOBILE ROBOTIC TESTBED FOR TELEPRESENCE

SARMAD ADNAN

A unique mobile telepresence testbed has been designed and implemented. This testbed is a tool for research in telepresence and tele-existence for control of remote robotic systems. An eight-degree-of-freedom, redundant manipulator has been designed and implemented for this system. Resolved acceleration control and impedance control have been demonstrated. An omnidirectional base has been built to provide human-like movement capabilities to the telepresence testbed.

Control software written for the system allows easy control of the base and the arm. Hand-controllers are used to guide the system trajectories. Ethernet, serial links, or wireless radio modems can be used as the control medium. Use of individual motor control processors for each motor allows high servo update rates to be achieved. A high level, modular and extensible library of routines has been written to allow easy programming of the system by future researchers. A head-tracking platform with color stereo cameras provides video feedback to the operator with depth perception to allow fine manipulation tasks.
Acknowledgments

I would like to thank Dr. Cheatham for his encouragment, support and the free rein of the robotics lab to pursue my numerous, and sometimes eccentric and expensive, technological interests; thus allowing my research to also become my hobby. I would not have been able to finish any of this in a timely manner had it not been for the help, support and understanding of my loving wife, Melissa. I am thankful to my father for his tremendous support of my decision to come to this country for my education. Finally I would like to thank my mother, without whose love and devotion none of this would have ever been possible.

The research documented in this thesis was supported by NASA Johnson Space Center project grants. I would like to express my appreciation for the help and guidance from Peter Galacki, and James Brock (NASA/JSC project NAG 9-372), Dr. Tim Cleghorn (NASA/JSC project NAG 9-443-Basic), and John Chladek (NASA/JSC project NAG 9-510).
# Table of Contents

Abstract ................................................................. ii  
Acknowledgments ..................................................... iii  
Chapter 1: Introduction ............................................. 1  
    Objectives: ....................................................... 3  
    State of the Art .................................................. 5  
Chapter 2: The Redundant Arm .................................... 13  
    Mechanical Design ............................................... 16  
    Kinematic Analysis .............................................. 27  
Chapter 3: The Omnidirectional Base ............................. 33  
    Mechanical Design ............................................... 38  
    Kinematic Analysis .............................................. 45  
Chapter 4: The Robot Torso ........................................ 55  
Chapter 5: Control Hardware ....................................... 60  
    Motor Control System .......................................... 66  
    Control Interfaces and Mediums ............................... 72  
Chapter 6: Control Algorithms and Software .................... 76  
    Resolved-Acceleration Control ................................. 79  
    Impedance Control .............................................. 85  
Chapter 7: Vision Feedback ........................................ 90  
    NASA Helmet Display and Camera Platform .................... 95  
    Rice Stereo Display and Camera Platform .................... 97  
Chapter 8: Future Extensions of Research ....................... 99  
Chapter 9: Summary ................................................ 103  
References ............................................................ 106  
Appendix A: Manufacturer Specification Sheets .................. 111  
    DC Servo Motors ............................................... 111  
    Harmonic Drives ............................................... 115
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motor Controllers</td>
<td>117</td>
</tr>
<tr>
<td>M3 Power Amplifiers Schematic</td>
<td>121</td>
</tr>
<tr>
<td>Bus Interface</td>
<td>122</td>
</tr>
<tr>
<td>Appendix B: Illustrations</td>
<td>123</td>
</tr>
<tr>
<td>Appendix C: Software Source Code</td>
<td>126</td>
</tr>
<tr>
<td>C Language Header Files</td>
<td>128</td>
</tr>
<tr>
<td>C Language Source Files</td>
<td>150</td>
</tr>
</tbody>
</table>
# Table of Figures

1.1: The Telepresence Testbed .................................. 3
2.1: Eight Degree of Freedom Redundant Robotic Arm .. 15
2.2: Operation of a Harmonic Drive ....................... 18
2.3: First Two Joints of the Redundant Arm ............ 18
2.4: Third and Fourth Joint of the Redundant Arm .... 19
2.5: Four Degree of Freedom Wrist ....................... 22
2.6: Electromagnetic Disc-Brakes ....................... 22
2.7: Overall View of the 8 Degree of Freedom ........ 25
2.8: Coordinate Axis for the 8 dof manipulator ...... 29
2.9: Projection on the X-Z Plane ......................... 30
3.1: First Prototype of Omnidirectional Wheels ...... 35
3.2: Computer Model of the New Omnidirectional Wheel 37
3.3: Support Bracket Aligned Along Roller Axis ....... 38
3.4: Clearances for Support bracket Between Rollers .. 38
3.5: Dimensions of the Rollers .......................... 39
3.6: Support Bracket Jigged for Final Turning ......... 40
3.7: Assembled Wheel ..................................... 41
3.8: Coordinate Axis of the Omnidirectional Base ..... 46
4.1: Solid Model of the Robot Torso ..................... 56
4.2: Side and Top view of the Torso Frame ............. 57
5.1: Block Diagram of The Control System .............. 63
5.2: Block Diagram of Motor Control Sub-System ...... 63
5.3: Three Different Handcontrollers Used with the .. 63
6.1: Resolved-Acceleration Control Block Diagram ..... 81
6.2: Pseudo-Code for Resolved-Acceleration Control .. 82
6.3: Impedance Control Block Diagram .................. 85
6.4: Pseudo Code for Impedance Control ................. 87
7.1: Configuration of the Original Head-Tracker ...... 93
7.2: Camera Platform Developed at NASA .................... 95
7.3: Rice Stereo Camera Platform ............................ 96
7.4: Control System for the Rice Camera Platform ...... 97
B.1: First Prototype of the Omnidirectional Base ...... 123
B.2: 8 dof Redundant Manipulator .......................... 123
B.3: The New Wheel Design ................................. 124
B.4: Telepresence Robot ...................................... 124
B.5: Rice Head Tracking System ............................. 125
B.6: Completed Telepresence Testbed ........................ 125
Chapter 1: Introduction

This chapter explains and outlines the objectives of this research project. Starting with a description of the concept of telepresence, it then goes on to give a review of the state of the art in the fields of telepresence and teleexistence.

Telepresence is a combination of kinesthetic, tactile, visual and proprioceptive feedback, which is presented in such a manner, from a telerobot to a human operator that the operator gets the feeling of being present at the remote site. There can be many levels of telepresence. In its simplest form, telepresence can be visual feedback to the operator using video cameras and monitors. Sophisticated implementations can involve head-tracking stereo vision systems, tactile sensing, teleproprioception, kinesthetic sensing and bilateral feedback. Low modal disparity, high bandwidth, and fidelity tend to increase the perceived quality of telepresence for the operator. Modal disparity is a measure of the deviation in the method of presentation of the feedback data to the operator. If the information is presented to the operator in the same manner and form as it was sensed at the remote site, then the system has little or no modal disparity. The system bandwidth determines the amount of information that the system provides to the operator. The system fidelity is the measure of the quality
and resolution of the information provided by the system to the operator.

Teleproprioception is defined as sensing the relative position and configuration of the remote telerobot with respect to the camera system and its body. Human operators in telepresence control of a robot identify with the robot in terms of their own body image [Sheridan 1986]. In essence, the operator makes an identification of the remote manipulator and his own arm. Kinesthetic or force feedback" is another important sensory input for telepresence. Force feedback requires the end-effector forces to be sensed either directly or resolved from joint torques. Proper feedback and utilization of this information can result in safer operation and allow the remote manipulator to be operated at higher speeds.

Tactile sensory feedback includes information gathered by tactile sensors such as touch switches, switch arrays, pressure sensors, Hall-effect sensors and infra-red proximity sensors. The main purpose behind tactile feedback is to sense initial surface contact with an object. Tactile information can also be very useful in detecting low speed collisions of compliant manipulators.

* Pronouns "he" and "his" used in this document are not meant to be gender-specific unless used in reference to a particular person.
** In this document the term "force feedback" is used to imply both force and torque feedback.
Objectives:

The main objective of this research was to design and implement a testbed for remote telepresence experimentation and to implement an integrated control scheme to make the system useful. A system such as this would allow research and implementation of new schemes and techniques to help advance the applicability, capability, and expectations of telepresence and tele-existence systems. This thesis documents all the different phases of the project, the results, and conclusions. It is also meant to serve as a reference manual for the mobile telepresence testbed. Finally another objective of this document is to compare the capabilities of this system with other implementations.

Figure 1.1: The Telepresence Testbed
An integrated testbed for telepresence (Figure 1.1) has been built at the Rice University mechanical engineering robotics laboratory. The system consists of an eight-degree-of-freedom redundant manipulator, a three-wheeled omnidirectional base, a stereo vision system, a head-tracking camera system, robot torso, on-board power supplies, hand-controllers, computers and other control equipment. Software has been written to properly control each of the individual elements of the system and to allow the system components to be used together in a useful and coherent manner. Many different control schemes have been implemented. It is possible to use handcontrollers, radio modems, ethernet, and function generators to control the system. Care has been taken during the design and implementation of the system to keep it as flexible and extensible as possible, thus allowing researchers in the future to implement and test their own control schemes and theories.
State of the Art

Teleoperation has repeatedly proved its worth during the past few decades. The fields of nuclear laboratory remote handling, deep sea explorations and more lately space exploration have reaped its benefits. Teleoperation, although feasible in slow tasks requiring coarse control, is not adequate for tasks requiring dexterity, manipulability, precision, fine control, and constrained motion. For these purposes, telepresence is desirable. With telepresence the operator works under the illusion of being at the remote site, generally resulting in superior task performance [Adnan 1991]. Depending on the environment and task, additional benefits of telepresence can include reduced operator fatigue, increased operational safety, and reduced operating costs.

Potential applications for telepresence include space operation, rescue, deep sea explorations, fire fighting, detoxification, and hot nuclear handling. Different tasks require different levels of telepresence. It is advantageous to determine exactly how many senses are needed to perform a certain task and to provide an adequate illusion of telepresence to the operator. In the cases of free manipulation and manipulation without physical contact of objects with the environment, simple visual feedback may be enough. While in remote assembly tasks, additional
tactile, kinesthetic, auditory, and teleproprioceptive feedback may be essential. Feedback of phenomena such as impact forces, high amplitude oscillations and loud noises may be detrimental to an operator's well being. Hence clipping and amplification of feedback are also issues that need to be considered. One solution is to use logarithmic feedback gains. Logarithmic amplification of force feedback can be very beneficial. Whereas at the low magnitudes of force the full magnitude may be presented to the operator, at higher force magnitudes a reduced portion can be fed back. Thus near the lower portion of the spectrum, maximum sensitivity can be achieved; near the higher end, operator fatigue can be reduced by the using a higher forward amplifications, thereby magnifying the forces applied by the operator [Telerobotics, 2].

The fields of telepresence and virtual reality are closely related. Whereas it is the goal of telepresence to provide the operator with a feel of being at a remote physical site, virtual reality strives to provide the operator with the feel of being at some imaginary site. Intricate computer-based models of some environment are made, and the operator is allowed to interact with them. Whereas telepresence can be useful in performing actual immediate tasks, virtual reality can be useful for training, evaluation and planning.
Since both telepresence and virtual reality involve providing sensory information to the operator, the same types of operator workstations can be used for both purposes. While the telepresence workstation needs a remote telerobot at the other end, the virtual reality workstation obtains inputs from the computer models and algorithms.

Among the equipment and paraphernalia used for telepresence systems are handcontrollers, force sensors, force impedance handcontrollers, visual display systems, video input devices, head and body tracking sensors, tactile sensors, audio input and output systems, and hand configuration devices.

Recently handcontrollers have been replacing master-slave type systems for positional input. The main advantage that handcontrollers offer over master-slave arrangements is flexibility. A master arm similar in kinematic configuration to the slave arm was often used in early teleoperated systems to control the remotely located slave. This similarity allowed simple control schemes that utilized joint-to-joint control to be used [Tachi 1990, 385-390]. In the past, lack of sufficient computer power prevented more elaborate schemes from being implemented. Handcontrollers are more general purpose then master-slave systems. Since they utilize input in Cartesian space, they provide a uniform interface to the operator independent of the
kinematic configuration of the telerobot. The need for more complex inverse kinematic control schemes is no longer a problem because of the easy availability of high-powered computer equipment.

Force feedback in telepresence systems can take two forms: unilateral force feedback and bilateral force feedback. Unilateral force feedback implies the sensing of end-effector forces at the remote telemanipulator and using them as gains in the control law. The operator may be visually informed of the forces, i.e., on a visual display unit (VDU). In the case of bilateral force reflection or feedback, end-effector forces at the remote telemanipulator are fed back to the operator through the hand controller or an equivalent input device. Force feedback handcontrollers usually manifest these forces as either displacement or impedance (resistance to displacement). Thereby requiring the different degrees of freedom of the handcontroller to be actuated. Forces exerted by the operator on the handcontroller are converted to joint torques at the remote telemanipulator. Almost all current bilateral force feedback mechanisms fail to model impact forces. Modeling of impact forces requires motors and joints capable of very high speeds and accelerations. Most existing systems use low speed and high torque actuators which, though suitable for feeding back gradual magnitude and sign changes of forces, are inadequate for the very rapid force changes
required to model impact. In master-slave type telemanipulators where the configurations of the master and the slave manipulators are identical, direct sensing and control of joint torque can yield adequate bilateral force reflection. Force amplification can also be achieved by increasing the feedback to the slave telerobot. Such systems found initial application in the nuclear laboratory and in industry.

Tactile sensors on the end-effectors and manipulator arms can be used to sense contact [Nowlin 1991, 380]. Feedback from such sensors is usually not presented to the operator in its natural form. Visual or auditory alarms are perhaps the most common methods of indicating contact. They can involve sounding an alarm or flashing a light indicator. Contact may also be indicated to operator by means of a graphic representation superimposed on the visual scene. In a limited manner direct stimulation of the operator's skin can also be used to indicate contact. However, because of the difficulties associated with covering large areas of the operator's skin with such devices, these techniques have not been implemented in the telepresence context.

Visual feedback for telepresence is accomplished in a variety of ways. Stereo-monitor-equipped helmets with head tracking systems provide the most realistic sense of telepresence [Adnan 1991]. Three-dimensional, stereo LCD
(liquid crystal diode) shutter glasses can also be used. They provide a realistic sense of depth perception [Visual Research Corporation 1991]. However, since the display monitor itself is stationary, no head tracking is possible.

In order to sense hand and arm configuration and positions, devices like the VPL Dataglove [VPL Research Inc. 1991, 1] and the hand exoskeleton [Exos Corporation 1991] can be utilized. The Dataglove, as the name suggests, is an instrumented glove. It uses optical fibers to sense the joint configurations of the fingers and the wrist. The hand exoskeleton, developed by Exos Corporation, is an exoskeleton structure that is worn on the back of the hand. Although more cumbersome to wear than the Dataglove, it provides greater resolution and precision. Unlike the Dataglove, it also allows sensing of the abduction motion of the fingers.

A tele-existence system has been implemented at the mechanical engineering laboratory at MITI in Tsukuba Science City, Japan [Tachi 1990]. The salient features of this system include a seven degree-of-freedom anthropomorphic robotic manipulator, a head-tracking stereo display, and a direct-drive master manipulator. Lack of mobility and the cumbersome master-slave input scheme are the main drawbacks of this system.
The Virgule teleoperator, developed by the French Commissariat a l'Energie Atomique, is a dual-armed mobile system sporting bilateral force-feedback, master-slave control and the ability to climb stairs [Vertut 1976]. This system can only be controlled in a master-slave mode and also lacks vision feedback for effective telepresence.

Among the applications often cited for telepresence are hazardous material handling, space operations, fire fighting, and bomb disposal. All of these tasks except space operations can be performed from a control site within a short distance from the remote teleoperator. In telepresence for space operations, a new factor needs to be considered: time delay. Time delay, due to telemetry, signal propagation and computational overhead, can cause instabilities in robotic control systems. For control of telepresence systems in orbit around the earth, time delay caused by signal propagation can be in the order of small fractions of a second. Researchers state that for stable force feedback control, time delays of 0.1 sec are near the upper threshold of acceptability [Noyes 1984]. Time delay in the order of a second or more makes even simple positional-control difficult for the operator, thereby suggesting that pure telepresence with visual and kinesthetic feedback would not be possible for earth based control on the Lunar or Martian surface. Research has been
done to overcome the problems caused by time delay using time clutches and time brakes [Conway 1990]. Time clutches involve the use of forward simulations of the manipulator motions presented to the operator. The operator can use the time clutch to run the simulation and the real manipulator in synchrony or with a variable time lag. For telepresence, these time clutches offer a non-intuitive solution that can only lead to increased modal disparity and subsequently a loss of "telepresence."
Chapter 2: The Redundant Arm

In order to make a telepresence system useful, it is necessary to provide it with some means to interact with its environment. The need for a robotic manipulator for this interaction was realized during the conceptual design stage of the telepresence project. The idea behind telepresence is to provide the operator with a feeling of being present at the remote site, and one of the requirements for telepresence is teleproprioception. In order for the operator to identify his own arm with the robotic manipulator, the robotic manipulator needs to have capabilities that are somewhat similar to those of the human arm. The operator may be able to overcome the disparity caused by differences of size and scale between the arms, but it would be not be possible for him to overcome configuration and functional differences between the two. For example, if the robot present at the remote site has only three degrees of freedom or is planar, then the operator would not get any teleproprioceptive feedback.

The human arm is a versatile, dexterous, and redundant manipulator. For manipulators, redundancy means seven or more degrees of freedom, since this gives self-motion capability to the manipulator. If one were to emulate the human arm with a robotic manipulator, only a redundant manipulator could be capable of approaching its
capabilities. Very few commercial redundant robotic arms exist today [Farrel 1990]. Among them, the most widely known and used is the Robotics Research seven dof K1207 Arm. This arm has an anthropomorphic design with seven revolute joints. All of the joints present on the arm are of the roll-pitch type. The large size of this arm makes it unsuitable for mounting on a mobile platform. Another redundant manipulator is an augmented PUMA robot [Hollerbach 1985]. This augmented robot is a PUMA 500 series robot with a roll joint added at the upper joint link. Researchers at Stanford have mounted a PUMA 200 series robots on a mobile platform with some degree of success. The PUMA 500 series robots are much larger and hence unsuitable for use on the omnidirectional mobile platform.

Initially, the RTX robot was thought to be a suitable candidate for being mounted on the mobile omnidirectional base. Among the factors contributing to its consideration were availability, cost, weight, and ease of interfacing. The RTX robot was already in the mechanical engineering robotics laboratory. It was cost-effective, light-weight and had standard interfaces for direct control from personal computers. A detailed survey of the requirements for the telepresence manipulator showed that the RTX was unsuitable. Among the requirements laid out for the manipulator were redundancy, dexterity, and anthropomorphy. (Anthropomorphy implies a human-like arm in appearance and structure.)
Unfortunately, the RTX robot did not fit these criteria since it is neither anthropomorphic nor redundant.

Figure 2.1: Eight Degree of Freedom Redundant Robotic Arm

The need for a redundant manipulator and the lack of any cost-effective commercial options led to the decision to build such a manipulator in-house. This decision resulted in an ambitious two year project. The positional degrees of freedom, i.e., the shoulder and elbow, were committed to design and construction during the first year. Orientation degrees of freedom of the arm, i.e., the wrist, was to be designed during the second year. This division of the design project into two parts helped streamline the design process and resulted in a successful design. Figure 2.1 is a sketch of the current state of the redundant manipulator.
Mechanical Design

The first year design team [Fessler et. al. 1990] consisted of four mechanical engineering design seniors. Their objective was to design the first four joints of an eight-degree-of-freedom redundant arm. The author served as the design advisor for the team. After evaluating different configuration possibilities for the wrist, a spherical configuration with an additional pitch joint was decided upon. This configuration was to be achieved by a roll-pitch-roll-pitch arrangement. Assisted by Dr. J. B. Cheatham and the author, the design team embarked upon a step-wise refinement of this design concept.

In an effort to reduce backlash and simplify the design, the motor and gear reduction unit were to be housed in close proximity to the relevant joint. Since each joint was to house its drive, the shape and size of the motor-reducer unit were going to have an overbearing effect on the design of each individual joint. Selection of the motors and reduction units was the first step in design. The joints were to be built around them.

One of the design goals of the telepresence system was that it should be self-contained, meaning that all required power needed to be carried on board in the form of batteries. Four, gel-type twelve volt, lead-acid batteries
comprised the power supply of the telepresence robot. Therefore, the motors selected for the arm joints were all forty-eight volt, direct current, permanent magnet motors. Stepper and brushless motors were not considered for this application because of their poor torque to weight ratios and increased complexities. Selection of the motors was based on output torque to weight ratio and price. Since the required output torque had been calculated to be approximately 1400 in-lb. for the first two joints, the GNM4150 motors from Micro Mo Electronics [Micro Mo Electronics 1988] were selected. The motors had a rated torque output of 7.875 in-lb. at 1635 rpm. Using a gear reduction of 200:1, an output torque of 1575 in-lb. could be attained. For the second and third joints, the torque requirements were approximately 500 in-lb. The GNM4125 motors selected for these joints have a rated torque output of 4.179 in-lb. at 1776 rpm. A gear reduction of 200:1, would result in output torque of more than 835 in-lb. Manufacturer specification sheets for this and other motors are included in Appendix A.

Compact size and minimum weight were the prerequisites when it came to selecting the gear reducers for the arm joints. Most types of spur gear sets and planetary reducers could not meet the weight restrictions. Worm drives meet the weight criteria but have, inherent in their design, large frictional losses which rendered them unsuitable for
our purpose. Therefore, harmonic drive gearing was the obvious choice. Harmonic drives were developed some thirty years ago primarily for use in aerospace applications. Aside from being light weight and compact, they offer other benefits such as high torque capacity, low backlash, high single stage reduction ratios, and an in-line configuration [Harmonic Drive 1989, 2]. These unconventional reducers consist of three parts, a rigid circular spline internal gear, a flexible cup-shaped flex-spline external gear, and an elliptical disc-shaped wave generator. The shaft of the motor is connected to the wave generator and is the input. The rigid spline is kept fixed, and the flex-spline is attached to the output shaft.

\[ \alpha \quad \text{Input Shaft Angle} \quad \varphi \quad \text{Output Shaft Angle} \]

\[ r \quad \text{reduction ratio} \]

\[ \varphi = 0 \quad \varphi = \frac{180}{r} \quad \varphi = \frac{360}{r} \]

\[ \alpha = 0^\circ \quad \alpha = 90^\circ \quad \alpha = 180^\circ \]

Figure 2.2: Operation of a Harmonic Drive
Figure 2.2 shows various phases of the operation of a harmonic drive. The gear teeth on the two splines are in constant engagement. The flex-spline has two fewer teeth than the rigid circular spline. As the elliptical wave generator rotates through 180°, the flex-spline rotates through a distance of two gear teeth. The harmonic drives selected for the redundant arm joints have a gear ratio of two-hundred to one.

Figure 2.3: First Two Joints of the Redundant Arm

The first joint of the redundant arm is built on an aluminum tubing frame. A bulkhead plate is used to mount
the motor to this frame. Figure 2.3 shows the drive train arrangement for the first joint of the redundant arm. Another bulkhead plate holds the harmonic drive. A 1" diameter steel output shaft carries power from the harmonic drive flex-spline cup to the second joint. This shaft is supported on two Seal Master NF series pillow blocks.

Figure 2.4: Third and Fourth Joint of the Redundant Arm

The second joint is a pitch joint. The motor and harmonic drive gear reduction are mounted along its axis. Calculations for the bending moment at this joint showed moments of nearly 750 in-lb. Instead of using conventional ball bearings, the decision was made to use a Kaydon Reali-Slim bearing [Kaydon 1988]. These ball bearings have a large ball race diameter, but their ball races are very thin
and narrow. The thin and narrow races minimize weight, while the large race diameter allows support of greater thrust and radial loads. Two of Kaydon KC060AR0 ball bearings are used in this pitch joint. They are separated from each other by aluminum ring-shaped spacers.

The third and fourth joints of the redundant arm are designed very much like the first two joints. They are scaled down versions of the first two. A tubular aluminum housing, 4.5" in diameter, covers the third joint. The entire roll mechanism is enclosed within this tube. Wall thickness of the tube is 0.125". Torsional stresses of the third joint can be carried by this tube with an adequate factor of safety. Design of the fourth joint is modeled after the second one. The main difference between the two joints is smaller joint size, smaller motor, and a smaller harmonic drive. Figure 2.4 shows details of this joint.

The second year design team [Bartosh et. al. 1990] was comprised of five mechanical engineering design seniors. Their objective was to design a four-degree-of-freedom redundant robotic wrist. Anthropomorphic design of the arm as a whole was a concern during this design process. In order to keep the four-degree-of-freedom wrist kinematically spherical, a roll-pitch-pitch-roll configuration was selected. The first joint on the wrist is a roll joint. Its range of motion is a full 360°. A pitch and a yaw joint
follow. These have approximately 270° and 90° of rotational capability. The final joint is another roll, and it too can rotate through a full 360°.

The selection of motors and gear heads for the wrist was the first design step. Motor specifications and requirements for the wrist were very different from those of the arm. Whereas torque and speed were paramount for arm motor selection, minimum weight and compact size were the leading motor selection criteria for the wrist. Since the torque requirements for all of the wrist joints were very much alike, the same type of motor was selected for them all. A motor manufactured by Soho Motors, part number GBL 35-DH-21090-10Y, was selected [Servo Systems]. The motors have a stall torque in excess of 27 in-lb. at 12 volts DC. An integral gear-head mounted on the motors provides an output speed of 40 rpm.

The first rotation of the wrist, also called the wrist roll, is directly driven through a spur gear arrangement by one of the motors. The last three rotations, pitch, yaw, and tool roll, sport a differential-type bevel gear design. The motors are mounted near the base of the wrist, while the bevel gear setup is at the other end. Some mechanism for power transmission also had to be provided. It was decided to transmit power from the motor to the bevel gears using plastic chains. These chains run on aluminum sprockets and
drive the bevel gears. Three such chains are utilized on the wrist. In an effort to reduce the weight of the wrist assembly, the large diameter bevel gears used on the last three rotations have been extensively lightened. Lightening holes were drilled, and unnecessary material was removed. The structure of the wrist and the arrangement of the bevel gears can be seen in Figure 2.5.

![Four Degree of Freedom Wrist Diagram]

Figure 2.5: Four Degree of Freedom Wrist

During initial testing of the design some problems were uncovered. It was discovered that the motors in the first two joints were using too much current. Consequently their amplifiers would overheat. This was a direct result of the gear reduction at these joints being inadequate. A solution
was needed that would be cost-effective and that would require the least amount of modifications to the arm. It was determined that electromagnetic disc-type normally-on brakes would provide an adequate solution to the problem.

![Electromagnetic Disc-Brakes Diagram]

**Figure 2.6: Electromagnetic Disc-Brakes**

The brakes have been installed on the output jack-shaft (see Figure 2.6) of the motor that transmits power to the harmonic drive. Due to their compact size, it was possible to install them with few modifications to the joints. The fact that the brakes are installed upstream of the gear reduction units gives them the benefit of mechanical advantage. These brakes are normally-on to keep their respective joints stationary, unless electrically released.
Figure 2.7: Overall View of the 8 Degree of Freedom Manipulator
The complete eight degree of freedom manipulator is shown in Figure 2.7. As is clear from the illustration, all the different parts of the manipulator have been integrated into a proportionally correct unit. Controls laws, electronics aspects and software for the redundant arm are discussed in chapters five and six.
Kinematic Analysis

Kinematics analysis is a necessary step for the control of any robotic manipulator. Most industrial robots and manipulators have six joints or less. The redundant arm used in this research has eight actuated joints. Analysis of this type of manipulator requires special consideration for the redundant joints. Researchers have proposed many different redundancy resolution techniques for kinematically redundant manipulators. Some of these techniques include weighted pseudo-inverse control, global minimization, minimum energy, configuration control, torque optimization, and parameterization [Siciliano 1990][Nenchev 1989].

The weighted pseudo-inverse control [Kircanski 1984] uses the concept of a pseudo-inverse of the redundant manipulator to minimize some of the joint velocities. The main problem with this technique is the non-cyclicity of the solution. Non-cyclity implies that for the same end-effector position this technique results in a different set of joint angles at different times. The authors of this technique incorrectly assumed that minimizing the joint velocities would avoid singularities, since singularities are usually accompanied by high joint velocities. This was an incorrect assumption since the joint velocities are a function of the trajectory used to reach the singular positions. Torque optimization [Hollerbach 1987] is another
technique based on pseudo-inverses and as the name implies involves optimization of joint torques. The minimum energy approach is similar to the velocity minimizing pseudo-inverse scheme, except the goal here is to optimize the total energy of the manipulator. An inherent shortcoming of these techniques is the high computational overhead due to the optimization process.

The parameterization method [Lee 1991] for redundant manipulator control involves the parameterization of properly chosen redundant joints. A "parameterized" closed form inverse kinematic solution is obtained as the \( n \) dof redundant manipulator transforms to an \( m \) dof non-redundant manipulator. Some \((n-m)\) of the links of this parameterized arm are adjustable and are set by the \( n-m \) redundant joints.

Configuration control [Seraji 1989] is another technique used for redundancy resolution in robotic manipulators. This technique utilizes a set of \( r \) \((r=n-m)\) arbitrary kinematic functions in joint or Cartesian space. These kinematic functions serve as task constraints, and the problem of redundancy resolution then becomes one of constraint satisfaction. Since redundancy resolution is done by satisfying arbitrary kinematic constraints, this method does not provide the operator with much control over the manipulator configuration unless a new set of kinematic functions is specified for each type of manipulator task.
Rather than use one of these time-consuming, and CPU intensive techniques for the control of the redundant joints, it was decided to allow the operator direct control of these joints. Thus the kinematic analysis of the eight degrees of freedom arm could be performed as the analysis of a six degree of freedom arm with variable geometry. The third and the fifth joints were selected for this purpose.

Figure 2.8: Coordinate Axis for the 8 dof manipulator

The forward kinematic solution of this manipulator is uncomplicated. The wrist point is defined as the origin of the joint coordinate axis six.

\[ x = S_1C_1S_4C_3L_3 - C_1S_4S_3L_3 + S_1S_2C_4L_3 + S_1S_2L_2 \]  \hspace{1cm} (2.1)

\[ y = -S_2S_4C_3L_3 + C_2C_4L_3 + C_2L_2 + L_1 \]  \hspace{1cm} (2.2)

\[ z = -C_1C_2S_4C_3L_3 - S_1S_3S_4L_3 - C_1S_2C_4L_3 - C_1S_2L_2 \]  \hspace{1cm} (2.3)
The inverse kinematic solution for the redundant arm were derived by two separate methods. The first method used by the author involved a geometric approach to the inverse kinematic solution [Shahinpoor 1987, 132-137]. The second method used was the conventional wrist partitioning scheme using D-H [Hartenberg et. al. 1955, 215-221] parameter tables. Shuxin Gu [Gu 1991, 1-8] performed the inverse kinematics using the conventional method as a research project. Results derived from both of the analyses match.

![Diagram](image)

**Figure 2.9: Projection on the X-Z Plane**

Using the geometric approach, the first step is to draw a projection of the manipulator on the world x-z plane. This projection can then be used with the cosine law to determine $\theta_4$. Remember that $\theta_3$ is a known quantity since it is specified by the operator. From the geometry of the figure and the cosine law we get
\[
\cos(\alpha) = \frac{w^2 - L_2^2 - L_3^2}{2L_2L_3}
\] (2.4)

\[\theta_4 = 180 - \alpha\] (2.5)

If we replace \(\frac{w^2 - L_2^2 - L_3^2}{2L_2L_3}\) by \(Z\), then from 2.4 and 2.5,

\[
\theta_4 = \text{atan} \left[ \frac{\sqrt{1 - Z^2}}{Z} \right], 0^\circ < \theta_4 < 180^\circ
\] (2.6)

This is the solution for joint four. Simultaneous solution of 2.1 through 2.3 yields two equations of interest. This solution was performed using the Maple symbolic algebra software [Maple].

\[
S_4S_3L_3 = -(C_1x + S_1z)
\] (2.7)

\[
S_2(S_4C_3L_3) - C_2(C_4L_3 - L_2) = L_1 - y
\] (2.8)

From Wolovich [Wolovich 1987, 110-111], we have a general analytical formulation.

\[
ac_\theta + b_\sin(\theta) = c; \theta = \text{atan} 2 \left[ \frac{b}{a} \right] + \text{atan} 2 \left[ \frac{\pm \sqrt{a^2 + b^2 - c^2}}{c} \right]
\] (2.9)

Inserting 2.7 in 2.9 we get the solution for the first joint.

\[
\theta_1 = \text{atan} 2 \left[ \frac{z}{x} \right] + \text{atan} 2 \left[ \frac{\pm \sqrt{x^2 + z^2 - (S_4S_3L_3)^2}}{-S_4S_3L_3} \right]
\] (2.10)
Inserting 2.8 in 2.9 we get the solution for the second joint.

\[ \theta_2 = \text{atan} 2 \left[ \frac{-S_4 C_3 L_3}{C_4 L_3 - L_2} \right] + \text{atan} 2 \left[ \frac{\pm \sqrt{(C_4 L_3 - L_2)^2 + (S_4 C_3 L_3)^2} - (y - L_1)^2}{(y - L_1)} \right] \quad (2.11) \]

Equations 2.6, 2.10, and 2.11 give the inverse kinematic solution for the manipulator. Solutions for the sixth, seventh, and eighth angles follow.

If the desired orientation of the manipulator arm is given in euler angles as,

\[
\begin{bmatrix}
 n_x & s_x & a_x & 0 \\
 n_y & s_y & a_y & 0 \\
 n_z & s_z & a_z & 0 \\
 0 & 0 & 0 & 1
\end{bmatrix}
\]

\[ \theta_6 = \text{atan} \left( \frac{a_y}{a_x} \right) \quad (2.12) \]

\[ \theta_7 = \text{atan} \left( \frac{-a_x \cos(\theta_6) - a_y \sin(\theta_6)}{a_z} \right) \quad (2.13) \]

\[ \theta_8 = \text{atan} \left( \frac{-s_x \cos(\theta_6) + s_y \sin(\theta_6) \cos(\theta_7) - s_z \sin(\theta_6)}{(s_x \sin(\theta_6) - s_y \cos(\theta_6))} \right) \quad (2.14) \]

Thus equations 2.6, 2.10, 2.11, 2.13, 2.14, and 2.15 describe the complete inverse kinematic solution for the manipulator.
Chapter 3: The Omnidirectional Base

Robot mobility is an important factor in the design of a telepresence system. Since ideally a telepresence system would have at least the same capabilities as its human operator and should also be able to move around like one, it is essential to have a mobile platform for the system. Mobile robot bases with conventional wheels and steering have very limited motion capabilities and can in no way approach the mobility of a human. Motion capabilities of a human can only be adequately mirrored by an omnidirectional mobile robot. Omnidirectional mobile robots are robots that can move in any direction in the plane of the floor. These robots can translate forward-back, right-left, and rotate around a vertical axis independently. This freedom of motion allows omnidirectional robots to have great versatility and maneuverability. The obvious advantages of omnidirectionality over conventional wheeled movement are the ability to maneuver through narrow passageways, move sideways, and display human like motion freedom.

Among the different omnidirectional mobile bases, the most commonly mentioned ones are the Unimation robot and the CMU Uranus [Muir 1986]. The Unimation robot is a three-wheeled omnidirectional robot built around the Stanford omnidirectional wheel design. The wheels are arranged along the sides of an imaginary equilateral triangle. This robot
is probably one of the first omnidirectional mobile robot base. The passive rollers are positioned with their axis at 90° from the wheel axis. The CMU Uranus is a four wheeled omnidirectional robot. The rollers on the wheels of this robot are positioned with their axis at 45° to the wheel axis. The Uranus has four actuated wheels and only three degrees of freedom. It is thus an over-determined system and has the possibility of actuator conflicts. Muir [Muir 1986] describes the actuation structure of Uranus as adequate but not robust. The omnidirectional wheels designed by both the Stanford and the CMU designers are very complex and have large number of moving parts. Another problem with the Stanford design is that there are sizable gaps between the rollers causing the wheels to "bump" as the gap approaches the ground.

One competitor of omnidirectional mobile bases is the Synchro-Drive [Muir 1986] base. These are two dof mobile robots. A chain and sprocket arrangement is used to drive all of the wheels of the robot from the same actuator. In addition all of the wheels are pivoted and constrained to turn together. The result is a robot that does not have to turn its body to move in any direction. The wheels, however, have to first face the intended direction of motion. The drawback of this type of robot base is that it only has two degrees of freedom and thus can not completely emulate the movements of a human operator. It is thus best
to use an omnidirectional mobile base to form the basis of a telepresence system.

The omnidirectional base used in this research project is a three-wheeled mobile robot that serves as the mobility subsystem of the telepresence robot. This mobile robot is a second generation system, an improved design based on the lessons learned in the design of the first prototype. The original omnidirectional platform was designed as part of a design project by a group of mechanical engineering senior students [Steub 1988, 5-25]. Kinematic analysis and implementation of a control system for this platform were completed by the author [Adnan 1989] in partial fulfillment of his Master's degree research requirements.

![Omnidirectional Wheel with Six Rollers](image)

*Figure 3.1: First Prototype of Omnidirectional Wheels*
There were four main problems with the first design. Rollers on these wheels consisted of two halves. Figure 3.1 illustrates the fact that the front half of the roller ended in a point. This caused the circumference of the rollers at this point to be close to zero. Consequently, a large amount of force was required to turn these rollers, this resulted in jerky motion each time one of the points touched the ground. The second problem with the first wheel design was the low coefficient of friction of the wheel material. Aluminum, due to its low weight and excellent strength, was used to construct the wheels. However its low coefficient of friction with the hard floor resulted in a loss of traction during situations of high wheel accelerations. Custom cast polyurethane covers for the rollers were then employed to partially alleviate the problem, but it was soon discovered that keeping these covers on the rollers was not easy. Traditional bonding methods, such as epoxy, contact cement, cyano-acrylic adhesive, and rubber compounds, were not entirely successful in permanently bonding the polyurethane coating to the aluminum rollers. Prolonged use and repeated stress sometimes tore off the coatings. Lack of sufficient weight capacity was a third problem with these wheels. These wheels had been designed and built for a maximum load capacity of only seventy-five pounds [Steub 1988]. Weight projections of the telepresence system were closer to four hundred pounds. Lastly the original design,
although innovative and unique, suffered from complexity. Each one of these omnidirectional wheels was made up of one hundred and three different parts. Disassembling a wheel to replace a failed part or to align a roller was very time consuming and difficult. To solve these problems and to build a reliable omnidirectional mobile base, new wheels were designed [Oldham et. al. 1990].

Figure 3.2: Computer Model of the New Omnidirectional Wheel
Mechanical Design

R. A. Cunningham, of the mechanical engineering faculty, provided the initial inspiration for the improved wheels by means a conceptual design sketch. A surface model of this design was built on a CAD system by the author to determine its feasibility. It was found that there were enough clearances between the rollers to place support brackets.

Figure 3.3: Support Bracket Aligned Along Roller Axis
The author worked very closely with the undergraduate design group during all stages of the design and fabrication process. Early in the design phase it was decided that the wheels would have ten rollers. Less than ten rollers would have resulted in insufficient clearance between rollers for the support bracket. The size of the support bracket required to hold the four hundred pounds projected weight of the telepresence system was the determining factor in deciding the overall size of the wheels.

Figure 3.4: Clearances for Support bracket Between Rollers

At its thinnest section, the support bracket was to be no less than a quarter of an inch in thickness. If we allowed for 1/16" clearance on either side of the bracket we
needed 3/8" distance between the rollers. The minimum diameter of a ten roller wheel that allowed such a clearance was approximately fourteen inches. This size was also considered proportionate to the rest of the telepresence system. Figure 3.3 shows the final dimensions of the support bracket for the rollers.

The calculation of the roller diameter was simple once the diameter of the wheel and the number of rollers was determined. The diameter of the rollers varies from 5.1" at the large end to 2.625" at the small end. The minimum shell thickness for the roller occurs at the large end and is 1/16". Simple shell calculations with this thickness showed it to be sufficient for a roller made out of 6061 aluminum.

---

**Figure 3.5: Dimensions of the Rollers**
For minimum rotary friction the rollers are provided with ball bearings, which allow the rollers to rotate freely about the support bracket. As the wheel rolls different parts of the roller contact the ground. This means that load on the ball bearing is not always radial. Depending upon which part of the roller is in contact with the ground the loading can change from $-16^\circ$ to $+16^\circ$ from radial. This clearly calls for some kind of an angular contact bearing.

Figure 3.6: Support Bracket Jigged for Final Turning

A Fafnir 5206K double row angular contact bearing was selected. This particular bearing is designed for loading up to $30^\circ$ off the vertical. Another option could have been a pair of single row angular contact bearings but this would
have unnecessarily complicated the design and added more parts. Figure 3.5 shows the dimensions of the roller. Both the rollers and the support brackets have been machined out of solid aluminum discs. Thirty identical rollers and brackets were to be machined. In addition to the large number of parts, some of the surfaces required on the parts were spherical. The author wrote programs for the mechanical engineering department's Monarch CNC lathe. This allowed the students in the design team to carry out the machining of the parts at a very accelerated rate. The CNC lathe not only speeded up the machining process but also allowed much closer tolerances to be kept. Since the bearings were to be press fit, high tolerances (+0.000, -0.002) were required on the bearing seats on both the roller and the support bracket.

Machining the brackets was a three step process. First rough cuts on the discs were made with a band saw. These were then bolted to a custom built jig and machined on the CNC lathe about the shaft axis. The final step consisted of fixing them in another jig and machining about an axis 360° from the shaft axis. This is shown in Figure 3.6.

Machining of the rollers was also a three step process. The aluminum discs for the rollers were first drilled on a vertical drill-press. They were then placed on a mandrill and the outer surface was machined on the CNC lathe.
Machining of the inside curve on the back end of the rollers was the third step.

Figure 3.7: Assembled Wheel

After completion of the rollers and support bracket machining, the rollers were sent out for vulcanization of a 3/32" coating of black poly-urethane rubber. Vulcanization is the only reliable method of bonding rubber to metal. Once the rubber coating was vulcanized on the rollers they
were press fit on the bearing and mounted on the support brackets.

Ten such units were assembled together on a hub to form a wheel. A mild steel axle shaft, 11/16" in diameter, is mounted in the center of the hub. The shaft, held by two pillow-blocks, attaches the wheel to the omnidirectional base. A photograph of the completed wheels is included in the appendix.

A permanent magnet DC servo motor drives each wheel. The motor has a gear-reduction built into the housing. A further 7 to 1 reduction is achieved by a pair of spur gears, one mounted on the output shaft of the motor and one on the axle shaft of the wheel. The motors themselves are installed on 1/4" aluminum mounts with mounting holes drilled through them. All gears are secured on their shafts by spring pins. These spring pins are sized to act as mechanical fuses in case of extreme overloads to the motors and drive trains.
Kinematic Analysis

The D-H matrix notation for the analysis of kinematic mechanisms introduced by Denavit and Hartenberg [Hartenberg 1955] is the commonly accepted standard in the kinematic analysis of robots and other kinematic mechanisms. The main drawback of this method is that it is applicable only to lower-pair joints. A lower pair joint is one created by contacting surfaces. Revolute, prismatic, screw and ball-socket joints are all lower pair joints. Joints created by point or line contacts between two surfaces are upper-pair joints. Examples of such joints are balls, discs and cylinders discs rolling on flat surfaces or on each other.

To correctly analyze the kinematics of the omnidirectional base, one needs to keep in mind the fact that omnidirectional wheels form a point contact with the ground. This is an upper pair joint. One method for analyzing upper-pairs is the Sheth-Uicker [Sheth 1971] generalized symbolic notation.

Sheth-Uicker generalized notation requires two instantaneously coincident coordinate systems to be defined at every upper-pair joint. One coordinate system is attached to the joint while another remains stationary with respect to the absolute coordinate system. A new instantaneously coincident coordinate system is assigned at
every instant such that the two coordinate systems have their origins at the same point and they are aligned.

Muir and Neuman [Muir 1987] used this notion of an instantaneously coincident coordinate system in their analysis of wheeled mobile robots. They defined an instantaneously coincident coordinate system $\tilde{A}$, for a system $A$, as one that has the same orientation and displacement in space as $A$ but is stationary relative to the absolute coordinate system. Velocities and acceleration of the joint link are not necessarily zero. All of this mathematical juggling gives us one advantage: It is simple to calculate the velocity and acceleration of the moving coordinate system relative to its instantaneously current position and orientation. Our interest lies in deriving an analytical expression for the position, velocity, and acceleration of the contact point of the wheel.

Figure 3.8: Coordinate Axis of the Omnidirectional Base
Since the current design of the wheel is kinematically similar to the original design [Adnan 1989], the kinematic analysis is identical. For each wheel, a local wheel coordinate system has its origin at the contact point of the wheel and the ground. The coordinate axis of the omnidirectional base has its origin at the center of the base of the robot. Figure 3.8 shows these coordinate axes. The local coordinate axis for each wheel is displaced $120^\circ$ radially and is located at distance $\rho$ from the center of the robot. $h$ defines the vertical distance between the origins of the omnidirectional base axis and the local wheel coordinate axes. All coordinate systems selected are orthogonal and right-handed.

If $^k\Pi_{c_i}$ is a homogenous transformation matrix and transforms the coordinates in the wheel $i$ coordinate system $C_i$ to the omnidirectional base coordinate system $R$. Then

$$R = ^k\Pi_{c_i}C_i$$  \hspace{1cm} (3.1)

For the three wheeled omnidirectional mobile robot a general transformation matrix can be written as

$$^k\Pi_{c_i} = \begin{bmatrix}
\cos^k \theta_{c_i} & -\sin^k \theta_{c_i} & 0 & \rho \cos^k \theta_{c_i} \\
\sin^k \theta_{c_i} & \cos^k \theta_{c_i} & 0 & \rho \sin^k \theta_{c_i} \\
0 & 0 & 1 & h \\
0 & 0 & 0 & 1
\end{bmatrix}$$  \hspace{1cm} (3.2)
Inserting angular values known from the kinematics described in Figure 3.1,

\[ r\theta_{c_1} = \frac{3\pi}{2} \]  \hspace{1cm} (3.3)

The transformation matrix for the first wheel may be written by substituting equation 3.3 in equation 3.2.

\[ r\Pi_{c_1} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & -\rho \\ 0 & 0 & 1 & h \\ 0 & 0 & 0 & 1 \end{bmatrix} \]  \hspace{1cm} (3.4)

In a similar manner, we can derive the coordinate transformation matrices for wheels 2 and 3, using

\[ r\theta_{c_2} = \frac{\pi}{6}, \quad r\theta_{c_1} = \frac{5\pi}{6} \]  \hspace{1cm} (3.5)

which are derived from the angular displacements of the second and third wheels from the robot coordinate system.

\[ r\Pi_{c_2} = \begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \rho \\ \frac{\sqrt{3}}{2} & \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \rho \\ \frac{\sqrt{3}}{2} & \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \rho \\ 0 & 0 & 1 & h \\ 0 & 0 & 0 & 1 \end{bmatrix} \]  \hspace{1cm} (3.6)
\[
\begin{bmatrix}
\frac{-\frac{\sqrt{3}}{2}}{2} & \frac{\sqrt{3}}{2} & \frac{1}{2} & 0 \\
\frac{\sqrt{3}}{2} & \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \\
0 & 0 & 1 & h \\
0 & 0 & 0 & 1 
\end{bmatrix}
\]
(3.7)

If \( \hat{\mathbf{x}} \) is the robot position vector and \( \hat{\mathbf{\theta}} \) is the wheel position vector then we can write these out as
\[
\hat{\mathbf{x}}^T = [x \ y \ \theta]
\]
(3.8)
\[
\hat{\mathbf{\theta}}^T = [\theta_x \ \theta_y \ \theta_z]
\]
(3.9)

Here vector \( \hat{\mathbf{x}} \) consists of the two translations and one rotation possible for the omnidirectional base. Vector \( \hat{\mathbf{\theta}} \) for each of the wheels describes the three possible rotations. Angle \( \theta_x \) is a measure of the rotation of the wheel about its axile. Angle \( \theta_y \) is the rotation angle of the roller about its axis and angle \( \theta_z \) is the rotation of the entire wheel about the z-axis.

Just as equation 3.2 relates \( \hat{\mathbf{x}} \) and \( \hat{\mathbf{\theta}} \), we can write a relation between \( \hat{\mathbf{\nu}} \) and \( \hat{\mathbf{\Omega}} \) defined below by (3.11) and (3.12). This is a Jacobian matrix or a matrix of the partial derivatives,
\[
J_i = \begin{bmatrix}
w_i \sin^R \theta_{ci} & r_i \cos^R \theta_{ci} & -\rho \sin^R \theta_{ci} \\
-w_i \cos^R \theta_{ci} & r_i \sin^R \theta_{ci} & \rho \cos^R \theta_{ci} \\
0 & 0 & 1
\end{bmatrix}
\]
(3.10)
and it relates the wheel velocity vector to the omnidirectional base velocity vector.

\[ \hat{\mathbf{V}}^r = \hat{\mathbf{X}}^r = [V_x, V_y, \omega] \]  

\[ \hat{\mathbf{\Omega}}^r = \hat{\mathbf{\Theta}}^r = [\omega_x, \omega_y, \omega_z] \]  

The Jacobian matrix for each wheel can now be derived.

For the first wheel the Jacobian matrix is given by.

\[
J_1 = \begin{bmatrix}
  -w & 0 & \rho \\
  0 & -r & 0 \\
  0 & 0 & 1
\end{bmatrix}
\]  

(3.13)

For the second and third wheel, respectively,

\[
J_2 = \begin{bmatrix}
  \frac{1}{2}w & \frac{\sqrt{3}}{2} r & -\frac{1}{2} \rho \\
  -\frac{\sqrt{3}}{2} w & \frac{1}{2} r & \frac{\sqrt{3}}{2} \rho \\
  0 & 0 & 1
\end{bmatrix}
\]  

(3.14)

\[
J_3 = \begin{bmatrix}
  \frac{1}{2}w & -\frac{\sqrt{3}}{2} r & -\frac{1}{2} \rho \\
  \frac{\sqrt{3}}{2} w & \frac{1}{2} r & -\frac{\sqrt{3}}{2} \rho \\
  0 & 0 & 1
\end{bmatrix}
\]  

(3.15)
From equation 3.10 through 3.15, we may write the forward solution for each of the three wheels of the omnidirectional base

\[
\begin{bmatrix}
V_x \\
V_y \\
\omega_z
\end{bmatrix} =
\begin{bmatrix}
-w & 0 & \rho \\
0 & -r & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\omega_{1x} \\
\omega_{2y} \\
\omega_{3z}
\end{bmatrix}
\] (3.16)

\[
\begin{bmatrix}
V_x \\
V_y \\
\omega_z
\end{bmatrix} =
\begin{bmatrix}
\frac{1}{2}w & \frac{\sqrt{3}}{2} & -\frac{1}{2}\rho \\
\frac{\sqrt{3}}{2} & 2 & \frac{\sqrt{3}}{2} \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\omega_{2x} \\
\omega_{2y} \\
\omega_{2z}
\end{bmatrix}
\] (3.17)

\[
\begin{bmatrix}
V_x \\
V_y \\
\omega_z
\end{bmatrix} =
\begin{bmatrix}
\frac{1}{2}w & -\frac{\sqrt{3}}{2} & -\frac{1}{2}\rho \\
\frac{\sqrt{3}}{2} & 2 & -\frac{\sqrt{3}}{2} \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\omega_{3x} \\
\omega_{3y} \\
\omega_{3z}
\end{bmatrix}
\] (3.18)

The next step is to determine the inverse kinematic solution for the omnidirectional base. The three wheels of the omnidirectional base, when in contact with the ground, create a closed kinematic chain. This constricts some of the degrees of freedom of the system. Thus even though each wheel has three degrees of freedom and there are three wheels the entire system has only three degrees of freedom.
For each wheel we can actuate one and only one degree of freedom. All other degrees of freedom are restricted.

Inversion of the Jacobian derived in equation 3.10 is the next step. Dividing the adjoint of the Jacobian by its determinant results in the inverse of the matrix.

\[
J_i^{-1} = \begin{bmatrix}
\sin \theta & -\cos \theta & -\rho \\
\frac{w}{w} & \frac{w}{w} & \frac{w}{w} \\
\cos \theta & \sin \theta & 0 \\
\frac{r}{r} & \frac{r}{r} & 0 \\
0 & 0 & 1
\end{bmatrix}
\] (3.19)

From equation 3.19, we can write the inverse Jacobian for each individual wheel

\[
J_1^{-1} = \begin{bmatrix}
-\frac{1}{2w} & 0 & -\rho \\
0 & \frac{1}{w} & 0 \\
0 & 0 & 1
\end{bmatrix}
\] (3.20)

\[
J_2^{-1} = \begin{bmatrix}
\frac{1}{2w} & -\frac{\sqrt{3}}{2w} & -\rho \\
\frac{\sqrt{3}}{2w} & \frac{1}{2w} & 0 \\
0 & 0 & 1
\end{bmatrix}
\] (3.21)
Using equation 3.20, 3.21, and 3.22, we can construct the overall inverse kinematics matrix.

\[
\begin{bmatrix}
\frac{1}{2w} & \frac{\sqrt{3}}{2w} & -\frac{\rho}{w} \\
\frac{\sqrt{3}}{2r} & \frac{1}{2r} & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
\omega_{1x} \\
\omega_{1y} \\
\omega_{1z} \\
\omega_{2x} \\
\omega_{2y} \\
\omega_{2z} \\
\omega_{3x} \\
\omega_{3y} \\
\omega_{3z}
\end{bmatrix} = \begin{bmatrix}
-\frac{1}{2w} & 0 & -\frac{\rho}{w} \\
0 & \frac{1}{r} & 0 \\
0 & 0 & 1 \\
\frac{1}{2w} & \frac{\sqrt{3}}{2w} & -\frac{\rho}{w} \\
\frac{\sqrt{3}}{2r} & \frac{1}{2r} & 0 \\
0 & 0 & 1 \\
\frac{1}{2w} & \frac{\sqrt{3}}{2w} & -\frac{\rho}{w} \\
\frac{\sqrt{3}}{2r} & \frac{1}{2r} & 0 \\
0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
V_x \\
V_y \\
\omega_z
\end{bmatrix}
\]

Since \( \theta_{1x} \), \( \theta_{2x} \), and \( \theta_{3x} \) are the only actuated degrees of freedom, we can eliminate the rows of the matrix representing the unactuated or constrained degrees of freedom. Thus we have computed the final inverse kinematic relation for the omnidirectional base.
\[
\begin{bmatrix}
\omega_{1z} \\
\omega_{2z} \\
\omega_{3z}
\end{bmatrix} = \frac{1}{w} \begin{bmatrix}
-1 & 0 & -\rho \\
\frac{1}{2} & -\frac{\sqrt{3}}{2} & -\rho \\
\frac{1}{2} & \frac{\sqrt{3}}{2} & -\rho
\end{bmatrix} \begin{bmatrix}
V_x \\
V_y \\
o_z
\end{bmatrix}
\]  
(3.24)

In concise notation the inverse kinematic solution can be written as,

\[
\hat{\Omega}_x = \frac{1}{w} M \hat{V}_r
\]  
(3.25)

Where \( \hat{\Omega}_x \) is the reduced wheel velocity vector, \( w \) is the wheel radius, \( M \) is the reduced inverse Jacobian, and \( \hat{V}_r \) is the omnidirectional base velocity vector in Cartesian coordinates.
Chapter 4: The Robot Torso

A less glamorous, but nonetheless crucial aspect of the design of the telepresence system was the construction of the robot torso or body. This torso had to be carefully designed to fit the requirements for the desired end-system. The torso needed to allow the telepresence system to be self-contained, to increase its utility in the development of real-world applications. This "self-contained-ness" dictated that the configuration of the torso be such that it could efficiently house all of the required equipment with a minimum of construction difficulty. Also the general dimensions of the robot needed to be practical for laboratory use. The torso needed to provide a housing for the control equipment, and also protection from accidental physical damage.

Since one of the goals of this research was to build a robot that was self-contained, all power supplies, communications, and control equipment needed to be carried onboard. Thus a torso or body for the robot was required that was large enough to carry the equipment but was light and rigid. After evaluating the different possibilities it was decided to construct the robot using welded tubular aluminum. One inch square 6065 aluminum tubing was used for construction of the robot torso. This has allowed us to maximize the frame rigidity while keeping the weight low.
Figure 4.1: Solid Model of the Robot Torso

The size and configuration of a mobile robot's wheel base are important factors in its stability and tipping resistance. As already discussed in chapter three, the wheels of the robot are aligned along the sides of an equilateral triangle. Thus a triangular torso would have been one option. However, this shape was not selected because its low area to circumference ratio would not have allowed sufficient room for all of the required onboard equipment. A circle, on the other hand, has an optimal area to circumference ratio, but it would have been difficult to
fabricate from aluminum tubing. A hexagonal cross-section was selected for the robot torso. Not only did this configuration allow adequate room inside the robot torso but it also allowed a generous wheel base. Solid modeling techniques [Autocad 1991] were used for visualizing the sizes and placements of the tubular members and the required onboard equipment. Figure 4.1 shows the solid model of the robot frame that was used design and construction.

![Diagram of the robot frame]

Figure 4.2: Side and Top view of the Torso Frame

The general dimensions of the robot are as follows. The robot torso stands 31 inches tall. At its smallest dimension it is 31 inches wide with a wheel base of 29 inches. Maneuvering through standard doorways is possible due to this small width. The large wheel base makes the system highly stable and resistant to tipping. Figure 4.2
shows two orthogonal views of the robot torso and also gives the overall dimensions.

In order to maximize frame strength, welded construction was used. Thin-walled aluminum tubing does not lend itself well to arc welding. A Heli-arc welding system, which shields the welding arc in an inert gas (helium) curtain, is the preferred method for welding aluminum. This system, available in the mechanical engineering department shop, was used in the construction of the frame. First, the aluminum tubes were cut to the correct sizes, clamped, and tack-welded together. The frame was then welded a little bit at a time at each joint to prevent heat buildup and consequential warping due to excessive thermal stresses. An undergraduate student* assisted the author in the construction of the tubular frame for the robot torso.

Load bearing members of the frame have been constructed from 1/8 inch wall thickness aluminum tube. These include the two beams that serve as mounts for the redundant robotic arm. Provisions have been made in the top of the frame to provide stable mounts for the camera tracking system. This system is described in detail in chapter seven. Heavy duty battery pans have been build in the bottom of the torso to carry the four gel-type batteries that power the robot.

* Louis A. Waters assisted the author with the construction of the robot frame in the mechanical engineering robotics laboratory, where he was working during the summer of 1990.
Mounts have been provided in the bottom of the frame for six Fafnir BR-11/16 pillow blocks. These pillow blocks support the three wheel axles and hence carry the load of the entire robot. Special felt gaskets have been placed between the frame and the pillow blocks to minimize shock.

Aluminum panels have been mounted on the insides of this frame to house the different control equipment and computers. Details regarding the location and type of equipment are given in chapter five. The outside of the robot torso is also covered with aluminum sheeting. To minimize the weight of the system, 1/32 inch thick aluminum sheet has been used for this purpose. The main function of this aluminum cover is to protect the delicate electronic components and wiring inside the robot from accidental damage. This metal cover has already proved its worth in preventing damage to the telepresence system during transportation to and from Johnson Space Center. The bead-blasted aluminum sheet also serves to enhance the aesthetic appearance of the robot. Removable cover panels on the sides and top of the robot torso allow easy access to the components inside for modifications, adjustments, and repairs.
Chapter 5: Control Hardware

To perform useful work with any robotic system, coordinated motion control is needed. On the mobile telepresence robotic system, this is achieved by using a distributed control scheme. The distributed control scheme is implemented using many different control devices and computers working cooperatively. These include onboard control devices, operator command computers, radio modems, ethernet network, graphic displays, and handcontrollers. The onboard control hardware of the mobile telepresence system consists of the motor control system, ethernet network, and control computers. The motor control system is composed of five components. These include a bus interface card, motor control processors, power amplifiers, incremental encoders, and DC motors.

There are two computers onboard the mobile telepresence robot: a supervisory computer and a base control computer. The functions performed by the supervisory computer include controlling the eight dof arm, passing control commands to the base control computer, and communicating with the operator base-station using serial ports, ethernet network, or the radio modem. The base control computer is responsible for the motion of the omnidirectional base. The two computers are linked together by an RS-232 serial link.
The supervisory computer is an Intel 386SX based PC compatible computer. It is equipped with a SCSI hard-drive, an ethernet interface, a joystick interface, a bus interface, a VGA graphic board, two serial ports, and a 9600 baud wireless modem. The reason for selecting a SCSI hard-drive was the potential capability of this type of drive to be shutdown by a software command. This could be useful to prevent damage to the hard-drive while the system is in motion. This feature of the system has not currently been exploited. The ethernet interface is used for controlling the robot from a remote location over the ethernet network. The radio modems are useful for controlling the system without any wire link between it and the operator. The joystick interface allows the local control of the telepresence system using a small three dof joystick mounted on top of the unit. The bus interface card is used to interface the computer to the motor controllers. The VGA graphic adapter and the separate monitor are only useful during debugging of system software and hardware and are disabled during normal operation. The base control computer is an Ampro single board computer [Ampro 1987]. It is based on the NEC V40 CPU. On the telepresence system, this computer has been equipped with a bus interface card, motor controllers, and power amplifiers. Software written by the author allows it to be used in one of two modes: direct control mode or indirect control mode. In the direct control
mode of operation, the computer takes inputs directly from a hardwired hand-controller and works as a teleoperated system in Cartesian coordinates. In the indirect control mode, the computer takes motion commands from the supervisory computer and executes them.

For remote operation of the telepresence system an operator command computer is used. This is located at the remote control station near the operator. The operator command computer is usually an Intel 386DX based computer equipped with an ethernet interface, radio modem, VGA graphic display, and serial ports. The location of this computer and the operator is not fixed. It can be a few feet from the telepresence system and connected to it with a serial cable; it can be in a different room using wireless remote modems for communications; or it can be a thousand miles away using the ethernet network to communicate with the telepresence system. This allows tremendous potential for flexibility in the use and operation of the system. Figure 5.1 gives a block diagram representation of the telepresence control system. Figure 5.2 shows the motor control sub-system. While the different joysticks and handcontrollers used in the system are depicted in Figure 5.3.
Figure 5.1: Block Diagram of The Control System
Figure 5.2: Block Diagram of Motor Control Sub-System
Figure 5.3: Three Different Handcontrollers Used with the System
Motor Control System

Generally closed loop computer control of DC servo motors is accomplished in one of two ways. One method involves use of the computer within the servo loop. An ADC* is used to monitor the motor position while a DAC** is used to convert the digital computer commands to analog signals for the motor. The method works well only if a few motors need to be controlled and the servo rates are not very high. The second method involves a dedicated device (a motor control system) for monitoring the motor position, generating the error signals and monitoring abnormal events such as the activation of limit switches.

When there is more than one or two motors to be controlled, then if one were to use the first control scheme the computer spends most of its time simply analyzing error signals and generating control commands for the motors. Very little processing time is left over for the calculation of the forward or inverse kinematics, path-planning, communication, and graphics. The second control scheme that utilizes a dedicated motor controller has an advantage. In order to accomplish closed loop control, the computer only needs to communicate with this motor controller. Once the

---

* Analog to Digital Converter.
** Digital to Analog Converter.
commands have been given to the motor controller the computer is free to perform other tasks. This allows ample time for the computer to do kinematics, path-planning, communication, and graphics. Control of the telepresence system required simultaneous supervision of thirteen closed loop servos. This provided more than ample justification to use dedicated motor controllers throughout the system. Figure 5.2 is a block diagram of the layout of the motor control sub-system.

Since large amounts of current may be required to run the DC motors, both closed loop control methods require the use of amplifiers. An amplifier is necessary not only to magnify the current and voltage but also to provide electrical isolation between the computer and the motors. Either analog or digital amplifiers may be used for this purpose. Traditional analog amplifiers use linear power transistors to directly scale up the input voltages and currents, while digital amplifiers use a technique known as Pulse Width Modulation (PWM) [Robillard 1983] to accomplish the same task.

Selection of the appropriate motor controllers for the robot was an important issue. Since the telepresence system involved simultaneous control of a large number of motors, it was important to select motor controllers that caused the least processor overhead, were cost effective, and were
easily available. The Hewlett Packard HCTL-1000 [Hewlett Packard 1990] is a single chip motion control IC that offers the advantages of speed, flexibility, compactness, low power consumption and ease of interfacing. Many different motor controllers are available that are based on this device and have somewhat similar capabilities. On the basis of its small size and cost effectiveness, the decision was made to select the LAB 40-6 [Computer Continuum 1987] smart two axis motor controllers. Each one of these dedicated motor control boards can control two DC servo motors. The motor controller allows a maximum servo update rate of eight kilo-Hertz to be maintained. This high servo update rate, used in conjunction with the high resolution optical motor shaft encoders, makes it possible to have smooth, vibration free motions. If the servo update rate is too low or the resolution of the shaft encoder is not high enough, then the controller can induce vibrations and jerkiness in the motion of the manipulator at slow speeds. Special care has been take to route the computer bus and encoder cables as far away as possible from the power lines. This has helped minimize power line interference. Power line interference can be the cause of unexplained crashes, system lockups, and untraceable bugs.

Prior experience with power amplifiers [Adnan 1987] had taught that the theoretical and actual power consumption of motors may greatly differ. This can be due to factors such
as friction, misalignment and inertial loads. During earlier experiments with the first prototype of the omnidirectional base, power amplifiers for the motors had often overheated and failed due to overload caused by the high starting torques of the DC motors. Back-emf generated by the motor during sudden stops was also sometimes responsible for failure of the amplifiers. For this reason, high continuous and peak capacity was a major consideration in the selection of power amplifiers. The decision was also made to stay with digital power amplification in order to benefit from its high efficiency, reliability, and noise immunity. Other considerations in the selection of suitable amplifiers were over-current shutoff, overheat shutoff, under-current shutoff, back-emf protection, optical isolation, and modular design. Back-emf is a surge of current caused by DC motors while accelerating or decelerating. This current can cause damage to control circuitry unless provisions are made for protection against it. Optical isolation is the complete electrical separation of the control and power sections of a circuit by using photo-diodes and photo-sensors. The two sections even have their own power supplies. Optical isolation prevents electrical noise caused by the motor brushes from making its way to the control section. Several digital power amplifiers were found that fulfilled these requirements. Specifications for amplifiers from many different sources
were studied and compared. The M-3 motor amplifier from Computer Continuum was selected. Modularity, compactness, high power output, and built-in safety features were among the salient features that led to its selection.

Software written by the author executes on the on-board computer and generates commands for the motor controller. These commands are routed to the motor controllers (HCTL-1000) through the bus interface card. The error signals generated by the motor controllers are in the form of PWM pulse train. These are fed to the power amplifiers through the optical isolators. The purpose of the optical isolators is to prevent high voltage noise (direct and induced) generated by the DC motors from making its way back to the control computer. The power amplifiers scale-up both the voltage and current of the error signal and drive the motors.

Incremental encoders have been used on all of the DC motors to obtain position feedback. The HEDS-5500 incremental encoders [Hewlett Packard 1987] used have a resolution of 1440 quadrature counts per revolution. They are insensitive to small axial and radial misalignments. Output from these encoders is in the form of two square waves that are 90° out of phase. The total number of pulses gives the relative position of the encoder, while the sign of the phase angle give the direction of motion. Feedback
from the motor encoders is passed back to the motor control after routing through the optical isolators. This prevents any induced noise from reaching the motor controllers or even the control computer. Manufacturer specification and technical data sheets for the motor controllers, amplifiers, bus interface and the computers are included in Appendix A.
Control Interfaces and Mediums

Telepresence and teleoperation systems require interactive control by a human operator. Thus, human interface devices and controllers are an essential part of any telepresence or teleoperation system. Among the different devices used for providing input to teleoperation systems are handcontrollers, master-slave arms, tracking systems, data-gloves and exo-skeletal devices. Handcontrollers are by far the most versatile of all. Hand controllers allow control of the teleoperation systems in Cartesian coordinates and thus have the flexibility to be used with almost any system.

Master-slave devices are usually pairs of similar arms: one, the slave, mimics all the joint configurations of the other, the master. The master arm does not need to be actuated. It is, however, instrumented to measure its joint angles. The drawbacks of the master-slave systems are the added expense of the master arm, lack of flexibility, i.e., the master can not be used for controlling a different arm, and the fact that they are cumbersome to use.

The data-glove [VPL Research] is a relatively new introduction to the field of control interfacing products. It consists of a close fitting glove, instrumented with optical fibers, such that the configurations of the
different fingers can be determined. The data-glove shares some of the same drawbacks of the master-slave systems. An additional problem with the use of the data glove is that it is not easy to accomplish a control transfer to another operator without having to either stop the system or using more that one data-glove in order to transfer control.

Exo-skeletal devices are mechanical linkages actually worn over parts of the operator's body. They can be as small as the Hand Exo-skeleton [Exos Corporation] which just covers the operator's hand, or as large as the proposed Man Amplifier [Rosheim 1989]. The drawbacks of exo-skeletal devices are also similar to those of master-slave setups in that they are cumbersome to use and lack flexibility.

The telepresence system has been designed to accept operator input from a variety of devices. These include dual three dof handcontrollers, joysticks, computer keyboard, and the Polhemus Isotracker system. The dual handcontrollers have been incorporated into the armrests of a chair. This handcontroller chair has been designed by Peter Galicki of NASA/JSC [Adnan 1991]. The operator can use one hand to control each handcontroller. A three dof rotational handcontroller with roll-pitch-yaw control has been installed on the right side, while a translational handcontroller with x-y-z control has been installed on the left. Triggers, buttons, sliders, and knobs are also
present to control auxiliary functions. A large panic button has also been provided. All of the degrees of freedom of the handcontroller have eight bits of resolution. This allows ample resolution and accuracy for most robotic control.

A three degree of freedom joystick handcontroller, built by the author, is mounted on the top of the telepresence system. This unit has only 5 bits of precision and also exhibits a zero drift. Thus it is only used for small scale movements and control.

Control of the telepresence system is also possible using the computer keyboard. The operator can use the cursor and function keys on the keyboard to control the different parts of the system. Using the arrow keys the operator can direct the omnidirectional base of the robot in different directions, while the function keys can be used to control the arm joints.

All of the different control interfaces described above can be linked with the robot using one of four available control mediums. These include direct link, hardwire cable serial communications, wireless modem communications and ethernet communication links. Software has been implemented in the system to make the control medium transparent to the operator. This in important for the reducing the modal disparities and helps create the feeling of telepresence.
All of the control interfaces can be directly linked to the onboard computer on the telepresence system, but this method is usually only used for testing and debugging. Hardwire serial RS-232 communications are another method for linking control interface devices to the system. This is useful when the distance between the devices and the telepresence system is short. Wireless remote modems, combined with the onboard batteries, allow the telepresence system to operate in a totally untethered manner. These modems allow distances of up to two hundred feet between the operator control interfaces and the system. Also, present on board the telepresence system is an ethernet interface. This interface, when used with the software developed by the author, can be used to control the telepresence system from any location on the ethernet network.
Chapter 6: Control Algorithms and Software

Programmable robots have evolved from numerical control machine tools, automated gantry cranes, and mechanical teleoperators. As a result, most of the commonly used control techniques for robots have their roots in the control of NC* machines. The main objective in the design of most control systems is to successfully direct the system to a desired position or state while faced with input disturbances. Implementation of robot controllers by early researchers in robotics, although more than adequate for the purposes for which they were designed, largely reflected the experience of the control community in designing NC machinery using open loop precision actuators and rigid members. Significant contributions were made by researchers in the design of minimum-time [Kahn 1969] and minimum-energy trajectories [Whitney 1969] using the open-loop control approach.

Presently, most industrial and laboratory robots are controlled using the independent joint control technique. Each joint is regarded as a separate sub-system and is controlled using a linear control law, such as PD or PID control. The main and obvious drawback of this technique is

* Numerical Control (NC) machine tools are a precursor of the Computer Numerical Control (CNC) machines.
the lack of any inherent coordination between the motion of the different robot joints. This shortcoming can be avoided by using control schemes that perform error analysis in operational space. Examples of such schemes are the global nonlinear control law and the resolved acceleration control law [Freund 1982][Johnson 1981]. In this research, the author has made use of the resolved acceleration control law to control the eight dof redundant robotic arm and the omnidirectional mobile base.

The software for the research has been written in the form of a modular hierarchical library of subroutines. Three robotic manipulators (i.e., the redundant arm, the RTX, and the PUMA 560) are supported by this library. Input devices that are supported by this library include the NASA-chair, the three dof joystick handcontroller, mouse, and computer keyboards. In addition, commands can also be generated using text scripts. For the control medium, the operator has a choice of using RS-232 serial cables, wireless modems, or ethernet. Care has been taken in the design of the software routines to keep them modular. This has been done by keeping the input-output parameters for all function performing similar tasks the same. For example, all manipulator control functions have the same number and types of parameters. The main benefit achieved from this modularity is that any of the input devices may be used with any of the manipulators while using any of the control
mediums. Control communication for the telepresence system is possible using the ethernet, serial ports, and the wireless radio-modems. In every case the communication programs have been written using interrupt driven schemes. Interrupt-driven communication requires the least computational overhead of all the different schemes available.

In addition to the control libraries written for the different manipulators and input devices, a software simulation of manipulator kinematics has also been produced. This simulation, written by the author, was later enhanced by Gu*.

* Shuxin Gu, a graduate student in the mechanical engineering department extended the author's four dof redundant manipulator simulation to a full eight dof simulation, during fall 1991.
Resolved-Acceleration Control

Individual joint PID control is a simplistic approach to robotic control. It involves the calculation of the initial and final positions of the manipulator in terms of joint coordinates. The joint velocities are then calculated and the individual joints are commanded to move with the calculated velocities. This is relatively simple, since the control problem can be broken down to the control of the individual joint. Unfortunately, this very aspect of the technique is also its major drawback. Since the joints are controlled independently of each other, any lag or lead in one or more of the joints has no effect on the commanded trajectory, and the manipulator may have significant deviations from the required Cartesian trajectory at any given instant.

Resolved-Acceleration control of mechanical manipulators involves specification of the manipulator accelerations and errors in Cartesian space [Luh 1980]. The main difference from individual joint PID control is in the control space. Whereas the individual joint PID control is performed in the joint space, the Resolved-Acceleration control is performed in the Cartesian space. Trajectory specification, feedback, trajectory errors, and the various gains are all specified in terms of Cartesian coordinates. If one or more of the manipulator joints leads or lags, the
resulting Cartesian trajectory changes. This causes a Cartesian trajectory error to be generated, and the algorithm automatically tries to bring the manipulator back on the desired trajectory. Thus at all times during the execution of the trajectory, the manipulator closely follows the trajectory command. This control technique requires the computation of both the forward and inverse kinematic solutions at each control iteration. The result is an increased computational requirement. It was determined that the Intel-386 based control computer on-board the telepresence system was capable of performing these calculations at a rate that was high enough for stable control.

For a serial link manipulator, the Resolved-Acceleration control law can be written as:

\[ \ddot{q}(t) = N^{-1}(q)\left( \ddot{x}^d(t) + k_1(\ddot{x}^d(t) - \ddot{x}(t)) + k_2 e(t) - \dot{N}(q, \dot{q})\dot{q}(t) \right) \]  

(6.1)

Where, \( q \) is the joint position vector, \( x \) is the Jacobian Cartesian position vector, \( N \) is a 6 x 6 kinematic configuration matrix, \( e \) is the positional error vector, and \( k_1 \) and \( k_2 \) specify the system gains. The implementation of the Resolved-Acceleration control algorithm on the telepresence system is based on the control law specified by equation (6.1).
Figure 6.1: Resolved-Acceleration Control Block Diagram

The actual implementation of the control algorithm on the telepresence system differs slightly from the original John Luh implementation [Luh 1980]. The main differences arise from the fact that the control hardware for the system requires current input to the motors. Luh's simulation dealt with supplying torques directly to the manipulator joints.

The actual control algorithm has been implemented in the C programming language. This allows for easy portability to other platforms without unnecessarily sacrificing speed or performance. Without using any specialized floating point co-processor, a control rate or
sampling frequency of 35 Hz. has been achieved on an Intel-386 based computer. This is more than adequate for stable control of the manipulator. Use of a floating point co-processor results in an order of magnitude improvement in the sampling frequency and allows the processor to have ample idle time for performing communication and housekeeping functions.

PROGRAM: Parametric Robot Control Subroutine

DESCRIPTION: This function controls a robot manipulator or base using the resolved-acceleration control algorithm. Command trajectory is from one of the control input devices.

INPUT PARAMETERS: Running time, time step, current robot position, velocity & acceleration, desired velocity vector

INPUT DEVICES: NASA chair, 3 dof handcontroller, radio modem, ethernet, pre-programmed script.

OUTPUT: None

SIDE EFFECT: Robot manipulator or omnidirectional base follows the input trajectory with only small deviations.

BEGIN:
dt=time step
iterations=0
current time=0
start time=READ CLOCK

WHILE current time LESS THAN running Time
    INCREMENT iterations
    desired trajectory=READ CONTROL INPUT DEVICE
    current joints=READ JOINT ENCODERS
    current trajectory=FORWARD KINEMATICS
    error trajectory=desired trajectory-current trajectory
    command velocity=RESOLVE ACCELERATION CONTROL LAW
    command joints=INVERSE KINEMATICS
    CONTROL JOINTS
    current time=READ CLOCK-start time
    time step=current time/iterations
    CHECK FOR ERROR

CONTINUE

END:

Figure 6.2: Pseudo-Code for Resolved-Acceleration Control
The Resolved-Acceleration control algorithm, as implemented on the telepresence system by the author, has been presented in pseudo-code form in Figure 6.2. Care has been taken in the actual coding of the algorithm to ensure efficient execution. Software traps* have been created to provide for graceful shutdown of the system. If a divide-by-zero error, amplifier over-current error, or limit switch triggering error takes place the system turns off all power to the motors and goes into an idle state. The complete text of the software for the project has been included in Appendix C. It contains the specific functions for performing the control of the redundant arm, the omnidirectional mobile base, the head tracking camera platform, the RTX, and the PUMA robot. All of the control routines have been designed to be modular, and are independent of the control interface. This means that the same control routines can be used with any input device available in the robotics lab: like the NASA handcontroller chair, the three dof joystick, the keyboard, or the trackball. In addition these control routines also allow trajectory input from an independent control computer communicating with the telepresence robot controller over

* Software traps or interrupt routines are program segments written for special purposes and placed in special locations. These segments are executed automatically by the CPU when a hardware or software condition is met.
the serial port, the radio modems or the ethernet. This flexibility means that the programmer using these functions for control of the telepresence system only needs to change one input device name in his program to use a different input device. Software written for the RTX and the PUMA robots in the robotics laboratory, like that for the telepresence system, allows that robot to be controlled in a manner suitable for interactive control. This software too has been written in the form of a flexible and modular library of control routines.
Impedance Control

Use of force feedback is important for robotic control in unstructured environments. One of the goals of the telepresence system was that it be usable in remote and unmapped regions, hence it was decided to implement force feedback in the control system. Among the many different techniques of incorporating force feedback in control, hybrid force control and impedance control are the most commonly used [Lasky 1991].

Figure 6.3: Impedance Control Block Diagram

Hybrid control is mostly used when the manipulator end-effector is in physical contact with a surface. Movement along the surface is possible and movement perpendicular to
it is constrained. This prevents damage to both the work surface and the manipulator. Impedance control is based on specifying a desired dynamic relationship between the velocity of the manipulator and the force exerted by the manipulator on the environment [Hogan 1985]. Simply put this means that impedance control scales the command velocity of the end-effector depending on the external force. For this particular implementation of resolved acceleration control the impedance control law can be written as,

\[ \ddot{q}(t) = N^{-1}(q)[\ddot{x}^d(t) + k_1(\dot{x}^d(t) - \dot{x}(t)) + k_2c(t) - \dot{N}(q, \dot{q})\dot{q}(t) ] - N^T \tau^e \]  (6.2)

where \( \tau^e \) is the joint torque vector caused by the external force, \( f \), and is given by,

\[ \tau^e = N^T f \]  (6.3)

In a teleoperated environment, there is an advantage in using impedance control over hybrid control. The advantage mainly arises from the fact that, in the absence of any contact forces, the impedance controller acts as a resolved-rate controller. This means that the same control algorithm can be used whether or not the robot is in contact with other objects. No esoteric mathematical techniques need to be used during the transition from contact to non-contact segments of the motion trajectory. Figure 6.3 gives the
block diagram of the impedance control algorithm as implemented on the telepresence system.

**PROGRAM:** Impedance Control Subroutine

**DESCRIPTION:** This function controls a robot manipulator or base using impedance control. Command trajectory input is taken from any of the control input devices.

**INPUT PARAMETERS:** Running time, time step, current robot position, velocity & acceleration, desired velocity vector

**INPUT DEVICES:** NASA chair, 3 dof handcontroller, radio modem, ethernet, pre-programmed script.

**OUTPUT:** None

**SIDE EFFECT:** Robot manipulator follows the input trajectory and reacts to any applied forces by modifying its trajectory to minimize these forces while staying as close to the command trajectory as possible.

**BEGIN:**

\[ dt=\text{time step} \]
\[ \text{iterations}=0 \]
\[ \text{current time}=0 \]
\[ \text{start time}=\text{READ CLOCK} \]

**WHILE** current time **LESS THAN** running Time

**INCREMENT** iterations

current forces=\text{READ FORCE TRANSDUCER}
desired trajectory=\text{READ CONTROL INPUT DEVICE}
current joints=\text{READ JOINT ENCODERS}
current trajectory=\text{FORWARD KINEMATICS}
error trajectory=desired trajectory-current trajectory
command velocity=\text{IMPEDANCE CONTROL LAW}
command joints=\text{INVERSE KINEMATICS}

**CONTROL JOINTS**
current time=\text{READ CLOCK}-start time
time step=\text{current time/iterations}
CHECK FOR ERROR

**CONTINUE**

**END:**

Figure 6.4: Pseudo Code for Impedance Control

The force moment transducer available in the robotics lab for force feedback was the JR3 [JR3 Inc.]. This is a six axis force-moment device. At the time that this part of
the research was conducted the redundant arm was not functional. Thus, the JR3 was not physically mounted on the redundant arm. Instead, the author installed the force-moment transducer on the RTX robot [UMI]. The RTX is a six dof cylindrical coordinate robot. The JR3 has been mounted between the sixth link and the gripper. Hence it measures the forces exerted on the gripper in terms of the tool coordinate system. These forces need to be transformed to the world coordinate system using the inverse tool coordinate transformation. For the sake of consistency, the author used the Resolved-Acceleration control scheme as the basic framework and layered the impedance control algorithm on it. The pseudo-code for the author's implementation of impedance control for the RTX is given in Figure 6.4.

The program code written by the author was tested in actual working demonstrations where different operators were able to use the system. One such demonstration was the implementation of force impedance teleoperated control for a simulated ORU* replacement task. In this demonstration, a mockup of a section of the space-station freedom with removable ORU was used. The operator uses the dual three dof handcontrollers on the NASA chair to control the RTX robot and to guide the ORU into the close fitting receptacle

* Orbital Replacement Unit (ORU) are multifunctional modules that can be changed in orbit on the NASA space-station.
on the mockup. This task would normally not be feasible with the RTX robot due to its poor repeatability. However, the complementary use of active force impedance control with teleoperation made this task relatively easy. The impedance control algorithm successfully guides the ORU into the target receptacle using force feedback from the wrist mounted JR3 force moment transducer. At other times, the operator was allowed to remotely control the ORU replacement task using video monitors. The lack of depth perception on the video link made such a task more challenging, but due the use of force impedance control any small errors on part of the operator were corrected by the system. The source code for this and other control programs discussed in this thesis is included in Appendix C.
Chapter 7: Vision Feedback

Vision is the basic feedback mechanism that humans use for interacting with their environment. It seems that the absence of vision severely curtails the interaction capabilities of humans [Worochel 1952]. Although visually impaired persons can perform the majority of tasks using tactile, kinesthetic, and auditory feedback, there is a loss of efficiency in task performance that occurs due to the lack of visual feedback. This loss of efficiency made it apparent that vision would have to be an integral part of the telepresence system.

Monocular vision has been used extensively in remote teleoperation, handling and guidance operations. While adequate for non-contact tasks and coarse handling, monocular vision severely limits the capabilities of the operator [Martin 1985][Ariyaeinia 1989]. Fine manipulation, precision tasks, and tasks performed in cluttered environments require depth perception. Depth perception is the ability of the human eyes and the visual cortex to perceive the distance and depth of objects within the field of view. This ability is extremely important for doing precision work at close range. Depth perception coupled with dexterity can be a very big asset in teleoperation. It seems that humans use three main types of cues in the visual scene to perceive depth. Pictorial cues
are one type of depth cue. These pictorial cues include occlusion, obstruction, shading and perspective. For many centuries western artists have exploited these cues in their work to present to the observer life-like three dimensional scenes in their paintings. The perspective cues were popularized by some Italian artists during the seventeenth century (the Renaissance) to trick certain aspects of the human depth perception into enhancing the illusion of depth [Janson 1986].

Another type of depth cue that the human visual system exploit is the binocular parallax [Rheingold 1991]. The binocular parallax arises from the fact that the human eyes have overlapping fields of view. While viewing three dimensional scenes, the images perceived by the two eyes are displaced through a precise angle. These images are not similar. This fact was first exploited in 1833 by an inventor, Wheatstone, and later perfected in 1844 by David Brewster in the form of the stereoscope [Rheingold 1991]. In its modern incarnation, a stereo-pair of video cameras is used to record the visual scene. This stereo scene can then be presented to the user using stereo pairs of helmet mounted CRTs or electronic shutter glasses. In case of the stereo pairs of CRTs, the two images are presented simultaneously; while in the case of the electronic shutter glasses, they are time-multiplexed.
The third type of depth cue that humans use is called the motion parallax. This arises from the fact that when we move our heads, the visual scene changes. Nearer objects appear to move more than distant objects. The visual cortex uses these differences in detectable motion to determine the relative distance between the different objects in the scene. In order to properly exploit this cue, the visual scene presented to operator must be appropriately adjusted in response to his head motion. This can be done by a head tracking camera system. The system can be as simple as a single degree of freedom system that tracks the turning of the head or it can have as many as four degrees of freedom: pan, tilt, roll and fore-aft translation (extension).

It seems then that a proper head tracking camera system used in conjunction with stereo-optic displays would be able to provide the user the three most important affordances* [Rheingold 1985] necessary for depth perception: these are pictorial cues, binocular parallax, and motion parallax.

This research involved the use of two different vision feedback and tracking systems. The first system that was used was a helmet-mounted stereo display and head tracking camera platform developed by Peter Galicki of NASA Johnson

* Affordances are described by psychologists as perceptual streams that enable us to create the world as we move through it.
Space Center and Jim Brock of CSC**. The three dof head tracking stereo camera system was mounted on top of the telepresence robot. The operator wears the helmet mounted display and is able to control the telepresence system using the NASA chair handcontrollers. The other system is a stereo-pair of cameras mounted on a Rhino [Sandhu 1986] two dof pan-tilt base. The image is displayed on a monitor and the operator views the stereo image while wearing a pair of electronic shutter glasses. The operator controls the camera platform by means of a trackball. Left-right motion of the trackball control the pan, while the up-down movement controls the tilt.

Figure 7.1: Configuration of the Original Head-Tracker

** Computer Science Corporation
The helmet mounted display and head tracking camera platform was designed at NASA Johnson Space Center. The display system consisted of a helmet in which was mounted a pair of miniature CRTs: one in front of each eye. The inter-ocular distance between the two CRTs is adjustable in order to accommodate different operators. A Polhemus [Polhemus] Isotracker receiver module is mounted on top of the helmet. The transmitter module for the Isotracker system is mounted on a stationary support a few feet above the operator's work area and is not physically connected to the receiver. The head tracking stereo camera platform consists of three rotary tables mounted orthogonal to each other. This setup results in three dof for the camera platform. The head-tracking camera platform is shown in Figure 7.1. Monochrome video signals from two cameras mounted on top of the camera platform are fed to the CRTs in the helmet. The convergence angle between the two cameras is adjustable for correct binocular parallax. Each axis is actuated by a stepper motor. Stepper motor drivers are used to drive these stepper motors from a computer.

A control program executing on the computer takes input from the Isotracker to calculate the orientation of the helmet with respect to the operator's workstation. This information is used to calculate the control commands to be
sent to the stepper motors on the camera platform. The net result is that as the operator turns, rolls, or tilts his head the camera platform follows his movements. This tracking of the head by the cameras and the corresponding change in the visual scene displayed on the helmet CRTs for the operator results in a feeling of visual telepresence.

Figure 7.2: Camera Platform Developed at NASA

This camera platform has some shortcomings. Among these is the problem of jerky motion of the platform. This is a result of using coarse-stepped stepper motors. Besides the problem of jerky motion the camera platform also displays a lag in tracking the head motions of the operator. Work is currently underway at NASA Johnson Space Center to design a second generation camera platform that would correct the design flaws in the first system.
Rice Stereo Display and Camera Platform

The Rice Stereo display and tracking platform has been integrated and programmed by the author. The system is made up of a pair of electronic LCD shutter type glasses [Visual Research] with frame switcher, a pair of color cameras and a Rhino two dof platform. A control computer is used to control the motion of the camera platform by means of trackball inputs from the operator.

![Figure 7.3: Rice Stereo Camera Platform](image)

The electronic LCD shutter glasses and frame switcher [Visual Research] work by interlacing the video images recorded by the left and right cameras. Interlacing is the mechanism of switching between two images on the CRT at a
rate of 60 Hz. By quickly switching the left and right images and at the same time blanking out the right and left lenses of the electronic LCD shutter glasses the illusion of three dimensionality can be created. The human brain fuses the two images together and picks up the depth cues of binocular parallax. This fused image appears to be three dimensional. The pair of color CCD* cameras are mounted on the Rhino two dof platform. The visual scene recorded by these cameras is routed to a color monitor through the BTX-3D frame switcher.

Figure 7.4: Control System for the Rice Camera Platform

An operator wearing a pair of electronic LCD shutter glasses can view the stereo scene on the monitor. A

* Charge Coupled Device
trackball connected to a control computer is used by the operator to guide the aim the platform. The control computer generates the correct velocity commands for the DC servo motors driving the camera platform and drives it under velocity control. The concept of using a hand operated pointing device for guiding the camera platform is not intuitive, and goes against the grain of telepresence. Ideally a head position sensor such as the Polhemus Isotracker should be used to provide control input to the camera platform control system. The Isotracker however was very expensive. Moreover the sensing of the head position was not considered to be an essential capability. Thus the decision was made by the author to control the camera platform by means of a trackball.
Chapter 8: Future Extensions of Research

The research described in this thesis is meant to lay down a foundation for a mobile robotic testbed for telepresence research. By using the telepresence system designed and implemented in this research, future researchers can perform experiments in telepresence and verify their algorithms and control strategies. Although the system described in the preceding chapters has laid the ground work for a telepresence testbed, some issues have not been addressed by it.

One of the most important of the issues not addressed in this research, and perhaps the one that should be resolved first, is the lack of an operational end-effector on the telepresence manipulator. Currently a mechanically complete four fingered end-effector is mounted on the last joint of the redundant arm. This end-effector, however, is not operational. Work needs to be done to design and implement a control system for this eleven dof end-effector. This control system could simply be an extension of the work done for the dextrous nine dof end-effector*. Grasp analysis research done by Chen can also be applied to this dexterous end-effector for semiautonomous grasping [Chen

---

* Sarmad Adnan and Louis A. Waters designed and build the nine dof end-effector as part of a research project during the summer of 1990.
Conversely an exo-skeletal device such as the Dexterous Hand Master [Exos Inc. 1989] or an instrumented glove such as the VPL Dataglove [VPL Research 1991] may be used to operate the end-effector in a master-slave setup.

A second avenue of research that can be pursued is the use of force feedback on the redundant manipulator. Force feedback can be very useful in remote teleoperation and telepresence tasks, specially when the tasks involve contact and fine manipulation. The author has implemented impedance control on the RTX robot as described in a previous chapter. However, impedance control could not be ported at that time on the redundant manipulator because its mechanical construction and controller design were still in progress. Now that the redundant manipulator is in a usable condition, a future researcher could port the impedance control algorithms to this manipulator. This would involve mounting the JR3 force-moment transducer on the redundant arm. The end-effector mounting flange on the redundant arm has been designed with this in mind.

Although not absolutely necessary, for telepresence and teleoperation tasks, redundancy resolution of the eight dof manipulator, mounted on the mobile telepresence testbed, is an important consideration. The current setup allows the operator to control the self-motion of the two redundant joints using knobs on the NASA handcontroller chair. One
direction for research could be the implementation of some redundancy resolution scheme. The use of energy or velocity minimization is very computationally expensive and tedious. A neural network based scheme that allows transparent control of the redundant links could be implemented. This would free the operator to concentrate more on the task rather than the tool.

The current system has stereo color cameras for three dimensional video feedback. These cameras require video cables to be run from the operator to the telepresence system. Since the rest of the system is completely untethered, this is awkward. Wireless video transmitters on the mobile robot coupled with receivers near the operator could be one solution. Preliminary testing has been done with one video transmitter and the results have been encouraging. It seems that indoor transmission range of the video transmitter that was tested is nearly a hundred feet.

The author has implemented the resolved acceleration control scheme on the redundant manipulator and the omnidirectional mobile base that are part of the telepresence system. Fuzzy logic control of a simulation of the omnidirectional base has been demonstrated in a preliminary form. A more robust fuzzy logic controller for the mobile base and the manipulator could replace the current control law.
Currently the operator uses a track-ball to control the stereo camera platform. This is not intuitive and detracts from the concept of telepresence. A head position sensor could be used to control the camera platform and thus relieve the operator from the burden of manual control of the camera platform. The NASA head tracking camera platform already incorporates this feature.

The issues of time delay have not been addressed in this research. This is important if the remote telepresence system is far removed from the operator. Some researchers have proposed the use of time and position clutches for solving this problem [Conway 1990]. The author feels that the direct use of time clutches and brakes detracts from the spirit of telepresence since their usage is not a natural action. The solution envisioned by the author is a model based hybrid telepresence-virtual reality system, that would use forward simulation to present to the operator a virtual workspace. The actual task would occur after the time lag. If necessary, the task feedback from the physical environment could be used to checkpoint the progress of the virtual task in the virtual environment at appropriate intervals. The enabling technologies for such a system currently exist and there is no reason a researcher could not utilize them to realize a coherent system for telepresence with time-delay.
Chapter 9: Summary

The research documented in this thesis has resulted in the implementation of a fully functional mobile testbed for remote telepresence research. The testbed provides a system of hardware and software to enable future researchers to delve deeper into the fields of telepresence and tele-existence. Although other researchers have also designed systems for telepresence, this system is unique by virtue of its omnidirectional mobility, redundant manipulator, and untethered operation capabilities. Plans are already underway at the mechanical engineering robotics laboratory to extend this work by incorporating some of the extensions to this research alluded to by the author in chapter 8.

Throughout this research the goals of extensibility and modularity have been kept in mind. This is apparent in both the hardware and the software. In an attempt to keep the hardware modular and extensible the computers used on-board the telepresence system are similar to the ones used everywhere else in the robotics laboratory. Thus programs can be developed and debugged on any one of the computers and simply transferred to the telepresence system. The same motor controllers and power amplifiers have been used for controlling all of the different joints and wheels of the robot. Moreover each individual amplifier and controller can be easily unplugged for replacement or repair. Due to
large variations in the torque requirements it was not feasible to use the same type of motor throughout the system. However the same motor encoders were used on all of the motors. These encoders allow for easy replacement for replacement or service. The extensive software library written for the control of the system also displays modularity and extensibility. It is organized in a hierarchical manner allowing other researchers to program the telepresence system using higher level routines. All of the intricacies and complications of low-level programming for interfacing and control are hidden behind a suite of easy to use routines.

The operational capabilities of the mobile telepresence system have been demonstrated on numerous occasions. An operator can guide the system with NASA handcontroller chair while wearing either the NASA helmet mounted displays or the LCD shutter glasses. In the case of the NASA helmet, the head tracking is automatic while in the case of the Rice stereo display the cameras are guided by means of a track-ball. In both cases the stereo cameras give the operator a view with depth where the objects have a three dimensional quality. The redundant robotic arm mounted on the telepresence system can be controlled by either the keyboard or the handcontroller chair. In either case the control of the self motion is manual.
In order to keep the control of the system flexible, provisions have been made in the software to allow it to be controlled using the ethernet, serial links, or wireless radio modems. In addition, any of the control devices such as the handcontroller chair, the keyboard or the three dof joystick can be directly plugged into the system. Onboard power supplies for the computers and the actuators allow untethered use. One of the limitations of the system is the short battery life. The current batteries allow approximately one half hour of continuous operation. This can be extended by adding another battery. Another minor problem with the system is the backlash in the first joint of the manipulator. This is a design flaw aggravated by poor machining. This problem can be easily corrected by replacing the aluminum flange coupling with a steel one. One problem with the software that also needs to be looked into is that it is not yet aware of the brakes installed on the first two joints of the manipulator. Thus a small amount of programming effort needs to be expended to add this enhancement to improve the system.

In conclusion the mobile telepresence testbed has been implemented successfully with some minor flaws. Its capabilities have been demonstrated and it is ready for use by other researchers to explore its potential and usefulness.
References


Kahn, M.E. 1969. The near minimum-time control of open-loop articulated kinematic chains. Stanford Artificial Intelligence Laboratory. AIM 106 (December).


Maple Rel. V. University of Waterloo, Waterloo, Ontario, Canada.


Nowlin, William. 1991. Experimental results on Bayesian algorithms for interpreting compliant tactile sensing


Appendix A: Manufacturer Specification Sheets

DC Servo Motors

GNM-4150 Motor

PM DC Motors Series GNM 4150

- 6 Standard Models in a 70 mm Ø Size (2.8" Ø)
- Fits Our Gearhead Series G11.2
- Available with DC Tachs, and Encoders up to 1024 Pulses per Revolution.
- Available in 24, 34, 42, and 180 Volt Versions Standard.

Continuous Duty Rating

Speeds up to 3,000 RPM  
Torque up to 50 oz-in (0.25 ft.lbs)  
Power Output up to 100 Watts

Electrical Specifications:

For Motor Type GNM  
4150  4150  4150  4150  4150

Supply Voltage nom. (Volts)  
24  24  34  42  180

Armature Resistance (Ohm) ±10%  
1.27  0.52  0.87  1.27  66.50

Max. Power Output (Watts)  
94  220  212  218  105

Max. Efficiency (%)  
74  72  72  72  72

No Load Speed (RPM) ±7%  
1875  3308  3348  3270  1912

No Load Current (mA) +10%−25%  
291  811  580  458  51

Fricition Torque (@No Load Speed) (oz-in)  
4.96  7.79  7.79  7.79  6.31

Stall Torque (oz-in)  
270  360  343  360  297

Velocity Constant (RPM/Volt)  
79.59  140.87  100.75  79.59  10.85

Back EMF Constant (mV/RPM)  
12.56  7.10  9.92  12.56  92.14

Torque Constant (oz-in/Amp)  
17.00  9.60  13.42  17.00  124.61

Maximum Current (Amp)  
24  42  29  24  3.25

Nominal Speed (RPM)  
1600  3000  3000  3000  1600

Mechanical Specifications:

Mechanical Time Constant (mS)  
8.3  10.6  9.1  8.3  8.1

Armature Inertia (x10^-3 oz-in-sec^2)  
13.4  13.4  13.4  13.4  13.4

Maximum Ambient Temperature (°C)  
40  40  40  40  40

Thermal Resistances Rotor to Ambient (°C/W)  
5.1  3.2  3.2  3.2  5.1

Thermal Time Constant (min)  
40  40  40  40  40

Weight (oz)  
72  72  72  72  72

Rotor Temperature Range  
-20°C to +100°C
PM DC Motors Series GNM 4150

Dimensional Outlines:

Dimensions are in mm (in.)

Ordering Information:

Motor Series GNM 4150, Nominal Voltage 24V, Nominal Speed 3K
PM DC Motors Series GNM 4125

- 6 Standard Models in a 70mm Ø Size (2.8" Ø).
- Fits Our Gearhead Series G8N.
- Available with DC Tachs, and Encoders up to 1024 Pulses per Revolution.
- Available in 24, 34, 42, and 180 Volts Versions Standard.

Continuous Duty Ratings:

- Speeds up to 3,000 RPM
- Torque up to 30 oz-in (0.15 ft.lbs)
- Power Outputs up to 60 Watts

Electrical Specifications:

<table>
<thead>
<tr>
<th></th>
<th>4125</th>
<th>4125</th>
<th>4125</th>
<th>4125</th>
<th>4125</th>
</tr>
</thead>
<tbody>
<tr>
<td>For Motor Type GNM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supply Voltage nom. (Volts)</td>
<td>24</td>
<td>24</td>
<td>34</td>
<td>42</td>
<td>180</td>
</tr>
<tr>
<td>Armature Resistance (Ohm) ±10%</td>
<td>2.66</td>
<td>1.00</td>
<td>2.04</td>
<td>2.66</td>
<td>151</td>
</tr>
<tr>
<td>Max. Power Output (Watts)</td>
<td>46</td>
<td>122</td>
<td>115</td>
<td>125</td>
<td>45</td>
</tr>
<tr>
<td>Max. Efficiency (%)</td>
<td>69</td>
<td>69</td>
<td>68</td>
<td>69</td>
<td>66</td>
</tr>
<tr>
<td>No Load Speed (RPM) ±7%</td>
<td>2032</td>
<td>3460</td>
<td>3495</td>
<td>3553</td>
<td>2058</td>
</tr>
<tr>
<td>No Load Current (mA) +10%-25%</td>
<td>227</td>
<td>622</td>
<td>444</td>
<td>364</td>
<td>37</td>
</tr>
<tr>
<td>Friction Torque (@No Load Speed) (oz-in)</td>
<td>3.54</td>
<td>5.66</td>
<td>5.66</td>
<td>5.66</td>
<td>4.25</td>
</tr>
<tr>
<td>Stall Torque (oz-in)</td>
<td>123</td>
<td>191</td>
<td>178</td>
<td>191</td>
<td>117</td>
</tr>
<tr>
<td>Velocity Constant (RPM/Volt)</td>
<td>67.18</td>
<td>148.54</td>
<td>106.12</td>
<td>87.18</td>
<td>11.86</td>
</tr>
<tr>
<td>Back EMF Constant (mV/RPM)</td>
<td>11.47</td>
<td>6.73</td>
<td>9.42</td>
<td>11.47</td>
<td>84.3</td>
</tr>
<tr>
<td>Torque Constant (oz-in/Amp)</td>
<td>15.58</td>
<td>9.10</td>
<td>12.74</td>
<td>15.58</td>
<td>113.28</td>
</tr>
<tr>
<td>Maximum Current (Amp)</td>
<td>14</td>
<td>24</td>
<td>16</td>
<td>14</td>
<td>1.9</td>
</tr>
<tr>
<td>Nominal Speed (RPM)</td>
<td>1600</td>
<td>3000</td>
<td>3000</td>
<td>3000</td>
<td>1600</td>
</tr>
</tbody>
</table>

Mechanical Specifications:

<table>
<thead>
<tr>
<th></th>
<th>12.3</th>
<th>13.5</th>
<th>14.1</th>
<th>12.3</th>
<th>13.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechanical Time Constant (mS)</td>
<td>7.9</td>
<td>7.9</td>
<td>7.9</td>
<td>7.9</td>
<td>7.9</td>
</tr>
<tr>
<td>Armature Inertia (x10^-3 oz-in-Sec^2)</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Maximum Ambient Temperature (°C)</td>
<td>6.1</td>
<td>3.95</td>
<td>3.95</td>
<td>3.95</td>
<td>5.5</td>
</tr>
<tr>
<td>Thermal Resistances Rotor to Ambient (°C/W)</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Thermal Time Constant (min)</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>-20°C to +100°C</td>
</tr>
<tr>
<td>Weight (oz)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotor Temperature Range</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PM DC Motors Series GNM 4125

Dimensional Outlines:

Dimensions are in mm (in.)

Ordering Information:

Motor Series: GNM 4125, 24V, 3K
## Motor Controllers

### HCTL-1000 pin description

#### INPUT/OUTPUT SIGNALS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Pin Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00/DB0-AD5/DBS</td>
<td>2-7</td>
<td>Address/Data bus — Lower 6 bits of 8-bit I/O port which are multiplexed between address and data.</td>
</tr>
<tr>
<td>D6, D7</td>
<td>8, 9</td>
<td>Data bus — Upper 2 bits of 8-bit I/O port used for data only.</td>
</tr>
</tbody>
</table>

#### INPUT SIGNALS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Pin Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHA/CHB</td>
<td>31, 30</td>
<td>Channel A,B — Input pins for position feedback from an incremental shaft encoder. Two channels, A and B, 90 degrees out of phase are required.</td>
</tr>
<tr>
<td>Index</td>
<td>33</td>
<td>Index Pulse — Input from the reference or index pulse of an incremental encoder. Used only in conjunction with the Commutator. Either a low or high true signal can be used with the index pin. See Timing Diagrams and Encoder Interface section for more detail.</td>
</tr>
<tr>
<td>R/W</td>
<td>37</td>
<td>Read/Write — Determines direction of data exchange for the I/O port.</td>
</tr>
<tr>
<td>ALE</td>
<td>38</td>
<td>Address Latch Enable — Enables lower 6 bits of external data bus into internal address latch.</td>
</tr>
<tr>
<td>CS</td>
<td>39</td>
<td>Chip Select — Performs I/O operation dependent on status of R/W line. For a Write, the external bus data is written into the internal addressed location. For Read, data is read from an internal location into an internal output latch.</td>
</tr>
<tr>
<td>OE</td>
<td>40</td>
<td>Output Enable — Enables the data in the internal output latch onto the external data bus to complete a Read operation.</td>
</tr>
<tr>
<td>Limit</td>
<td>14</td>
<td>Limit Switch — An internal flag which when externally set, triggers an unconditional branch to the Initialization/Idle mode before the next control sample is executed. Motor Command is set to zero. Status of the Limit flag is monitored in the Status register.</td>
</tr>
<tr>
<td>Stop</td>
<td>15</td>
<td>Stop Flag — An internal flag that is externally set. When flag is set during Integral Velocity Control mode, the Motor Command is decelerated to a stop.</td>
</tr>
<tr>
<td>Reset</td>
<td>38</td>
<td>Reset — A hard reset of internal circuitry and a branch to Reset mode.</td>
</tr>
<tr>
<td>ExtClk</td>
<td>34</td>
<td>External Clock</td>
</tr>
<tr>
<td>Vcc</td>
<td>11, 35</td>
<td>Voltage Supply — Both VCC pins must be connected to a 5.0 volt supply.</td>
</tr>
<tr>
<td>Vss</td>
<td>10, 32</td>
<td>Circuit Ground</td>
</tr>
<tr>
<td>NC</td>
<td>1</td>
<td>Not Connected — This pin should be left floating.</td>
</tr>
</tbody>
</table>

#### OUTPUT SIGNALS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Pin Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC0-MC7</td>
<td>18-25</td>
<td>Motor Command Port — 8-bit output port which contains the digital motor command adjusted for easy bipolar DAC interfacing. MC7 is the most significant bit (MSB).</td>
</tr>
<tr>
<td>Pulse</td>
<td>16</td>
<td>Pulse — Pulse width modulated signal whose duty cycle is proportional to the Motor Command magnitude. The frequency of the signal is External Clock/100 and pulse width is resolved into 100 external clocks.</td>
</tr>
<tr>
<td>Sign</td>
<td>17</td>
<td>Sign — Gives the sign/direction of the pulse signal.</td>
</tr>
<tr>
<td>PHA-PHD</td>
<td>26-29</td>
<td>Phase A, B, C, D — Phase Enable outputs of the Commutator.</td>
</tr>
<tr>
<td>Prof</td>
<td>12</td>
<td>Profile Flag — Status flag which indicates that the controller is executing a profiled position move in the Trapezoidal Profile Control mode.</td>
</tr>
<tr>
<td>Init</td>
<td>13</td>
<td>Initialization/Idle Flag — Status flag which indicates that the controller is in the Initialization/Idle mode.</td>
</tr>
<tr>
<td>Register (Hex)</td>
<td>Function</td>
<td>Mode Used</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>R00H</td>
<td>R00D Flag Register</td>
<td>All</td>
</tr>
<tr>
<td>R06H</td>
<td>R06D Program Counter</td>
<td>All</td>
</tr>
<tr>
<td>R07H</td>
<td>R07D Status Register</td>
<td>All</td>
</tr>
<tr>
<td>R08H</td>
<td>R08D 8 bit Motor Command Port</td>
<td>All except</td>
</tr>
<tr>
<td>R09H</td>
<td>R09D PWM Motor Command Port</td>
<td>All except</td>
</tr>
<tr>
<td>R0CH</td>
<td>R12D Command Position (MSB)</td>
<td>All except</td>
</tr>
<tr>
<td>R0DH</td>
<td>R13D Command Position</td>
<td>All except</td>
</tr>
<tr>
<td>R0EH</td>
<td>R14D Command Position (LSB)</td>
<td>All except</td>
</tr>
<tr>
<td>R0FH</td>
<td>R15D Sample Timer</td>
<td>All</td>
</tr>
<tr>
<td>R12H</td>
<td>R18D Actual Position (MSB)</td>
<td>All</td>
</tr>
<tr>
<td>R13H</td>
<td>R19D Actual Position</td>
<td>All</td>
</tr>
<tr>
<td>R14H</td>
<td>R20D Actual Position (LSB)</td>
<td>All</td>
</tr>
<tr>
<td>R18H</td>
<td>R24D Commutator Ring</td>
<td>All</td>
</tr>
<tr>
<td>R19H</td>
<td>R25D Commutator Velocity Timer</td>
<td>All</td>
</tr>
<tr>
<td>R1AH</td>
<td>R26D X</td>
<td>All</td>
</tr>
<tr>
<td>R1BH</td>
<td>R27D Y Phase Overlap</td>
<td>All</td>
</tr>
<tr>
<td>R1CH</td>
<td>R28D Offset</td>
<td>All</td>
</tr>
<tr>
<td>R1FH</td>
<td>R31D Maximum Phase Advance</td>
<td>All except</td>
</tr>
<tr>
<td>R20H</td>
<td>R32D Filter Pole, A</td>
<td>All except</td>
</tr>
<tr>
<td>R21H</td>
<td>R33D Filter Pole, B</td>
<td>All except</td>
</tr>
<tr>
<td>R22H</td>
<td>R34D Gain, K</td>
<td>All</td>
</tr>
<tr>
<td>R23H</td>
<td>R35D Command Velocity (LSB)</td>
<td>All</td>
</tr>
<tr>
<td>R24H</td>
<td>R36D Command Velocity (MSB)</td>
<td>Proportional Velocity</td>
</tr>
<tr>
<td>R26H</td>
<td>R38D Acceleration (LSB)</td>
<td>Integral Velocity and Trapezoidal Profile</td>
</tr>
<tr>
<td>R27H</td>
<td>R39D Acceleration (MSB)</td>
<td>Integral Velocity and Trapezoidal Profile</td>
</tr>
<tr>
<td>R28H</td>
<td>R40D Maximum Velocity</td>
<td>Trapezoidal Profile</td>
</tr>
<tr>
<td>R29H</td>
<td>R41D Final Position (LSB)</td>
<td>Trapezoidal Profile</td>
</tr>
<tr>
<td>R2AH</td>
<td>R42D Final Position</td>
<td>Trapezoidal Profile</td>
</tr>
<tr>
<td>R2BH</td>
<td>R43D Final Position (MSB)</td>
<td>Trapezoidal Profile</td>
</tr>
<tr>
<td>R34H</td>
<td>R52D Actual Velocity (LSB)</td>
<td>Proportional Velocity</td>
</tr>
<tr>
<td>R35H</td>
<td>R53D Actual Velocity (MSB)</td>
<td>Proportional Velocity</td>
</tr>
<tr>
<td>R3CH</td>
<td>R60D Command Velocity</td>
<td>Integral Velocity</td>
</tr>
</tbody>
</table>

Notes:
1. Consult appropriate section for data format and use
2. Upper 4 bits are read only
3. Writing to R0EH (LSB) latches all 24 bits
4. Reading R14H (LSB) latches data into R12H and R13H
5. Writing to R13H clears Actual Position Counter to zero
6. The scalar data is limited to positive numbers (00H to 7FH)
7. The commutator registers (R18H, R19H, R1FH) have further limits which are discussed in the Commutator section of this data sheet
Appendix B: Illustrations

Figure B.1: First Prototype of the Omnidirectional Base

Figure B.2: 8 dof Redundant Manipulator
Figure B.5: Rice Head Tracking System

Figure B.6: Completed Telepresence Testbed
Appendix C: Software Source Code

The software for the control and simulation of the telepresence testbed and related programmable equipment is included in this appendix. All software is written in C. The header files for compiling this source code have also been included. A machine readable copy of this software has been archived on the telepresence system in the mechanical engineering robotics laboratory at Rice University. Each file is preceded by its file name in bold print.
All software included in this Appendix has been written by the author of this document as part of his Ph.D. research. The author grants permission to all for its free copying, use, modification, and distribution, and hopes due credit is given to him for such use.
C Language Header Files

motor.h

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

PROJECT:
A TESTBED FOR TELEPRESENCE RESEARCH

AUTHOR:
Sarmad Adnan
MECHANICAL ENGINEERING ROBOTICS LABORATORY
DEPARTMENT OF MECHANICAL ENGINEERING & MATERIAL SCIENCES
RICE UNIVERSITY
HOUSTON, TX 77005.

ADVISORS:
Dr. John B. Cheatham Jr.
(Professor of Mechanical Engineering)
Dr. Ian Walker
(Professor of Electrical Engineering)
RICE UNIVERSITY
HOUSTON, TX 77005.

PROJECT SPONSOR:
NASA/JSC

 *******************************************
 *******************************************
 *******************************************
 *******************************************
 *******************************************
 *******************************************

/**
 */

/* Current Program Version # */
#define VERSION BASE 3.4
#define VERSION_ARM 1.2

/* MSC 5.1 Include Files */
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <math.h>
#include <float.h>
#include <time.h>
#include <signal.h>
#include <conio.h>
#include <memory.h>

/* Greenleaf Comm-Library Include Files */
#define LINT_ARGS
#include <as1ports.h>
#include <qf.h>
#include <ibmkeys.h>

/* Debug Output Definitions */
/*DEBUG1: screen updates in the parametric_omnibase routine*/
#define DEBUG1

/*DEBUG2: trajectory and encoder files made during the parametric_omnibase routine*/
#define DEBUG2
/*DEBUG3:  continuous controller signal outputs in the local/remote_3d_rpy_control
routine*/
#define DEBUG3

/*DEBUG4:  NASA CHAIR output debugging * /
#define DEBUG4

/*DEBUG5:  velocity (world coords) output from the resolved acceleration routine*/
#define DEBUG5

/*DEBUG6:  output of the TX (txstk.c) of the handcontroller positions*/
#define DEBUG6

/*DEBUG7:  output before fwd and inverse kinematic routines*/
#define DEBUG7

/*DEBUG8:  output for position desired and actual positions*/
#define DEBUG8

/*DEBUG9:  At the end of a control loop the resolved acceleration routine prints out
the servorate in the end*/
#define DEBUG9

/*DEBUG10:  output for the trigger used in World coord movement of the robot*/
#define DEBUG10

/*DEBUG11:  only to signal entry and exit from fwd_kin_vel_omnibase()*/
#define DEBUG11

/*DEBUG12:  only to signal entry and exit from read_6d_stick()*/
#define DEBUG12

/
Some Common definitions
*/
#endif
#define TRUE 1
#endif
#define FALSE 0
#endif

#define max(a, b) ( (a) > (b) ) ? (a) : (b)
#endif
#define endif
#endif

#define SIGN(a) (( (a) > 0)? (1): (-1))
#endif
#define ABS(a) (( (a) > 0)? (a): (-a))
#endif
#define CR 13
#endif

/*
Parallel Port Definitions for
our setup of the IAB-40-PC Card
It uses the Intel-8255 P-I/O chip
*/
PA, PB, PC are the programmable registers.
CO is the control register. These are mapped
into the I/O space of the PC.

#define PA 736
#define PB 740
#define PC 744
#define CO 748
#define CH0 736
#define CH1 737
#define CH2 738
#define CH3 739

/*
Definitions of different Board Addresses
*/
#define DIG CTL 739
#define DIGITAL 4
#define ANALOG 737

/*
Clock Rate for the HCTL-1000 2MHz
*/
#define CLOCK 2

/*
Encoder Resolution
*/
#define ENCODE 1440

/*
Gear Ratio depends of the
Motor Currently in use for the Base
*/
#define GEAR_RATIO 525.0

/*
The Ratio of the Encoder Count
to the Output Shaft Speed c/s
*/
#define ENCODER_SCALE (GEAR_RATIO*ENCODE)

/*
The Radius of the Circle defined by the
three wheels of the Omnidirectional Base in m.m.
*/
#define ROBOT_RADIUS 237.5

/*
Radius of the Omnidirectional Wheels in m.m.
*/
#define WHEEL_RADIUS 178.05

/*
To convert from Encoder counts to
millimeters of travel multiply encoder
count by this
*/
#define ENCODER_2_MM (2.0 * PI * WHEEL_RADIUS / ENCODER_SCALE)

/*
The Inverse of the last definition
*/
#define MM_2_ENCODER (ENCODER_SCALE / WHEEL_RADIUS / 2.0 / PI)

/*
Number of wheels on the omnidirectional robot
*/
#define WHEELS 3

/*
Value of PI
*/
#define PI (double)3.141592654
#define PI_by_2 (PI/2.0)

/*
To convert radians to degrees
multiply by
*/
#define RADIUS 57.29583

/*
Gain, Pole, Zero, Acceleration Settings
setting in the Integral Velocity Control
mode. We almost exclusively use this mode
of control.
*/
#define GAIN 6
#define POLE 75
#define ZERO 200
#define MAX_SAFE_WHEEL ACCELERATION 450

/*
This definition used as a parameter
during the initialization of the
HCTL-1000 determines the speed
range within which the Motor Controller
operates. Its range is 0-255
*/
#define SAMPLE_TIME 255

/*
SAMPLE TIME in secs is
{ (SAMPLE_TIME + 1) * 16 / 2E6 }
it is hardcoded and only pertains
to the HCTL-1000 chip
*/
#define SAMPLE_SEC 0.002048

/*
Kp and Kv are the position and
velocity error gains for the omnibase
*/
#define Kp 0.05
#define Kv 0.30

/*
The different types of hand-controIlers
that can be used
*/
#define KRAFT_STICK 0x0001
#define NASA CHAIR 0x0002
#define CURSOR STICK 0x0003
#define MICRO STICK 0x0004

/*
The com port used with each
hand controller. Basically
only the NASA Chair is connected
through the com port. We have
decided to always connect the
NASA chair to comm port 2.
*/
#define NASA_CHAIR_PORT COM2
#define MAX_MISSED_RESPONSE 4

/*
This definition determines
the type of handcontroller
currently in use
*/
#define CURRENT_STICK NASA_CHAIR

/*
Push-button and trigger
bitpatterns for the NASA chair
*/
#define TR1 0xbf
#define TR2 0x3f
#define TOGGLE 0xdf
#define TOGGLEUP 0xf3
#define PBTN1 0xfb
#define PBTN2 0xdf
#define PBTN3 0xef
#define PBTN4 0xf7

/*
control messages are passed in
the structure COMM_PACKET in the
mesg field
*/
#define PANIC_ALL 0xff /**<Catastrophic failure: shutdown power*/
#define STOP_ALL 0xfe
#define STOP_BASE 0xfd
#define STOP_ARM 0xfc
#define TRIGGER1 0xfb /**<trigger position 1 on the nasa stick*/
#define TRIGGER2 0xfa /**<trigger position 2 on the nasa stick*/
#define TOGGLEUP 0xf9 /**<toggle switch up on the nasa stick*/
#define TOGGLEDOWN 0xf8 /**<toggle switch down on the nasa stick*/
#define PUSHBTN1 0xf7 /**<big red button near joystick*/
#define PUSHBTN2 0xf6 /**<button1 on joystick*/
#define PUSHBTN3 0xf5 /**<button2 on joystick*/
#define PUSHBTN4 0xf4 /**<button3 on joystick*/
#define ALL_OK 0x01 /**<no errors continue*/
#define END_ALL 0x02
#define HELLO 0x20
#define HI 0x21

/*
Zero positions and gains
for the hand-controllers
Remember that at zero position
the output of a controller
is not necessarily zero
*/
#define NASA_CHAIR_GAIN 0.3
#define NASA_CHAIR_ZERO_X 127
#define NASA_CHAIR_ZERO_Y 127
#define NASA_CHAIR_ZERO_Z 127
#define NASA_CHAIR_ZERO_WX 128
#define NASA_CHAIR_ZERO_WY 127
#define NASA_CHAIR_ZERO_WZ 128
#define NASA_CHAIR_ZERO_SLIDE 127
#define NASA_CHAIR_ZERO_DIAL 127
#define NASA_CHAIR_ZERO_BUTTONS 255
/*
definitions for the communications with base. The communications for the base always happen over the comm port 1. It has been decided that both the ampro little board computer and the command computer will use comm port 1 for the purpose of controlling the wheels */
#define PORT_BASE COM1

/*
packets are queued up in order to increase the servo update rate, but this causes a lag between the handcontroller and the robot */
#define PACKET_Q_SIZE 0x02

/*
the wait time for the comm port for the initialization of the communication in milliseconds */
#define MAX_HELLO_WAIT 10000

/*
the wait time for the comm port for receiving an acknowledge while communication are happening in milliseconds */
#define MAX_ACK_WAIT 1600
#define MAX_CMAP_WAIT 1200 /*retry comm port 600 m.secs max*/

/* The size of the comm buffer */
#define BUF_LEN (PACKET_SIZE * 100)
/* Other communication parameters that we have decided to have */
#define WORD_LEN 8
#define STOP_BIT 1
#define BAUD 9600

/*
we must define PACKET_SIZE manually every
time we change the packet struct because
we do not want to use the packed structures
*/
#define PACKET_SIZE 13

/*
The structure of the comm packet
check is the checksum info
msg is the communication message
x y z are the translations
wx wy wz are the rotations
*/
#define COMM_PACKET struct comm_packet
COMM_PACKET
{
  unsigned int check;
  unsigned char msg;
  unsigned char x;
  unsigned char y;
  unsigned char z;
  unsigned char pr;
  unsigned char wx;
  unsigned char wy;
  unsigned char wz;
  unsigned char rr;
  unsigned char grip;
  unsigned char packet_end;
};

/*
function declarations for lint
*/

/*
panic routines to prevent disorderly shutdowns
filename: panic.c
in case of panic conditions these routines
turn off all motors and comm interrupts
*/
void install_panic(void); /* starts up the panic checks */
void panic_int(int); /* °C condition */
void panic_fpe(int, int); /* floating point error */
void panic_abrt(int); /* abnormal termination */
void dummyRoutine(float); /* dummy to prevent fpe install error */

/* serial packet communication routines
filename: comm.c
all these routines make calls to the greenleaf comm library */
int sendCommPacket(int, int, int, int, int, int, int, int, int, int);
/* send a packet and get result of previous one*/
void readCommPacket(int*, int*, int*, int*, int*, int*, int*, int*, int*, int*, int*, int*, int*);
/* receive packet*/
void requestPacket(int, int); /* send a request for a packet, first time only*/
void sendAck(int, int); /* send an ack of receipt */
int waitAck(int); /* wait for result of previous packet */
int initComm(int); /* setup the comm port, 4800 bauds etc */
void helloComm(int);
/* 
handcontroller routines
filename: stick.c
routines read numbers from joysticks
and handcontrols
*/
void read_6d_stick(int*, int*, int*, int*, int*, int*, int*, int*);
void read_3d_rpy_stick(int*, int*, int*, int*);  /* read the joystick using the game
controller card */
void init_stick(void); /* initialize the current input device */
int  resolve_message(int);  /* determine the message to send with the packet */

/*
omni_base routines
filename: omni_base.c
these routines pertain to the omni-base control only
*/

/*
motor controller routines
filename: motor.c
these routines do the low level stuff on
the motor controllers and must be
made more generalized.
*/
int  query_profile(int);
void idle_motor(int);
void do_profile(int, long);
void set_trap_vel(int, int);
void set_accl(int, int);
void pwrinit(int);
void trapinit(int);
void hardreset(int, int, int, int);
char hpr_read(int, int);
void hpr_write(int, int, int);
long get_crd_pos(int);
int get_crd_vel(int);
int get_cmd_vel(int);
void amps_on(void);
void amps_off(void);
void bus_init(void);
void read_analog(int*, int*, int*);
void digital_setup(void);
void digital_out(int);
int digital_in(void);
long get_curr_pos(int);
void clear_pos(void);

/*
parametric movement routines for the omni-base
filename: pmv.c
these routines do the mid level
parametric movement they are called
by the routines doing resolved rate
and resolved acceleration control
*/
void pnm_path_base(float, float, float*, float*, float*, float*, float*,
float*, float*, float*, float*, float*, float*, float*, float*,
float*, float*, float*, float*, float*, float*, float*, int*);
void local_3d_rpy_control(float, float, float*, float*, float*, float*,
float*, float*, float*, float*, float*, float*, float*, float*,
float*, float*, float*, float*, float*, float*, float*, float*,
float*, float*, float*, float*, float*, float*, int*);
void remote_3d_rpy_control(float, float, float*, float*, float*, float*,
float*, float*, float*, float*, float*, float*, float*, float*,
float*, float*, float*, float*, float*, float*, float*, int*);

/*
support routines that cannot be otherwise catgorised
filename: support.c
*/
void sleep(int); /* sleep for the given number of clock ticks */
void msleep(int); /* sleep for the given number of clock ticks */

/*
version display routines
filename: version.c
These routine display the
relevant version number of
the software
*/
void base_version(void);
void arm_version(void);
void ethernet_version(void);

/*
necessary routines that must be user supplied
filename: xxx.c
*/
void main(void); /* we have no command-line arguments*/
void shutdown(void); /* checks for the shutdown conditions coded with
main*/
float pnm_x(float); /* user must supply these routines */
float pnm_y(float); /* they define the parametric curves */
float pnm_z(float); /* along which the robot must move */
float pnm_wx(float);
float pnm_wy(float);
float pnm_wz[float];
arm.h

/*
Program: Redundant Arm Controller Header File

Author: Samad Adnan
Dept of MEMS
Rice University
Houston, Texas
*/

#include "..\..\base\work\motor.h"

#define ARM_Kp 0.05 /*position error gain for resolved acceleration*/
#define ARM_Kv 0.01 /*velocity error gain for the resolved acceleration*/

#define JOINT_1_MAX (2*PI)
#define JOINT_2_MAX (PI/2)
#define JOINT_3_MAX (PI/2)
#define JOINT_4_MAX (PI/2)
#define JOINT_5_MAX (PI/3)
#define JOINT_6_MAX (PI/3)
#define JOINT_7_MAX (PI/3)
#define JOINT_8_MAX (PI/3)
#define JOINT_1_MIN (-2*PI)
#define JOINT_2_MIN (-PI/2)
#define JOINT_3_MIN (-PI/2)
#define JOINT_4_MIN (-PI/2)
#define JOINT_5_MIN (-PI/3)
#define JOINT_6_MIN (-PI/3)
#define JOINT_7_MIN (-PI/3)
#define JOINT_8_MIN (-PI/3)

#define ARM_JOINTS struct arm_joints

ARM_JOINTS
{
    double t1;
    double t2;
    double t3;
    double t4;
    double t5;
    double t6;
    double t7;
    double t8;
};

/* Definition "ARM_JOINTS_SIZE" must be fixed each time ARM_JOINTS definition changes*/
#define ARM_JOINTS_SIZE 64

/* remember that we are aiming for an eight dof manipulator*/
#define ARM_WORLD struct arm_world

ARM_WORLD
{
    double x;
    double y;
    double z;
    double px;
    double roll;
    double pitch;
    double yaw;
    double rr;
    /*This is theta3 the redundant positional link*/
    /*This is theta5 the redundant orientation link*/
/* This definition "ARM_WORLD_SIZE" must be manually changed every time we make a change in the structure ARM_WORLD*/
#define ARM_WORLD_SIZE 64
#define ARM_GEAR_RATIO 200
#define ARM_ENCODER 1440
#define ARM_ENC_RAD (double){ (1.0*ARM_ENCODER*ARM_GEAR_RATIO) / 2.0 / PI}
#define MAX_SAFE_ARM_ACCELERATION 200
#define ARM_VELOCITY_SCALE 0.05
#define ARM_MOTORS 8
#define LEN1 (double)500 /*Length of link #1*/
#define LEN2 (double)575 /*Length of link #2*/
#define LEN3 (double)300 /*Length of link #3*/
#define LEN1_2 (double)(LEN1 * LEN1)
#define LEN2_2 (double)(LEN2 * LEN2)
#define LEN3_2 (double)(LEN3 * LEN3)

/* definitions of debug flags if true debug is compiled in */
#define DEBUG91 FALSE
#define DEBUG92 FALSE
#define DEBUG93 FALSE
#define DEBUG94 TRUE
#define DEBUG95 FALSE
#define DEBUG96 FALSE
#define DEBUG101 FALSE /*printout in local_6d_control*/

/* parametric path generation routines*/
double parametric_x(double);
double parametric_y(double);
double parametric_z(double);
double parametric_pr(double);
double parametric_yaw(double);
double parametric_roll(double);

/* Routine to adjust arm to a desired initial configuration by using the keyboard */
void move_arm_keyboard(void);
void cursor_move_arm(void);
int arm_fwd_kin_solve(ARM_JOINTS*, ARM_WORLD*);
int arm_inv_kin_solve(ARM_JOINTS*, ARM_WORLD*);
void init_vel_mode_arm(int);
void get_world_arm(double, ARM_WORLD*, ARM_WORLD*, ARM_JOINTS*);
void put_world_arm(double, ARM_WORLD*, ARM_JOINTS*);
void set_arm_vel(ARM_JOINTS*);
void parametric_arm(double, ARM_WORLD*, void(*)(double, double, ARM_WORLD*, ARM_WORLD*, ARM_WORLD*, int*)));
void set_int_vel_arm(ARM_JOINTS*);

/* pararm.c */
void pam_path_arm(double, double, ARM_WORLD*, ARM_WORLD*, ARM_WORLD*, int*);
void local_6d_control(double, double, ARM_WORLD*, ARM_WORLD*, ARM_WORLD*, int*);

/* init.c */
int home_arm(void);
void set_test_vel(int, int);
int move_servo(int, int, long);
3d.h

/*
Program: Redundant Arm Simulator Header File

Author: Sarmad Adnan
        Dept of MEMS
        Rice University
        Houston, Texas
*/
#include "../..\arm\work\arm.h"
#include <graph.h>

#define LINE
struct line
LINE
{
    float x1, y1, z1;
    float x2, y2, z2;
};

#define POINT
struct point
POINT
{
    double x, y;
};

/* display.c */
void endgraphics(void);
void begingraphics(void);
void drawrobot(float, float, float, float, ARM_JOINTS*, int);
void parametric_graphic(double, ARM_WORLD*, void(*) (double, double, ARM_WORLD*,
    ARM_WORLD*, ARM_WORLD*, int* ));
void puttext(short, short, short, unsigned char _far*);
void main(void);
jr3.h

/*
Program: JR3 Force moment Transducer Header File

Author: Sarmad Adnan
Dept of MEMS
Rice University
Houston, Texas
*/
#define ZERO 0x0000
#define MASK_WORD 0x0000ee
#define MAX_WAIT_JR3 5000

#define FX 0
#define FY 1
#define FZ 2
#define MX 3
#define MY 4
#define MZ 5

#define SIX_FORCE struct six_force
SIX_FORCE
{
    double force[6];
};

int jr3(SIX_FORCE*);
int read_jr3_valid(int*);
void dump_trailers(void);
void jr3_init(void);
pantilt.h

/*
Program: Include file for Pantilt camera base

Author: Sarmad Adnan
Dept of MEMS
Rice University
Houston, Texas
*/

#define X_MIN 240
#define X_MAX 639
#define X_OFFSET 0
#define Y_MIN 0
#define Y_MAX 199
#define Y_OFFSET 0
#define X_CEN 440
#define Y_CEN 100

/* The next are specific to the Rhino pan tilt base */
#define ENCODER_PER_REV 1440
#define SCALER 1000
#define PAN_MOTOR 0
#define TILT_MOTOR 1
#define GEAR_RATIO 1 66.1
#define GEAR_RATIO 2 3.12
#define DEG_PER_ROT 360

#define ANGLE ((double) 1/ENCODER_PER_REV/GEAR_RATIO_1/GEAR_RATIO_2*DEG_PER_ROT)
#define MAX_PAN_ENC 1000
#define MAX_TILT_ENC 1000
#define VEL_SCALE 397
#define MAX_SAFE_PANTILT_ACCELERATION 600
#define PANTILT_GAIN 64
#define PANTILT_SAMPLE_TIME 255

/* Function Declarations */
void main(void);
void shutdown(void);
void init_pt(void);
void set_vel_pt(int, int);
void get_vel_pt(int*, int*);
void set_vel_servo(int, int);
void curr_pos_pt(int*, int*);
void curr_ang_pt(double*, double*);
void print_blurb(void);
void install_panic(void);
mouselib.h

/*
Program: Include File for Mouse Usage

Author: Sarmad Adnan
    Dept of MEMS
    Rice University
    Houston, Texas
*/

int mouse_check(void);
void show_mouse(void);
void hide_mouse(void);
void mouse_get(int*, int*, int*);
void mouse_set(int, int);
void mouse_range_x(int, int);
void mouse_range_y(int, int);
void mouse_sensitivity(int, int);

#define TRUE 1
#define FALSE 0
rtx.h

/*
Program: RTX Robot Controller Header File

Author: Samad Achan
Dept of MEMS
Rice University
Houston, Texas
*/

/* debug conditional compile declarations */
#define DEBUG1    /* prevents robot motion*/
#define DEBUG2    /* debug printouts*/
#define DEBUG3    /* debug printouts*/
#define DEBUG4    /* timestep fixing*/
#define DEBUG5    /* function completion notices*/
#define DEBUG6
#define NO_BLACKBOARD/*
#define BLACKBOARD /* if consortium code is used*/
#define NO_SOCKETS*/
#define SOCKETS

/* include files, some conditional */
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <memory.h>
#include <time.h>
#include <dos.h>
#include <bios.h>

#define SOCKETS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#endif

/* misc constants */
#define TRUE    1
#define FALSE   0
#define MOTOR_READ_ON TRUE
#define MOTOR_READ_OFF FALSE
#define HOME_POSITION 505,0,935,0,0,0,0
#define ZERO     
#define SCALER1  837.168
#define SCALER2  1674.183
#define SCALER3  0.0000214
#define SCALER4  279.056
#define SCALER5  558.05
#define SCALER6  0.07415
#define VEL_SCALE 0.95
#define ZEE_MAX   0
#define ZEE_MIN  -3554
#define MAX_REACH 507
#define GRIP_MIN  -2
#define GRIP_MAX  70
#define ZEE_SCALE 3.7495
#define ROLL_MAX_DEG 181
#define ROLL_MIN_DEG -132
#define PITCH_MAX_DEG 4
#define PITCH_MIN_DEG -98
#define YAW_MAX_ENCODE 2072
#define YAW_MEAN_ENCODE 1071
#define ELBOW_MIN_RAD -3.141
#define ELBOW_MAX_RAD 2.635
#define MAX_ZEE 935
#define MIN_ZEE 280
#define MAX_RAD 500
#define MIN_RAD 100
#define MAX_GRIP (GRIP_MAX-1)
#define MIN_GRIP (GRIP_MIN+1)
#define MAX_ROLL (ROLL_MAX_DEG-1)
#define MIN_ROLL (ROLL_MIN_DEG+1)
#define MAX_PITCH (PITCH_MAX_DEG+1)
#define MIN_PITCH (PITCH_MIN_DEG+1)
#define MAX_YAW 100
#define MIN_YAW -100
#define MAX_ELLOW (ELBOW_MAX_RAD+.1)
#define MIN_ELLOW (ELBOW_MIN_RAD+.1)
#define INIT 0
#define DELTA_TIME 1
#define TIME_ZERO 0.00001
#define AVE_TICS 22
#define DEG_TO_RAD (double)(57.29577951)
#define PI BY 2 (PI/2.0)
#define SPEED 5.0 /*mm per second*/
#define PI (double)(3.141592654)

/* motor names */
#define ELBOW 0
#define SHOULDER 1
#define ZED 2
#define WRIST1 3
#define WRIST2 4
#define YAW 5
#define GRIP 6
#define ZEDOWN 7

/* definitions for computer supervised manual control */
#define GOAL X 396.001
#define GOAL Y 100.001
#define GOAL Z 435.001
#define GOAL Z2 335.001
#define GOAL ROLL 0.001
#define GOAL PITCH -90.001
#define GOAL YAW 0.001
#define MAX_FORCE 2.5

/* Speed Increment from cursor pad */
#define INC 1

/* Positional Error gain would determine how you integrate the positional error */
#define Kp 1.0

/* Force Control Error Gain, currently 2.0 */
#define Kf 4.0

/* error codes returned by rtx_inverse function and others */
#define ENV_ERROR 8
#define 2 RANGE ERROR 1
#define SHOULDER ERROR 2
#define ELBOW ERROR 4
#define ROLL ERROR 8
#define PITCH ERROR 16
#define YAW ERROR 32
#define GRIPPER ERROR 64
#define TSR ABSENT 0xFFFF
#define RTX FAILED_INIT 0xFFFF
#define RTX FAILED_READ 0xFFFC
#define STATUS_ERROR 0xFFFF
/* Scan Codes for numeric keypad keys on an enhanced keyboard used when we drive
the arm from the key pad */
#define ENRY 0x4f
#define CURRN 0x50
#define PAGDN 0x51
#define CURLF 0x4b
#define CURRT 0x4d
#define HOMKY 0x47
#define CURUP 0x48
#define PAGUP 0x49
#define INSKY 0x52
#define DELKY 0x53
#define DIVKY 0x35
#define MILKY 0x37
#define MINKY 0x4a
#define PLISKY 0x4e
#define FUN01 0x3b
#define S_DESIRED 1
#define S_ACTUAL 2

/* To the untrained eye the following may look like a Super Kludge. But it is
necessary for checking the presence of the TSR. The TSR in question, by the way,
handles the interrupt driven serial communication between the PC and the robot. When
the TSR is present the location "SEGMENT:OFFSET" contains its signature */
#define OFFSET 0x5AED
#define SEGMENT 0x0000

#define RTX_WORLD struct rtx_world
RTX_WORLD
    double x, y, z, roll, pitch, yaw, gripper;
    
#define RTX_JOINT struct rtx_joint
RTX_JOINT
    int j1, j2, j3, j4, j5, j6, j7;
    
/* The following lines are dependant on the definition on the control program on the
command computer and hence have been demarcated to ease modification */
#define ARM_CARTESIAN_STATE struct arm_cartesian_state
ARM_CARTESIAN_STATE
    
/* Function Declarations */
int comp_sup_control(RTX_WORLD*);
int velocity_control(RTX_WORLD*);
int position_control(ARM_CARTESIAN_STATE*);
int rtx_read(int, int, int *,int);
int rtx_interrupt(int);
int encoder_feed(RTX_JOINT *, int);
int rtx_init_com(int, int, int);
int rtx_forward(RTX_WORLD*, RTX_JOINT*, int);
int rtx_inverse(RTX_WORLD*, RTX_JOINT*);
int rtx_start_move(void);
int rtx_end(void);
int rtx_init(RTX_WORLD*, RTX_JOINT*);
#define NO_BLACKBOARD
#ifndef NO_BLACKBOARD
void main(void);
int is_event_waiting(void);
#endif
sockets.h

/*
Program: Socket Based Communication Header File

Author: Samad Adnan
    Dept of MEMS
    Rice University
    Houston, Texas
*/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/udp.h>
#include <sys/socket.h>

#define PORT 0x1234
#define ODDENAME "oh please\0"
#define ROBOT_HOST "robotical.rice.edu"

#define PANIC_ALL 0xff
#define STOP_ALL 0xfe
#define STOP_BASE 0xfd
#define STOP_APM 0xfc
#define TRIGGER1 0xfb
#define TRIGGER2 0xfa
#define TRIGGER3 0xf8
#define TRIGGER4 0xf7
#define TRIGGER5 0xf6
#define TRIGGER6 0xf5
#define TRIGGER7 0xf4
#define TRIGGER8 0xf3
#define TRIGGER9 0xf2
#define TRIGGER10 0xf1
#define TRIGGER11 0xf0
#define ALL_OK 0x01
#define END_ALL 0x02
#define HELLO 0x20
#define HI 0x21
#define NASA_CHAIR_GAIN 0.3
#define NASA_CHAIR_GAIN 0.6
#define NASA_CHAIR_ZERO X 0.123
#define NASA_CHAIR ZERO Y 0x129
#define NASA CHAIR ZERO Z 0x129
#define NASA CHAIR ZERO WX 0x128
#define NASA CHAIR ZERO WY 0x138
#define NASA CHAIR ZERO WZ 0x129
#define NASA CHAIR ZERO SLIDE 0x127
#define NASA CHAIR ZERO DIAL 0x127
#define NASA CHAIR ZERO BUTTONS 0x255
#define VEL SCALE 0.5
#define PACKET SIZE 12
#define COMM PACKET COMM_PACKET
    {
        unsigned short int check;
        unsigned char msg;
        unsigned char x;
        unsigned char y;
        unsigned char z;
        unsigned char pr;
        unsigned char wx;
        unsigned char wy;
    }

struct comm_packet
unsigned char wz;
unsigned char rr;
unsigned char grip;

#define ARM_WORLD
ARM_WORLD
{ double x;
  double y;
  double z;
  double pr;
  double roll;
  double pitch;
  double yaw;
  double rr;
  double gripper;
};

#define RTX_WORLD struct rtx_world
RTX_WORLD
{ double x, y, z, roll, pitch, yaw, gripper;
};

#define RTX_JOINT struct rtx_joint
RTX_JOINT
{ int j1, j2, j3, j4, j5, j6, j7;
};

/*
The following lines are dependant on the definition on the control
program on the command computer and hence have been demarcated to
ease modification */
#define ARM_CARTESIAN_STATE struct arm_cartesian_state
ARM_CARTESIAN_STATE
{ short cs_status;
  short cs_x, cs_y, cs_z;            /* in 1/8 mm*/
  short cs_roll, cs_pitch, cs_yaw;   /* in 1/64 deg*/
  short cs_gripper;                 /* in 1/8 mm*/
};

#define S_DESTINED 1
#define S_ACTUAL 2

int sd, sd_current, addrlen;
struct sockaddr_in ssin;
struct sockaddr_in pin;
struct hostent *hp;
tracker.h

/*
Program: Fuzzy Logic Based Camera Tracker Header File

Author: Sarmad Adnan
Dept. of MEMS
Rice University
Houston, Texas
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#include <targgraf.h>
#include <time.h>
#include <graph.h>
#include <math.h>
#include <float.h>
#include <signal.h>
#include <conio.h>
#include <memory.h>

/*
screen size is 512 x 400 at 16 bits. 7c00 is the brightest red,
3ff is brightest other ie. fully saturated blue and green
*/
#define MOUSE_RATIO 0x20
#define TARG_BASE_ADDR 0x220
#define TRUE 1
#define FALSE 0
#define RED 0x7c00L
#define GREEN 0x3e00L
#define BLUE 0x1f0L
#define RED_MAX 0x7c00L
#define GREEN_MAX 0x3e00L
#define BLUE_MAX 0x1f0L
#define RED_MIN 0x1c00L
#define X_RES 512
#define Y_RES 400
#define X_CEN (X_RES/2)
#define Y_CEN (Y_RES/2)
#define X_SML (X_RES/10)
#define Y_SML (Y_RES/10)
#define X_MED (X_RES/3)
#define Y_MED (Y_RES/3)
#define X_LRG (X_RES/2)
#define Y_LRG (Y_RES/2)
#define GEN_SZ1 (Y_RES*0.33)
#define GEN_SZ2 (Y_RES*0.33+10)
#define GEN_SZ3 (X_CEN+Y_RES*0.33)
#define GEN_SZ4 (X_CEN+Y_RES*0.33+10)
#define VMAX 100

/* Parallel Port Definitions for our setup of the homebrew card. It uses the Intel 8255 P10 chip PA, PB, PC are the programmable registers CO is the control register. These are mapped into the I/O space of the PC */
#define IO_BASE_ADDR 0x360
#define PA IO_BASE_ADDR
#define PB IO_BASE_ADDR+1
#define PC IO_BASE_ADDR+2
#define CO IO_BASE_ADDR+3
#define STOP_ALL 18
#define SETUP_PIO 128
#define MOTOR_CHANNEL PB
#define PITCH_SLOW_UP 32
#define PITCH_FAST_UP 0
#define PITCH_SLOW_DOWN 40
#define PITCH_FAST_DOWN 8
#define PITCH_STOP_ALL 16
#define PAN_SLOW_LEFT 4
#define PAN_FAST_LEFT 0
#define PAN_SLOW_RIGHT 5
#define PAN_FAST_RIGHT 1
#define PAN_STOP_ALL 2
#define FAST 2
#define SLOW 1
#define RIGHT 1
#define LEFT -1
#define UP 1
#define DOWN -1

/* Membership function shoulder values (horizontal axis) before it was /3 /3 */
#define CX (X_CEN/2)
#define CY (Y_CEN/2)
#define CP (3*VARMAX/4)
#define CT (3*VARMAX/4)

/* Membership grades */
#define MEMS_MAX 100
#define MEMS_MIN 0
#define MOMENTS 0
#define AREA 1

/*functions defined in tracker.c */
void find_red_blob(int*, int*);
void track_red_blob(int*, int*, int*, int*, int*, unsigned long*);
void draw_cross(int, int);
void draw_simulation(double, double, double, double, double, double, double, int,
unsigned long);
void beengraphics(void);
void endgraphics(void);
void puttext(short, short, short, unsigned char _far*);
void setupFioBoard(void);
void setMotorSpeeds(int, int);
void stopMotors(void);
void simFuzzyTrack(int, int);
void install_panic(void);
void panic_int(int);
void panic_fpe(int, int);
void panic_abrt(int);
void shutdown(void);
void dummy_routine(float);
void sleep(int);

/* functions defined in fuzztrak.c */
void FuzzyTrack(int, int);
void precompute(void);
C Language Source Files

arm.c

#include "arm.h"

void parametric_arm(double end_time, ARM_WORLD* pinit, void(*path)(double, double, ARM_WORLD*, ARM_WORLD*, ARM_WORLD*, int*))
{
    ARM_WORLD pd, pe, pa, pf, vd, ve, va, vr, ad;
    ARM_JOINTS ja;
    double time, ave_time_step;
    clock_t start_time, now_time;
    int total_itr, message;

    total_itr=0;                  /* reset count of iterations*/
    ave_time_step=(double)0.05;   /* make actual position zero */
    clear_pos();                 /* make all hardware counters zero*/
    start_time=clock()-(clock_t)5;         /* obtain time at function entry minus 5
milliseconds*/
    time=(double)0.0;
    message=ALL_OK;               /* initial message is OK */
    memcpy(&pa, pinit, (size_t)ARM_WORLD_SIZE); /*make a copy of the initial world
coords*/

    while(time < end_time)
    {
        /* increment the record of the total iterations */
        total_itr++;

        /* obtain desired trajectory velocity vector */
        (*path)(time, ave_time_step, &pd, &vd, &ad, &message);

        if(message==STOP_ALL)
        {
            arm_inv_kin_vel(vxr, vyr, vwr);
            break;
        }

        /* obtain actual trajectory */
        get_world_arm(ave_time_step, &pa, &va, &ja);

        /* obtain the error signal position & velocity vector */
        pe.x=pd.x-pa.x, pe.y=pd.y-pa.y, pe.z=pd.z-pa.z, pe.pr=pd.pr-pa.pr;
        ve.x=vd.x-va.x, ve.y=vd.y-va.y, ve.z=vd.z-va.z, ve.pr=vd.pr-va.pr;

        /* resolve the acceleration into velocity by multiplying with ave_time_step
and then calculate the resultant velocity vector. currently only being done for x
y z and pr*/
        vr.x=ad.x * ave_time_step + vd.x + ARM_Kv * ve.x + ARM_Kp * pe.x;
        vr.y=ad.y * ave_time_step + vd.y + ARM_Kv * ve.y + ARM_Kp * pe.y;
        vr.z=ad.z * ave_time_step + vd.z + ARM_Kv * ve.z + ARM_Kp * pe.z;
        vr.pr=ad.pr * ave_time_step + vd.pr + ARM_Kv * ve.pr + ARM_Kp * pe.pr;

        /*Using the average timestep calculate the final position of the
manipulator. currently only being done for x y z and pr*/
        pf.x=pa.x+vr.x*ave_time_step;
        pf.y=pa.y+vr.y*ave_time_step;
        pf.z=pa.z+vr.z*ave_time_step;
        pf.pr=pa.pr+vr.pr*ave_time_step;
/* Perform Inverse kinematics */
put_world_arm(ave_time_step, &pf, &ja);

/* time step computations in terms of real-time These calculations sometimes
result in a divide by zero if the PC clock has not ticked before we have
done another iteration hence in such cases we will use the previous time
step The clock on a PC has 0.055 resolution and we are on the borderline
so we may get some time steps that are zero*/

now_time=clock() - start_time;
ave_time_step=(double)now_time/CLOCK_TCK/total_itr;
time+=ave_time_step;
}

/*/ get_world_arm() obtains the actual/current trajectory(position&velocity)
by reading the position registers on the HCTL-1000 and differentiating the
results. In the parm list pact & vact are structures in which we will place
the current position and current velocity, while jact is structure to hold
actual joint coordinates in radians. This function additionally calls on
arm_fwd_kin_solve() to perform the forward kinematics */
void get_world_arm(double ave_t, ARM_WORLD* pact, ARM_WORLD* vact, ARM_JOINTS* jact)
{
    ARM_WORLD ppre;    /* previous step positions and velocities*/
    jact->t1=(double)get_curr_pos(0)/ARM_ENC_RAD, jact->
    t2=(double)get_curr_pos(1)/ARM_ENC_RAD;
    jact->t3=(double)get_curr_pos(2)/ARM_ENC_RAD, jact->
    t4=(double)get_curr_pos(3)/ARM_ENC_RAD;
    jact->t5=(double)get_curr_pos(4)/ARM_ENC_RAD, jact->
    t6=(double)get_curr_pos(5)/ARM_ENC_RAD;
    jact->t7=(double)get_curr_pos(6)/ARM_ENC_RAD, jact->
    t8=(double)get_curr_pos(7)/ARM_ENC_RAD;
    memcpy(&ppre, pact, (size_t)ARM_WORLD_SIZE); /* make a copy of the previous world
coordinates*/
    arm_fwd_kin_solve(jact, pact); /* convert the joint coords to world coords*/
    vact->x=(pact->x - ppre.x)/ave_t;
    vact->y=(pact->y - ppre.y)/ave_t;
    vact->z=(pact->z - ppre.z)/ave_t;
    vact->pr=(pact->pr - ppre.pr)/ave_t;
}

/* put_world_arm() sets the required velocity on the registers of the HCTL-1000
In the parm list pf is a structure pointer in which we place the final world
coord position required at the end of the timestep, ja is current joint angles
in radians. The function additionally calls on arm_inv_kin_solve() to perform
the inverse kinematics to convert the world velocity to the joint velocity */
void put_world_arm(double ave_t, ARM_WORLD* pf, ARM_JOINTS* ja)
{
    ARM_JOINTS jf;
    ARM_JOINTS v;
    arm_inv_kin_solve(&jf, pf);
    v.t1=ARM_ENC_RAD*SAMPLE_SEC*(jf.t1 - ja->t1)/ave_t;
    v.t2=ARM_ENC_RAD*SAMPLE_SEC*(jf.t2 - ja->t2)/ave_t;
    v.t3=ARM_ENC_RAD*SAMPLE_SEC*(jf.t3 - ja->t3)/ave_t;
    v.t4=ARM_ENC_RAD*SAMPLE_SEC*(jf.t4 - ja->t4)/ave_t;
    v.t5=ARM_ENC_RAD*SAMPLE_SEC*(jf.t5 - ja->t5)/ave_t;
    v.t6=ARM_ENC_RAD*SAMPLE_SEC*(jf.t6 - ja->t6)/ave_t;
void set_int_vel_arm(int v)
{
    int velocity[8], servo;
    int cur, hold, set;

    velocity[0]=(int)v.t1, velocity[1]=(int)v.t2;
    velocity[2]=(int)v.t3, velocity[3]=(int)v.t4;
    velocity[4]=(int)v.t5, velocity[5]=(int)v.t6;
    velocity[6]=(int)v.t7, velocity[7]=(int)v.t8;

    for (servo=0; servo<ARM_MOTORS; servo++)
    {
        velocity[servo] *= ARM_VELOCITY_SCALE;
        velocity[servo] = (velocity[servo]>127)? 127 : velocity[servo];
        velocity[servo] = (velocity[servo]<-127)? -127 : velocity[servo];
        cur = hp_read(servo, 60);
        hold = velocity[servo] - cur;

        if (abs(hold)>127)
            set = (unsigned char)(cur+126*SIGN(hold));
        else
            set = (unsigned char)(velocity[servo]);

        hp_write(servo, 60, set);    /*write the actual or modified velocity request to
                                      the velocity register*/
    }
}

void init_vel_mode_arm(int acceleration)
{
    int arm_servo;

    hardreset(GAIN, POLE, ZERO, SAMPLE_TIME);
    for (arm_servo=0; arm_servo<ARM_MOTORS; arm_servo++)
    {
        pvinit(arm_servo);
        set_accl(arm_servo, acceleration);
    }
}
#include "..\..\graph\work\3d.h"

void main(void)
{
    ARM WORLD pinit;
    static ARM_JOINTS
        speed={(double)0, (double)0, (double)0, (double)0, (double)0, (double)0, (double)0, (double)0};
    struct tm *curtime;
    time_t bintime;

    arm_version();
    init_stick();
    install_panic();
    time(&bintime);
    curtime=localtime(&bintime);
    printf("\n\time is %s\n", asctime(curtime));

    /*Initializing Everthing*/
    home_arm();
    init_vel_mode_arm(MAX_SAFE_ARM_ACCELERATION);
    pinit.x=(double)4, pinit.y=(double)1370.0, pinit.z=(double)-4, pinit.pr=(double)0;
    parametric_arm(10, &pinit, parm_path_arm);

    set_int_vel_arm(speed);
    move_arm_keyboard();
    set_Int_vel_arm(speed);
    shutdown();
}

double parametric_x(double t)
{
    return(4+(double)t*3);
}

double parametric_y(double t)
{
    return((double)1370-t*3);
}

double parametric_z(double t)
{
    return((double)-4-t*3);
}

double parametric_pr(double t)
{
    return((double)0.0*t);
}

double parametric_rr(double t)
{
    return((double)0.0*t);
}

double parametric_pitch(double t)
{
    return((double)0.0*t);
}

double parametric_roll(double t)
{ return((double)0.0*t);
}

double parametric_yaw(double t)
{
    return((double)0.0*t);
}

void shutdown()
{
    static ARM_JOINTS speed={ (double)0, (double)0, (double)0, (double)0, (double)0, (double)0,
        (double)0, (double)0};

    printf("shutdown: ....\n");
    set_int vel arm(speed);
    hardreset(GAIN, POLE, ZERO, SAMPLE_TIME);
}
comm.c

#include "motor.h"

/*
This file contains all the routines that pertain to serial port communications
Packets with checksum info and messages are made out of the trajectory desired
and sent over the comm port.
*/

/* send_comm_packet():
this function takes as parameters six trajectory variables a message
and a comm port number and sends a packet over. It receives the reply
for the previous packet */
int send_comm_packet(int x, int y, int z, int wx, int wy, int wz, int msg, int port)
{
    COMM PACKET packet;
    int temp;
    temp=wait_ack(port); /* wait until a request is made or previous packet is
acknowledged by ALL_OK*/
    packet.msg=(unsigned char)msg;
    packet.x=(unsigned char)x;
    packet.y=(unsigned char)y;
    packet.z=(unsigned char)z;
    packet.wx=(unsigned char)wx;
    packet.wy=(unsigned char)wy;
    packet.wz=(unsigned char)wz;
    packet.pr=packet.grp=0;
    packet.packet_end=CR;
    packet.check=(packet.x + packet.y + packet.z + packet.wx +
    packet.wy + packet.wz + packet.msg + packet.pr +
    packet.grp);
    if(asiputb(port,(char *)&packet, PACKET_SIZE)!=PACKET_SIZE)
    {
        fprintf(stderr, "send_comm_packet: Transmit Error, Aborting\n");
        exit(0);
    }
    return(temp);   /* give the received message back to the calling function*/
}

/* read_comm_packet():
This function listens at the desired comm port and obtains a comm packet */
void read_comm_packet(int *x, int *y, int *z, int *wx, int *wy, int *wz, int *msg,
    int *port)
{
    COMM PACKET packet;
    clock_t retry;
    int message;
    message=*msg;
    retry=clock(); /* entry time */
    while(getrxcnt(*port)< PACKET_SIZE)
    {
        if((retry+MAX_COMM_WAIT)< clock())
        {
            send_ack(*port, PANIC_ALL);
            fprintf(stderr,"read_comm_packet: packet timeout\n");
        }
exit(0);
}

if(asigetb(*port, (char *)&packet, PACKET_SIZE)!=PACKET_SIZE)
{
    send_ack(*port, PANIC_ALL);
    fprintf(stderr,"read_comm_packet: packet size incorrect\n");  
    exit(0);
}

if(packet.check != (packet.x + packet.y + packet.z + packet.wx + packet.wy + 
    packet.wz + packet.msg + packet.pr + packet.rr + packet.grip))
{
    send_ack(*port, PANIC_ALL);  
    fprintf(stderr,"read_comm_packet: packet checksum incorrect\n");  
    exit(0);
}

*x=packet.x, *y=packet.y, *z=packet.z;  
*msg=packet.msg;

if(message != PANIC_ALL)
    send_ack(*port, *msg);
else
    send_ack(*port, PANIC_ALL);

/* wait_ack():  
   This function waits for an acknowledgment at the comm port for a small period 
   of time and returns the acknowledgment token to the calling function */
int wait_ack(int port)
{
    clock_t retry;
    retry=clock();
    while(getrxcnt(port)<1)
    {
        if((clock() - retry) >= MAX_ACK_WAIT)
        {
            fprintf(stderr,"\n\nwait_ack: packet timeout\n");
            exit(0);
        }
        return(asigetc(port));
    }

    /* send_ack():  
       This function just sends an acknowledgment to a given port */
    void send_ack(int port, int msg)
    {
        asiputc(port, msg);
    }

    /* request_packet():  
       Used to start of the communication It sends a few requests for packets 
       to the designated comm port */
    void request_packet(int port, int msg)
    {
        int count;
        for(count=0; count<PACKET_Q_SIZE; count++)
        {
            asiputc(port, msg);
        }
    }

    /* init_comm():
Initializes the interrupt communication on the comm port given by port_num */
int init_comm(int port_num)
{
    int status;
    unsigned mode=ASINOUT|BINARY|NORMALRX;

    if((status=asioopen(port_num, mode, BUF_LEN, BUF_LEN, BAUD, P_NONE, STOP_BIT, WORD_LEN, ON, ON)) != ASSUCCESS)
    {
        printf("init_comm_rx: Open COM%d Failed return code:%d\n", port_num, status);
        exit(status);
    }

    asiflow(port_num, 50, 80, ON, ON);
    hello_comm(port_num);
    return(0);
}

/* hello_comm();
   since the receiver and transmitter can not be synchronized at the start (because
   they run on different computers) the hello routine if run on both machines
   synchronizes the communications */
void hello_comm(int port)
{
    clock_t retry;
    int test;

    retry=clock();
    printf(stderr, "\nhello comm: synching packets\n");
    asiputc(port, HELLO);

    while(getrxcnt(port)<1)
    {
        if(clock() > MAX_HELLO_WAIT+retry)
        {
            printf(stderr, "\nhello_comm: Timed Out......\n");
            exit(0);
        }
    }

    test=asigetc(port);
    if(test==HELLO)
    {
        send_ack(port, HI);
        test=wait_ack(port);
        if(test !=HI)
        {
            printf(stderr, "\nhello_comm: Synchronization Failure, Got 0\n",test);
            exit(0);
        }
    }
    else if(test==HI)
    {
        send_ack(port, HI);
    }
    else
    {
        printf(stderr, "\nhello_comm: Synch Failure, Got 0\n",test);
        exit(0);
    }
    endif

    printf(stderr, "..DONE\n");
}
control.c

#include <stdio.h>
#include <math.h>
#include <graph.h>
#include <time.h>
#include "pantilt.h"
#include "mouselib.h"
#include "..\..\base\work\motor.h"

extern float ave_time_step;

void shutdown(void)
{
    setvideomode(DEFAULTMODE);
    printf("Average timestep %f\n", ave_time_step);
    printf("Shutdown...");
    hardreset(PT_GAIN, PT_POLE, PT_ZERO, PT_SAMPLE_TIME);
    printf("...
");
}

void init_pt(void)
{
    hardreset(PT_GAIN, PT_POLE, PT_ZERO, PT_SAMPLE_TIME);
    pinipt(PAN MOTOR);
    set_acc1(PAN MOTOR, MAX_SAFE_PT_ACCELERATION);
    pinipt(TILT MOTOR);
    set_acc1(TILT MOTOR, MAX_SAFE_PT_ACCELERATION);
}

void set_vel_pt(int pan_vel, int tilt_vel)
{
    set_vel_servo(PAN MOTOR, pan_vel);
    set_vel_servo(TILT MOTOR, tilt_vel);
}

void get_vel_pt(int *pan_vel, int *tilt_vel)
{
    *pan_vel=hp_read(PAN MOTOR, 0x35)*256+hp_read(PAN MOTOR, 0x34);
    *tilt_vel=hp_read(TILT MOTOR, 0x35)*256+hp_read(TILT MOTOR, 0x34);
}

void curr_pos_pt(int *pan_pos, int *tilt_pos)
{
    *pan_pos=(int) get_curr_pos(PAN MOTOR);
    *tilt_pos=(int) get_curr_pos(TILT MOTOR);
}

void curr_ang_pt(float *pan_ang, float *tilt_ang)
{
    *pan_ang=(float) get_curr_pos(PAN MOTOR) * ANGLE;
    *tilt_ang=(float) get_curr_pos(TILT MOTOR) * ANGLE;
}
void set_wel_servo(int servo, int sp)
{
    int cur, hold, set;
    sp = (sp > 127) ? 127 : sp;
    sp = (sp < 127) ? -127 : sp;

    cur = hp_read(servo, 0);
    hold = sp - cur;
    if (abs(hold) > 127)
        set = (unsigned char) (cur + 126 * SIGN(hold));
    else
        set = (unsigned char) (sp);
    endif

    if (servo == 0)
        {
            outp(PC, 1);
            outp(PA, 60); /* setup adress*/
            outp(PB, 8); /* r/w low reset high*/
            outp((PA + 1), set); /* output the data */
        }
    else if (servo == 1)
        {
            outp(PC, 1);
            outp((PA + 2), 60); /* setup adress*/
            outp(PB, 8); /* r/w low reset high*/
            outp((PA + 3), set); /* output the data */
        }
    }

void print_blurb(void)
{
    char str[80];

    _rectangle(GORDER, X_MIN + X_OFFSET, Y_MIN, X_MAX + X_OFFSET, Y_MAX);
    _rectangle(GORDER, 0, 0, X_MIN + X_OFFSET, Y_MAX);
    _rectangle(GORDER, 4, 2, X_MIN + X_OFFSET - 4, Y_MAX - 2);
    /* Draw Origin */
    _moveto(X_CEN + X_OFFSET - 20, Y_CEN), _lineto(X_CEN + X_OFFSET + 20, Y_CEN);
    _moveto(X_CEN + X_OFFSET, Y_CEN - 10), _lineto(X_CEN + X_OFFSET, Y_CEN + 10);

    _settextposition(2, 2);
    _spritef(str, " Pan-Tilt Stereo Camera ");
    _outtext(str);
    _settextposition(3, 2);
    _spritef(str, " Platform Controller ");
    _outtext(str);
    _settextposition(4, 2);
    _spritef(str, " By Samad Adnan ");
    _outtext(str);
    _settextposition(7, 2);
    _spritef(str, " Platform Position (deg) ");
    _outtext(str);
    _settextposition(10, 2);
    _spritef(str, " Platform Velocity (deg/sec) ");
    _outtext(str);
    _settextposition(13, 2);
    _spritef(str, " Trackball Position (pixels) ");
    _outtext(str);
}
#include "3d.h"

double base_coord[15][2] = {
  -228.6, 393.7, -457.2, 0, -228.6, -469.9, 228.6, -469.9, 457.2, 0, 228.6, 393.7,
  -228.6, 393.7, 228.6, -393.7, 76.2, 76.2, -76.2, 76.2, -228.6, 393.7,
  -76.2, 393.7, -76.2, 673.1, 76.2, 673.1, 76.2, 393.7
};

void begingraphics(void)
{
  struct videoconfig vc;
  int vertex;

  _getvideoconfig(&vc);
  if((vc.monitor==ANALOG)&&(vc.adapter==VGA))
    _setvideomode(_VRES16COLOR);
  else
    {
      perror("Only VGA adapter w/ VGA monitor supported");
      exit(0);
    }

  /* draw the borders, windows and axis*/
  _remapalette(8, BLACK);
  _setbkcolor(_GRAY);
  _setcolor(7);
  _rectangle(_GBORDER, 0, 0, 639, 479);    /* screen border */
  _rectangle(_GBORDER, 0, 77, 401, 479);   /* big window */
  _rectangle(_GBORDER, 401, 239, 639, 479);
  _rectangle(_GBORDER, 401, 0, 639, 239);
  _rectangle(_GTILEINTERIOR, 2, 2, 399, 75);
  _setcolor(0);
  _rectangle(_GBORDER, 1, 1, 400, 76);

  /* Write the title etc*/
  if(_registerfonts("tmsrb.fon")<=0)
    {
      printf("display: font registration failed\n");
      exit(0);
    }
  if(_setFont("t. tms mm'h26w16")<=0)
    {
      printf("display: font size incorrect\n");
      exit(0);
    }
  puttext(20, 20, 9, (unsigned char _far*)"Telepresence Robot System");
  if(_setFont("t. tms mm'h12w6")<=0)
    {
      printf("font registration error\n");
      exit(0);
    }
  puttext(155, 50, 4, (unsigned char _far*)"Rice University, Department of M.E.M.S.");
  puttext(25, 50, 4, (unsigned char _far*)"Sarmad Agha");

  /* first window top right*/
  _setviewport(402, 1, 638, 238);
  _setwindow(1, -500, -1000, 1500, 1000);
  _setcolor(1);
void endgraphics(void)
{
    _setvideomode(_DEFAULTMODE);
}

void drawrobot(float time, float x, float y, float theta, ARM_JOINTS *j, int wipe)
{
    static LINE o1, o2, o3;
    LINE l1, l2, l3;
    static double old_base[15][2];
    double S1, S2, S3, S4, C1, C2, C3, C4, SB, CB;
    int vertex;
    char time_text[80];

    CB=cos(theta), SB=sin(theta);
    /* arm calculations*/
    C1=cos(j->t1), C2=cos(j->t2), C3=cos(j->t3), C4=cos(j->t4);
    S1=sin(j->t1), S2=sin(j->t2), S3=sin(j->t3), S4=sin(j->t4);
    11.x1=0, 11.y1=LEN1-100, 11.z1=0;
    11.x2=0, 11.y2=LEN1, 11.z2=0;
    12.x1=0, 12.y1=LEN1, 12.z1=0;
    12.x2=s1*s2*LEN2, 12.y2=LEN1+C2*LEN2, 12.z2=-c1*s2*LEN2;
    13.x1=12.x2, 13.y1=12.y2, 13.z1=12.z2;
    13.x2=12.x2+s1*c2*C3*LEN3-C1*s3*s4*LEN3+c1*s2*c4*LEN3;
    13.y2=12.y2+c2*C3*LEN3-C2*C3*s4*LEN3;
    13.z2=12.z2-c1*s2*c4*LEN3-s1*s3*s4*LEN3-c1*c2*C3*LEN3;

    /* yz window, bottom right*/
    _setviewport(402, 239, 638, 478);
    _setwindow(1, -500, -1000, 1500, 1000);
    if(wipe)
        _moveto_w(o1.y1, o1.z1); _lineto_w(o1.y2, o1.z2); _lineto_w(o2.y2, o2.z2);
        _lineto_w(o3.y2, o3.z2); _setcolor(14);
    _moveto_w(l1.y1, l1.z1); _lineto_w(l1.y2, l1.z2); _lineto_w(l2.y2, l2.z2);
        _lineto_w(l3.y2, l3.z2);

    /* xy window, top right*/
    _setviewport(402, 1, 638, 238);
    _setwindow(1, -500, -1000, 1500, 1000);
    if(wipe)
        _setcolor(0);
    _moveto_w(o1.y1, o1.x1); _lineto_w(o1.y2, o1.x2); _lineto_w(o2.y2, o2.x2);
        _lineto_w(o3.y2, o3.x2); _setcolor(14);
    _moveto_w(l1.y1, l1.x1); _lineto_w(l1.y2, l1.x2); _lineto_w(l2.y2, l2.x2);
        _lineto_w(l3.y2, l3.x2);

    /* xy window, large, left, shows top view of base and no motion on arm */
    _setviewport(1, 78, 400, 478);
    _setwindow(1, -5000, -5000, 5000, 5000);
    /* base draw*/
    if(wipe)
        _setcolor(0);
    _moveto_w(old_base[0][0], old_base[0][1]);
    for(vertex=1; vertex<15; vertex++)
        _lineto_w(old_base[vertex][0], old_base[vertex][1]);
    for(vertex=0; vertex<15; vertex++)
        { /* base draw*/

    }
old_base[vertex[0]] = x + CB * base_coord[vertex[0]] + SB * base_coord[vertex[1]];
old_base[vertex[1]] = y + CB * base_coord[vertex[1]] - SB * base_coord[vertex[0]];

_setcolor(14);
_moveto_w(old_base[0][0], old_base[0][1]);
for (vertex = 1; vertex < 15; vertex++)
    _lineto_w(old_base[vertex][0], old_base[vertex][1]);
_moveto(5, 5);
_setcolor(0);
_rectangle(GFILLINTERIOR, 5, 5, 400, 15);
_setcolor(7);
sprintf(time_text, "%.1f: %.1f %.1f %.1f: %.3f %.3f %.3f", time, l3.x2, l3.y2, l3.z2, j->t1, j->t2, j->t3, j->t4);
_outgtext((unsigned char _far *)time_text);
_setcolor(14);

memcpy(&s01, &s1, sizeof(LINE));
memcpy(&s02, &s2, sizeof(LINE));
memcpy(&s03, &s3, sizeof(LINE));

}

void puttext(short x, short y, short color, unsigned char _far *txt)
{
    _moveto(x, y);
    _setcolor(color);
    _setgtextvector(1, 0);
    _outgtext(txt);
}
#include "motor.h"

/* calls the assembly language routine instk() on the game controller card. 
don't use with the greenleaf comm routines. This function is useful for 
use on a computer with a pc type rom which does not have joystick support */

void read_3d_rpy_stick(wx, wy, wz, panic)
    int *x, *y, *w, *panic;
    {
        int pos[4];
        instk(&pos[0]);

        *wx=pos[0];
        *wy=pos[1];
        *wz=pos[2];
        *panic=FALSE;
    }
jr3.c

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "jr3.h"

SIX_FORCE scale=
{ 
  1.52587891e-03, 1.52587891e-03, 3.05175781e-03,
  4.57763671e-03, 4.57763671e-03, 4.57763671e-03
};

long retries=0;

int jr3(SIX_FORCE* applied)
{
  int axis=0, previous, current=0xff, raw;

  do
  {
    previous=current; /*synchronize output of sensor necessary normally
    only the first time but is used each time*/
    if(axis > 20)  
      { 
        fprintf(stderr,"\njr3: Transducer did not Sync, Check Power &
          Cables\n"");
        return(1);
      }

    if(read_jr3_valid(&current))
      Jr3_Init();

    if(((previous===(int)ZERO) & ((current===MASK_WORD))))
      break;  /*have synchronized with the packet header*/

  } while(+axis);  /*cycle through two complete packets i.e. 20 words*/

  for(axis=FX; axis<=MZ; axis++)
  {
    if(read_jr3_valid(&raw))
      { 
        fprintf(stderr, "\njr3: Force readings timed out\n");
        return(1);
      }

    applied->force[axis]=raw * scale.force[axis];
  }

cleanup_trailers();
  return(0);
}

int read_jr3_valid(int* raw)
{
  int count=0;
  unsigned a;

  while(!((inp(0x30a) & 0x01)))  /*check for the data valid line*/
  {
    if(count++==MAX_WAIT_JR3)
    { 
      outp(0x30a, 0x00);  /*request another piece of data*/
      outp(0x30a, 0x10);  /*by toggling the data-req line*/
    }
  }

  return(0);
}
fprintf(stderr, "read JrValid: Retrying\n");
}
if (count > (2*MAX_WAIT_JR3))
{
  fprintf(stderr, "nread JrValid: Timeout\n");
  retries += count;
  return(1); /*line did not change state*/
}

a=inpw(0x308);
*raw=a;
outp(0x30a, 0x00); /*request another piece of data*/
outp(0x30a, 0x10); /*by toggling the data-req line*/
retries += count;
return(0);

void dump_trailers(void)
{
  outp(0x30a, 0x00); /*request another piece of data*/
  outp(0x30a, 0x10); /*by toggling the data-req line*/
  outp(0x30a, 0x00); /*discard data rather than read*/
  outp(0x30a, 0x10); /*in the trailer words*/
}

/* void jr3_init(void):
Sets Up the Jr3 DMA interface and sends the first request for data */
void jr3_init(void)
{
  static int first_time=1;
  if (first_time)
  {
    printf("jr3_init: The JR3 performs best at 200 Hz\n");
    first_time=0;
  }
  outp(0x30b, 0x93);
  outp(0x30a, 0x00);
  outp(0x30a, 0x10);
kin.c

#include "arm.h"

/* int arm_fwd_kin_solve()
This function takes the joint angles of the arm and returns the cartesian position currently on the positional portion. A non zero value is returned if a certain configuration is not possible */
int arm_fwd_kin_solve(ARM_JOINTS *j, ARM_WORLD *p)
{
    double S1, S2, S3, S4, C1, C2, C3, C4;
    /* S1 S2 S3 S4 and C1 C2 C3 C4 will contain the values of sin(theta1) sin(theta2) so as to calculate them only once*/

    S1=sin(j->t1);
    S2=sin(j->t2);
    S3=sin(j->t3);
    S4=sin(j->t4);
    C1=cos(j->t1);
    C2=cos(j->t2);
    C3=cos(j->t3);
    C4=cos(j->t4);
    p->x=S1 * C2 * C3 * S4 * LEN3 - C1 * S3 * S4 * LEN3 + S1 * S2 * C4 * LEN3 + S1 * S2 * LEN2;
    p->y=-S2 * C3 * S4 * LEN3 + C2 * S4 * LEN3 + C2 * LEN2 + LEN1;
    p->z=C1 * C2 * C3 * S4 * LEN3 - S1 * S3 * S4 * LEN3 - C1 * S2 * C4 * LEN3 - C1 * S2 * LEN2;
    p->pr=j->t3;
    return(0);
}

/* int arm_inv_kin_solve()
This function takes the cartesian position of the arm and returns the corresponding joint angles. OVER and LEFTY configuration as defined for a PUMA560 are the preferred configurations A non zero value is returned if a certain position is out of range(yet to be implemented). */
int arm_inv_kin_solve(ARM_JOINTS *j, ARM_WORLD *p)
{
    double X_2, Y_2, Z_2, S3, C3, S4, C4;
    ARM_JOINTS prev;

    memcpy(&prev, j, sizeof(ARM_JOINTS));
    /* X_2 thru Z_2 will contain the squares of x, y, z S3, C3, S4, C4 contain sin(theta4) etc to prevent repeated calculation*/
    X_2=p->x * p->x;
    Y_2=p->y * p->y;
    Z_2=p->z * p->z;

    /*Calculation of angle theta 3, the redundancy angle*/
    { j->t3=p->pr; /* solution for theta 3, already known */
      if(j->t3 > JOINT_3_MAX)
          j->t3=JOINT_3_MAX;
      else if(j->t3 < JOINT_3_MIN)
          j->t3=JOINT_3_MIN;
    }

    /*Calculation of angle theta 4, the second pitch angle*/

double u, v, t;
t=(X_2 + Z_2 + ((p->y - LEN1)*(p->y - LEN1))) - LEN3_2 - LEN2_2)/(2 * LEN2 * LEN3);
if(t >=1)
    goto NO_SOLUTION;
else
{
    u=t;
    v=sqrt(1 - t*t);
    j->t4=atan2(v, u);
}
S3=sin(j->t3);
C3=cos(j->t3);
S4=sin(j->t4);
C4=cos(j->t4);

/*Calculation of angle theta 1, the first roll angle*/
{
double u, v, t;
    u=S3 * S4 * LEN3;
t=X_2 + Z_2 - u * u;
if(t < 0)
    goto NO_SOLUTION;
else
{
    v=sqrt(t);
    t=atan2(p->z, p->x)+ atan2(v, u);
    /* Doing a mod PI and -PI on the angle */
    j->t1=atan2(sin(t), cos(t));
}

/*Calculation of angle theta 2, the first pitch angle*/
{
double u, v, w, t;
    u=sin(j->t1)* p->x - cos(j->t1)* p->z;
    v=LEN1 - p->y;
t=u * u + v * v -(S4 * C3 * LEN3)*(S4 * C3 * LEN3);
if(t < 0)
    goto NO_SOLUTION;
else
{
    w=sqrt(t);
    j->t2=atan2(v, u) + atan2(w, (S4 * C3 * LEN3));
}
}

if((j->t1 > JOINT_1_MAX) ||
    (j->t2 > JOINT_2_MAX) ||
    (j->t3 > JOINT_3_MAX) ||
    (j->t4 > JOINT_4_MAX) ||
    (j->t1 < JOINT_1_MIN) ||
    (j->t2 < JOINT_2_MIN) ||
    (j->t3 < JOINT_3_MIN) ||
    (j->t4 < JOINT_4_MIN))
{
    memcpy(j, &prev, sizeof(ARM_JOINTS));
    return(1);
}
else
    return(0);

NO_SOLUTION:
    memcpy(j, &prev, sizeof(ARM_JOINTS));
    return(1);
#include "..\..\graph\work\3d.h"

void main(void)
{
    ARM_WORLD pinit;
    static ARM_JOINTS
    speed={(double)0, (double)0, (double)0, (double)0, (double)0, (double)0, (double)
    0};
    struct tm *curtime;
    time_t bintime;

    arm version();
    /*init stick();*/
    install panic();
    time (bintime);
    curtime=localtime (bintime);
    printf("\time is %s\n", asctime (curtime));

    /*Initializing Everthing*/
    hardreset (GAIN, POLE, ZERO, SAMPLE_TIME);
    cursor move arm();
    shutdown();
}

void shutdown()
{
    static ARM_JOINTS
    speed={(double)0, (double)0, (double)0, (double)0, (double)0, (double)0, (double)
    0};

    printf("shutdown: .......\n");
    set int vel arm (speed);
    hardreset (GAIN, POLE, ZERO, SAMPLE_TIME);
}

double parametric_x (double t) {return (4+(double)t*3);}

double parametric_y (double t) {return ((double)1.370-t*3);}

double parametric_z (double t) {return ((double)-4-t*3);}

double parametric_pr (double t) {return ((double)0.0*t);}

double parametric_pp (double t) {return ((double)0.0*t);}

double parametric_pitch (double t) {return ((double)0.0*t);}

double parametric_roll (double t) {return ((double)0.0*t);}

double parametric_yaw (double t) {return ((double)0.0*t);}
motor.c

#include "motor.h"

/* This file contains routines that handle the HCTL-1000 directly and are
independant of the the equipment they are used on. This same file is usable
on the omibase and the arms */

/* int query_profile()
checks if the last trapezoidal command on the servo in the parameter list has
completed or not. It
is useful only in the trapezoidal mode. */
int query_profile(int servo)
{
return(((16 & hp_read(servo,7))==16)? TRUE: FALSE); /* ret the status b4 */
}

/* void idle_motor() immediately shuts down the HCTL-1000 referred to in the parameter
list. Thus cutting
power to its motor and placing the motion control processor in an idle mode. */
void idle_motor(int servo)
{
int stop;
if((servo<0)||(servo>7))
{
for(stop=0; stop<=7; stop++) /* if servo num bad*/
idle_motor(stop); /* stop all servos*/
fprintf(stderr,"idle_servo: bad servo number, all servos idled");
exit(1);
}
endif
hp_write(servo, 5, 1); /* change to idle mode */
}

/* void do_profile()
initiates the trapezoidal profile mode and takes the motor to the position defined
by
the parameter position */
void do_profile(int servo, long position)
{
int hi, md, lo;
if(position>8388607)
{
int stp;
for(stp=0; stp<=7; stp++)
idle_motor(stp);
}
fprintf(stderr,"do_profile: position out range");
exit(1);
}
endif
if(position<0)
position=-16777216;
endif
hi=(int) (position/65536);
md=(int) ((position-hi*65536)/256);
lo=(int) (position-hi*65536-md*256);
/* write position data */
hp_write(servo, 43, hi);  /* write position data */
hp_write(servo, 42, md);  /* -do- */
hp_write(servo, 41, lo);  /* -do- */
hp_write(servo, 0, 8);  /* start trapezoidal move */

/* void set_trap_vel()*/
sets the ceiling velocity for the trapezoidal mode only useful in the trapezoidal
mode */
void set_trap_vel(int servo, int vel)
{
  if((vel>127)||(vel<0))
    {
      int stp;
      for(stp=0;stp<7;stp++)
        idle_motor(stp);
      fprintf(stdout,"set_vel: velocity out of range");
      exit(1);
    }
  hp_write(servo, 40, abs(vel));  /* set max velocity */
}

/* void set_accl()*/
sets the acceleration parameter for the HCTL-1000 it is useful in integral velocity
control mode and
trapezoidal velocity control mode. */
void set_accl(int servo, int accl)
{
  int hi, lo;
  if((accl<0)||(accl>32767))
    {
      int stp;
      for(stp=0;stp<7;stp++)
        idle_motor(stp);
      fprintf(stdout,"set_accl: acceleration out of range");
      exit(1);
    }
  hi=accl/256;
  lo=accl-hi*256;
  hp_write(servo, 39, hi);  /* set the acceleration */
  hp_write(servo, 38, lo);  /* -do- */
}

/* void pvinit()*/
is used to initialize the integral velocity mode of the HCTL-1000 initially the
command velocity is set to zero */
void pvinit(int servo)
{
  hp_write(servo, 5, 1);  /* change to idle mode */
  hp_write(servo, 0, 0);  /* clear f0 */
  hp_write(servo, 0, 3);  /* clear f3 */
  hp_write(servo, 0, 13);  /* set f5 */
/* change in velocity may not exceed 7 bits hence do it in 2 steps*/
if(hp_read(servo, 60) > 170)
    hp_write(servo, 60, 170);
if(hp_read(servo, 60) < 83)
    hp_write(servo, 60, 85);

hp_write(servo, 60, 0); /* zero the velocity */
hp_write(servo, 39, 0); /* zero the accel */
hp_write(servo, 38, 0); /* zero the accel */
hp_write(servo, 5, 3);  /* jump into int-vel mode*/
}

/* void trapinit()
   initializes the trapezoidal velocity mode on the servo refered in the parameter servo */
void trapinit(int servo)
{
hp_write(servo, 5, 1);    /* change to idle mode */
hp_write(servo, 0, 0);    /* clear f0 */
hp_write(servo, 0, 3);    /* clear f3 */
hp_write(servo, 0, 5);    /* clear f5 */
hp_write(servo, 19, 0);   /* zero actual position */
hp_write(servo, 12, 0);   /* zero the command position */
hp_write(servo, 13, 0);   /* -do- */
hp_write(servo, 14, 0);   /* -do- */
hp_write(servo, 41, 0);   /* zero the final position */
hp_write(servo, 42, 0);   /* -do- */
hp_write(servo, 43, 0);   /* -do- */
hp_write(servo, 39, 0);   /* zero the acceleration */
hp_write(servo, 38, 0);   /* -do- */
hp_write(servo, 5, 3);    /* jump to position lock mode */
}

/* void hardreset()
   is a function call which causes a hard restart of the HCTL-1000. This reverts the chip back
to its power on condition */
void hardreset(int gain, int pole, int zero, int time)
{
int servo, flag, addr, board;
outp(C0, 192); /* high impedance mode of bus*/
for(board=1; board<4; board++)
{
    outp(PC, board);    /* select board */
    outp(FB, 25); /* force reset high */
    outp(FB, 1); /* strobe low */
    outp(FB, 25); /* take up again */
}

for(serv0=0; servo<=7; servo++)
{
    for(flag=0; flag<6; flag++)
        hp_write(servo, 0, flag); /* reset all flags */
    if(hp_read(servo, 60) > 170)
        hp_write(servo, 60, 170);
    if(hp_read(servo, 60) > 83)
        hp_write(servo, 60, 85);
    for(addr=5; addr<60; addr++)
        hp_write(servo, addr, 0); /* zero all registers */
hp_write(servo, 32, zero); /* preset zero */
hp_write(servo, 33, pole); /* preset pole */
hp_write(servo, 34, gain); /* preset gain */
hp_write(servo, 15, time); /* preset time */
hp_write(servo, 7, 0x81); /* sign reverse inhibit */
}

/*
char hp_read(). This function returns an 8-bit value from the register
number referred to by addr parameter, from the HCTL-1000 referred
as servo
*/
char hp_read(int servo, int addr)
{
    int svo;

    switch(servo)
    {
    case 0:
    case 1:
        outp(PC, 1), svo=servo;
        break;
    case 2:
    case 3:
        outp(PC, 2), svo=servo-2;
        break;
    case 4:
    case 5:
        outp(PC, 3), svo=servo-4;
        break;
    case 6:
    case 7:
        outp(PC, 4), svo=servo-6;
        break;
    default:
    {
        int stp;
        for(stp=0;stp<=7;stp++)
            idle_motor(stp);
        fprintf(stdout,"hp_read: serv0 not 0 thru 7");
        exit(1);
    }
    }

    outp(PA, 9); /* r/w high reset high*/
    outp((PA+2*svo), addr); /* setup adress*/
    outp((PA+1+2*svo), 0); /* latch the adress */
    inp((PA+1+2*svo)); /* enable output by dummy oper*/
    svo=inp((PA+1+2*svo)); /* read */

    return((char)svo);
}

/*
void hp_write(). This function write an 8-bit value num to the register
number addr on the control chip number servo
*/
void hp_write(int servo, int addr, int num)
{
    int svo;

    switch(servo)
{ }  
  case 0:  
  case 1:  
    outp(PC, 1), sv0=servo;  
    break;  
  case 2:  
  case 3:  
    outp(PC, 2), sv0=servo-2;  
    break;  
  case 4:  
  case 5:  
    outp(PC, 3), sv0=servo-4;  
    break;  
  case 6:  
  case 7:  
    outp(PC, 4), sv0=servo-6;  
    break;  
  default:  
    {  
      int stp;  
      for(stp=0;stp<=7;stp++)  
        idle_motor(stp);  
        fprintf(stdout,"hp_write: servo not 0 thru 7");  
        exit(1);  
    }  
}

if(num<0 && num>255))  
  {  
    int stp;  
    for(stp=0;stp<=7;stp++)  
      idle_motor(stp);  
    fprintf(stdout,"hp_write: Attempted bad value to hp register");  
    exit(1);  
  }
endif

outp((PA+2*svo), addr);  /* setup adress*/  
outp(PB, 8);  /* r/w low reset high*/  
outp((PA+1+2*svo), num);  /* output the data */  
outp(PB, 9);  /* r/w high reset high*/  
}

/*
  long get_cmd_pos(). reads in the commanded position for the trapezoidal move from the Final position register
*/
  long get_cmd_pos(int servo)  
  {  
    return((long)hp_read(servo, 41)+(long)hp_read(servo, 42)*256+(long)hp_read(servo, 43)*65536);  
  }

/*
int get_curr_vel() reads in the actual velocity value being given to the amplifiers from the FWM motor command port
*/
int get_curr_vel(int servo)  
  {  
    return(hp_read(servo, 9));  
  }
/* 
int get_cmd_vel(). This function reads in the commanded velocity for the 
trapezoidal mode. */
int get_cmd_vel(int servo)
{
    return(hp_read(servo, 40));
}

/*
void amps_on(). Now mostly defunct. This command turned the power to the 
analog amplifiers on.
*/
void amps_on()
{
    hp_write(0,5,3);
}

/*
void amps_off(). Now mostly defunct. This command turned the power to the 
analog amplifiers off.
*/
void amps_off()
{
    hp_write(0,5,1);
}

/*
void bus_init(). Needed to initialize the LAB-40-PC board
*/
void bus_init()
{
    outp(CO,192); /* high impedance mode of bus*/
}

/*
void read_analog(). Now mostly defunct. Used to read in 3 values from 
a 3-axis joystick connected to the A/D board.
*/
void read_analog(int* x, int* y, int* w)
{
    int delay, low, high;
    outp(PC, 3); /*board-sel#3 is LAB40-2 A/D board*/

    /* CHANNEL #1 */
    outp(CH1,0); /*set channel select to ch1*/
    for(delay=0; delay<100; delay++); /*delay loop */
    outp(PB,0); /*set byte select to read low byte*/
    low=inp(ANALOG); 
    low=inp(ANALOG); /*two accesses are required*/
    outp(PB,1); /*set byte select to read high byte*/
    high=inp(ANALOG); high=inp(ANALOG); /*two accesses are required*/
    *x=low+256*high; /*concatenate bytes*/

    /* CHANNEL #2 */
    outp(CH2,0); /*set channel select to ch2*/
    for(delay=0; delay<100; delay++); /*delay loop */
    outp(PB,0); /*set byte select to read low byte*/
    low=inp(ANALOG); low=inp(ANALOG); /*two accesses are required*/
    outp(PB,1); /*set byte select to read high byte*/
    high=inp(ANALOG); high=inp(ANALOG); /*two accesses are required*/
*/
*y=low+256*high; /*concatenate bytes*/
*/

/* CHANNEL #3 */
outp(CH3, 0); /*set channel select to ch3*/
for(delay=0; delay<100; delay++); /*delay loop */
outp(PB, 0); /*set byte select to read low byte*/
low=inp(ANALOG); low=inp(ANALOG); /*two accesses are required*/
outp(PB, 1); /*set byte select to read high byte*/
high=inp(ANALOG); high=inp(ANALOG); /*two accesses are required*/
*y=low+256*high; /*concatenate bytes*/
outp(PC, 1); /*board-sel#3 is LAB40-2 A/D board*/
*/

/*
void digital_setup(). This sets up the digital I/O board for us. This is
the board that is used to interface to the Industrial relay racks. This was
the one used for the Genetic Design Inc. work.
*/
void digital_setup()
{
    outp(CO, 194);
    outp(PC, DIGITAL);
    outp(DIG_CTRL, 139); /*port A is output B&C are input*/
    outp(736, 00); /*736 is port A on digital I/O*/
    outp(CO, 192);
}

/*
void digital_out(). This function outputs the value in the parameter
list to the digital I/O boards port.
*/
void digital_out(int value)
{
    outp(CO, 194);
    outp(PC, DIGITAL);
    value &= 0x000FF;
    outp(736, value);
    outp(CO, 192);
}

/*
int digital_in(). This function reads in a value from the digital I/O board
*/
int digital_in()
{
    int value;
    outp(CO, 194);
    outp(PC, DIGITAL);
    value=inp(737);
    outp(CO, 192);
    return value;
}

/*
long get_curr_pos(). This function reads in the current position(actual) from
the encoder count register on the HCIL-1000
*/
long get_curr_pos(int servo)
{
    long b1, b2, b3, holder=0;
    int flg, svo;

if((servo==0) || (servo==1))
    outp(PC, 1), flg=FALSE, svo=servo; /*select board*/
else if((servo==2) || (servo==3))
    outp(PC, 2), flg=TRUE, svo=servo-2;
else if((servo==4) || (servo==5))
    outp(PC, 3), flg=TRUE, svo=servo-4;
else if((servo==6) || (servo==7))
    outp(PC, 4), flg=TRUE, svo=servo-6;
else
    {
        int stp;
        for(stp=0; stp<7; stp++)
            idle_motor(stp);
        fprintf(stdout,"get_servo_pos: servo not 0 thru 7");
        exit(1);
    }
endif

outp(PB, 9);
outp((PA+2*svo), 20);
outp((PA+1+2*svo), 0);
inp((PA+i+2*svo));
b1=inp((PA+i+2*svo));

outp(PB, 9);
outp((PA+2*svo), 19);
outp((PA+1+2*svo), 0);
inp((PA+i+2*svo));
b2=inp((PA+i+2*svo));

outp(PB, 9);
outp((PA+2*svo), 18);
outp((PA+1+2*svo), 0);
inp((PA+i+2*svo));
b3=inp((PA+i+2*svo));

b1=b1;
b2-=256;
b3-=65536;

holder=b1 | b2 | b3;

if(holder>8388607)
    holder-=16777216;
if(flg)
    outp(PC, 1);
return(holder);
}

/*void clear_pos(). Clears the position registers on the HCTL-1000 to zero. */
void clear_pos()
{
    hp_write(0, 19, 0);
    hp_write(1, 19, 0);
    hp_write(2, 19, 0);
    hp_write(3, 19, 0);
    hp_write(4, 19, 0);
    hp_write(5, 19, 0);
    hp_write(6, 19, 0);
    hp_write(7, 19, 0);
}
mouselib.c

#include "mouselib.h"
#include <dos.h>

int mouse_check(void)
{
    int status;
    union REGS regs;
    regs.x.ax=0;
    int86(0x33,&regs,&regs);
    status=regs.x.ax;
    if(status==-1)
        return(TRUE);
    else
        return(FALSE);
}

void show_mouse(void)
{
    union REGS regs;
    regs.x.ax=1;
    int86(0x33,&regs,&regs);
}

void hide_mouse(void)
{
    union REGS regs;
    regs.x.ax=2;
    int86(0x33,&regs,&regs);
}

void mouse_get(int *x, int *y, int *button)
{
    union REGS regs;
    regs.x.ax=3;
    int86(0x33,&regs,&regs);
    *x=regs.x.cx;
    *y=regs.x.dx;
    *button=regs.x.bx;
}

void mouse_set(int x, int y)
{
    union REGS regs;
    regs.x.ax=4;
    regs.x.cx=x;
    regs.x.dx=y;
    int86(0x33,&regs,&regs);
}

void mouse_range_x(int min, int max)
{
    union REGS regs;
    regs.x.ax=7;
    regs.x.cx=min;
    regs.x.dx=max;
    int86(0x33,&regs,&regs);
void mouse_range_y(int min, int max)
{
    union REGS regs;
    regs.x.ax=8;
    regs.x.cx=min;
    regs.x.dx=max;
    int86(0x33,&regs,&regs);
}

void mouse_sensitivity(int sens_x, int sens_y)
{
    union REGS regs;
    regs.x.ax=15;
    regs.x.cx=sens_x;
    regs.x.dx=sens_y;
    int86(0x33,&regs,&regs);
}
#include "sockets.h"

int net_init_stick(void)
{
    char msg[10];

    if((sd=socket(AF_INET, SOCK_STREAM, 0))== -1)
    {
        perror("socket");
        exit(1);
    }

    memset(&ssin, 0, sizeof(ssin));
    memset(&spin, 0, sizeof(ssin));
    addrlen = sizeof(ssin);
    ssin.sin_family = AF_INET;
    ssin.sin_addr.s_addr = INADDR_ANY;
    ssin.sin_port = htons(PORT);

    if(bind(sd, (struct sockaddr *)&ssin, sizeof(ssin))== -1)
    {
        perror("bind");
        exit(1);
    }

    if(listen(sd, 5)== -1)
    {
        perror("listen");
        exit(1);
    }

    if((sd_current = accept(sd, (struct sockaddr *)&spin, &addrlen))== -1)
    {
        perror("net_init_stick: accept error");
        exit(1);
    }

    if((recv(sd_current, msg, 10, 0))== -1)
    {
        perror("net_init_stick: recv error");
        exit(1);
    }

    if(strcmp(msg, CODENAME, 10))
    {
        printf("net_init_stick: incorrect password\n", msg);
        msg[0] = PANIC_ALL;
        send(sd_current, msg, 1, 0);
        close(sd_current);
        close(sd);
        return(1);
    }

    msg[0] = ALL_OK;
    if(send(sd_current, msg, 1, 0)== -1)
    {
        perror("net_init_stick: acknowledge failed");
        exit(1);
    }
    return(0);
}

int net_close_stick(void)
{

close(sd_current);
close(sd);

#ifdef DEBUG201
printf("net_close_stick: exiting\n");
#endif

exit(0);
}
#include "motor.h"

void parametric_omnibase(float end_time, void(*path)(float, float, float*, float*, float*, float*, float*, float*, float*, int*))
{
    float pxd, pyd, pwd, pwe, pxa, pya, pwa;
    float vx, vy, wvd, vwe, vxa, vya, vwa, v1r, vyr, vwr;
    float axd, ayd, awd;
    float time, ave_time_step, rotation=(float)0.0;
    clock_t start_time, now_time;
    int total_itr, message;

    total_itr=0;  /* reset count of iterations*/
    ave_time_step=(float)0.05;
    pxa=pya=pwe=(float)0.0;      /* make actual position zero */
    vxd=vxd=(float)0.0;          /* make actual velocity zero */
    vxd=vxd=(float)0.0;          /* make desired velocity zero */
    clear_pos();                /* make all hardware counters zero*/
    start_time=clock()-(clock_t)5;  /* obtain time at function entry minus 10 milliseconds*/
    time=(float)0.0;
    message=ALL_OK;            /* initial message is OK */

    while(time < end_time)
    {
        /* increment the record of the total iterations */
        total_itr++;

        /* obtain desired trajectory velocity vector * /
        (*path)(time, ave_time_step, &pxd, &pyd, &pwd, &vxd, &vyd, &v1d, &axd, &ayd, &awd, &message);

        if(message==STOP_ALL)
            /* if a STOP_ALL message has been received then stop base and exit loop*/
            {
                inv_kin_vel_omnibase(vxr, vyr, vwr);
                break;
            }

        /* obtain actual trajectory */
        fwd_kin_vel_omnibase(time, ave_time_step, &pxa, &pya, &pwe, &vxa, &vya, &vwa);

        /* obtain the error signal position & velocity vector */
        pxe=pxd-pxa, pye=pyd-pya, pwe=pwd-pwa;
        vxe=vxd-vxa, vye=vyd-vya, vwe=vxd-vwa;

        /* resolve the acceleration into velocity by multiplying with ave_time_step*/
        /* and then calculate the resultant velocity vector*/
        vxe=(axd * ave_time_step + vxd + Kv * vxe + Kp * pxe);
        vye=(ayd * ave_time_step + vyd + Kv * vye + Kp * pye);
        vwe=(awd * ave_time_step + vwd + Kv * vwe + Kp * pwe);
        /* Perform Inverse Kinematics */
        inv_kin_vel_omnibase(vxr, vyr, vwr);
        /* Time Step computations in terms of real-time */
        These calculations sometimes result in a divide by zero if the PC clock has not ticked before we have done another iteration hence in such cases we will use the previous time step.
        The clock on a PC has 0.055 resolution and we are on the border line so we may get some time steps that are zero*/

        now_time=clock() - start_time;
    }
```c
ave time_step=(float)now_time/CLK_TCK/total_itr;
time+=ave time_step;
}

/*
fwd_kin_vel_omibase() obtains the actual
trajectory(position & velocity) by reading the position registers
and differentiating the results.
In the parm list px, py, pw are the positions
vx, vy, vw are the velocities,
time is time, and dt is the interval or time-step.
The forward solution is given by
-1
[M] w=R^V (here M is the reduced Jacobian)

M=
-0.66670.3333 0.3333
 0 -0.57740.5774
0.3333 0.3333 0.3333
*/
void fwd_kin_vel_omibase(float time, float dt, float *x, float *y, float *w, float
*vx, float *vy, float *vw)
{
  long ps1, ps2, ps3;
do ble xx, yy, ww;
    *vx=*x;
    *vy=*y;
    *vw=*w;
  ps1=get_curr_pos(0), ps2=get_curr_pos(1), ps3=get_curr_pos(2);
  xx=(double)ENCODER_2_MM *(-0.6667 * ps1 + 0.3333 * ps2 + 0.3333 * ps3);
  yy=(double)ENCODER_2_MM *(-0.5774 * ps2 + 0.5774 * ps3);
  ww=(double)ENCODER_2_MM *(0.3333 * ps1 + 0.3333 * ps2 + 0.3333 * ps3);

    *wx=(float)xx;
    *wy=(float)yy;
    *w=(float)ww;
    *vx=(*x - *vx)/dt;
    *vy=(*y - *vy)/dt;
    *vw=(*w - *vw)/dt;
}

void inv_kin_vel_omibase(float vx, float vy, float vw)
{
do ble vs1, vs2, vs3;
do ble v1, v2, v3;
  vs1=(double)MM_2_ENCODER * SAMPLE_SEC *(-vx - vw);
  vs2=(double)MM_2_ENCODER * SAMPLE_SEC *(0.5 * vx - 0.866025*vy - vw);
  vs3=(double)MM_2_ENCODER * SAMPLE_SEC *(0.5 * vx + 0.866025*vy - vw);

  v3=(int)vs3, v1=(int)vs1, v2=(int)vs2;
  set_int_vel_omibase(v1, v2, v3);
}

void set_int_vel_omibase(int sp1, int sp2, int sp3)
{
do ble curl, hold1, set1, cur2, hold2, set2, cur3, hold3, set3;
```
/* servo 0 read check */
sp1=(sp1>127)? 127: sp1;
sp1=(sp1<-127)? -127: sp1;

outp(PC, 1);
outp(PB, 9);  /* r/w high reset high*/
outp(PA, 60);  /* setup adress*/
outp((PA+1), 0);  /* latch the adress */
inp((PA+1));  /* enable output by dummy oper*/
cur1=inp((PA+1));  /* read */
cur1=char(inp((PA+1)));
hold1=sp1-curl;

if(abs(hold1)>127)
  set1=(unsigned char)(cur1+126*SIGN(hold1));
else
  set1=(unsigned char)(sp1);
endif

/* read and check servo 1 */
sp2=(sp2>127)? 127: sp2;
sp2=(sp2<-127)? -127: sp2;

outp(PC, 1);
outp(PB, 9);  /* r/w high reset high*/
outp((PA+2), 60);  /* setup adress*/
outp((PA+3), 0);  /* latch the adress */
cur2=inp((PA+3));  /* enable output by dummy oper*/
cur2=char(inp((PA+3));  /* read */
hold2=sp2-curl2;

if(abs(hold2)>127)
  set2=(unsigned char)(cur2+126*SIGN(hold2));
else
  set2=(unsigned char)(sp2);
endif

/* read and check servo 2 */
sp3=(sp3>127)? 127: sp3;
sp3=(sp3<-127)? -127: sp3;

outp(PC, 2);
outp(PB, 9);  /* r/w high reset high*/
outp(PA, 60);  /* setup adress*/
outp((PA+1), 0);  /* latch the adress */
inp((PA+1));  /* enable output by dummy oper*/
cur3=inp((PA+1));  /* read */
cur3=char(inp((PA+1)));
hold3=sp3-curl3;

if(abs(hold3)>127)
  set3=(unsigned char)(cur3+126*SIGN(hold3));
else
  set3=(unsigned char)(sp3);

/* write speed on servo 0 */
outp(PC, 1);
outp(PA, 60);  /* setup adress*/
outp(PB, 8);  /* r/w low reset high*/
outp((PA+1), set1);  /* output the data */

/* write speed on servo 1 */
outp(PC, 1);
outp((PA+2), 60);  /* setup adress*/
outp(PB, 8);  /* r/w low reset high*/
outp((PA+3), set2);  /* output the data */
/* write speed on servo 2 */
outp(FC, 2);
outp(FA, 60);  /* setup address*/
outp(FB, 8);   /* r/w low reset high*/
outp((FA+1), set3); /* output the data*/
outp(FC, 1);
}

void get_coords_omnibase(float *x, float *y, float *w)
{
float pos1, pos2, pos3;

pos1=(float) get_curr_pos(0)*ENCODER_SCALE/2.0/PI;
pos2=(float) get_curr_pos(1)*ENCODER_SCALE/2.0/PI;
pos3=(float) get_curr_pos(2)*ENCODER_SCALE/2.0/PI;

*x=(float) (2.0/3.0*WHEEL_RADIUS*(pos3-0.5*pos1-0.5*pos2));
*y=(float) (1.0/1.0 *WHEEL_RADIUS*(pos1-pos2));
*w=(float) (1.0/3.0*WHEEL_RADIUS/ROBOT_RADIUS*(pos1+pos2+pos3));
}

float normalize_omnibase(float *x, float *y, float *w)
{
double store;
float magnitude;

store=*x * *x + *y * *y + *w * *w;

if(store==0.0)
    return((float)0.0);
endif

magnitude=(float)sqrt(store);
*x=*x/magnitude,  *y=*y/magnitude,  *w=*w/magnitude;
return magnitude;
}

int move_omnibase(int x_eng, int y_eng, int w_eng, int speed_eng)
{
double x, y, w, servo[4], speed[4], vel, big, s1, s2, s3;
float r_accl;
long svc[4];
int spd[4], motor, accl, hold, flag, f[4];

x=(double)x_eng, y=(double)y_eng, w=(double)w_eng;

if((speed_eng>126)||(speed_eng<0))
{
    int stp;
    for(stp=0; stp<4; stp++)
    idle_motor(stp);
    fprintf(stdout,"omni_move: speed range error");
    amps_off();
    exit(1);
}

vel=(double)1.0 * speed_eng;
servo[2]=(double)1.0/WHEEL_RADIUS*(-x-ROBOT_RADIUS*w/RADIUS);
servo[0]=(double)1.0/WHEEL_RADIUS*(x/2.0+y*0.866025403-ROBOT_RADIUS*w/RADIAN);
servo[1]=(double)1.0/WHEEL_RADIUS*(x/2.0-y*0.866025403-ROBOT_RADIUS*w/RADIAN);

s1=(double)fabs(servo[0]);
s2=(double)fabs(servo[1]);
s3=(double)fabs(servo[2]);
s1=max(s1, s2);
big=max(s1, s3);

for(motor=0; motor<WHEELS; motor++)
{
    speed[motor]=vel*servo[motor]/big;
    svo[motor]=(long)(servo[motor]*ENCODER_SCALE);
    spd[motor]=(int)fabs(speed[motor]);
    if(spd[motor]<1) spd[motor]=1;
}

r_accel=(float)1.0;
hold=256.0*(r_accel-(int)r_accel); /* deliberate data conversion */
accel=(int)(r_accel*256+hold*256);

for(motor=0; motor<WHEELS; motor++)
{
    trapinit(motor);
    set_accel(motor, accel);
    set_trap_val(motor, spd[motor]);
    f[motor]=TRUE;
}

for(motor=0; motor<WHEELS; motor++)
do_profile(motor, svo[motor]);

flag=TRUE;
while(flag)
{
    for(motor=0; motor<WHEELS; motor++)
    {
        if(!query_profile(motor))
            f[motor]=FALSE;
        endif
    }
    flag=f[0]|f[1]|f[2];
    if(kbhit())
    {
        int stp;
        for(stp=0; stp<WHEELS; stp++)
            idle_motor(stp);
        printf(stdout,"omni_move: Keyboard interrupt, motion terminated\n");
        return(FALSE);
    }
    endif
}

{ int stp;
  for(stp=0; stp<WHEELS; stp++)
    idle_motor(stp);
}
return(TRUE);
void init_vel_mode_omniBase(int acceleration)
{
    int wheel_servo;

    hardreset(GAIN, POLE, ZERO, SAMPLE_TIME);
    for(wheel_servo=0; wheel_servo<WHEELS; wheel_servo++)
    {
        pvinit(wheel_servo);
        set_accl(wheel_servo, acceleration);
    }
}
#include "motor.h"

/* All the functions in this file have to do with setting up the panic
case handling and making sure that orderly shutdowns take
place. This includes shutting down power to all motors and turning
off the interrupts on the com ports. */

/*
install_panic():
  a call to this function results in setting up of traps for floating point,
  interrupt and abort conditions. For example a divide by zero would cause
  a call to the function panic_fpe and program would stop. */
void install_panic(void)
{
  if(signal(SIGFPE,panic_fpe)==SIG_ERR)
    {
      perror("Could not install SIGFPE\n");
      exit(0);
    }

  if(signal(SIGINT,panic_int)==SIG_ERR)
    {
      perror("Could not install SIGINT\n");
      exit(0);
    }

  if(signal(SIGABRT, panic_abrt)==SIG_ERR)
    {
      perror("Could not install SIGABRT\n");
      exit(0);
    }

  if(onexit(shutdown)==NULL)
    {
      perror("Could not setup exit function stack\n");
      exit(0);
    }
}

void panic_int(int sig)
{
  signal(SIGINT, SIG_IGN);
  printf("\npanic: Aborting! on Control C : code 0x%lx\n", sig);
  exit(0);
}

void panic_fpe(int sig, int sig_err_num)
{
  signal(SIGFPE, SIG_IGN);
  printf("\npanic: Aborting! on floating point error: 0x%lx 0x%lx\n",sig, sig_err_num);
  exit(0);
}

void panic_abrt(int sig)
{
  signal(SIGABRT, SIG_IGN);
  printf("\npanic: Aborting! on abnormal condition: 0x%lx\n",sig);
  exit(0);
}

#include <stdio.h>
#include <math.h>
#include <graph.h>
#include <time.h>
#include "pantilt.h"
#include "mouselib.h"

float ave_time_step;

void main(void)
{
    int mx=0, my=0, old_mx=0, old_my=0, vel_pan=0, vel_tilt=0, pan=0, tilt=0;
    int cmd_pan=0, cmd_tilt=0, vel_act_pan=0, vel_act_tilt=0, pos_pan=0, pos_tilt=0;
    int button=0, del_pan=0, del_tilt=0;
    long total_itr=0;
    float pan_angles, tilt_angles;
    char str[80];
    clock_t start_time, now_time;

    if(!mouse_check())
    {
        perror("pantilt: unable to detect mouse or trackball");
        exit(0);
    }

    mouse_sensitive(32, 64);

    install_panic();
    init_pt();

    setvideomode(HRES0);
    mouse_range_x(X_MIN+5, X_MAX-5);
    mouse_range_y(Y_MIN+3, Y_MAX-3);
    mouse_set(X_CEN, Y_CEN);
    mx=X_CEN, my=Y_CEN;
    print_blurb();
    show_mouse();
    start_time=now_time();
    setcolor(1);

    while(!kbhit())
    {
        mouse_get(&mx, &my, &button);
        if(button!=0)
        {
            mx=X_CEN, my=Y_CEN;
            mouse_set(mx, my);
        }

        curr_ang_pt(pantilt_angles, &tilt_angles);

        cmd_pan=(X_CEN-mx)/2, cmd_tilt=-(Y_CEN-my)/3;
        vel_pan=((float)(cmd_pan-pantilt angles)*10);
        vel_tilt=((float)(cmd_tilt-tilt_angles)*10);

        vel_pan=(vel_pan>60) ? 60: vel_pan;
        vel_pan=(vel_pan<60) ? -60: vel_pan;
        vel_tilt=(vel_tilt>60) ? 60: vel_tilt;
        vel_tilt=(vel_tilt<-60) ? -60: vel_tilt;

        /* control motor here*/
        set_vel_pt(vel_pan, vel_tilt);
if(!(total_itr%20))
{
    _settextposition(8,8);
    sprintf(str, "%4.1f %4.1f %4.1f", pan_ang, tilt_ang);
    _outtext(str);
    get_vel_pt(&val_act_pan, &val_act_tilt);
    val_act_pan/=VEL_SCALE, val_act_tilt/=VEL_SCALE;
    _settextposition(11,8);
    sprintf(str, "%d %d",val_act_pan,val_act_tilt);
    _outtext(str);
    _settextposition(14,8);
    sprintf(str, "%d %d",cmd_pan, cmd_tilt);
    _outtext(str);
}
    total_itr+=;
}
now_time=clock()- start_time;
ave_time_step=(float)now_time/CLK_TCK/total_itr;
}
#include "motor.h"

/* these are the parametric movement routines a pointer to one of these must be
   passed
to the parametric_omni_base() routine as one of the arguments */

/* parm_path_base()
this routine uses the supplied functions parm_x(), parm_y(), parm_wz()
to generate a trajectory for the omni_base. */
void parm_path_base(float t, float dt, float *x, float *y, float *z, float *vx, float
  *vy, float *vz, float *ax, float *ay, float *az, int *message)
{
  /* for real parameter t=time*/
  /* parmetric_x=a + b t */
  /* parmetric_y=a + b t */
  /* parmetric_w=a + b t */
  x = parm_x(t);
  y = parm_y(t);
  z = parm_wz(t);
  vx = (parm_x(t + dt) - x) / dt;
  vy = (parm_y(t + dt) - y) / dt;
  vz = (parm_wz(t + dt) - z) / dt;
  ax = (*vx - (*x - parm_x(t + dt))/dt)/dt;
  ay = (*vy - (*y - parm_y(t + dt))/dt)/dt;
  az = (*vz - (*z - parm_wz(t + dt))/dt)/dt;
  message = ALL_OK;
}

/* parm_vel_base()
this routine uses the supplied functions parm_x(), parm_y(), parm_wz()
to generate a velocity trajectory for the omni_base. */
void parm_vel_base(float t, float dt, float *x, float *y, float *z, float *vx, float
  *vy, float *vz, float *ax, float *ay, float *az, int *message)
{
  float vpx, vpy, vpz;
  vpx = *vx;
  vpy = *vy;
  vpz = *vz;
  *x = *vx * dt;
  *y = *vy * dt;
  *z = *vz * dt;
  *vx = parm_x(t);
  *vy = parm_y(t);
  *vz = parm_wz(t);
  *ax = (*vx - vpx)/ dt;
  *ay = (*vy - vpy)/ dt;
  *az = (*vz - vpz)/ dt;
  *message = ALL_OK;
}

/* This routine uses a local hand controller, (keyboard 3d joystick, NASA chair)
to generate a trajectory for the omni_base */
void local_3d_rpy_control(float t, float dt, float *wx, float *wy, float *wz, float
  *vwx, float *vwy, float *vwz, float *awx, float *awy, float *awz, int *message)
{
  int stick wx, stick wy, stick wz, msg;
  float vpx, vpy, vpz;
  double sin_theta, cos_theta, tx, vy;
static double rotation=0.0;

vpx=*vwx;
vpy=*vwy;
vpz=*v wz;
read_3d_rpy_stick(&stick_wx, &stick_wy, &stick_wz, &msg);

#if (CURRENT_STICK==KRAFT_STICK)
*vwx=(float) (stick_wx-KRAFT_STICK ZERO WX) * KRAFT_STICK_GAIN;
*vwy=(float) (KRAFT_STICK ZERO WY-stick_wy) * KRAFT_STICK_GAIN;
*v wz=(float) (KRAFT_STICK ZERO WZ-stick_wz) * KRAFT_STICK_GAIN;
#elif (CURRENT_STICK==MICRO_STICK)
*vwx=(float) (stick_wx-MICRO_STICK ZERO WX) * MICRO_STICK_GAIN;
*vwy=(float) (MICRO_STICK ZERO WY-stick_wy) * MICRO_STICK_GAIN;
*v wz=(float) (MICRO_STICK ZERO WZ-stick_wz) * MICRO_STICK_GAIN;
#elif (CURRENT_STICK==NASA_CHAIR)
*vwx=(float) (stick_wx-NASA CHAIR ZERO WX) * NASA CHAIR_GAIN;
*vwy=(float) (stick_wy-NASA CHAIR ZERO WY) * NASA CHAIR_GAIN;
*v wz=(float) (NASA CHAIR ZERO WZ-stick_wz) * NASA CHAIR_GAIN;
#elif (CURRENT_STICK==CURSOR_STICK)
*vwx=(float) (stick_wx-CURSOR_STICK ZERO WX) * CURSOR_STICK_GAIN;
*vwy=(float) (stick_wy-CURSOR_STICK ZERO WY) * CURSOR_STICK_GAIN;
*v wz=(float) (CURSOR_STICK ZERO WZ-stick_wz) * CURSOR_STICK_GAIN;
#endif

if ((msg==TRIGGER1) || (msg==TRIGGER2))   /*if trigger in either position */
{
    rotation+=*vwx / ROBOT_RADIUS * dt;
    sin_theta=sin(rotation);
    cos_theta=cos(rotation);
    vwx=*vwx * cos_theta + *vwy * sin_theta;
    vwy=*vwx * sin_theta + *vwy * cos_theta;
    *vwx=(float)vwx, *vwy=(float)vwy;
}
else
{
    rotation=0.0;
}
#endif

/* remote_3d_rpy_control()*/
This routine uses a handcontroller on a remote computer to generate
a trajectory for the omniBase */
void remote_3d_rpy_control(float t, float dt, float *wx, float *wy, float *wz, float
*vwx, float *vwy, float *v wz, float *awx, float *awy, float *awz, int *message)
{
    static int x, y, z, wxx, wwy, wwx, msg, port;
double sin theta, cos theta, vx, vy;
static double rotation=0.0;

    port=PORT_BASE;
    msg=*message;
    *awx=*vwx;
    *awy=*vwy;
    *awz=*v wz;
}
read_comm_packet(&x, &y, &z, &wwx, &wwy, &wwz, &msg, &port);

#if(CURRENT_STICK==KRAFT_STICK)
  *wwx=(float) (wwx-KRAFT_STICK ZERO WX) * KRAFT_STICK_GAIN;
  *wwy=(float) (wwy-KRAFT_STICK ZERO WY-wwy) * KRAFT_STICK_GAIN;
  *wwz=(float) (wwz-KRAFT_STICK ZERO WZ-wwz) * KRAFT_STICK_GAIN;
#elif(CURRENT_STICK==MICRO_STICK)
  *wwx=(float) (wwx-MICRO_STICK ZERO WX) * MICRO_STICK_GAIN;
  *wwy=(float) (wwy-MICRO_STICK ZERO WY-wwy) * MICRO_STICK_GAIN;
  *wwz=(float) (wwz-MICRO_STICK ZERO WZ-wwz) * MICRO_STICK_GAIN;
#elif(CURRENT_STICK==NASA_CHAIR)
  *wwx=(float) (wwx-NASA CHAIR ZERO WX) * NASA_CHAIR_GAIN;
  *wwy=(float) (wwy-NASA CHAIR ZERO WY) * NASA_CHAIR_GAIN;
  *wwz=(float) (wwz-NASA CHAIR ZERO WZ-wwz) * NASA_CHAIR_GAIN;
#elif(CURRENT_STICK==CURSOR_STICK)
  *wwx=(float) (wwx-CURSOR_STICK ZERO WX) * CURSOR_STICK_GAIN;
  *wwy=(float) (wwy-CURSOR_STICK ZERO WY) * CURSOR_STICK_GAIN;
  *wwz=(float) (wwz-CURSOR_STICK ZERO WZ-wwz) * CURSOR_STICK_GAIN;
#endif

if((msg==TRIGGER1)||(msg==TRIGGER2)) /*if trigger in either position */
{
  rotation+=wwz / ROBOT_RADIUS * dt;
  sin_theta=sin(rotation);
  cos_theta=cos(rotation);
  vx=wwx * cos_theta + wwy * sin_theta;
  vy=wwx * sin_theta + wwy * cos_theta;
  *wwx=(float)vx, *wwy=(float)vy;

} else
{
  rotation=0.0;
}

endif

*wx+=wwx * dt;
*wy+=wwy * dt;
*wwz+=wwz * dt;
*awx=(*wwx - *awx)/dt;
*awy=(*wwy - *awy)/dt;
*awz=(*wwz - *awz)/dt;
*message=msg;
}
parmarm.c

#include "arm.h"

/* these are the parametric movement routines. A pointer to one of these is passed to the parametric_omnibase() routine as one of the arguments */

/* parm_path_arm()  
this routine uses the supplied functions parametric_x() parametric_y()  
parametric_z() parametric_pr() to generate a trajectory for the omnibase. */
void parm_path_arm(double t, double dt, ARM_WORLD *p, ARM_WORLD *v, ARM_WORLD *a, int *message)
{
    /* for real parameter t=time*/
    /* parametric_pos=a + b t */

    p->x=parametric_x(t);
    p->y=parametric_y(t);
    p->z=parametric_z(t);
    p->pr=parametric_pr(t);
    p->roll=parametric_roll(t);
    p->pitch=parametric_pitch(t);
    p->yaw=parametric_yaw(t);
    p->rr=parametric_rr(t);

    v->x=(parametric_x(t+dt)- p->x) / dt;
    v->y=(parametric_y(t+dt)- p->y) / dt;
    v->z=(parametric_z(t+dt)- p->z) / dt;
    v->pr=(parametric_pr(t+dt)- p->pr) / dt;
    v->roll=(parametric_roll(t+dt)- p->roll) / dt;
    v->pitch=(parametric_pitch(t+dt)- p->pitch) / dt;
    v->yaw=(parametric_yaw(t+dt)- p->yaw) / dt;
    v->rr=(parametric_rr(t+dt)- p->rr) / dt;

    a->x=(v->x - (p->x - parametric_x(t-dt)))/dt)/dt;
    a->y=(v->y - (p->y - parametric_y(t-dt)))/dt)/dt;
    a->z=(v->z - (p->z - parametric_z(t-dt)))/dt)/dt;
    a->pr=(v->pr - (p->pr - parametric_pr(t-dt)))/dt)/dt;
    a->roll=(v->roll - (p->roll - parametric_roll(t-dt)))/dt)/dt;
    a->pitch=(v->pitch - (p->pitch - parametric_pitch(t-dt)))/dt)/dt;
    a->yaw=(v->yaw - (p->yaw - parametric_yaw(t-dt)))/dt)/dt;
    a->rr=(v->rr - (p->rr - parametric_rr(t-dt)))/dt)/dt;

    *message=ALL_OK;
}

/* This routine uses a local handcontroller, currently NASA chair and cursor only to generate a trajectory for the arm */
void local_6d_control(double t, double dt, ARM_WORLD *p, ARM_WORLD *v, ARM_WORLD *a, int *message)
{
    int x, y, z, roll, pitch, yaw, slide, dial, msg;
    float vpx, vpy, vpz, vpr, vroll, vppitch, vyaw, vrr;

    /* correct for the acceleration and postion calcs*/
    /*
    vpx=v->x, vpy=v->y, vpz=v->z, vpr=v->pr;
    vroll=v->roll, vppitch=v->pitch, vyaw=v->yaw, vrr=v->rr;
    *
read_6d_stick(&x, &y, &z, &roll, &pitch, &yaw, &slide, &dial, &msg);
    */
#if(CURRENT_STICK==NASA_CHAIR)

*/
v->x=(float) (x-NASA CHAIR ZERO X) * NASA CHAIR GAIN;
v->y=(float) (y-NASA CHAIR ZERO Y) * NASA CHAIR GAIN;
v->z=(float) (z-NASA CHAIR ZERO Z) * NASA CHAIR GAIN;
v->roll=(float) (roll-NASA CHAIR ZERO WX) * NASA CHAIR GAIN;
v->pitch=(float) (pitch-NASA CHAIR ZERO WY) * NASA CHAIR GAIN;
v->yaw=(float) (NASA CHAIR ZERO WZ-yaw) * NASA CHAIR GAIN;
v->rr=(float) (NASA CHAIR ZERO DIAL-dial) * NASA CHAIR GAIN;
#endif
v->x=(float) (x-CURSOR STICK ZERO X) * CURSOR STICK GAIN;
v->y=(float) (y-CURSOR STICK ZERO Y) * CURSOR STICK GAIN;
v->z=(float) (z-CURSOR STICK ZERO Z) * CURSOR STICK GAIN;
v->pr=(float) (slide-CURSOR STICK ZERO SLIDE) * CURSOR STICK GAIN;
v->roll=(float) (roll-CURSOR STICK ZERO WX) * CURSOR STICK GAIN;
v->pitch=(float) (pitch-CURSOR STICK ZERO WY) * CURSOR STICK GAIN;
v->yaw=(float) (CURSOR STICK ZERO WZ-yaw) * CURSOR STICK GAIN;
v->rr=(float) (CURSOR_STICK ZERO DIAL-dial) * CURSOR_STICK_GAIN;
#endif
p->x+=v->x * dt;
p->y+=v->y * dt;
p->z+=v->z * dt;
p->pr+=v->pr * dt;
p->roll+=v->roll * dt;
p->pitch+=v->pitch * dt;
p->yaw+=v->yaw * dt;
p->rr+=v->rr * dt;
p->x=p->y=p->z=p->pr=0;
p->roll=p->pitch=p->yaw=p->rr=0;
a->x=a->y=a->z=a->pr=0;
a->roll=a->pitch=a->yaw=a->rr=0;
*message=mesg;
}
#include "sockets.h"

/* This routine reads a network handcontroller, NASA chair only
to generate a trajectory for the arm */
void socket_6d_control(double t, double dt, ARM_WORLD *p, ARM_WORLD *v, ARM_WORLD *a,
int message)
{
COMM_PACKET s;
char msg[10];

#ifdef DEBUG201
printf("socket_6d_control: entered\n");
#endif

if((recv(sd_current, (char *)&s, PACKET_SIZE, 0))<1)
{
    perror("socket_6d_control: recv ");
    exit(1);
}

if(!(s.msg >=STOP_ARM)&&(s.msg <=PANIC_ALL))
msg[0]=ALL_OK;

if(send(sd_current, msg, 1, 0)<1)
{
    perror("socket_6d_control: send ");
    exit(1);
}

v->x=(double)(s.x-NASA_CHAIR_ZERO X) * NASA_CHAIR_GAIN;
v->y=(double)(s.y-NASA_CHAIR_ZERO Y) * NASA_CHAIR_GAIN;
v->z=(double)(s.z-NASA_CHAIR_ZERO Z) * NASA_CHAIR_GAIN;
v->pr=(double)(s.pr-NASA_CHAIR ZERO SLIDE) * NASA_CHAIR_GAIN;
v->roll=(double)(s.wx-NASA_CHAIR ZERO WX) * NASA_CHAIR_GAIN;
v->pitch=(double)(s.wx-NASA_CHAIR ZERO WY) * NASA_CHAIR_GAIN;
v->yaw=(double)(NASA_CHAIR ZERO WZ-s.wz) * NASA_CHAIR_GAIN;
v->rr=(double)(NASA_CHAIR_ZERO.Dial-s.rr) * NASA_CHAIR_GAIN;

v->x=(double)(v->x - NASA_CHAIR_GAIN) : v->x;
v->y=(double)(v->y + NASA_CHAIR_GAIN) : v->y;
v->z=(double)(v->z + NASA_CHAIR_GAIN) : v->z;

v->roll=(v->roll > 0) ? (v->roll - NASA_CHAIR_GAIN) : v->roll;
v->roll=(v->roll < 0) ? (v->roll + NASA_CHAIR_GAIN) : v->roll;
v->pitch=(v->pitch > 0) ? (v->pitch - NASA_CHAIR_GAIN) : v->pitch;
v->pitch=(v->pitch < 0) ? (v->pitch + NASA_CHAIR_GAIN) : v->pitch;
v->yaw=(v->yaw > 0) ? (v->yaw - NASA_CHAIR_GAIN) : v->yaw;
v->yaw=(v->yaw < 0) ? (v->yaw + NASA_CHAIR_GAIN) : v->yaw;

*message=s.msg;

if(s.msg==TOGGLEUP)
{
    v->grippers=8;
    *message=ALL_OK;
}
else if(s.msg==TOGGLEDN)
{
v->gripper=8;
    *message=ALL_OK;
}

else
    {
    *message=s.msg;
    v->gripper=0;
    }

/fix here for getting accelerations and positions*/
p->x=p->y=p->z=p->px=0;
p->roll=p->pitch=p->yaw=p->rr=0;
a->x=a->y=a->z=a->px=0;
a->roll=a->pitch=a->yaw=a->rr=0;
#include "arm.h"

/* This program uses the parm_path_arm function and is called by the parametric arm. 
The parametric_x, parametric_y, parametric_z, parametric_pr function can be modified
to generate a different path for the arm. */

void main(void)
{
    static ARM JOINTS
    speed=(double)0, (double)0, (double)0, (double)0, (double)0, (double)0, (double)0, (double)0;
    ARM WORLD home;
    struct tm *curtime;
    time_t bintime;

    arm_version();

    move_arm_keyboard();
    /*Initialzing Everthing*/
    install_panic();
    init_vel_mode_arm(MAX_SAFE_ARM_ACCELERATION);

    time(&bintime);
    curtime=localtime(&bintime);
    printf("\nReal time is %s\n", asctime(curtime));

    home.x=(double)0, home.y=(double)1370.0, home.z=(double)0, home.pr=(double)0;
    parametric_arm((double)10.0, &home, parm_path_arm);

    time(&bintime);
    curtime=localtime(&bintime);
    printf("\nTime is %s\n", asctime(curtime));

    set_int_vel_arm(speed);
    move_arm_keyboard();
    set_int_vel_arm(speed);
}

double parametric_x(double t)
{
    return((double)0.0*t);
}

double parametric_y(double t)
{
    double tmp;
    if(t<1.0)
        return((double)1370-5.0*t*t);
    if(t>9.0)
        return((double)1370-5.0*t*t);
    return((double)1370-5.0*t*t);
}

double parametric_z(double t)
{
    return((double)0.0*t);
}

double parametric_pr(double t)
{
    return((double)0.0*t);
double parametric_rr(double t)
{
    return((double)0.0*t);
}

double parametric_pitch(double t)
{
    return((double)0.0*t);
}

double parametric_roll(double t)
{
    return((double)0.0*t);
}

double parametric_yaw(double t)
{
    return((double)0.0*t);
}

void shutdown()
{
    static ARM_JOINTS
    speed=((double)0, (double)0, (double)0, (double)0, (double)0, (double)0, (double)0, (double)0);

    printf("shutdown: .......
");
    set_int_vel_arm(speed);
    hardreset(GAIN, POLE, ZERO, SAMPLE_TIME);
}
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <float.h>
#include <time.h>

/*
Greenleaf Comm-Library Include Files
*/
#define LINT_ARGS
#include <asiports.h>
#include <gf.h>
#include <binkeys.h>

/* Keyboard movement program allows the puma to be moved around
using the cursor keys */
#define INC 1

void sleep(int);

void main(void)
{
  int status, i=0, j=0, k=0, length, num_sent;
  char cart[80];
  unsigned mode=ASINOUT|ASCII|NORMALRX;
  clock_t retry;

  /*initialize the communication port COM1*/
  if((status=asiopen(COM1, mode, 256, 256, 9600, P_NONE, 1, 8, ON, ON))!=ASSUCCESS)
  {
    printf("pumatalk: Open COM1 Failed return code:%d\n",status);
    exit(status);
  }
  length=sprintf(cart, "ex serve");
  if((num_sent=asiputs(COM1,(char *)cart, -2)!=(length+2)))
  {
    fprintf(stderr, "pumatalk: Transmit Count Error(%d sent), Code
    [%d]\n",num_sent, _aserror);
    exit(0);
  }
  do
  {
    retry=clock();
    while(getrxcnt(COM1)<7)
    {
      if((retry+2000)<clock())
      {
        fprintf(stderr, "pumatalk: Receiver Timed out\n");
        exit(0);
      }
      sleep(100);
    }
    num_sent=asigets(COM1,(char *)cart, 40, -2);
    if(strcmp(cart, "PIMAS60", 7)!=NULL)
    {
      fprintf(stderr, "pumatalk: initial garbage [%s]\n",cart);
    }
    else
    {
printf("==&gt; received %s\n", cart);
break;
}

while(TRUE);

while(TRUE)
{
    if(i&gt;10) i=10;
    if(i&lt;-10) i=-10;
    if(j&gt;10) j=10;
    if(j&lt;-10) j=-10;
    if(k&gt;10) k=10;
    if(k&lt;-10) k=-10;

    if(kbhhit())
        switch(getkey())
        {
            case PGUP:
                k+=INC; break;
            case HOME:
                k-=INC; break;
            case CURUP:
                j+=INC; break;
            case CURDN:
                j-=INC; break;
            case CURRT:
                i+=INC; break;
            case CURLF:
                i-=INC; break;
            default:
                i=0; j=0; k=0; break;
        }

    length=sprintf(cart, "%d %d %d", i, j, k);

    if((num_sent=asiputs(COML, (char *)cart, -2) != (length+2)))
        fprintf(stderr, "pumatax: Transmit Count Error(%d sent), Code [%d]\n", num_sent, _aserror);
        exit(0);
    }

    retry=clock();
    while(getc(xmt(COML)) &lt; 5)
    {
        if((retry=5000) &lt; clock())
            {
                fprintf(stderr, "pumatax: Receiver Timed out\n");
                exit(0);
            }
    }

    num_sent=asigets(COML, (char *)cart, 40, -2);
    if(strcmp(cart, "sss", 3))
        fprintf(stderr, "pumatax: received garbage [%s]\n", cart);
        exit(0);
    }

    printf("good received%\n", cart);

    num_sent=asigets(COML, (char *)cart, 40, -2);
    fprintf(stderr, "garbage [%2x] %s\n", cart[0], cart);
void sleep(int ticks)
{
    clock_t entry_tick;

    entry_tick=clock()+ticks;
    while(entry_tick+ticks > clock());
}
pumacomm.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <float.h>
#include <time.h>

/* Greenleaf Comm-LibraryInclude Files */
#define LINT_ARGS
#include <asiports.h>
#include <gf.h>
#include <hmkeys.h>

/* Keyboard movement program allows the puma to be moved around
using the cursor keys */
#define INC 1
#define BEGIN_COMM "ex serve"
#define SIZE_BEGIN_COMM 8
#define OK_STR "O.K"
#define SIZE_OK_STR 3

void sleep(int);
void comm(char *, int);
#define SEND_PUMA(u) comm(u, TRUE);
#define GET_PUMA(u) comm(u, FALSE);

void main(void)
{
    int status, i=0, j=0, k=0, length, num_sent;
    char cart[80];
    unsigned mode=ASINOUT|ASCII|NORMALRX;
    clock_t retry;

    /*initialize the communication port COM1*/
    if((status=asiopen(COM1, mode, 256, 256, 9600, P_NONE, 1, 8, ON, ON))!=ASSUCCESS)
    {
        printf("pumatalk: Open COM1 Failed return code:%d\n",status);
        exit(status);
    }

    asiputs(COM1, (char *)"ex serve", 0x0D); /*initialize puma*/
    sleep(100);

    /*keyboard polling loop*/
    while(TRUE)
    {
        GET_PUMA(cart); /*obtain response*/
        printf("gothere \%s\n", cart);
        if(i>10) i=10;
        if(i<-10) i=-10;
        if(j>10) j=10;
        if(j<-10) j=-10;
        if(k>10) k=10;
        if(k<-10) k=-10;

        if(kbhit())
            switch(getkey())
            {
            case FGUP:
                k=INC; break;
            }
case HOME:
    k=INC; break;
case CURUP:
    j=INC; break;
case CURDN:
    j=INC; break;
case CURRT:
    i=INC; break;
case CURLF:
    i=INC; break;
default:
    i=0; j=0; k=0; break;
}
sprintf(cart, "%d %d %d", i, j, k);
SEND_PUMA(cart);
}

void sleep(int ticks)
{
    clock_t entry_tick;
    entry_tick = clock() + ticks;
    while (entry_tick + ticks > clock());
}

void comm(char* msg, int mode)                  /* mode TRUE means transmit */
{
    static char str[80]=BEGIN_COMM, tmp[80];
    static int length=SIZE_BEGIN_COMM;
    int number;
    char sample;
    time_t retry;
    if (length)
    {
        retry = clock();
        while (getrxcnt(COML)<(length+3))
        {
            if ((retry+300)< clock())
            {
                fprintf(stderr, "pumatalk: receiver timed out\n");
                exit(0);
            }
        }
        if ((number=asigets(COML, (char *)tmp, (length+2), -2))!=(length+1))
        {
            fprintf(stderr, "pumatalk: character count error [%x]\n", number);
            exit(0);
        }
        if (strcmp(str, tmp, length)!=_NULL)
        {
            fprintf(stderr, "pumatalk: duplex mismatch <%s>\n", tmp);
            exit(0);
        }
        if ((sample=asigetc(COML))!=0x0a)
        {
            fprintf(stderr, "pumatalk: duplex mismatch CR\n");
            exit(0);
        }
    }
    if (mode)
    {
        sleep(1);
    }
strcpy(str, msg);
length=strlen(str);
asiputs(COM1,(char*)str, 0xD);
}
else
{
    length=0;
    if(getrxcnt(COM1)<(SIZE_OK_STR+2))
        sleep(1);
    asigets(COM1,(char*)msg, 40, -2);
    if(strcmp(msg, OK_STR, SIZE_OK_STR))
    {
        fprintf(stderr, "pumata: puma signals error <\s>\n",msg);
        exit(0);
    }
}
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <float.h>
#include <time.h>
#include <malloc.h>
#include <conio.h>

/* Socket communications Include Files */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

/* Greenleaf Comm-Library Include Files */
#define LINT ARGS
#include <ascports.h>
#include <gf.h>
#include <imkeys.h>

/* Keyboard movement program allows the puma to be moved around using the cursor keys */
#define INT 1
#define BEGIN_COMM "ex serve"
#define SIZE_BEGIN_COMM 8
#define OK_STR "O.K"
#define SIZE_OK_STR 3
#define MAX_TIME 30
#define PORT 0x1234
#define EXIT MSG "exit"
#define endif

void wait(int);
void comm(char*);
int count=0;

void main(void)
{
    int status, i=0, j=0, k=0, rate, count, ii, jj, kk;
    char cart[80], pumacart[80];
    unsigned mode=ASINOUT|ASCII|NORMALRX ;
    clock_t timer, start;
    int sd, sd current, addr len, char_count;
    struct sockaddr_in sin, pin;

    /* sockets initialization begins here */
    if((sd=socket(AF_INET, SOCK_STREAM, 0))<=-1)
    {
        perror("pumaseck, socket");
        exit(1);
    }

    memset(&sin, 0, sizeof(sin));
    memset(&pin, 0, sizeof(pin));
    addr len=sizeof(sin);
    sin.sin_family=AF_INET;
    sin.sin_addr.s_addr=INADDR_ANY;
    sin.sin_port=s htons(PORT);
}
```
if(bind(sd, &sin, sizeof(sin))==-1)
{
    perror("pumsock, bind");
    exit(1);
}

if(listen(sd, 5)==-1)
{
    perror("pumsock, listen");
    exit(1);
}

if((sd_current=accept(sd, &sin, &addrlen))==-1)
{
    perror("pumsock, accept");
    exit(1);
}

/* sockets initialization ends here */

if((status=aslopen(COM1, mode, 1024, 1024, 9600, P_NONE, 1, 8, ON, ON))!=ASSUCCESS)
{
    printf("pumsock: Open COM1 Failed return code:%d\n",status);
    exit(status);
}

asiputs(COM1, (char *)"ex serve", 0xD); /*initialize puma*/
wait(1000);
start=clcock();
timer=start;&VAR_TIME*CLK_TCK;

do
{
    bzero(cart, sizeof(cart));
    if((char_count=recv(sd_current, cart, sizeof(cart), 0))==-1)
    {
        perror("pumsock, recv");
        exit(1);
    }

    printf("Message Recieved: 's' \t", cart);
    count=sscanf(cart, "%d%d%d", &ii, &jj, &kk);
    printf("count is %d \t", count);
    if(count!=3)
    {
        printf(cart, EXIT_MSG);
        printf("\n", "Cart");
    }
    else
    {
        /*since loop rate is approx 2 Hz and we do not want a speed of zero*/
        i=ii, j[jj, k=kk;
        if(i>10)i=10;
        if(i<10)i=-10;
        if(j>10)j=10;
        if(j<10)j=-10;
        if(k>10)k=10;
        if(k<10)k=-10;
        rate=(int)sqrt((double)i*i+(double)j*j+(double)k*k) + 1;
        sprintf(pumacart, "%d %d %d", rate, i, j, k);
        printf("\n", pumacart);
        comm(pumacart);
    }
}
endif

if(send(sd_current, cart, strlen(cart), 0)==-1)
{

perror("pumasock, send");
exit(1);
}

while((timer > clock()) && (strcmp(cart, EXIT_MSG, 4))){

putchar(0x07); /* bell because I am across the room on the other computer*/
printf("\nTime %d secs. Iterations %d\n", (timer-start)/CLK_TCK, count);
comm("quit");
putchar(0x07); /* bell*/

/* terminating socket */
close(sd_current);
close(sd);
/* done terminating sockets */
wait(500);
exit(0); /* automatically closes the comm interrupts */
}

void comm(char* msg)
{
static char str[80]=BEGIN COMM;
static int length=SIZE_BEGIN_COMM;
char sample;
time_t retry;
retry=clock();

while(TRUE)
{
    if(getxcount(COM1)>(SIZE_OK_STR+1))
    {
        sample=(char) asigetc(COM1);
        if(sample=='O')
        {
            if(asigetc(COM1)=='.'
                if(asigetc(COM1)=='K')
                {
                    sample=(char) asigetc(COM1);
                    break;
                }
        }
    }
    wait(1);
    if((retry+3000)<clock())
    {
        fprintf(stderr, "pumasock: sync timed out. packets=%d\n", count);
        exit(0);
    }
}

count++;
asiputs(COM1, (char *)msg, 0xD);
}

void wait(int ticks)
{
clock_t entry_tick;
entry_tick=clock() + ticks;
while(entry_tick + ticks > clock());
}
rtx.c

#include "rtx.h"

/* global variables */
int pam[8];
double current_time, ave_time_step=0.02; /*initial small positive value*/
clock_t start_tick;
RTX WORLD cur_pos;
RTX_JOINT cur_int;

int position_control(RTX_CARTESIAN_STATE* request)
{
  RTX WORLD initial, final, delta, vel;
  RTX_JOINT links;
  double exec_time, norm_time=1.0, step_time, hold;
  int status, positional_iteration=0;

  if(request->cs_status != S_DESIRED)
  {
    printf(stderr, "position_control: status conflict or bad\n");
    return(STATUS_ERROR);
  }

  status=rtx_forward(initial, &links, MOTOR READ ON);
  memcpy(&cur_pos, initial, (size_t)sizeof(RTX WORLD));
  memcpy(&cur_int, &links, (size_t)sizeof(RTX_JOINT));

  final.x=(double)request->cs_x / 8.0; /*convert from JD coordinates*/
  final.y=(double)request->cs_y / 8.0; /*to world coordinates*/
  final.z=(double)request->cs_z / 8.0;
  final.roll=(double)request->cs_roll / 64.0;
  final.pitch=(double)request->cs_pitch / 64.0;
  final.yaw=(double)request->cs_yaw / 64.0;
  final.gripper=(double)request->cs_gripper / 8.0;

  delta.x=final.x - initial.x;
  delta.y=final.y - initial.y;
  delta.z=final.z - initial.z;
  delta.roll=final.roll - initial.roll;
  delta.pitch=final.pitch - initial.pitch;
  delta.yaw=final.yaw - initial.yaw;
  delta.gripper=final.gripper - initial.gripper;

  hold=(fabs(delta.x)>fabs(delta.y))? fabs(delta.x): fabs(delta.y);
  hold=(fabs(delta.z)>hold)? fabs(delta.z): hold;
  exec_time=hold / SPEED; /*execution time for the move*/
  current_time=TIME_0;

  start_tick=clock()-(clock_t)AVE_TICS; /*fake the first time steps*/

  while(norm_time > 0)
  {
    positional_iteration+=1;
    step_time=(double)((clock()-(double)start_tick)/CLK_TCK/positional_iteration);

    vel.x=(final.x - delta.x * norm_time - cur_pos.x)/step_time;
    vel.y=(final.y - delta.y * norm_time - cur_pos.y)/step_time;
    vel.z=(final.z - delta.z * norm_time - cur_pos.z)/step_time;
    vel.roll=(final.roll - delta.roll * norm_time - cur_pos.roll)/step_time;
    vel.pitch=(final.pitch - delta.pitch * norm_time - cur_pos.pitch)/step_time;
    vel.yaw=(final.yaw - delta.yaw * norm_time - cur_pos.yaw)/step_time;
vel.grripper=(final.grripper - delta.grripper * norm_time - 
cur_pos.grripper)/step_time;

status=velocity_control(&vel);
current_time+=step_time;
/*check to see if another object is pending*/
if(is_event_waiting())
    norm_time=-1;
else
    norm_time=(exec_time-current_time)/exec_time;
return(status);

int rtx_init(RTX_WORLD* cur_pos, RTX_JOINT* cur_int)
{i
    int status;
    start_tick=clock()- DELTA_TIME; /*One time step, nominal*/
    current_time=(double)start_tick/CLK_TCK;
    if(!status=rtx_start_move()))
        status=rtx_forward(cur_pos, cur_int, MOTOR_READ_ON);
    printf("rtx_init: complete status %0x \n", status);
    return(status);
}

/* int velocity_control(req_vel): Start moving the RTX along the real world 
velocity vector, mm/secs. Read the code to find out more. */
int velocity_control(RTX_WORLD* req_vel)
{i
    RTX WORLD dst_pos, nxt_pos;
    RTX_JOINT req_int, inc_int;
    static RTX WORLD err_pos=(ZERO);
    static long total_iteration=0;
    double angle_phi, check_rad, angle_elb, max_yaw, min_yaw;
    int status;

    total_iteration++;
    current_time+=ave_time_step;
    nxt_pos.x=cur_pos.x + req_vel->x * ave_time_step + err_pos.x * Kp;
    nxt_pos.y=cur_pos.y + req_vel->y * ave_time_step + err_pos.y * Kp;
    nxt_pos.z=cur_pos.z + req_vel->z * ave_time_step + err_pos.z ;
    nxt_pos.roll=cur_pos.roll + req_vel->roll * ave_time_step + err_pos.roll * Kp;
    nxt_pos.pitch=cur_pos.pitch + req_vel->pitch * ave_time_step + err_pos.pitch * Kp;
    nxt_pos.yaw=cur_pos.yaw + req_vel->yaw * ave_time_step + err_pos.yaw * Kp;
    nxt_pos.grripper=cur_pos.grripper + req_vel->grripper * ave_time_step + err_pos.grripper
        * Kp;

    check_rad=sqrt(nxt_pos.x * nxt_pos.x + nxt_pos.y * nxt_pos.y);
    angle_phi=atan2(nxt_pos.y, nxt_pos.x);
    if(check_rad > MAX_RAD)/* must fix the x a y coords*/
    {
        nxt_pos.x=cos(angle_phi)* MAX_RAD;
        nxt_pos.y=sin(angle_phi)* MAX_RAD;
    }

    check_rad=sqrt(nxt_pos.x * nxt_pos.x + nxt_pos.y * nxt_pos.y);
    angle_elb = acos(check_rad / MAX REACH)* DEG_TO_RAD;
    max_yaw=MAX_YAW + angle_elb / 3.0 * 2.0;
min_yaw = MIN_YAW + angle_elb / 3.0 * 2.0;

nxt_pos.yaw = (nxt_pos.yaw > max_yaw) ? max_yaw : nxt_pos.yaw;
nxt_pos.yaw = (nxt_pos.yaw < min_yaw) ? min_yaw : nxt_pos.yaw;

nxt_pos.z = (nxt_pos.z > MAX_ZEE) ? MAX_ZEE : nxt_pos.z;
if (nxt_pos.z < MIN_ZEE) nxt_pos.z = MIN_ZEE;

nxt_pos.roll = (nxt_pos.roll > MAX_ROLL) ? MAX_ROLL : nxt_pos.roll;
nxt_pos.roll = (nxt_pos.roll < MIN_ROLL) ? MIN_ROLL : nxt_pos.roll;

nxt_pos.pitch = (nxt_pos.pitch > MAX_PITCH) ? MAX_PITCH : nxt_pos.pitch;
nxt_pos.pitch = (nxt_pos.pitch < MIN_PITCH) ? MIN_PITCH : nxt_pos.pitch;

if(status = rtx_inverse(&nxt_pos, &req jint))
{
    fprintf(stderr, "RTX inverse kinematics error: code %d\n", status);
    return(status);
}

status = rtx_inverse(&nxt_pos, &req jint);

inc jint.j1 = req jint.j1 - cur jint.j1;
inc jint.j2 = req jint.j2 - cur jint.j2;
inc jint.j3 = req jint.j3 - cur jint.j3;
inc jint.j4 = req jint.j4 - cur jint.j4;
inc jint.j5 = req jint.j5 - cur jint.j5;
inc jint.j6 = req jint.j6 - cur jint.j6;
inc jint.j7 = req jint.j7 - cur jint.j7;

inc jint.j1 = (inc jint.j1 > 7) ? 7 : inc jint.j1;
inc jint.j2 = (inc jint.j2 > 7) ? 7 : inc jint.j2;
inc jint.j3 = (inc jint.j3 > 7) ? 7 : inc jint.j3;
inc jint.j4 = (inc jint.j4 > 7) ? 7 : inc jint.j4;
inc jint.j5 = (inc jint.j5 > 7) ? 7 : inc jint.j5;
inc jint.j6 = (inc jint.j6 > 7) ? 7 : inc jint.j6;
inc jint.j7 = (inc jint.j7 > 7) ? 7 : inc jint.j7;

cur jint.j1 = cur jint.j1 + inc jint.j1;
cur jint.j2 = cur jint.j2 + inc jint.j2;
cur jint.j3 = cur jint.j3 + inc jint.j3;
cur jint.j4 = cur jint.j4 + inc jint.j4;
cur jint.j5 = cur jint.j5 + inc jint.j5;
cur jint.j6 = cur jint.j6 + inc jint.j6;
cur jint.j7 = inc jint.j7;

status = rtx_forward(&dst_pos, &cur jint, MOTOR_READ_OFF);

err_pos.x = nxt_pos.x - dst_pos.x;
err_pos.y = nxt_pos.y - dst_pos.y;
err_pos.z = nxt_pos.z - dst_pos.z;
err_pos.roll = nxt_pos.roll - dst_pos.roll;
err_pos.pitch = nxt_pos.pitch - dst_pos.pitch;
err_pos.yaw = nxt_pos.yaw - dst_pos.yaw;

memcpy(&cur_pos, &dst_pos, (size_t)sizeof(RTX_WORLD));

return(status);

/* int comp_sup_control(req vel); Start moving the RTX along the real world velocity vector, m/s. Read the code to find out more. The difference b/w this and velocity control() is that we have an inverted 5d pyramid within which the RTX end effector must stay. This pyramid has its peak at GOAL X, GOAL Y, GOAL Z, GOAL THETA. The slope of its sides is 1. */

int comp_sup_control(RTX_WORLD* req vel)
{
    RTX WORLD dst_pos, nxt_pos, jnt_pos;
    RTX_JOINT req jint, jnt jint;
    static RTX WORLD err_pos = (ZERO);
static long total_iteration=0;
double angle_total, check_rad, angle_elb, max_yaw, min_yaw, delta, delta_2;
int status;

total_iteration++;
current_time=ave_time_step;
nxt_pos.x=nxt_pos.x + req_vel->x * ave_time_step + err_pos.x * Kp;
nxt_pos.y=nxt_pos.y + req_vel->y * ave_time_step + err_pos.y * Kp;
nxt_pos.z=nxt_pos.z + req_vel->z * ave_time_step + err_pos.z;
nxt_pos.roll=nxt_pos.roll + req_vel->roll * ave_time_step + err_pos.roll * Kp;
nxt_pos.pitch=nxt_pos.pitch + req_vel->pitch * ave_time_step + err_pos.pitch * Kp;
nxt_pos.yaw=nxt_pos.yaw + reqVel->yaw * ave_time_step + err_pos.yaw * Kp;

for(i=0; i<nxt_pos.gripper; i++)
  nxt_pos.gripper[i].nxt_pos->gripper[i].ave_time_step + err_pos.gripper * Kp;

/*define the sides of the 5d pyramid*/
delta=(nxt_pos.x-GOAL_Z) ? (nxt_pos.z-GOAL_Z) : 0;
delta=delta * 0.5;
/*limit x and y to GOAL_Z*/
if(nxt_pos.z > GOAL_Z || nxt_pos.z < GOAL_Z) nxt_pos.z = GOAL_Z;

/*only vertical movement in the socket*/
lmt_pos.x = (GOAL X+delta) > nxt_pos.x) ? GOAL X+delta : nxt_pos.x;
lmt_pos.x = (GOAL X-delta) < nxt_pos.x) ? GOAL X-delta : nxt_pos.x;
lmt_pos.y = (GOAL Y+delta) > nxt_pos.y) ? GOAL Y+delta : nxt_pos.y;
lmt_pos.y = (GOAL Y-delta) < nxt_pos.y) ? GOAL Y-delta : nxt_pos.y;

/*limit roll of the body*/
lmt_pos.roll = (GOAL_ROLL+deltta 2) > nxt_pos.roll) ? (GOAL_ROLL+deltta 2) : nxt_pos.roll;
lmt_pos.roll = (GOAL_ROLL+deltta 2) < nxt_pos.roll) ? (GOAL_ROLL+deltta 2) : nxt_pos.roll;
lmt_pos.pitch = (GOAL_PITCH+deltta 2) > nxt_pos.pitch) ? (GOAL_PITCH+deltta 2) : nxt_pos.pitch;
lmt_pos.pitch = (GOAL_PITCH+deltta 2) < nxt_pos.pitch) ? (GOAL_PITCH+deltta 2) : nxt_pos.pitch;

/*limit yaw of the body*/
lmt_pos.yaw = (GOAL_YAW+deltta 2) > nxt_pos.yaw) ? (GOAL_YAW+deltta 2) : nxt_pos.yaw;
lmt_pos.yaw = (GOAL_YAW+deltta 2) < nxt_pos.yaw) ? (GOAL_YAW+deltta 2) : nxt_pos.yaw;

/*limit gripper if necessary*/
lmt_pos.gripper = nxt_pos.gripper;

check_rad=sqrt(lmt_pos.x * lmt_pos.x + lmt_pos.y * lmt_pos.y);
angle_phi=tan2(lmt_pos.y, lmt_pos.x);
if(check_rad > MAX_RAD)/*must fix the x a y coords*/
{
  lmt_pos.x = cos(angle_phi) * MAX_RAD;
  lmt_pos.y = sin(angle_phi) * MAX_RAD;
}

check_rad=sqrt(lmt_pos.x * lmt_pos.x + lmt_pos.y * lmt_pos.y);
angle_elb= acos(check_rad / MAX_REACH) * DEG_TO_RAD;
max_yaw=MAX_YAW + angle_elb / 3.0 * 2.0;
min_yaw=MN_YAW + angle_elb / 3.0 * 2.0;
lmt_pos.yaw = (lmt_pos.yaw > max_yaw) ? max_yaw : lmt_pos.yaw;
lmt_pos.yaw = (lmt_pos.yaw < min_yaw) ? min_yaw : lmt_pos.yaw;
lmt_pos.z = (lmt_pos.z > MAX_ZEE) ? MAX_ZEE : lmt_pos.z;
lmt_pos.z = (lmt_pos.z < MIN_ZEE) ? MIN_ZEE : lmt_pos.z;
lmt_pos.roll = (lmt_pos.roll > MAX_ROLL) ? MAX_ROLL : lmt_pos.roll;
lmt_pos.roll = (lmt_pos.roll < MIN_ROLL) ? MIN_ROLL : lmt_pos.roll;
lmt_pos.pitch = (lmt_pos.pitch > MAX_PITCH) ? MAX_PITCH : lmt_pos.pitch;
lmt_pos.pitch = (lmt_pos.pitch < MIN_PITCH) ? MIN_PITCH : lmt_pos.pitch;
lmt_pos.gripper = (lmt_pos.gripper > MAX_GFR) ? MAX_GFR : lmt_pos.gripper;
lmt_pos.gripper = (lmt_pos.gripper < MIN_GFR) ? MIN_GFR : lmt_pos.gripper;

if(status=rtxInverse(&lmt_pos, &req_jnt))
{
  fprintf(stderr, "RTX inverse kinematics error: code %x\n", status);
  return(status);
}
inc_jnt.j1=inc_jnt.j1+cur_jnt.j1;
inc_jnt.j2=inc_jnt.j2+cur_jnt.j2;
inc_jnt.j3=inc_jnt.j3+cur_jnt.j3;
inc_jnt.j4=inc_jnt.j4+cur_jnt.j4;
inc_jnt.j5=inc_jnt.j5+cur_jnt.j5;
inc_jnt.j6=inc_jnt.j6+cur_jnt.j6;
inc_jnt.j7=inc_jnt.j7+cur_jnt.j7;

if (inc_jnt.j1>7)? 7: inc_jnt.j1=(inc_jnt.j1<7)? -7: inc_jnt.j1;
if (inc_jnt.j2>7)? 7: inc_jnt.j2=(inc_jnt.j2<7)? -7: inc_jnt.j2;
if (inc_jnt.j3>7)? 7: inc_jnt.j3=(inc_jnt.j3<7)? -7: inc_jnt.j3;
if (inc_jnt.j4>7)? 7: inc_jnt.j4=(inc_jnt.j4<7)? -7: inc_jnt.j4;
if (inc_jnt.j5>7)? 7: inc_jnt.j5=(inc_jnt.j5<7)? -7: inc_jnt.j5;
if (inc_jnt.j6>7)? 7: inc_jnt.j6=(inc_jnt.j6<7)? -7: inc_jnt.j6;
if (inc_jnt.j7>7)? 7: inc_jnt.j7=(inc_jnt.j7<7)? -7: inc_jnt.j7;

cur_jnt.j1=inc_jnt.j1, cur_jnt.j2=inc_jnt.j2, cur_jnt.j3=inc_jnt.j3;
cur_jnt.j4=inc_jnt.j4, cur_jnt.j5=inc_jnt.j5, cur_jnt.j6=inc_jnt.j6;
cur_jnt.j7=inc_jnt.j7;

status=rtx_forward(&dst_pos, &cur_jnt, MOTOR_READ_OFF);

err_pos.x=lmnt.pos.x-dst_pos.x, err_pos.y=lmnt.pos.y-dst_pos.y;
err_pos.x=lmnt.pos.z-dst_pos.z, err_pos.roll=lmnt.pos.roll-dst_pos.roll;
err_pos.pitch=lmnt.pos.pitch-dst_pos.pitch, err_pos.yaw=lmnt.pos.yaw-dst_pos.yaw;
err_pos.gripper=lmnt_pos.gripper-dst_pos.gripper;

memcpy(&cur_pos, &dst_pos, (size_t)sizeof(RTX_WORLD));

if(status=encoder_feed(&inc_jnt, status))
{
    fprintf(stderr, "encoder_feed error: code %ixn", status);
    return(status);
}

return(status);

/* int rtx_inverse(pos, jnt): This function takes a world coordinate and
converts them into the appropriate robot joint coordinates
return=any one or combination of inverse kinematics error codes */
int rtx_inverse(RTX_WORLD* pos, RTX_JOINT* jnt)
{
    double theta1, theta2, theta3, phi, pitch, roll, yaw, grip, radius;
    int z_enc, retval=0;

    radius=sqrt(pos->x * pos->x + pos->y * pos->y);

    if(radius > MAX_REACH)    /*The position is out of the env*/
        retval=ENV_ERROR;

    z_enc=(int) (pos->z * ZEE_SCALE + ZEE_MIN); /*checking bounds on 2*/
    if((z_enc > ZEE_MAX) || (z_enc < ZEE_MIN))
        retval=Z_RANGE_ERROR;

    phi=atan2(pos->y, pos->x);
    theta1=acos(radius / MAX_REACH); /*507=2 * 253.5 link length */
    theta3=phi + theta1; /*lefty configuration only in order*/
    theta2=2 * theta1;  /*to prevent a config switch */

    if(theta3 > PI_BY_2)
    {
        theta3=PI_BY_2;
    }
else if(theta3 < -PI_BY_2) /* if lefty not possible */
    {
        theta3=-PI_BY_2;
        retval=SHOULD_ERROR; /* report a shoulder error */
    }

if(theta2 > ELBOW_MAX_RAD)
    {
        theta2=ELBOW_MAX_RAD;
        retval=ELBOW_ERROR; /* Elbow bounds check */
    }
else if(theta2 < ELBOW_MIN_RAD)
    {
        theta2=ELBOW_MIN_RAD;
        retval=ELBOW_ERROR; /* Elbow bounds check */
    }

if((pos->pitch > PITCH_MAX_DEG) || (pos->pitch < PITCH_MIN_DEG))
    retval=PITCH_ERROR; /* Check the pitch angle */

if((pos->roll > ROLL_MAX_DEG) || (pos->roll < ROLL_MIN_DEG))
    retval=ROLL_ERROR; /* Check the roll angle */

pitch=(double)(pos->roll + pos->pitch) / SCALER6;
roll=(double)(pos->pitch - pos->roll) / SCALER6;

theta1=SCALER5 * (pos->yaw / DEG_TO_RAD) - theta3 - theta2 / 2;

yaw=(theta2 * SCALER4 + YAW_MEAN_ENCODE;

if((theta1 < (yaw - YAW_MAX_ENCODE)) || (theta1 > yaw))
    retval=YAW_ERROR;

if((pos->gripper < GRIP_MIN) || (pos->gripper > GRIP_MAX))
    retval=GRIPPER_ERROR;

float grip=sqrt(0.00042 + 0.0000428 * pos->gripper) - 0.0584;
grip=grip / SCALER3; /* These 2 lines have converted */
/* Grip encoder - counts to m.m. */
if(!retval)
    {
        jnt->j1=(int)(theta2 * SCALER1);
        jnt->j2=(int)(theta3 * SCALER2);
        jnt->j3=(int)z_enc;
        jnt->j4=(int)roll;
        jnt->j5=(int)pitch;
        jnt->j6=(int)theta1;
        jnt->j7=(int)grip;
    }
return(retval);

/* int rtx_forward(current, link, toggle): */
This function would read the encoders on the arm and tell you the current position
of the arm in world coordinates return=results of the servo read performed */
int rtx_forward(RTX_WORLD* current, RTX_JOINT* link, int toggle)
{
    double theta2, theta3;
    int srv1, srv2, srv3, srv4, srv5, srv6, srv7;
    int status=0;

    #ifndef DEBUG1
    if(toggle)
        if((rtx_read(0, 1, &srv1, status) | rtx_read(1, 1, &srv2, status)
rtx_read(2, 1, &srv3, status) | rtx_read(3, 1, &srv4, status)
rtx_read(4, 1, &srv5, status) | rtx_read(5, 1, &srv6, status)
rtx_read(6, 1, &srv7, status))

return (RTX_FAILED_READ);
#endif

if (!toggle)
{
    srv1=link->j1, srv2=link->j2;
    srv3=link->j3, srv4=link->j4;
    srv5=link->j5, srv6=link->j6;
    srv7=link->j7;
}

theta2=(double) srv2 * 0.00059725;
theta3=(double) srv1 * 0.00119450;
current->y=253.5*(sin(theta2)+sin(theta2+theta3));
current->x=253.5*(cos(theta2)+cos(theta2+theta3));

if ((current->x==0) && (current->y==0))
    current->y=0.1;

current->x0=(int) ((srv3+3555)/3.7495);
current->yaw=(57.2958*(srv6/558.05+theta2+theta3/2));
current->pitch=(srv4 + srv5)* .07415/2;
current->roll=(srvv - srv4)* .07415/2;
current->gripper=((srv7* 0.0000214 + 0.0584)*(srv7 * 0.0000214 + 0.0584)) - 0.00342 / 0.0000428;
link->j1= srv1, link->j2= srv2, link->j3= srv3, link->j4= srv4, link->j5= srv5, link->j6= srv6, link->j7= srv7;

return(0);
}

/* int encoder_feed(joint, status)
Feeds the RTX_JOINT structure defined by 'joint' to the robot, returned conditions
are read back into 'status' return=new status of the arm */
int encoder_feed(RTX_JOINT* joint, int status)
{
    pam[0]=joint->j1;
pam[1]=joint->j2;
pam[2]=joint->j3;
pam[3]=joint->j4;
pam[4]=joint->j5;
pam[5]=joint->j6;
pam[6]=joint->j7;
pam[7]=status;

    status=rtx_interrupt(14);
    return(status);
}

/* int rtx_read(motor, data_code, *data, status): This function reads 'data' from
a servo defined by 'motor'. Information such as current location etc can be read.
return=current status of the arm */
int rtx_read(int motor, int data_code, int *data, int status)
{
    pam[0]=motor;
pam[1]=data_code;
    status=rtx_interrupt(12);
    *data=pam[6];
    return(status);
}
/* int rtx_write(motor, data_code, data, status): This routine writes encoder
   information 'data' to the arm controller's servo defined by 'motor'. Calling it is required only
   if you want joint interpolated movements. return=current status of the arm */
int rtx_write(int motor, int data_code, int data, int status)
{
   parm[0]=motor;
   parm[1]=data_code;
   parm[2]=data;
   parm[7]=status;
   status=rtx_interrupt(13);
   return(status);
}

/* rtx_init_comms(t, d, status):
   As the name suggests this routine is used to initialize the robot servo controllers
   or to put them in a specific state return=current status of the arm */
int rtx_init_comms(int t, int d, int status)
{
   parm[0]=t;
   parm[1]=d;
   parm[7]=status;
   status=rtx_interrupt(2);
   return(status);
}

/* int rtx_start_move()
   called without any parameters this function will prepare the rtx for working in
   the interpolated mode return=current status of the arm */
int rtx_start_move(void)
{
   #ifdef DEBUG1
   int status=0;
   status=rtx_init_comms(1,0,status);
   
   /*reloading the pids*/
   parm[7]=status;
   status=rtx_interrupt(6);
   status=parm[7];
   
   /*stoping any current motions*/
   parm[0]=3;
   parm[7]=status;
   status=rtx_interrupt(8);

   status=rtx_init_comms(1,0,status);
   if(status != 0)
      return(RTX_FAILED_INIT);
   else
      return(0);
   #endif
}

/* int rtx_end()
   Stops the arm interpolate mode once you are done recommended to prevent jerky motion
   in control modes other than the interpolate mode return=current status of the arm */
int rtx_end(void)
{
   #ifdef DEBUG1
   
   /* stoping any current motions*/
   parm[0]=3;
   parm[7]=status;
   status=rtx_interrupt(8);

   status=rtx_init_comms(1,0,status);
   if(status != 0)
      return(RTX_FAILED_INIT);
   else
      return(0);
   #endif
}
return(0);
#endif

#ifdef DEBUG1
int status;
status=rtx_init_comms(0, 0, status);
return(status);
#endif

/* int rtxinterrupt(number)
This is the low level communications routine, i.e. the gateway [:-)], between the
control
program and the Servo Controller. This is simply a messenger between the control code
and a
TSR. The TSR is needed to take care of error free serial communications between the
PC and the
RTX robot. The TSR sets up a state machine, well sort of. return=current status of
the arm */
int rtxinterrupt(int number)
{
char far *arguments=(char far *)&pam[0];
union REGS regs;
struct SEGS segs;
static int library_present=FALSE, library_check=FALSE;
#endif DEBUG1
if(! library_check)
{
    library_check=TRUE;
    regs.h.ah=0x35;
    regs.h.al=0x78;
    int86(0x21,&regs,&regs,&seg);
    if((regs.x.bx)!=OFFSET)
    {
        fprintf(stderr,"rtxinterrupt: communication TSR missing or corrupt\n");
        return(TSR_ABSENT);
    }
    library_present=TRUE;
}
if(!pam[7] && library_present)
{
    regs.x.ax=number;
    regs.x.bx=(unsigned int)FP_OFF((char far *)arguments);
    regs.x.cx=(unsigned int)FP_SEG((char far *)arguments);
    int86(0x78,&regs,&regs);
}
else
    return(TSR_ABSENT);

return(pam[7]);
#endif

#ifdef BLACKBOARD
int is_event_waiting()
{
    if(kbhit())
        if(((bios_keybrd(KEYBRD_READ) >> 8) == FUN01)
            return(TRUE);
    else
        return(FALSE);
}
#endif
rtx1.c

#include <dos.h>
#include <ctx.h>

#define offset 0x5AED
#define segment 00

rtx_int(number)
int number;
{
    char far *arguments = (char far *) & (parm[0]);
    union REGS regs;
    struct SEGREGS segs;

    if(! lib_check)
    {
        lib_check = 1;
        regs.h.au = 0x35;
        regs.h.al = 0x78;
        int86(0x21, &regs, &regs, &segs);
        if( (regs.x.bx) != offset ) parm[7]=1;
    }

    if(parm[7]==0)
    {
        regs.x.ax = number;
        regs.x.bx = (unsigned int) FP_OFF( (char far *) arguments);
        regs.x.cx = (unsigned int) FP_SEG( (char far *) arguments);
        int86(0x78, &regs, &regs);
    }

    return(parm[7]);
}
rtx2.c

/*
RTX library access routine
(high level part of the set to)
(be used with rtx1.asm & rtx.h)

for the
Microsoft C3.00 Compiler

COPYRIGHT october 17th 1987 by
SARVON ADNAN
284-84-1489
RICE UNIVERSITY
Houston, TX 77030

For more information check the rtx library
usage manual
*/

extern int parm[8];

arm_raw_command(ip,bytes,b1,b2,b3,ok)
int ip, bytes, b1, b2, b3, *ok ;
{
  parm[0] = ip ;
  parm[1] = bytes ;
  parm[2] = b1 ;
  parm[3] = b2 ;
  parm[4] = b3 ;
  parm[7] = *ok ;
  rtx_int(0) ;
  *ok = parm[7] ;
}

arm_raw_response(ip,bytes,b1,b2,b3,ok)
{
  parm[0] = ip ;
  parm[7] = *ok ;
  rtx_int(1) ;
  *bytes = parm[3] ;
  *b1 = parm[4] ;
  *b2 = parm[5] ;
  *b3 = parm[6] ;
  *ok = parm[7] ;
}

arm_init_comms(t,d,ok)
int t, d, *ok ;
{
  parm[0] = t ;
  parm[1] = d ;
  parm[7] = *ok ;
  rtx_int(2) ;
  *ok = parm[7] ;
}
arm_version(lib_ver, lib_rev, rom_ver, ok)
int *lib_ver, *lib_rev, *rom_ver, *ok ;
{
  parm[7] = *ok ;
  rtx_int(3) ;
  *lib_ver = parm[4] ;
  *lib_rev = parm[5] ;
  *rom_ver = parm[6] ;
  *ok = parm[7] ;
}

arm_restart(ok)
int *ok ;
{
  parm[7] = *ok ;
  rtx_int(4) ;
  *ok = parm[7] ;
}

arm_define_origin(ok)
int *ok ;
{
  parm[7] = *ok ;
  rtx_int(5) ;
  *ok = parm[7] ;
}

arm_reload_pids(ok)
int *ok ;
{
  parm[7] = *ok ;
  rtx_int(6) ;
  *ok = parm[7] ;
}

arm_set_mode(motor, motor_mode, ok)
int motor, motor_mode, *ok ;
{
  parm[0] = motor ;
  parm[1] = motor_mode ;
  parm[7] = *ok ;
  rtx_int(7) ;
  *ok = parm[7] ;
}

arm_stop(stop_mode, ok)
int stop_mode, *ok ;
{
  parm[0] = stop_mode ;
  parm[7] = *ok ;
  rtx_int(8) ;
  *ok = parm[7] ;
}

arm_go(go_mode, go_bits, ok)
int go_mode, go_bits, *ok ;
{
  parm[0] = go_mode ;
  parm[1] = go_bits ;
parm[7] = *ok ;
rtx_int(9) ;
*ok = parm[7] ;
}

arm_motor_status(motor, status, ok)
int motor, *status, *ok ;
{
parm[0] = motor ;
parm[7] = *ok ;
rtx_int(10) ;
*status = parm[6] ;
*ok = parm[7] ;
}

arm_general_status(status, ok)
int *status, *ok ;
{
parm[7] = *ok ;
rtx_int(11) ;
*status = parm[6] ;
*ok = parm[7] ;
}

arm_read(motor, data_code, data, ok)
int motor, data_code, *data, *ok ;
{
parm[0] = motor ;
parm[1] = data_code ;
rtx_int(12) ;
*data = parm[6] ;
*ok = parm[7] ;
}

arm_write(motor, data_code, data, ok)
int motor, data_code, data, *ok ;
{
parm[0] = motor ;
parm[1] = data_code ;
parm[2] = data ;
parm[7] = *ok ;
rtx_int(13) ;
*ok = parm[7] ;
}

arm_interpolate(i_data, ok)
int i_data[8], *ok ;
{
parm[0] = i_data[0] ;
parm[7] = *ok ;
rtx_int(14) ;
*ok = parm[7] ;
}
arm_soak(s, ok)
int s, *ok;
{
    parm[0] = s;
    parm[7] = *ok;
    rtx_int(15);
    *ok = parm[7];
}
rtx3.c

/*
RTX inverse kinematics solution
(high level part of the set to)
(use with rtx.lib & rtx.h)

for the
Microsoft C3.00 Compiler

COPYRIGHT october 23rd 1987 by
SARMAD AHNAN
284-84-1489

RICE UNIVERSITY
Houston, TX 77005

arm_move_cartesian is the routine that the
user should call to move the RTX in world
coordinates the routine arm_prepare() must
be called once at the start of a program
in order to initialize the arm, the
parameters to arm_move_cartesian are all
integers, the origin lies at the base of
the vertical track. x axis directly away
from the RTX the y axis to the left and
the zee upwards, orientations of the
gripper is defined follows, roll is the
angle in degrees that the gripper makes
around its axis (positive clockwise),pitch
is the inclination angle of the gripper
from the horizontal (negative downward),
yaw is the angle the whole wrist assembly
makes with the x axis (negative clockwise)
grp is the gripper opening (0 closed).
the pwx, pwy and pwz define coordinates
of the wrist point.

LIMITS
0 < pwx < 940 m.m
0 < pwy*2-pwy^2 < 508*2 m.m
-110 < yaw < 110 degrees
0 < grp < 85 m.m
-98 < pitch < 4 degrees
-132 < roll < 180 degrees
0 < speed < 160

care must be taken in moving downwards on
the zee axis considerable overshoot occurs
raising the possibility of the arm hitting
the end-stop or floor at high speed. "ok"
is the global variable defined for error-
detection by the lower level routines this
may be checked by the user to determine
the current error status of the arm these
error codes are listed in the RTX manuals.
errors may also however occur due to the
specification of impossible or singular
configurations, or positions outside the
working envelope these errors are flagged
by the integer returned by this function
error_code = arm_move_cartesian( );
ERRORS CODES
  error_code bit 0  zee coordinate error
  error_code bit 1  shoulder error
  error_code bit 2  elbow  error
  error_code bit 3  roll  angle  error
  error_code bit 4  pitch angle error
  error_code bit 5  yaw  angle  error
  error_code bit 6  gripper range error
  error_code bit 7  initialization error

  (warning: this program prefers a lefty configuration of the RTX, however if a
  point isn't accesible in this way then it
  automatically tries to access the point in
  a righty configuration, this may be a concern
  in your applications.) */

#include <math.h>
#define ABS(u) ( (u<0) ? (-u) : (u) )

extern int parm[8];
extern int ok;

int arm_prepare()
{
  int ret_code=0;

  ok = 0;
  arm_init_comms(1,0,&ok);
  arm_reload_pids(&ok);
  arm_stop(3,&ok);
  if(ok != 0)
    ret_code = 128;
  return(ret_code);
}

int arm_move_cartesian(pwx,pwy,pwz,roll,pitch,yaw,gr,precision);
int pwx,pwy,pwz,roll,pitch,yaw,gr,precision;

{
  int limit,th2,th3,z,w1,w2,y,arm_solve_inverse(),arm_move_joints();
  int gr, arm_calc_grip(),maxspeed = 160;

  if(ok == 0)
  {
    th2 = pwx;
    th3 = pwy;
    z = pwz;
    w1 = roll;
    w2 = pitch;
    y = yaw;
    gr = grp;

    limit = arm_solve_inverse(&th2,&th3,&z,&w1,&w2,&y);
    limit = limit + arm_calc_grip(&gr);
    if(limit == 0)
    {
      if((speed>maxspeed) || (speed<0))
        speed = maxspeed;
      arm_move_joints(th2,th3,z,w1,w2,y,gr,speed);
      arm_wait_if_busy(precision);
    }
```c
int arm_solve_inverse(th2, th3, z, w1, w2, y)
int *th2, *th3, *z, *w1, *w2, *y;
{
    int temp, temp1;
    double phi, yy, xx, r, theta, theta2, theta3, pitch, roll, yaw, hold;
    int retval = 0;

    if((*th2==0) && (*th3==0)) /* the atan2 function gives */
        *th3 = 1; /* a domain error at x=y=0 */
    xx = *th2;
    yy = *th3;
    r = xx * xx + yy * yy;
    r = sqrt(r);
    if(r > 507.0) /* out of reach */
        return(8);
    temp = *z * 3.7495 - 3555; /* 3.7495 is zed scale */
    if((temp > 0) || (temp < -3554)) /* zed not possible */
        retval = 1;
    phi = atan2(yy, xx);
    theta = acos(r / 507); /* 507 = 2 * 253.5 link length */
    theta2 = phi + theta; /* lefty configuration by default */
    theta3 = -2 * theta;
    if((theta2 > 1.5707) || (theta2 < -1.5707)) /* if lefty not possible*/
        {
            theta2 = phi - theta; /* try righty configuration */
            theta3 = 2 * theta;
            if((theta2 > 1.5707) || (theta2 < -1.5707)) /* if righty not possible*/
                retval = retval + 2; /* send error code */
        }
    if((theta3 > 2.635) || (theta3 < -3.1415)) /* can't bend elbow there */
        retval = retval + 4;
    if( (*w2 > 4) || (*w2 < -98) )
        retval = retval + 16;
    if( (*w1 > 181) || (*w1 < -132) )
        retval = retval + 8;
    pitch = (double)(*w1 + *w2) / .07415; /* roll and pitch are calculated*/
    roll = (double)(*w2 - *w1) / .07415; /* as a function of the w1 & w2 */
    theta = 558.05 * ((double)y/57.29577) - theta2 - theta3 / 2 ;
    yaw = (theta3 * 279.056) + 1071;
    if( (theta > yaw) || (theta < (yaw - 2072)) )
        retval = retval + 32;
    if(retval == 0)
        {
            *th2 = theta2 * 1674.183 ;
            *th3 = theta3 * 837.168 ;
            *z = temp ;
            *w1 = roll ;
            *w2 = pitch ;
            *y = theta ;
        }
    return(retval);
}
```
int arm_calc_grip(gr)
{
    int ret = 0;
    double temp1;

    temp1 = *gr;
    if(**gr < 0 || (**gr > 85))
        ret = 64;
    temp1 = sqrt(.00342 + 0.0000428 * temp1) - 0.0584;
    temp1 = temp1 / 0.0000214;
    *gr = temp1;
    return(ret);
}

int arm_move_joints(a,b,cc,dd,e,f,g,h)
{
    int status,motor;
    int limit,th2,th3,z,w1,w2,y;
    int th2o,th3o,z0,w1o,w2o,yo,gr0;
    int pwxsp,pwysp,pxwsp,pywsp,pwx2sp,pyw2sp,pwxosp,pwyosp;
    int t[8],s[8],c[8],count, dmax,d[8],r[8];
    int i,j,sum,temp;
    double store;

    r[1]=a; r[0]=b; r[2]=cc;
    r[6]=g; r[7]=h;

    if(ok != 0)
        return(0x8F00);

    arm_current_encoders(&c[1],&c[0],&c[2],&c[3],&c[4],&c[5],&c[6]);

    for(i=0;i<=6;++i)
        d[i] = ABS(r[i]-c[i]);

    d[2]=d[2];
    dmax=1;
    for(i=0;i<=6;++i)
        if( d[i] > dmax ) dmax=d[i];

    for(i=0;i<=6;++i)
    {
        store=r[7];
        store/=dmax;
        store*=d[i];
        s[i]=ABS(store);
        s[i]= (s[i]==0) ? (1) : (s[i]);
    }

    for(i=0;i<=6;++i)
    {
        arm_write(i,4,s[i],&ok);
        arm_write(i,3,r[i],&ok);
    }

    arm_go(1,0x1555 ,&ok);
int arm_wait_if_busy(precision)
int precision;
{
    int motor, sum=1;
    int ce[7];
    int cd[7];
    int status = 1;

    if(precision==0)
    {
    }
    else if(precision==1)
    {
        while((status & 1) != 0)
        {
            arm_general_status(&status,&ok);
            arm_stop(0,&ok);
        }
    }
    else
    {
        while((status & 1) != 0)
        {
            arm_general_status(&status,&ok);
            while(sum!=0)
            {
                for(motor=0;motor<=6;++motor)
                {
                    arm_read(motor,0,&ce[motor],&ok);
                    arm_read(motor,8,&cd[motor],&ok);
                    ce[motor] &= (0x38FF); /*kill 2 high bit*/
                    sum= ((ABS(cd[motor])-ABS(ce[motor])) < 0) ? (1) : (0);
                }
            }
            arm_stop(0,&ok);
        }
    }
}

int arm_move_wristpoint(pwx,pwy,pwz,speed,precision)
int pwx,pwy,pwz,speed,precision;
{
    int limit,th2,th3,z,arm_solve_inversel(),arm_move_joints1();
    int maxspeed = 160;

    if(ok == 0)
    {
        th2=pwx;
        th3=pwy;
        z =pwz;

        limit = arm_solve_inversel(&th2,&th3,&z);
        if(limit == 0)
        {
            if((speed>maxspeed)||(speed<0))
            {
                speed = maxspeed;
                arm_move_joints1(th2,th3,z,speed);
                arm_wait_if_busy(precision);
            }
        }
        return(limit);
    }
int arm_solve_inverse1(th2,th3,z)
int *th2,*th3,*z;
{
    int temp, temp1;
    double phi,yy,xx,r,theta,theta2,theta3,pltn,roll,yaw,hold;
    int retval = 0;

    if((*(th2==0)) && (*(th3==0))){
        /* the atan2 function gives */
        /* a domain error at x=y=0 */
        /*
        
        
        */
        *th3=1;
        xx = *th2;
        yy = *th3;
        r = xx * xx + yy * yy;
        r = sqrt(r);
        if(r > 507.0)
            return(8);
        return(8);
        temp = *z * 3.7495 - 3555;
        /* 3.7495 is zed scale */
        if((temp > 0) || (temp < -3554))
            /* zed not possible */
            /*
            */
            /*
            */
            retval = 1;
            phi = atan2(yy,xx);
            theta = acos(r / 507);
            /* 507 = 2 * 253.5 link length */
            theta2 = phi + theta;
            /lefty configuration by default */
            theta3 = -2 * theta;
            if((theta2 > 1.5707) || (theta2 < -1.5707))
                /* if lefty not possible*/
                { theta2 = phi - theta; /* try righty configuration */
                    theta3 = 2 * theta;
                    if((theta2 > 1.5707) || (theta2 < -1.5707)) /*if righty not possible*/
                        retval = retval + 2; /*send error code*/
                }
            if((theta3 > 2.635) || (theta2 < -3.1415)) /* cant bend elbow there */
                retval = retval + 4;

            if(retval == 0)
                { *th2 = theta2 * 1674.183;
                    *th3 = theta3 * 837.168;
                    *z = temp;
                }
            return(retval);
        }
}

int arm_move_joints1(th2,th3,z,speed)
int th2,th3,z,speed;
{
    int status,motor;

    arm_stop(3,&ok);
    if(Gk == 0)
    { for(motor=0;motor<=2;++motor)
        arm_write(motor,4,speed,&ok);
        arm_write(0,3,th3,&ok); /*feeding encoder counts into */
        arm_write(1,3,th2,&ok); /*individual motors */
        arm_write(2,3,z,&ok);
        arm_go(1,0x15,&ok); /*elbow,shoulder, zed motors go*/
    }
}
int arm_wait_if_busy1(precision)
    int precision;
{
    int motor;
    int current_error = 100;
    int current_deadband = 0;
    int status0, status1, status2, status = 1;

    if(precision==0)
    {
    }
    else if(precision==1)
    {
        while((status & 1) != 0)
        {
            arm_motor_status(0,&status0,&ok);
            arm_motor_status(1,&status1,&ok);
            arm_motor_status(2,&status2,&ok);
            status=((0x01 & (status0 | status1 | status2)));
        }
        arm_stop(0,&ok);
    }
    else
    {
        while((status & 1) != 0)
        {
            arm_motor_status(0,&status0,&ok);
            arm_motor_status(1,&status1,&ok);
            arm_motor_status(2,&status2,&ok);
            status=((0x01 & (status0 | status1 | status2)));
        }
        for(motor=0;motor<=2;++motor)
        {
            while((current_error>current_deadband) || (ok != 0))
            {
                arm_read(motor,0,&current_error,&ok);
                arm_read(motor,8,&current_deadband,&ok);
                current_error = (0x3FFF & current_error);    /*kill 2 high bit*/
            }
        }
        arm_stop(0,&ok);
    }
}

int arm_move_wrist(roll,pitch,yaw,speed,precision)
int roll,pitch,yaw,speed,precision;
{
    int limit,th2,th3,z,w1,w2,w3,arm_solve_inverse2(),arm_move_joints2() ;
    int maxspeed = 160 ;

    if(ok == 0)
    {
        w1 =roll;
        w2 =pitch;
        z =yaw;
        limit = arm_solve_inverse2(&w1,&w2,&z) ;
        if(limit == 0)
        {
            if((speed>maxspeed) || (speed<0))
                speed = maxspeed ;
}
arm_move_joints2(w1,w2,y,speed) ;
arm_wait_if_busy2(precision) ;
}
}

return(limit) ;
}

int arm_solve_inverse2(w1,w2,y)
int *w1,*w2,*y;
{
    int temp, temp1, th2, th3;
    double phi,yy,xx,r,theta,theta2,theta3,pitch,roll,yaw,hold;
    int retval = 0 ;

    arm_read(0,1,&th3,&ok);
    arm_read(1,1,&th2,&ok);
    theta2=(double)th2 * 0.00059725 ;
    theta3=(double)th3 * 0.00119450 ;

    xx = (253.5*(1+cos(theta3))*cos(theta2));
    yy = (253.5*(1+cos(theta3))*sin(theta2));
    r = xx * xx + yy * yy ;
    r = sqrt(r) ;
    if((yy==0)&(xx==0)) yy=1;
    phi = atan2(yy,xx) ;

    theta = acos(r / 507) ; /* 507 = 2 * 253.5 link length */
    if( (*w2 > 4) || (*w2 < -98) )
        retval = retval + 16 ;
    if( (*w1 > 181) || (*w1 < -132) )
        retval = retval + 8 ;

    pitch = (double)(*w1 + *w2) / .07415 ; /* roll and pitch are calculated*/
    roll = (double)(*w2 - *w1) / .07415 ; /* as a function of the w1 & w2 */
    theta = 558.05 * (((double)y/57.29577) - theta2 - theta3/2) ;
    yaw = (theta3 * 279.056) + 1071 ;

    if( (theta > yaw) || (theta < (yaw -2072)) )
        retval = retval + 32 ;

    if(retval == 0)
    {
        *w1 = roll ;
        *w2 = pitch ;
        *y = theta ;
    }
    return(retval) ;
}

int arm_move_joints2(w1,w2,y,speed)
int w1,w2,y,Speed ;
{
    int status,motor ;
arm_stop(3,&ok) ;
if(ok == 0)
{
    for(motor=3;motor<=5;++motor)
        arm_write(motor,4,speed,&ok) ;

    arm_write(3,3,w1,&ok); /*feeding encoder counts into */
    arm_write(4,3,w2,&ok); /*individual motors */
    arm_write(5,3,y,&ok);
    arm_go(1,0x540 ,&ok); /*roll,pitch,yaw motors go*/
}

int arm_wait_if_busy2(precision)
int precision;
{
    int motor ;
    int current_error = 100 ;
    int current_deadband = 0 ;
    int status3, status4, status5, status = 1;

    if(precision==0)
    { }
    else if(precision==1)
    { 
        while((status & 1) != 0)
        {
            arm_motor_status(3,&status3,&ok);
            arm_motor_status(4,&status4,&ok);
            arm_motor_status(5,&status5,&ok);
            status=( 0x01 & (status3 | status4 | status5));

            arm_stop(0,&ok) ;
        }
    }
    else
    { 
        while((status & 1) != 0)
        {
            arm_motor_status(3,&status3,&ok);
            arm_motor_status(4,&status4,&ok);
            arm_motor_status(5,&status5,&ok);
            status=( 0x01 & (status3 | status4 | status5));
        }

        for(motor=3;motor<=5;++motor)
        { 
            while((current_error>current_deadband) || (ok != 0))
            {
                arm_read(motor,0,&current_error,&ok) ;
                arm_read(motor,8,&current_deadband,&ok) ;
                current_error = (0x3FFF & current_error) ; /*kill 2 high bit*/
            }

            arm_stop(0,&ok) ;
        }
    }
}

int arm_move_grip(grp,speed,precision)
int grp,speed,precision ;
{ int limit, arm_move_joints();
  int gr, arm_calc_grip(), maxspeed = 160;

  if(ok == 0)
  {
    gr = grp;

    limit = arm_calc_grip(&gr);
    if(limit == 0)
    {
      if((speed>maxspeed) || (speed<0))
        speed = maxspeed;
      arm_move_joints3(gr, speed);
      arm_wait_if_busy(precision);
    }
  }
  return(limit);
}

int arm_move_joints3(g, speed)
int g, speed ;
{
  int status ;

  arm_stop(3, &ok);
  if(ok == 0)
  {
    arm_write(6, 4, speed, &ok);
    arm_write(6, 3, g, &ok);
    arm_go(1, 0x1000, &ok); /* all motors go, also gripper*/
  }
}

int arm_current_encoders(th2o, th3o, z, w1, w2, y, g)
int *th2o, *th3o, *z, *w1, *w2, *y, *g;
{
  int arm_read();
  int th2, th3, z, w1, w2, y, gr;

  arm_read(0, 1, &th3, &ok);
  arm_read(1, 1, &th2, &ok);
  arm_read(2, 1, &z, &ok);
  arm_read(3, 1, &w1, &ok);
  arm_read(4, 1, &w2, &ok);
  arm_read(5, 1, &y, &ok);
  arm_read(6, 1, &gr, &ok);

  *th2o = th2;
  *th3o = th3;
  *z = z;
  *w1o = w1;
  *w2o = w2;
  *yo = y;
  *g = gr;
}
rtx5.c

/*
 RTX library access routine
 (high level part of the set to)
 (be used with rtx1.c & rtx.h)
 for the
 Microsoft C5.00 Compiler
 COPYRIGHT Oct 20th 1990 by
 SARMAD ADVAN
 284-84-1489
 (713) 521-0548
 RICE UNIVERSITY
 Houston, TX 77030
*/

#include <stdio.h>
#include <math.h>
#include <get.h>
#include <lmbkeys.h>
extern int parm[8], ok;

main()
{
 static int out_encs[8]={0,0,0,0,0,0,0,0};
 int count;

 arm_init_comms(1,0,&ok);

 while(TRUE)
  {
   if(kbhit())
       {
         switch(getkey())
         {
           case PGUP:
             out_encs[0] += 1; break;
           case HOME:
             out_encs[0] -= 1; break;
           case CURUP:
             out_encs[1] += 1; break;
           case CURDN:
             out_encs[1] -= 1; break;
           case CURRT:
             out_encs[2] += 1; break;
           case CURLF:
             out_encs[2] -= 1; break;
           case PGDN:
             out_encs[3] += 1; break;
           case ENDKEY:
             out_encs[3] -= 1; break;
           default:
             out_encs[0]=0, out_encs[1]=0, out_encs[2]=0, out_encs[3]=0,
             printf("%d %d %d %d %d %d \r", out_encs[0], out_encs[1], out_encs[2],
             out_encs[3], out_encs[4], out_encs[5], out_encs[6]);
         }
         for(count=0; count<8; count++)
       }

 for(count=0; count<8; count++)
}

{ out_encs[count]=(out_encs[count]<7)? -7: out_encs[count];
 out_encs[count]=(out_encs[count]>7)? 7: out_encs[count];
 }

arm_interpolate(&out_encs[0],&ok);
if(!ok)
  break;

printf("code was %d \x\n", ok, ok);
arm_init_comms(0,0,&ok);
return(0);"
rtxchair.c

#include "..\..\arm\work\arm.h"
#include "rtx.h"

extern RTX WORLD cur_pos;
extern RTX_JOINT cur_int;
extern double current_time;

void main(void)
{
  RTX WORLD vel;
  ARM WORLD p, v, a;
  int message=ALL_OK, return_val;
  long int total_itr=1;
  double dt=0.1;

  init_stick();
sleep(100);

  if(rtx_init(&cur_pos, &cur_int))
    {
      perror("rtx_init: failed");
      exit(0);
    }

  while(message==ALL_OK)
    {
    dt=current_time / total_itr++;

    local_6d_control(current_time, dt, &p, &v, &a, &message);

    if(message!=ALL_OK)
      exit(0);

    printf("%4.2f %4.2f %2x: %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f \r",

    vel.x=v.x/5, vel.y=v.y/5, vel.z=v.z/5, vel.roll=v.roll/5, vel.pitch=v.pitch/5, vel.yaw=v.yaw/5, vel.gripper=v.pr/5;

    return_val=velocity_control(&vel);

    if((return_val=velocity_control(&vel)))
      printf("velocity_control: %d 0x%x\n", return_val, return_val);
    }
}

double parametric_x(double x)
{
  return(-.5*x);
}
double parametric_y(double x)
{
  return(0*x);
}
double parametric_z(double x)
{
  return(0*x);
}
double parametric_pr(double x)
return(0*x);
}
double parametric_roll(double x)
{
    return(0*x);
}
double parametric_pitch(double x)
{
    return(0*x);
}
double parametric_yaw(double x)
{
    return(0*x);
}
double parametric_rroll(double x)
{
    return(0*x);
}
rtxsuprx.c

#include "sockets.h"

extern RTX WORLD cur_pos;
extern RTX_JOINT cur_int;
extern double current_time;

void main(void)
{
  RTX WORLD vel;
  ARM WORLD p, v, a;
  int message=ALL_OK, return_val, count, ndx;
  long int total_itr=1;
  char msg[250];
  double t=1.0, dt=0.1;

  if(rtx_init(&cur_pos, &cur_int))
  {
    perror("rtx_init: failed");
    exit(0);
  }

  if(net_init_stick())
    net_close_stick();

  printf("rtxsuprx: running ....\n");

  /*prime the transmitter program*/
  msg[0]=ALL_OK;

  if(send(6d_current, msg, 1, 0)==-1)
  {
    perror("socket_6d_control: send ");
    exit(1);
  }

  do
  {
    dt=current_time / total_itr++;
    socket_6d_control(t, dt, &p, &v, &a, &message);

    #if (CURSOR STICK==CURSOR STICK)
    v.roll=0;
    v.pitch=0;
    v.yaw=0;
    #endif

    if((message==TRIGGER) || (message==TRIGGER2))
      {
        printf("%4.2f:vel %4.2f %4.2f %4.2f : pos %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f \\n", current_time, v.x, v.y, v.z, cur_pos.x, cur_pos.y, cur_pos.z, cur_pos.roll,
          cur_pos.pitch, cur_pos.yaw);

        vel.x=v.y*VEL_SCALE, vel.y=-v.x*VEL_SCALE, vel.z=-v.z*VEL_SCALE;
        vel.roll=v.roll*VEL_SCALE/3, vel.pitch=v.pitch*VEL_SCALE/3;
        vel.yaw=v.yaw*VEL_SCALE/3, vel.gripper=v.gripper;

        if((return_val=comp_sup_control(&vel))
          printf("comp_sup_control: code %d=0x%x\n", return_val, return_val);
      }
  }

  printf("\n");

  while(1);

  printf("\n");

  exit(0);
}
while (message==ALL_OK) || (message==TRIGGER1) || (message==TRIGGER2));
    net_close_stick();
}

double parametric_x(double x)
{
    return(-.5*x);
}
double parametric_y(double x)
{
    return(0*x);
}
double parametric_z(double x)
{
    return(0*x);
}
double parametric_pr(double x)
{
    return(0*x);
}
double parametric_roll(double x)
{
    return(0*x);
}
double parametric_pitch(double x)
{
    return(0*x);
}
double parametric_yaw(double x)
{
    return(0*x);
}
double parametric_rr(double x)
{
    return(0*x);
}
rtxsuptx.c

#include "sockets.h"

void init_stick(void);
void read_6d_stick(int*, int*, int*, int*, int*, int*, int*, int*);

main()
{
  COMM_PACKET coords;
double count=0.0;
int x, y, z, roll, pitch, yaw, slide, dial, message;
clock_t st_secs, end_secs;
double ave_secs;

init_socket_connection();
printf("socket done\n");
init_stick();
printf("chair done\n");

st_secs=clock();
do
{
  count+=1;
  read_6d_stick(&x, &y, &z, &roll, &pitch, &yaw, &slide, &dial, &message);
  coords.x=x, coords.y=y, coords.z=z;
  coords.wx=roll, coords.wx=pitch, coords.wx=yaw;
  coords.grip=slide, coords.pr=0, coords.rr=dial;
  coords.msg=message;
  printf("%d %d %d %d %d %f \r", x, y, z, roll, pitch, yaw, count);
  if(send_get((char *)&coords) == PANIC_ALL)
    
    printf("rtxmettx: panic button pressed or robot error occurred\n");
    break;
 }
while(message != PANIC_ALL);

end_secs=clock();
if(shutdown(sd, 2) == -1)
{
  perror("rtxmettx: shutdown");
  exit(1);
}
close(sd);

ave_secs=(double)1/count*((double)end_secs-(double)st_secs)/CLK_TCK;
printf("\nrtxmettx: total iterations %d servo rate %f\n", count, ave_secs);
}

init_socket_connection()
{
  char msg[10];
  if((hp=gethostbyname(ROBOT_HOST)) == 0)
    
    perror("init_socket_connection: gethostbyname ");
exit(1);
}

memset(&pin, 0, sizeof(pin));

pin.sin_family=AF_INET;
pin.sin_addr.s_addr=((struct in_addr *) (hp->h_addr)) -> s_addr;
pin.sin_port=htons(PORT);

if((sd=socket(AF_INET, SOCK_STREAM, 0))== -1)
{
 perror("init_socket_connection: socket ");
 exit(1);
}

if(connect(sd, (struct sockaddr *)&pin, sizeof(pin))==-1)
{
 perror("init_socket_connection: connect");
 exit(1);
}

if(send(sd, CODENAME, 10, 0)==-1)
{
 perror("init_socket_connection: send ");
 exit(1);
}

if(reCV(sd, msg, 1, 0)==-1)
{
 perror("init_socket_connection: recv ");
 exit(1);
}

if(msg[0] != ALL_OK)
{
 printf("init_socket_connection: password refused\n");
 exit(0);
}

int send_get(char *buf)
{
 char msg[10];

if(reCV(sd, msg, 1, 0)==-1)
{
 perror("send_get: recv ");
 exit(1);
}

if(msg[0] != ALL_OK)
 return(msg[0]);

if(send(sd, buf, PACKET_SIZE, 0)==-1)
{
 perror("send_get: send ");
 exit(1);
}

return(msg[0]);
}
RX.C

#include "motor.h"

/*
 This program is the receiver for a communications check (rx and tx).
 It will simply echo to screen whatever is transmitted by the
 base computer and also give a final display of the maximum
 theoretical communication throughput
 */
void main(void)
{
  int x, y, z, wx, wy, wz, msg, port;
  float time;
  clock_t start_time, now_time;
  long packet_count=0;

  base_version();
  init_comm(PORT_BASE);
  install_panic();

  start_time=clock();
  port=PORT_BASE;
  request_packet(port, ALL_OK);

  for(;;)
  {
    now_time=clock();
    time=(float)(now_time - start_time)/CLK_TCK;
    msg=ALL_OK;
    read_comm_packet(&x, &y, &z, &wx, &wy, &wz, &msg, &port);
    packet_count++;
    printf("%d %d %d %d\t%f\tsecs\r", packet_count, wx, wy, wz, msg, time);
  }
  exit(0);
}

void shutdown()
{
}
rxstk.c

#include "motor.h"

/*
This is the receiver end of our omnibase control program
It assumes that the trajectory generation is taking place
on another computer and it is communicating over the com
port
*/

void main(void)
{
    base_version();
    init_comm(PORT_BASE);
    install_panic();
    init_vel_mode_omnibase(MAX_SAFE_WHEEL_ACCELERATION);

    request_packet(PORT_BASE, ALL_OK);
    parametric_omnibase((float)600.0, remote_3d_rpy_control);

    exit(0);
}

void shutdown()
{
    printf("shutdown: .......
");
    set_int_vel_omnibase(0, 0, 0);
    hardreset(GAIN, POLE, ZERO, SAMPLE_TIME);
}

float parm_x(float t)
{
    return((float)0.0 * t);
}

float parm_y(float t)
{
    return((float)0.0 * t);
}

float parm_wz(float t)
{
    return((float)0.0 * t);
}
sim.c

#include "3d.h"

void parametric_graphic(double end_time, ARM_WORLD* pinit, void(*path)(double, double, ARM_WORLD*, ARM_WORLD*, ARM_WORLD*, Int*))
{
    ARM_WORLD pf, pd, vd, ad;
    ARM_JOINTS jd;
    float base_x=0, base_y=0, base_theta=0;
    double time, ave_time_step;
    clock_t start_time, now_time;
    int total_itr, message;

    total_itr=0;
    ave_time_step=(double)0.05;
    clear_pos();
    start_time=clock()-(clock_t)5;
    time=(double)0.0;
    message=ALL_OK;

    arm_inv_kin_solve(&jd, pinit);
    memcpy(&pf, pinit, sizeof(pf));
    drawrobot(time, base_x, base_y, base_theta, &jd, 1);

    while(time < end_time)
    {
        /* increment the record of the total iterations */
        total_itr++;
        /* obtain desired trajectory velocity vector */
        (*path)(time, ave_time_step, &pd, &vd, &ad, &message);

        if(message==PANIC_ALL)
            break;
        pf.x+=vd.x*ave_time_step;
        pf.y+=vd.y*ave_time_step;
        pf.z+=vd.z*ave_time_step;
        pf.roll+=vd.roll*ave_time_step;
        pf.pitch+=vd.pitch*ave_time_step;
        pf.yaw+=vd.yaw*ave_time_step;
        pf.pr=vd.pr/RADIAN;
        pf.rz=vd.rz/RADIAN;

        /* Perform Inverse kinematics */
        arm_inv_kin_solve(&jd, &pf);
        arm_fwd_kin_solve(&jd, &pf);
        drawrobot(time, base_x, base_y, base_theta, &jd, 1);

        now_time=clock() - start_time;
        ave_time_step=(double)now_time/CIK_TCK/total_itr;
        time+=ave_time_step;
    }
}
simstick.c

#include "3d.h"

void main(void)
{
    ARM_WORLD pinit;
    begingraphics();
    init_stick();

    pinit.x=(double)1, pinit.y=(double)1370.0, pinit.z=(double)1, pinit.pr=(double)0;
    parametric_graphic(600, &pinit, local_6d_control);
    shutdown();
}

double parametric_x(double t){return(1.0+(double)10.0*t);}

double parametric_y(double t){return((double)1370-10.0*t);}

double parametric_z(double t){return(1.0+(double)20.0*t);}

double parametric_pr(double t){return((double)0.0*t);}

double parametric_rr(double t){return((double)0.0*t);}

double parametric_pitch(double t){return((double)0.0*t);}

double parametric_roll(double t){return((double)0.0*t);}

double parametric_yaw(double t){return((double)0.0*t);}

void shutdown()
{
    endgraphics();
}
socstick.c

#include "..\..\..\base\work\motor.h"

/*Only function that are related to the handcontroller are present
in this file. Currently the NASA chair, Kraft joystick the micro
measurements 3 dof stick and the cursor pad on the keyboard can
be used as a joystick. The Kraft stick will not work on the
leading edge or the ampro, because of lack of rom support for
the game controller*/

/* read_6d_stick():
 This function reads the value from the 6 d.o.f. hand controller
 returns it to the calling program. A message is also returned*/
void read_6d_stick(int *x, int *y, int *z, int *wx, int *wy, int *wz, int *slide, int *dial, int *message)
{
  #if(CURRENT_STICK==NASA_CHAIR)
  int tmp, pots[9];
  static clock_t entry;

  restart:
  if(getrxcnt (NASA_CHAIR_PORT)!="9) 
    {
      printf("\nread_6d_stick: NASA chair communication Failure, check power and
cable\n");
      exit(0);
    }

  entry=clock()+ 2 * CLK_TCK;

  while(getrxcnt (NASA_CHAIR_PORT)<9)
    {
      if(clock()> entry)
        {
          printf("socstick: chair did not respond resetting\n");
          asisquit(NASA_CHAIR_PORT);
          init_stick();
          goto restart;
        }
    }

  for(tmp=0; tmp<9; tmp++)
    pots[tmp]=asigetc(NASA_CHAIR_PORT);

  asiputc(NASA_CHAIR_PORT, '\0');
  *x=(pots[3] > 0xff) ? 0xff: pots[3];
  *y=(pots[5] > 0xff) ? 0xff: pots[5];
  *z=(pots[4] > 0xff) ? 0xff: pots[4];
  *wx=(pots[2] > 0xff) ? 0xff: pots[2];
  *wy=(pots[0] > 0xff) ? 0xff: pots[0];
  *wz=(pots[1] > 0xff) ? 0xff: pots[1];
  *slide=(pots[6] > 0xff) ? 0xff: pots[6];
  *dial=(pots[7] > 0xff) ? 0xff: pots[7];
  *message=resolve_message(pots[8]);

  #elif(CURRENT_STICK==KRAFT_STICK)
  union REGS input, output;

  input.x.dx=1;           /*getting the pot values*/
  input.h.ah=0x84;
  int86(0x15, &input, &output);
*x=*y=2*z=0 *slide=*dial=KRAFT_STICK_ZERO_X;
*wx=(output.x.ax > 0xff)? 0xff: output.x.ax;
*wy=(output.x.bx > 0xff)? 0xff: output.x.bx;
*wz=(output.x.dx > 0xff)? 0xff: output.x.dx;

input.x.dx=0;  /*getting the button status*/
input.h.ah=0x84;
int86(0x15, &input, &output);
if((output.x.ax & 0xff)==0x00f0)
  *message=ALL_OK;
else
  *message=STOP_ALL;

if(*wx==0 && *wy==0 && *wz==0)
{
  printf("\nread 6d stick: kraft joystick did not respond\n");
  exit(0);
}

#elif(CURRENT_STICK==CURSOR_STICK)
static int xx=128, yy=128, zz=128;

if(kbhit())
  switch(getkey())
  {
  case PGUP:
    zz+=10 ;break;
  case HOME:
    zz-=10 ;break;
  case CURUP:
    yy+=10 ;break;
  case CURDN:
    yy-=10 ;break;
  case CURRT:
    xx+=10 ;break;
  case CURLF:
    xx-=10 ;break;
  default:
    xx=yy=zz=128; *message=STOP_ALL; break;
  }

*x=*y=2*z=0 *slide=*dial=CURSOR_STICK_ZERO_X;
*wx=xx, *wy=yy, *wz=zz;

#else
  printf("\n#define CURRENT_STICK=?\n");
  printf("\ninit stick: Illegal Hand controller specification\n");
  exit(0);
  #endif


*/
read_3d_rpy_stick();
This function reads the value from the current 3d joystick and
returns it to the calling program. A message is also returned */
void read_3d_rpy_stick(int *wx, int *wy, int *wz, int *message)
{
#if(CURRENT_STICK==NASA_CHAIR)
int tmp;
static missed_count=0;
static int pots[9]=
  {NASA_CHAIR_ZERO_X,
   NASA_CHAIR_ZERO_Y,
   NASA_CHAIR_ZERO_Z,
   NASA_CHAIR_ZERO_WX,
   NASA_CHAIR_ZERO_WY,
   NASA_CHAIR_ZERO_WZ,
NAS_CAHING ZERO WZ,
NAS_CAHING ZERO SLIDE,
NAS_CAHING ZERO DIAL,
NAS_CAHING ZERO BUTTONS
);

if((getrxcnt (NASA_CHAIR_PORT) !=9))
{
    missed_count++;        
    if(missed_count > MAX_MISSED_RESPONSE)        
    {
        printf("\nread_3d_ply_stick: NASA chair communication Failure, check power and cable\n");
        ifdef DEBUG4
            printf("\ndebug output: %d characters received\n",getrxcnt (NASA_CHAIR_PORT));
        endif
        exit(0);
    }
}
else        
{
    missed_count=0;
    for(tmp=0; tmp<9; tmp++)        
        pots[tmp]=asigetc (NASA_CHAIR_PORT);
    asiputc (NASA_CHAIR_PORT, '\0');
}
endif

*wz=(pots[2] > 0xff)? 0xff: pots[2];
*wz=(pots[1] > 0xff)? 0xff: pots[1];

*message=resolve_message (pots[8]);

#elfiF(CURRENT_STICK==KRAFT_STICK)
union REGS input, output;

input.x.dx=1;        /*getting the pot values*/
input.h.ah=0x84;
int86(0x015, &input, &output);
*wz=(output.x.ax > 0xff)? 0xff: output.x.ax;
*wz=(output.x.bx > 0xff)? 0xff: output.x.bx;
*wz=(output.x.dx > 0xff)? 0xff: output.x.dx;

input.x.dx=0;        /*getting the button status*/
input.h.ah=0x84;
int86(0x015, &input, &output);
if((output.x.ax & 0xff)==0x00f0)
    *message=ALL_OK;
else
    *message=STOP_ALL;

if__(*wz==0) & (*wy==0) & (*wz==0))
{
    printf("\nread_3d_ply_stick: kraft joystick did not respond\n");
    exit(0);
}
#endif

#elfiF(CURRENT_STICK==MICRO_STICK)
union REGS input, output;

input.x.dx=1;        /*getting the pot values*/
input.h.ah=0x84;
int86(0x015, &input, &output);
*wz=(output.x.ax > 0xff)? 0xff: output.x.ax;
*wy=(output.x.bx > 0xff)? 0xff: output.x.bx;
*wz=(output.x.dx > 0xff)? 0xff: output.x.dx;

input.x.dx=0;  /*getting the button status*/
input.y.sh=0x84;
int8&(&input, &output);
if((output.x.ax & 0xff)==0x00f0)
  *message=ALL_OK;
else
  *message=STOP_ALL;

if((*wx==0) && (*wy==0) && (*wz==0))
{
  printf("\nread_3d_rpy_stick: onboad micro measurements stick did not respond\n");
  exit(0);
}

#if(CURRENT_STICK==CURSOR_STICK)
static int x=128, y=128, z=128;

if(kbhit())
  switch(getkey())
  {
    case PGUP:
      z+=10 ;break;
    case HOME:
      z=-10 ;break;
    case CURUP:
      y+=10 ;break;
    case CURDN:
      y=-10 ;break;
    case CURRT:
      x+=10 ;break;
    case CURRL:
      x-=10 ;break;
    default:
      x=y=z=128; *message=STOP_ALL; break;
  }

  *wx=x, *wy=y, *wz=z;
#endif

/*
init_stick();
This routine checks for the presence of the currently defined
handcontroller(CURRENT_STICK)and initializes it if necessary
*/
void init_stick()
{
#if(CURRENT_STICK==NASA CHAIR)
unsigned mode=ASINOUT|BINARY|NORMALRX ;
int tmp, status;

printf("init_stick: initializing serial port\n");
if(status=aslopen(NASA CHAIR PORT, mode, BUF_LEN, BUF_LEN, 9600, P_NONE,
  STOP_BIT, WORD_LEN, OFF, OFF)!=ASSUCCESS)
  {
    printf("\ninit_stick: COM%d port open failed, return
            code=%d\n", (NASA CHAIR _PORT+1), status);
    exit(0);
  }
}
*/


```c
printf("\ninit\n"");
if(gettxcnt(NASA_CHAIR_PORT)<9)
{
    printf("\ninit\n": NASA chair did not respond, check power and cable\n\nexit(\status);\n}"
    
for(tmp=0; tmp<9; tmp++)
    asigetc(NASA_CHAIR_PORT);
    asigetc(NASA_CHAIR_PORT, 'n');
    /*prime the chair*/

#if(CURRENT_STICK==KRAFT_STICK)
    int t1, t2, t3, t4;
    read_3d_rpy(t1, &t2, &t3, &t4);
    if((t1==0)&&(t2==0)&&(t3==0))
    { 
        printf("\ninit\n": kraft joystick did not respond\n\nexit(0);
    }
#endif

#if(CURRENT_STICK==MICRO_STICK)
    int t1, t2, t3, t4;
    read_3d_rpy(t1, &t2, &t3, &t4);
    if((t1==0)&&(t2==0)&&(t3==0))
    { 
        printf("\ninit\n": onboard microphone stick did not respond\n\nexit(0);
    }
#endif

#if(CURRENT_STICK==CURSOR_STICK)
    int a;
    a=0; /* just a nop until code is placed here for checking */
    /* for the presence of a keyboard.*/
    #else
    printf("\n#define CURRENT_STICK=?\n\ninit\n": Illegal Handcontroller specification\n\nexit(0);
    #endif
}

/* resolve_message()*/
The message generated by a certain hand controller (Currently only
the NASA Chair is converted into one of our generic messages as
defined in motor.h */
int resolve_message(int msg)
{
    switch(msg)
    {
    case TR1:
        return(TRIGGER1);
    case TR2:
        return(TRIGGER2);
    case PBTN1:
        return(PANIC_ALL);
    case PBTN2:
        return(PANIC_ALL);
    case PBTN3:
        return(TOGGLEUP);
    case PBTN4:
        return(TOGLLEDN);
```
case TOGUP:
    return (TOGGLEUP);
case TOGDN:
    return (TOGGLEDN);
default:
    return (ALL_OK);
}
#include "..\..\base\work\motor.h"

/* Support function that can not be otherwise classified are placed in this file. */
/* msleep():
   A call to this function results in the program sleeping for that many milliseconds. 
The granularity of the sleep timer is constrained by the system timer and is 
currently 18.2 Hz, or about 58 milliseconds. */
void msleep(int ticks)
{
    clock_t entry_tick;

    entry_tick=clock()+ticks;
    while(entry_tick+ticks > clock());
}
stick.c

#include "motor.h"

/* Only function that are related to the handcontroller are present in this file. Currently the NASA chair, Kraft joystick the micro measurements 3 dof stick and the cursor pad on the keyboard can be used as a joystick. The Kraft stick will not work on the leading edge or the ampro, because of lack of rom support for the game controller */

/* read_6d_stick():
This function reads the value from the 6 d.o.f. hand controller returns it to the calling program.
A message is also returned */
#include <stdio.h>
#include <unistd.h>

void read_6d_stick(int *x, int *y, int *z, int *wx, int *wy, int *wz, int *slide, int *dial, int *message)
{
    #ifdef CURRENT_STICK==NASA_CHAIR
    int tmp, pots[9];
    #ifdef DEBUG12
    printf("read_6d_stick: entered\n");
    #endif
    
    #ifdef DEBUG4
    msleep(1);
    #endif
    
    if(getrxcnt (NASA_CHAIR_PORT) !=9)
    msleep(1);
    if(getrxcnt (NASA_CHAIR_PORT) !=9)
    {
        printf("\nread_6d_stick: NASA chair communication Failure, check power and cable\n");
        #ifdef DEBUG4
        printf("\ndebug output: %d characters received\n",getrxcnt (NASA_CHAIR_PORT));
        #endif
        exit(0);
    }
    
    for(tmp=0; tmp<9; tmp++)
    pots[tmp]=asiggetc (NASA_CHAIR_PORT);
    asigputc (NASA_CHAIR_PORT, '\0');

    *x=(pots[3] > 0xff)? 0xff: pots[3];
    *y=(pots[5] > 0xff)? 0xff: pots[5];
    *z=(pots[4] > 0xff)? 0xff: pots[4];
    *wx=(pots[2] > 0xff)? 0xff: pots[2];
    *wy=(pots[0] > 0xff)? 0xff: pots[0];
    *wz=(pots[1] > 0xff)? 0xff: pots[1];
    *slide=(pots[6] > 0xff)? 0xff: pots[6];
    *dial=(pots[7] > 0xff)? 0xff: pots[7];
    *message=resolve_message(pots[8]);
    
    #elif(CURRENT_STICK==KRAFT_STICK)
    union REGS input, output;

    input.x.dx=1;  /*getting the pot values*/
    input.h.ah=0x84;
    int86(0x15, &input, &output);
    *x=*y=*z=*slide=*dial=KRAFT_STICK_ZERO_X;
    *wx=(output.x.ax > 0xff)? 0xff: output.x.ax;
*wy=(output.x.bx > 0xff)? 0xff: output.x.bx;
*wz=(output.x.dx > 0xff)? 0xff: output.x.dx;

input.x.dx=0;  /* getting the button status */
input.y=toxF4;
int86(0x15, &input, &output);
if((output.x.ax & 0xff)==0x00f0)
  *message=ALL_OK;
else
  *message=STOP_ALL;

if(*wy==0 && *wy==0 && *wz==0)
{
  printf("\nThe 64 stick: Kraft joystick did not respond\n");
  exit(0);
}

#else if(CURRENT_STICK==CURSOR_STICK)
static int xx=128, yy=128, zz=128, pr=128;
*message=ALL_OK;
if(kbhit())
  switch(getkey())
  {
    case PGUP:
      zz++10; break;
    case HOME:
      zz-=10; break;
    case CURUP:
      yy++10; break;
    case CURON:
      yy-=10; break;
    case CURRT:
      xx++10; break;
    case CURLF:
      xx-=10; break;
    case F12:
      pr+=5; break;
    case ENDCALL:
      pr-=5; break;
    case PGDN:
      pr+=5; break;
    default:
      xx=yy=zz=pr=128; *message=STOP_ALL; break;
  }

*wx=*wy=*wz=*dial=CURSOR_STICK_ZERO_X;
*x=xx, *y=yy, *z=zz, *slide=pr;

#else
  printf("\n\n#define CURRENT_STICK=?\n");
  printf("\ninit_stick: Illegal Handcontroller specification\n");
  exit(0);
#endif

#define DEBUG12
#endif

printf("\n\nread_6d_stick: complete\n");
#endif

/* read 3d rpy stick():
 * This function reads the value from the current 3d joystick and
 * returns it to the calling program. A message is also returned */

void read_3d_rpy_stick(int *wx, int *wy, int *wz, int *message)
{
  #ifdef CURRENT_STICK==NASA_CHAIR
    int tmp;

static missed_count=0;
static int pots[9] = {
    NASA_CHAIR_ZERO_X,
    NASA_CHAIR_ZERO_Y,
    NASA_CHAIR_ZERO_Z,
    NASA_CHAIR_zero_Wx,
    NASA_CHAIR_ZERO_Wy,
    NASA_CHAIR_ZERO_Wz,
    NASA_CHAIR_ZERO_SLIDE,
    NASA_CHAIR_ZERO_DIAL,
    NASA_CHAIR_ZERO_BUTTONS
};

#if define DEBUG4
msleep(1);
#endif

if((getrxcnt(NASA_CHAIR_PORT) != 9))
{
    missed_count++;
    if(missed_count > MAX_MISSED_RESPONSE)
    {
        printf("\nread_3d_rpy_stick: NASA chair communication Failure, check power and cable\n");
#if define DEBUG4
        printf("\ndebug output: %d characters received\n", getrxcnt(NASA_CHAIR_PORT));
#endif
        exit(0);
    }
}
else
{
    missed_count = 0;
    for(tmp = 0; tmp < 9; tmp++)
        pots[tmp] = asigetc(NASA_CHAIR_PORT);
    asputc(NASA_CHAIR_PORT, '\0');
}

#endif

#define CURRENT_STICK==KRAFT_STICK
union RS8 input, output;

input.x.dx=1;    /* getting the pot values*/
input.h.ah=0x84;
int86(0x15, &input, &output);
*xw=(output.x.ax > 0xff) ? 0xff : output.x.ax;
*wy=(output.x.bx > 0xff) ? 0xff : output.x.bx;
*wz=(output.x.dx > 0xff) ? 0xff : output.x.dx;

input.x.dx=0;    /* getting the button status*/
input.h.ah=0x84;
int86(0x15, &input, &output);
if((output.x.ax & 0xff)==0x00f0)
    *message=ALL_OK;
else
    *message=STOP_ALL;

if(strlen(wx) == 0 && (*wy == 0) && (*wz == 0))
{
    printf("\nread_3d_rpy_stick: kraft joystick did not respond\n");
}
exit(0);
}

#else if(CURRENT_STICK==MICRO_STICK)
union REGS input, output;

input.x.xx=1;  /*getting the pot values*/
input.h.ah=0x84;
int86(0x15, &input, &output);
*wx=(output.x.ax > 0xff)? 0xff: output.x.ax;
*wy=(output.x.bx > 0xff)? 0xff: output.x.bx;
*wz=(output.x.dx > 0xff)? 0xff: output.x.dx;

input.x.xx=0;  /*getting the button status*/
input.h.ah=0x84;
int86(0x15, &input, &output);
if((output.x.ax & 0xff)==0x00ff00)
  *message=ALL_OK;
else
  *message=STOP_ALL;

if((!*wx==0) & (*wy==0) & (*wz==0))
  {
printf("\nread_3d_xpy_stick: onboard micro measurements stick did not respond\n");
exit(0);
  }
#endif

static int x=128, y=128, z=128;

if(kbhit())
  switch(getkey())
  {
    case PGUP:
      z+=5 ;break;
    case HOME:
      z-=5 ;break;
    case CURUP:
      y+=10 ;break;
    case CURDN:
      y-=10 ;break;
    case CURRT:
      x+=3 ;break;
    case CURLF:
      x-=3 ;break;
    default:
      x=y=z=128; break;
  }
*wx=x, *wy=y, *wz=z;

#else
printf("\n#define CURRENT_STICK=?\n");
printf("init_stick: Illegal Handcontroller specification\n");
exit(0);
#endif

/*
init_stick():
This routine checks for the presence of the currently defined
handcontroller(CURRENT_STICK) and initializes it if necessary */

void init_stick()
{
#if(CURRENT_STICK==NASA_CHAIR)
unsigned mode=ASINOUT|BINARY|NORMALRX;
int tmp, status;

printf("init_stick: initializing comm port\n");
if((status=asopen(NASA_CHAIR_PORT, mode, BUF_LEN, BUF_LEN, 9600, P_NONE,
STOP_BIT, WORD_LEN, GET, OFF))!=ASSUCCESS)
{
    printf("init_stick: COM\d port open failed, return
code: %d\n", (NASA_CHAIR_PORT+1), status);
    exit(0);
}

printf("init_stick: testing NASA chair\n");
asputc(NASA_CHAIR_PORT, '\0');
msleep(500); /*give the chair a chance to respond*/
if(getrxcnt(NASA_CHAIR_PORT)<9)
{
    printf("init_stick: NASA chair did not respond, check power and cable\n");
    exit(status);
}
for(tmp=0; tmp<9; tmp++)
asgetc(NASA_CHAIR_PORT);
asputc(NASA_CHAIR_PORT, '\0'); /*prime the chair*/
#endif

#elif(CURRENT_STICK==KRAFT_STICK)
int t1, t2, t3, t4;
read_3d_rpy_stick(&t1, &t2, &t3, &t4);
if((t1==0) && (t2==0) && (t3==0))
{
    printf("init_stick: kraft joystick did not respond\n");
    exit(0);
}

#elif(CURRENT_STICK==MICRO_STICK)
int t1, t2, t3, t4;
read_3d_rpy_stick(&t1, &t2, &t3, &t4);
if((t1==0) && (t2==0) && (t3==0))
{
    printf("init_stick: onboard micro_measurements stick did not respond\n");
    exit(0);
}

#elif(CURRENT_STICK==CURSOR_STICK)
int a;
a=0; /* just a nop until code is placed here for checking */
    /* for the presence of a keyboard.*/
#else
printf("\n#define CURRENT_STICK=?\n");
printf("init_stick: Illegal Handcontroller spec\txtion\n");
exit(0);
#endif

resolve_message();
The message generated by a certain hand controller (Currently only
the NASA Chair is converted into one of our generic messages as
defined in motor.h
*/
int resolve_message(int msg)
{
    switch (msg)
    {
case TR1:
    return (TRIGGER1);

case TR2:
    return (TRIGGER2);

case PBIN1:
    return (STOP_ALL);

case PBIN2:
    return (PUSHBIN2);

case PBIN3:
    return (PUSHBIN3);

case PBIN4:
    return (PUSHBIN4);

default:
    return (ALL_OK);
#include "motor.h"

/* Control the omnibase using a stick that is installed locally. The
stick may be the NASA handcontroller chair, kraft stick or the cursor pad*/
void main(void)
{
    base_version();
    init_stick();
    install_panic();
    init_vel_mode_omnibase(MAX_SAFE_WHEEL_ACCELERATION);
    parametric_omnibase((float)60.0, local_3d_rpy_control);
    exit(0);
}

void shutdown()
{
    printf("shutdown: ....\n");
    set_int_vel_omnibase(0, 0, 0);
    hardreset(GAIN, POLE, ZERO, SAMPLE_TIME);
}

float param_x(float t)
{
    return((float)-25.0 * t);
}

float param_y(float t)
{
    return((float)0.0 * t);
}

float param_wz(float t)
{
    return((float)0.0 * t);
}
support.c

#include "motor.h"

/*sleep():
A call to this function results in the program sleeping
for that many milliseconds. The granularity of the sleep
timer is constrained by the system timer and is currently
18.2 Hz, or about 58 milliseconds.
*/
void sleep(int ticks)
{
    clock_t entry_tick;

    entry_tick=clock()+ticks;
    while(entry_tick+ticks > clock());
}

/*
msleep();
same as above except with a different name to avoid conflict with
the sockets library definition
*/
void msleep(int ticks)
{
    clock_t entry_tick;

    entry_tick=clock()+ticks;
    while(entry_tick+ticks > clock());
}
#include "motor.h"

/*
This is the transmission end of the remote-comm control program. It
reads the handcontroller and pumps out the positions to the comm ports
*/

void main(void)
{
  int x, y, z, zx=0, zy=0, zz=0, wx, wy, wz, mesg, port, reply;

  base_version();
  init_comm(PORT_BASE);
  init_stick();
  install_panic();

  port=PORT_BASE;
  x=0, y=0, z=0, wx=0, wy=0, wz=0, mesg=ALL_OK;
  sleep(60); /* to allow the nasa chair to have a chance to reply*/
  reply=ALL_OK; /* for the first iteration fake reply to be OK*/

  while(reply != STOP_ALL)
    {
      read_3d_rpy_stick(&wx, &wy, &wz, &mesg);
      wx=wx-zx;
      wy=wy-zy;
      wz=wz-zz;
      reply=send_comm_packet(x, y, z, wx, wy, wz, mesg, port);

      #ifdef DEBUG6
      printf("Hand Controller=> %d\t%d\t%d $\n", wx, wy, wz);
      #endif
    }
  exit(0);
}

void shutdown()
{

}
velocity.c

#include "motor.h"

void main(void)
{
    struct tm *curtime;
    time_t bintime;
    int velocity;

    install_panic();
    init_vel_mode_omniBase(150);
    
    time(&bintime);
    curtime=localtime(&bintime);
    printf("\ntime is %s\n", asctime(curtime));
    set_int_vel(5, 0, 0);

    while(TRUE)
    {
        scanf("%d", &velocity);
        if(velocity>10)
            break;
        set_int_vel(5, velocity, 0);
        while(!kbhit())
            printf("g%d\t"%d\t"%d\r", hp_read(1, 9), hp_read(1, 52),
            hp_read(1, 60));
        printf("\n");
    }
    set_int_vel(0, 0, 0);
}

void shutdown()
{
    printf("shutdown: ......\n");
    set_int_vel_omniBase(0, 0, 0);
    hardreset(GAIN, POLE, ZERO, SAMPLE_TIME);
}
version.c

#include "motor.h"

/*
The following two routines
just display the software
version number. Which keeps
changing rather rapidly
*/

/*
base_version()
This prints the version number for
the base software
*/
void base_version(void)
{
printf("\nOmniDirectional Robot Base Controller\nRevision %4.2f: Compiled %s\n",
VERSION_BASE, __TIMESTAMP__);
printf("Samad Adnan, Dept M.E.M.S. Rice University\n\n");
}

/*
arm_version()
This prints the version number for
the arm software
*/
void arm_version(void)
{
printf("\nRedundant Manipulator Arm Controller\nRevision %4.2f: Compiled %s\n",
VERSION_ARM, __TIMESTAMP__);
printf("Samad Adnan, Dept M.E.M.S. Rice University\n\n");
}

/*
ethernet_version()
This prints the version number for
the arm software
*/
void ethernet_version(void)
{
printf("\nEthernet Communication Program\nRevision %4.2f: Compiled %s\n",
VERSION_ARM, __TIMESTAMP__);
printf("Samad Adnan, Dept M.E.M.S. Rice University\n\n");
}