[34] T. Tarn, A. Bejczy, and X. Yuan. Control of Two Coordinated Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1193–1202, 1986.

[35] J.C. Trinkle, J.M. Abel, and R.P. Paul. Enveloping, Frictionless, Planar Grasping. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 246–251, March-April 1987. Raleigh, North Carolina.

[36] VMEbus International Trade Association. *The VMEbus Specification*, Scottsdale, AZ 1987.

[37] Wind River Systems. *VxWorks Programmer's Guide*, 1990.

[38] Paul Wright and David Bourne. *Manufacturing Intelligence.* Addison-Wesley, Boston, MA, 1988.

[16] D. KIRKPATRICK, B. MISHRA, AND C. YAP. Quantitative Steinitz's Theorem with Applications to Multifingered Grasping. In *Proceedings of the Twenty-Second ACM Symposium on Theory of Computing*, pages 341–351, Baltimore, Maryland, May 1990. (To appear in *Discrete & Computational Geometry*, in press, 1991).

[17] D.E. KODITSCHECK. The application of total energy as a lyapunov function for mechanical control systems. *Control Theory and Multibody Systems* (To appear).

[18] D.E. KODITSCHECK AND E. RIMON. Robot navigation functions on manifolds with boundaries. *Advances in Applied Mathematics*, (To appear), 1989.

[19] Y. LAMDAN AND H. WOLFSON. Geometric Hashing: A General and Efficient Model-Based Recognition Scheme. Technical Report 152, Robotics Research Lab., Courant Institute of Mathematical Sciences, New York University, May 1988.

[20] M. LANDY, Y. COHEN, AND G. SPERLING. HIPS: A Unix Based Image Processing System. *Computer Vision, Graphics, and Image Processing*, 25(1984):331.

[21] S.T. LEVI AND A.K. AGRAWALA. *Real Time System Design.* McGraw Hill Publishing Company, 1990.

[22] T. LOZANO-PEREZ. Spatial Planning: A Configuration Space Approach. *IEEE Trans. Computers*, 1983.

[23] J.Y.S. LUH AND Y.F. ZHENG. Constrained Relations between two Coordinated Industrial Robots for Motion Control. *International Journal of Robotics Research*, 6-3, 1987.

[24] MATTHEW T. MASON AND JR. J. KENNETH SALISBURY. *Robot Hands and the Mechanics of Manipulation.* MIT Press, 1985.

[25] P.J. MCKERROW. *Introduction to Robotics.* Addison-Wesley, 1991.

[26] B. MISHRA, J.T. SCHWARTZ, AND M. SHARIR. On the Existence and Synthesis of Multifinger Positive Grips. *Algorithmica*, 2:541–558, 1987.

[27] BUD MISHRA AND NAOMI SILVER. Some discussion of static gripping and its stability. *IEEE Transactions on Systems, Man and Cybernetics*, 19:783–796, July/August 1989.

[28] D. MONTANA. *Tactile Sensing and Kinematics of Contact.* PhD thesis, Division of Applied Sciences, Harvard University, 1986.

[29] MOTOROLA, INC. *MVME147S MPU VMEmodule User's Manual*, Tempe, AZ 1989.

[30] Y. NAKAMURA, K. NAGAI, AND T. YOSHIKAWA. Mechanics of Coordinative Manipulation by Multiple Robotic Mechanisms. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 991–998, 1987.

[31] R. PAUL. *Robot Manipulators: Mathematics, Programming and Control.* MIT Press, 1981.

[32] LOU SALKIND. *SAGE: A Real-Time System for Robotic Supervisory Control.* PhD thesis, Courant Institute, New York University, 1990.

[33] R. SCHALKOFF. *Digital Image Processing and Computer Vision.* Wiley Press, 1989.

References

[1] C. An, C. Atkeson, and J. Hollerbach. *Model-Based Control of a Robot Manipulator*. MIT Press, 1988.

[2] Haruhiko Asada and Jean-Jacques E. Slotine. *Robot Analysis and Control*. Wiley-Interscience, 1986.

[3] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha and F. Wang. Quantifying the Advantage of Knowing the Future When Scheduling Sporadic Tasks. *Real Time Systems Journal*, 4(2): 125–144, 1992.

[4] A.K. Bejczy. Robot Arm Dynamics and Control. Technical report, Jet Propulsion lab. Rep. JPL TM 33-669, 1974.

[5] O. Bottema and B. Roth. *Theoretical Kinematics*. North-Holland Publishing Co., 1979.

[6] Dayton R. Clark. *Data Communication in Robot Control Systems*. PhD thesis, Courant Institute, New York University, 1989.

[7] J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley, 1989.

[8] Creonics. *VMEbus Compatible Motion Control Card User's Manual*, 1990.

[9] Mark R. Cutkosky. *Robotic Grasping and Fine Manipulation*. Kluwer Academic Publishers, Massachusetts, 1985.

[10] Mark R. Cutkosky and Paul K. Wright. Modeling Manufacturing Grips and Correlation with the Design of Robotic Hands. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1533–1539, 1986.

[11] I. Greenfeld, F. Hansen, J.Fehlinger and E. Pavlakos. *MOSAIC, System Description, Specification and Planning*, Technical Report No. 452, Robotics and Manufacturing Laboratory, Courant Institute, New York University, New York, 1989.

[12] J. Hong, B. Mishra and X. Tan. Techniques for Calibration with Applications to Telemanipulation. In *Progress in Robotics and Intelligent Systems*, Vol. **5**, (Edited by C.Y. (Pete) Ho and George W. Zobrist), Ablex Publishing Corporation, New Jersey, 1993.

[13] Robert A. Hummel. Representations Based on Zero-Crossing in Scale-Space. In *Proceedings of the IEEE Computer Vision and Pattern Recognition Conference*, pages 204–209, 1986.

[14] Maw kae Hor. *Control and Task Planning for a Four Finger Dextrous Manipulator*. PhD thesis, Robotics Research Lab., Courant Institute of Mathematical Sciences, New York University, 1987.

[15] J. Kerr and B. Roth. Analysis of Multifingered Hands. *International Journal of Robotics Research*, 4(4):3–17, 1986.

- **Experiment 1. Modeling of a D.C. Torque Motor**.
  Students are asked to obtain a model of the D.C. torque motor which drives the last joint of the ED I arm, simulate the transient response of the model and finally, compare the results with the experimental results. Finally, they evaluate the accuracy of the model.

- **Experiment 2. Controller Design**.
  Students are asked to design a proportional (P), proportional-and-derivative (PD) and lead-lag compensator for the D.C. torque motor model. Next, they evaluate the performance of the compensator using a step response.

- **Experiment 3. Modeling and Control of Inverted Pendulum**.
  Students are asked to connect an inverted pendulum to the tip of the ED I arm, obtain a nonlinear system description of the system and then, identify the equilibrium states. Using a nonlinear control law, they stabilize the system in the up-right position.

- **Experiment 4. Adaptive Control of the ED I Robot with Unknown Load**.
  Students are asked to derive the nonlinear dynamic model of the ED I robot arm assuming that the load at the distal point is unknown. Model-based linearization and parameter estimation techniques are to be applied to compute the inertial load. Adaptive techniques are to be used to achieve high-fidelity control[1].

B.8. Undergraduate Thesis Projects

This course is intended to provide senior students the opportunities of pursuing advanced studies in Robotics, AI, CAM and Vision. Under the direct supervision of a faculty member, the students first conduct a literature survey in the area of their interest. Then, they choose a more focused problem, propose a feasible approach and implement the approach on a testbed consisting of ED I robot, a vision sysytem and several manufacturing tools. A number of possible topics for thesis projects include, but are not limited to, the term projects outlined in the set of courses listed in the earlier subsections.

B.6. Artificial Intelligence and Learning in the Robotic Setting

This is a junior/senior level course concerning a variety of topics in artificial intelligence, which bridges the theory, artificial intelligence and robotics areas. The topics to be covered include machine learning from examples, neural networks, and applications to vision, robotics and manufacturing. The course aims to allow students who have a solid mathematical and theoretical foundations with ways of applying such theoretical principles to robotics-related problems in artificial intelligence. This course includes the following machine learning experiments to be undertaken in the robotics educational laboratory.

- **Experiment 1. Learning on Disc Throw, I**.
  A round disc is put in a holder fixed on the end of the robot arm, which is initially positioned at a fixed place. The students are asked to design a series of continuous movements of the robot arm with adjustable parameters so that the arm will throw the disc to a position as close as possible to a bar, but without touching that bar. Then the student should adjust the parameters to accomplish the job through experiments.

  The student should also design a learning procedure for the machine so that the machine can determine these parameters automatically through experiments. The result of an experiment, which is input to the learning algorithm, can be a real number representing the error.

- **Experiment 2. Learning on Disc Throw, II**.
  The students are asked to develop an automatic learning procedure for the case that the initial position is an arbitrary one. The initial position is input to the computer. They are also asked to design an automatic learning procedure to play a match with a human player so that the winning probability of the robot arm is maximized in the asymptotic limit.

- **Experiment 3. Monitoring a Moving Object**.
  A stick is fixed upwards on the end of the robot arm. A bright sign is fixed on the top of the stick and monitored by a camera. A person holds another sign which is also monitored by the camera. The students are asked to develop a fast algorithm to determine the positions of these two signs and to control the motion of the robot arm so that the sign on the stick always follows the other sign which is randomly moved by the person.

- **Experiment 4. Stick Balancing**.
  A stick is connected to the end of the ED I robot arm by a linkage which allows it to fall down along two opposite directions. A bright sign is fixed to the top of the stick and monitored by a camera. The students are asked to design a neural network program to control the robot arm so that the stick will not fall. The student should also design a program to balance the stick so that the stick always follows a bright sign which is randomly moved by a person.

B.7. Linear and Nonlinear Systems

This course is offered to junior/senior undergraduate and/or first year graduate students from the mathematics and the computer science departments, and makes extensive use of the robotics educational laboratory to teach basic principles and related design techniques in linear and nonlinear systems.

The syllabus includes: modeling of linear time-invariant(LTI) systems, linear differential equations, Laplace transform techniques, transfer function description of input-output system, feedback control and compensator design; transient response analysis; modeling and description of nonlinear dynamic systems, differential equations, critical points and Lyapunov stability analysis; control techniques for nonlinear systems.

B.5. Computer Vision

The proposed course is for juniors or seniors, and covers the fundamentals of computer vision and image processing. The course covers standard topics in image sensing, processing, and analysis, stressing algorithmic methods, as is typically covered in an introductory image processing and computer vision class. Recent texts (e.g. Schalkoff's book [33]) make it possible to cover topics in an accessible and rapid fashion, affording the students the option of implementing basic techniques in a suitably equipped laboratory. The course covers image formation and color theory, feature extraction, image enhancement, some signal processing theory, image segmentation, multiresolution image analysis, motion, stereo, pattern recognition and labeling, and model-based vision. Lectures and readings are used as guides to the projects and experiments that students will conduct on hardware and software tools integrated with the usual computing facilities available.

There will be four assigned experiments to supplement the first eight weeks of the course, and then an assigned term project chosen from a list of proposed projects, envisioned to encompass the last six weeks of the course. The proposed projects are as follows:

- **Experiment 1. Software Introduction**.
  The students are asked to demonstrate capability of acquiring digital imagery, running edge detectors and feature detectors, running convolutions and simple nonlinear enhancement processes, and programming in high-level vision application software. Supplied software enables interactive execution of individual image operations, so that only embodying scripts must be supplied by the students. These scripts, written in C, are used to build up a library of routines.

- **Experiment 2. Image Restoration and Reconstruction**
  Students are supplied with several images that have been subjected to degradation through manipulation and noise. The task is to undo the degradations, recording the steps. Examples of degradations that the students will have to contend with is blurring, grayscale changes, image rotation, line drop-outs, additive noise, and scanline variations.

- **Experiment 3. Multiresolution image analysis**.
  In this experiment, students are asked to perform an image analysis task, such as edge or feature extraction, or simple model matching, using a multiresolution image representation. The students develop scripts to perform these functions and they have to contend with issues related to reasoning about the choice of scale and combination of information from multiple scales.

- **Experiment 4. Stereo matching**.
  Stereo pairs are provided, and students are asked to implement any of a number of stereo matching algorithms in order to obtain depth estimates to object points visible in both images. Candidate algorithms include iterative disparity analysis, the iterative Marr/Poggio algorithm, zero-crossing feature matching, branch-and-bound search for feature matching, and Cepstral filter matching. Ground truth of actual object depths are provided.

- **Term project.** Each student is expected to implement a substantial image analysis system in the software environment provided, using a research paper or other published description as a guide. Example projects include image segmentation methods based on color and/or texture, motion extraction from image sequences, texture analysis or depth-from-texture studies, generalized Hough transformation studies, geometric hashing for object recognition, or comparative feature analysis.

- **Experiments 5 & 6. Sensor Based Manufacturing.**
  As a final development of experiments 5 and 6, a rudimentary two-dimensional vision system is used to inspect and measure the components that the students have manufactured and ground smooth. Students can acquire appreciation of different lighting methods for computer vision and manufacturing, the available technologies in terms of CCD cameras and lenses, and they, thus, gain experience in the general set-up of industrial sensors. As the laboratory is developed, it may be possible to use the vision information as a feedback to the system and to make corrections in the grinding part and in the routing trajectories. The overall goal of this last experiment is to make the students aware of quality control methods in industry.

B.4. Real Time Systems

This is a junior/senior level introductory course, covering the principles, implementation and applications of a real-time system. The topics covered in the course include: hard-real time constraints; periodic and aperiodic tasks; operating system issues: scheduling, communication, critical sections, priority inversion problems; real time system modeling: petri-nets, temporal logic, discrete event systems; real-time operating systems and languages; applications to robotics (with examples such as controlling a pin-ball machine or a juggling robot). The text by Levi and Agrawala covers most of these topics [21]. Prerequisites are basic robotics, introductory operating systems and moderate programming experience.

The following set of four experiments are to be conducted, with each experiment taking roughly three weeks. Additional term projects may be assigned, and will usually involve building real time controllers for an integrated robot system (e.g. a pin-ball machine, four-finger manipulator, the Utah/MIT hand or the NYU manufacturing system).

- **Experiment 1. Cyclic Executive**.
  The students are asked to implement a simple cyclic executive system in C programming language, capable of handling the proportional (P) and proportional-and-derivative (PD) control schemes for the ED I robot.

- **Experiment 2. Periodic Tasks**.
  In this experiment the objective is to model the sensor, actuator and data-logger servo tasks for the ED I robot controller as simple periodic tasks with different periods. The students are asked to implement the controller, by using a simple rate monotonic scheduler and then evaluate the performance as the periods of the tasks are uniformly scaled.

- **Experiment 3. Sporadic Tasks**.
  The students are asked to model the sensor, actuator and data-logger tasks for the ED I robot controller as aperiodic tasks with given minimum separation times and hard and soft real-time constraints. Additionally, each task may be given priorities, based on certain value-functions associated with the tasks. Next, they are asked to implement the controller, by using a variety of scheduling schemes, e.g. earliest-deadline-first, least-laxity-first, etc. and then evaluate the performance under different scheduling schemes and as the system is made overloaded [3].

- **Experiment 4. Hierarchical Controller**.
  In this experiment, first the system of two robots working cooperatively is decomposed into several levels of a hierarchical system. The students are asked to implement the lowest servo-level as a system of *periodic* tasks, and the higher levels as systems of *aperiodic* tasks. The communications between the successive levels are to be implemented via double-buffers. Finally, the students evaluate the performance of the system.

algorithms for a two-fingered robotic hand, (2) motion planning for a two-fingered robotic hand, (3) hand-eye coordination, and (4) simulation studies of legged locomotion robots or space robots.

B.3. Computer-Aided Manufacturing, CAM

This is a junior/senior level course aimed at familiarizing computer science students with future manufacturing technologies. The topics covered in the laboratory are: computer-aided design, simulation, control, sensors, and quality control. The course is designed to emphasize the computer science skills of the undergraduate students with the engineering components being taken care of by the laboratory technicians.

- **Experiment 1. Computer-Aided Design Methodologies.**
  Students are asked to design a component of their choice on a computer-aided design system (e.g. the Anvil 5000 software package). Using this system, students can create their components in wire frame images, with boundary representations, and, to a limited extent, using constructive solid geometry. This experiment will familiarize the students with basic design methods, the operation of computer-aided design systems, and, simultaneously, with standard graphics interface languages (e.g. IGES and PDES).

- **Experiment 2. Manufacturing Simulations and Tool Path Generations.**
  In the second experiment, the students are asked to generate a software simulation of a simple manufacturing process, building on their computer-aided design work. In the process, they become familiar with technology that creates tool path simulations on their design. They are then introduced to a simple expert system, running in the local CAD/CAM environment that advises on the usable set-up procedures and operating instructions. Via simulation, they can then see how their particular part can be made on a full scale CNC machining center and turn into the component ready for use in steel or aluminum.

- **Experiment 3. Simple Manufacturing Technologies.**
  Using the ED I robot arm, the students are asked to create plastic hard copies of the components that they have designed and simulated in Experiments 1 and 2. In the laboratory, the students use prototyping wax as the component and limit their operations to two-dimensional routing on the wax. They are then able to program the ED I robot arm to carry out the two-dimensional cutting operations and create the outer profile of their desired part. In addition, other small cutting tools are supplied as an ancillary to the robot and the students can drill holes and carve lettering onto their components. In this way, they become experienced in the complete CAD/CAM loop even though these experiments are only carried out in simplified shapes and with the prototype wax.

- **Experiment 4. Force Position Control.**
  In this experiment students are made familiar with a simple grinding operation as an example of carrying out manufacturing operations that require both force and position control. For this experiment, the ED I robot is equipped with small finish grinding tools and the students are asked to smooth an arbitrarily rough contour. They are required to program the robot to the coarse profile of the object and then grind the surface smooth. Thus, they demonstrate position control along the direction of the profile and force control normal to the profile in order that the grinding tool presses against the surface with uniform load. This experiment overlaps with the Experiment 5 of *Basic Robotics*, but extends the students' understanding of the importance of such force control in manufacturing processes.

force terms for three ranges of joint velocity (0 to $\frac{1}{3}$ of maximum joint velocity, $\frac{1}{3}$ to $\frac{2}{3}$ of maximum joint velocity and $\frac{2}{3}$ to 1 of maximum joint velocity). (2) Next, they design, implement and compare position control algorithms, using following two approaches: *joint-space control* and *task space control*.

- **Experiment 5. Interfacing with the External World: Force Control**.
  The main objectives are to study (1) force control with fixed contact and (2) hybrid position/force control. First, the students design and implement a simple feedback control algorithm that tracks a desired force trajectory in the normal direction, while stabilizing a fixed point of contact between the tip of the robot and a fixed surface. Next, they design and implement position/force control algorithms, where the tip may be allowed to move on some surface of constraint.

- **Term Project.** After consulting with the instructor, the students may choose a topic for their term project. They are asked to read one or two research papers in the area of their choice. Possible topics include (1) contact motion and force control, (2) coordinated manipulation by two robot arms, (3) dynamics and simulation of a one legged hopping robot using simulation and graphics tools, (4) motion-planning with obstacle avoidance and (5) planning and execution of a simple task.

B.2. Intermediate Robotics

This is a senior/first year graduate level course covering concepts and methods underlying current research in robotics: geometric approaches to manipulator kinematics, contact kinematics and grasp kinematics of a robotic hand system; dynamic models of, and linear and nonlinear control techniques for, robotic hands, space robots and legged locomotion robots; algorithmic approaches to motion planning for robotic manipulators and mobile robots, subject to holonomic and nonholonomic constraints. ([15,24,26,27,28]). Accompanying the lectures are four laboratory experiments designed to aid the students in mastering the principles and techniques.

- **Experiment 1. Kinematic and Dynamic Modeling of a Two-Robot Manipulation System**.
  Two robots (in our case, ED I and an IBM SCARA robot) are integrated to form a *two-fingered robotic hand system* which performs a coordinated manipulation task, such as moving a box on a circular trajectory. Using position, velocity and force sensors, students are asked to determine the kinematics of the *two-fingered robotic hand*, including the contact models, hand Jacobian, grasp map ([9,15,24,35]) and the kinematics of contact ([28]).

- **Experiment 2. Coordinated Manipulation.**
  Using the hand kinematics of the previous experiment, students are asked to implement and evaluate the dynamic coordinated control algorithms ([30]), the master-slave control ([23,34]) and the PD type control algorithms ([2,17]).

- **Experiment 3. Path Planning with Obstacle Avoidance.**
  Students are asked to implement the Schwartz-Sharir motion planning algorithm on the 3-DOF planar arm amidst obstacles. Also see [22].

- **Term Project.** The term project consists of a well defined research problem. After consulting with the instructor, the students are asked to choose a topic, analyze the problem and then propose a possible solution. Finally, they proceed to experimentally test their solution and give a presentation of their results. The list of topics include: (1) robust coordinated control

Appendix: B
A Proposed Course Sequence

B.1. Basic Robotics

This is a junior/senior level introductory course, covering the fundamental concepts of robotics and aimed at students from computer science, engineering and mathematics departments. The course develops analytic models of robot mechanisms, actuators and the robot's operating environment. It emphasizes foundational concepts, involving the geometry and algebra of rigid motions, forward and inverse kinematics of mechanical linkage systems, dynamics of actuators and linkages, task and trajectory planning as well as the algorithms for robot command and control. ([2,4,5,7,22,24,31]). Prerequisites are introductory calculus, basic linear algebra and moderate experience with high level programming languages.

The following is a suggested list of experiments for this course; each takes about two weeks. (Also see §4).

- **Experiment 1: System Familiarization and Robot Kinematics**.
  Students are introduced to the system, consisting of the robot, its actuators, its control units, the interface and the computer system. A description of the functionality of the system (via flow-charts) is presented, followed by a discussion of the safety rules (see the box on page 34) and the conduct in the laboratory. A demonstration of the subroutines to read feedback signals and to command joint movement of the robot.

  (1) Students are asked to define the coordinate frames for the joints of ED I, to identify the Denavit-Hartenberg parameters, and using these D-H parameters, to compute forward kinematics and manipulator Jacobian. (2) Next they are asked to conduct an experiment to identify kinematic singularities by moving the robot few times, each time coming close to the singularities. Finally they are asked to experimentally verify the non-commutativity of rigid motions.

- **Experiment 2. Inverse Kinematics and Trajectory Generation**.
  (1) Students are asked to develop a software package capable of moving the object in the task-space. This involves combining inverse kinematics with some simple interpolation techniques (e.g. cubic polynominals method or straight line with parabolic blends method). Each student program is tested by writing the figure eight (∞) with a marker. (2) This is followed by several other simple path planning experiments.

- **Experiment 3. Control of Simple Mechanical Systems**.
  (1) Students are asked to derive an *exact* model of the actuator (a D.C. torque motor), located at the last joint and loaded by the last link. Simplification of the model by appropriate approximations is allowed. Students then compare the approximate model with the exact model. (2) Next, they experiment with and compare proportional (P) and proportional-and-derivative (PD) feedback controllers, designed for the position control of the last joint (e.g. the robot's response to a step input signal).

- **Experiment 4. Robot Arm Dynamics and Control**.
  (1) Students are asked to derive the dynamics of the ED I robot using a given mass distribution of the links and evaluate and compare the inertia force terms with the centrifugal and Coriolis

```
>> time2edges: t = 1.209, msec = 1209
SEND:  1  0  2 31 13  3 FC FC  4 6A 64 B9  4
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 3 -4 -4 4 106 100 -71 4
>> MCC track message setup
>> MCC setup track command: X<-3.59, 1.27>, Y<14.12, 1.20>
>> incr2edges: i = -3.590, e = -1021
>> time2edges: t = 1.276, msec = 1276
>> incr2edges: i = 14.120, e = 25706
>> time2edges: t = 1.209, msec = 1209
SEND:  1  0  2 31 13  3 FC FC  4 6A 64 B9  4
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 3 -4 -4 4 106 100 -71 4
>> MCC track message setup
>> MCC setup track command: X<-0.07, 0.05>, Y<1.36, .23>
>> incr2edges: i = -0.079, e = -22
>> time2edges: t = 0.056, msec = 56
>> incr2edges: i = 1.362, e = 2480
>> time2edges: t = .233, msec = 233
SEND:  1  0  2 31 13 EA FF 38  0 B0  9 E9  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 -22 -1 56 0 -80 9 -23 0
>> MCC send track commands: all commands queued
>> MCC send track commands: sending closing message.
>> MCC  Tracking Handling: message received [3]
REC: 1 0 105
>>> Track Done!

-> off 'a'
off 'a'
value = 0 = 0x0
-> logout
logo
Connection closed.
ed1@robocop 12>
```

```
>> time2edges: t = .112, msec = 112
SEND:  1  0  2 31 13 1E  0  B  0  C  3 70  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 30 0 11 0 12 3 112 0
>> MCC track message setup
>> MCC setup track command: X<8.23, .87>, Y<5.30, .80>
>> incr2edges: i = 8.230, e = 2341
>> time2edges: t = .875, msec = 875
>> incr2edges: i = 5.307, e = 9661
>> time2edges: t = .801, msec = 801
SEND:  1  0  2 31 13 25  9 6B  3 BD 25 21  3
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 37 9 107 3 -67 37 33 3
>> MCC track message setup
>> MCC setup track command: X<.56, 0.07>, Y<.36, 0.04>
>> incr2edges: i = .567, e = 161
>> time2edges: t = 0.075, msec = 75
>> incr2edges: i = .368, e = 671
>> time2edges: t = 0.047, msec = 47
SEND:  1  0  2 31 13 A1  0 4B  0 9F  2 2F  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 -95 0 75 0 -97 2 47 0
>> MCC track message setup
>> MCC setup track command: X<.28, 0.07>, Y<.48, 0.04>
>> incr2edges: i = .282, e = 80
>> time2edges: t = 0.075, msec = 75
>> incr2edges: i = .480, e = 875
>> time2edges: t = 0.047, msec = 47
SEND:  1  0  2 31 13 50  0 4B  0 6B  3 2F  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 80 0 75 0 107 3 47 0
>> MCC track message setup
>> MCC setup track command: X<1.33, .71>, Y<8.92, .78>
>> incr2edges: i = 1.332, e = 378
>> time2edges: t = .714, msec = 714
>> incr2edges: i = 8.920, e = 16238
>> time2edges: t = .785, msec = 785
SEND:  1  0  2 31 13 7A  1 CA  2 6E 3F 11  3
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 122 1 -54 2 110 63 17 3
>> MCC track message setup
>> MCC setup track command: X<0.03, 0.04>, Y<0.03, 0.00>
>> incr2edges: i = 0.032, e = 9
>> time2edges: t = 0.046, msec = 46
>> incr2edges: i = 0.036, e = 67
>> time2edges: t = 0.003, msec = 3
SEND:  1  0  2 31 13  9  0 2E  0 43  0  3  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 9 0 46 0 67 0 3 0
>> MCC track message setup
>> MCC setup track command: X<-0.07, 0.04>, Y<0.03, 0.00>
>> incr2edges: i = -0.076, e = -21
>> time2edges: t = 0.046, msec = 46
>> incr2edges: i = 0.037, e = 68
>> time2edges: t = 0.003, msec = 3
SEND:  1  0  2 31 13 EB FF 2E  0 44  0  3  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 -21 -1 46 0 68 0 3 0
>> MCC track message setup
>> MCC setup track command: X<-3.59, 1.27>, Y<14.12, 1.20>
>> incr2edges: i = -3.590, e = -1021
>> time2edges: t = 1.276, msec = 1276
>> incr2edges: i = 14.120, e = 25706
```

```
>> time2edges: t = 0.020, msec = 20
SEND:  1  0  2 31 13 27  0 14  0 91 FE 14  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 39 0 20 0 -111 -2 20 0
>> MCC track message setup
>> MCC setup track command: X<.16, 0.02>, Y<-.18, 0.02>
>> incr2edges: i = .160, e = 45
>> time2edges: t = 0.020, msec = 20
>> incr2edges: i = -.180, e = -327
>> time2edges: t = 0.020, msec = 20
SEND:  1  0  2 31 13 2D  0 14  0 B9 FE 14  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 45 0 20 0 -71 -2 20 0
>> MCC track message setup
>> MCC setup track command: X<9.19, 1.12>, Y<-9.04, 1.11>
>> incr2edges: i = 9.190, e = 2614
>> time2edges: t = 1.126, msec = 1126
>> incr2edges: i = -9.044, e = -16465
>> time2edges: t = 1.115, msec = 1115
SEND:  1  0  2 31 13 36  A 66  4 AF BF 5B  4
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 54 10 102 4 -81 -65 91 4
>> MCC track message setup
>> MCC setup track command: X<.20, 0.02>, Y<-.24, 0.03>
>> incr2edges: i = .206, e = 58
>> time2edges: t = 0.023, msec = 23
>> incr2edges: i = -.249, e = -454
>> time2edges: t = 0.034, msec = 34
SEND:  1  0  2 31 13 3A  0 17  0 3A FE 22  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 58 0 23 0 58 -2 34 0
>> MCC track message setup
>> MCC setup track command: X<.23, 0.02>, Y<-.19, 0.03>
>> incr2edges: i = .233, e = 66
>> time2edges: t = 0.023, msec = 23
>> incr2edges: i = -.190, e = -346
>> time2edges: t = 0.034, msec = 34
SEND:  1  0  2 31 13 42  0 17  0 A6 FE 22  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 66 0 23 0 -90 -2 34 0
>> MCC track message setup
>> MCC setup track command: X<13.99, 1.33>, Y<-5.68, 1.21>
>> incr2edges: i = 13.993, e = 3980
>> time2edges: t = 1.330, msec = 1330
>> incr2edges: i = -5.680, e = -10341
>> time2edges: t = 1.218, msec = 1218
SEND:  1  0  2 31 13 8C  F 32  5 9B D7 C2  4
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 -116 15 50 5 -101 -41 -62 4
>> MCC track message setup
>> MCC setup track command: X<.11, 0.01>, Y<-.20, .11>
>> incr2edges: i = .113, e = 32
>> time2edges: t = 0.011, msec = 11
>> incr2edges: i = -.207, e = -378
>> time2edges: t = .112, msec = 112
SEND:  1  0  2 31 13 20  0  B  0 86 FE 70  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 32 0 11 0 -122 -2 112 0
>> MCC track message setup
>> MCC setup track command: X<.10, 0.01>, Y<.42, .11>
>> incr2edges: i = .107, e = 30
>> time2edges: t = 0.011, msec = 11
>> incr2edges: i = .428, e = 780
```

```
>> incr2edges: i = -0.076, e = -21
>> time2edges: t = 0.046, msec = 46
>> incr2edges: i = 0.037, e = 68
>> time2edges: t = 0.003, msec = 3
>> MCC prepare data: 16 17 X<45.07, 9.94> Y<-1.36, 9.76>
>> incr2edges: i = -3.590, e = -1021
>> time2edges: t = 1.276, msec = 1276
>> incr2edges: i = 14.120, e = 25706
>> time2edges: t = 1.209, msec = 1209
>> incr2edges: i = -3.590, e = -1021
>> time2edges: t = 1.276, msec = 1276
>> incr2edges: i = 14.120, e = 25706
>> time2edges: t = 1.209, msec = 1209
>> MCC prepare data: 17 19 X<44.99, 10.00> Y<-0.00, 9.99>
>> incr2edges: i = -0.079, e = -22
>> time2edges: t = 0.056, msec = 56
>> incr2edges: i = 1.362, e = 2480
>> time2edges: t = .233, msec = 233
>> Track: sending....
>> MCC send track commands: TM_Box = 4
>> MCC Axis Driver: TRACK_MOVE received
>>                  axis number = 0
>> MCC Axis Driver: sending to mcc_driver
>> MCC Axis Driver: sent to mcc_driver
>> MCC Driver: TRACK Request received
>> MCC Tracking Handling (TM_Box = 4; #msg 0)
>> MCC send track commands: given up semaphore
>> MCC send track commands: retaken semaphore for AXES_FIRST
>> MCC send track commands: blocking all other axes
>> MCC send track commands: blocking axis 1
>> MCC send track commands: about to send commands
>> MCC track message setup
>> MCC setup track command: X<.36, .12>, Y<-1.03, .20>
>> incr2edges: i = .367, e = 104
>> time2edges: t = .121, msec = 121
>> incr2edges: i = -1.039, e = -1892
>> time2edges: t = .203, msec = 203
SEND:  1  0  2 31 13 68  0 79  0 9C F8 CB  0
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 104 0 121 0 -100 -8 -53 0
>> MCC track message setup
>> MCC setup track command: X<8.69, 1.43>, Y<-14.19, 1.39>
>> incr2edges: i = 8.690, e = 2471
>> time2edges: t = 1.433, msec = 1433
>> incr2edges: i = -14.194, e = -25841
>> time2edges: t = 1.392, msec = 1392
SEND:  1  0  2 31 13 A7  9 99  5  F 9B 70  5
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 -89 9 -103 5 15 -101 112 5
>> MCC track message setup
>> MCC setup track command: X<8.69, 1.43>, Y<-14.19, 1.39>
>> incr2edges: i = 8.690, e = 2471
>> time2edges: t = 1.433, msec = 1433
>> incr2edges: i = -14.194, e = -25841
>> time2edges: t = 1.392, msec = 1392
SEND:  1  0  2 31 13 A7  9 99  5  F 9B 70  5
>> MCC  Tracking Handling: message received [13]
REC: 1 0 2 49 19 -89 9 -103 5 15 -101 112 5
>> MCC track message setup
>> MCC setup track command: X<.13, 0.02>, Y<-.20, 0.02>
>> incr2edges: i = .138, e = 39
>> time2edges: t = 0.020, msec = 20
>> incr2edges: i = -.201, e = -367
```

```
>> incr2edges: i = -.201, e = -367
>> time2edges: t = 0.020, msec = 20
>> MCC prepare data:  3  4 X<18.04, 3.03> Y<-29.81, 3.03>
>> incr2edges: i = .160, e = 45
>> time2edges: t = 0.020, msec = 20
>> incr2edges: i = -.180, e = -327
>> time2edges: t = 0.020, msec = 20
>> MCC prepare data:  4  5 X<27.23, 4.15> Y<-38.85, 4.14>
>> incr2edges: i = 9.190, e = 2614
>> time2edges: t = 1.126, msec = 1126
>> incr2edges: i = -9.044, e = -16465
>> time2edges: t = 1.115, msec = 1115
>> MCC prepare data:  5  6 X<27.44, 4.18> Y<-39.10, 4.18>
>> incr2edges: i = .206, e = 58
>> time2edges: t = 0.023, msec = 23
>> incr2edges: i = -.249, e = -454
>> time2edges: t = 0.034, msec = 34
>> MCC prepare data:  6  7 X<27.67, 4.20> Y<-39.29, 4.21>
>> incr2edges: i = .233, e = 66
>> time2edges: t = 0.023, msec = 23
>> incr2edges: i = -.190, e = -346
>> time2edges: t = 0.034, msec = 34
>> MCC prepare data:  7  8 X<41.67, 5.53> Y<-44.97, 5.43>
>> incr2edges: i = 13.993, e = 3980
>> time2edges: t = 1.330, msec = 1330
>> incr2edges: i = -5.680, e = -10341
>> time2edges: t = 1.218, msec = 1218
>> MCC prepare data:  8  9 X<41.78, 5.54> Y<-45.18, 5.54>
>> incr2edges: i = .113, e = 32
>> time2edges: t = 0.011, msec = 11
>> incr2edges: i = -.207, e = -378
>> time2edges: t = .112, msec = 112
>> MCC prepare data:  9 10 X<41.89, 5.55> Y<-44.75, 5.65>
>> incr2edges: i = .107, e = 30
>> time2edges: t = 0.011, msec = 11
>> incr2edges: i = .428, e = 780
>> time2edges: t = .112, msec = 112
>> MCC prepare data: 10 11 X<50.12, 6.43> Y<-39.44, 6.46>
>> incr2edges: i = 8.230, e = 2341
>> time2edges: t = .875, msec = 875
>> incr2edges: i = 5.307, e = 9661
>> time2edges: t = .801, msec = 801
>> MCC prepare data: 11 12 X<50.68, 6.50> Y<-39.07, 6.50>
>> incr2edges: i = .567, e = 161
>> time2edges: t = 0.075, msec = 75
>> incr2edges: i = .368, e = 671
>> time2edges: t = 0.047, msec = 47
>> MCC prepare data: 12 13 X<50.97, 6.58> Y<-38.59, 6.55>
>> incr2edges: i = .282, e = 80
>> time2edges: t = 0.075, msec = 75
>> incr2edges: i = .480, e = 875
>> time2edges: t = 0.047, msec = 47
>> MCC prepare data: 13 14 X<52.30, 7.29> Y<-29.67, 7.34>
>> incr2edges: i = 1.332, e = 378
>> time2edges: t = .714, msec = 714
>> incr2edges: i = 8.920, e = 16238
>> time2edges: t = .785, msec = 785
>> MCC prepare data: 14 15 X<52.33, 7.34> Y<-29.64, 7.34>
>> incr2edges: i = 0.032, e = 9
>> time2edges: t = 0.046, msec = 46
>> incr2edges: i = 0.036, e = 67
>> time2edges: t = 0.003, msec = 3
>> MCC prepare data: 15 16 X<52.25, 7.39> Y<-29.60, 7.34>
```

```
# mcc_axis_init("zeta_axis") # still missing
# mcc_axis_init("spindle_axis") # still missing
-> >> Starting mcc_axis_driver
>> pos_conv_factor = 1820.444444
>> vel_conv_factor = 119304.647111
>> acc_conv_factor = 119.304647
>> Axis  1:              theta_2 driver:Alive and well
Axis  1: Is a ROTARY axis


->

-> on 'a'
on 'a'
value = 0 = 0x0
-> home 'a'
home 'a'
>>> Target pos = .000 (axis x)
>>> Target pos = .000 (axis y)
value = 0 = 0x0
-> zero
zero
Last_Pos_In_Joint[0]: -60.263000
Last_Pos_In_Joint[1]: -107.263000
Move mode: 0
value = 0 = 0x0
->

-> coord
coord
value = 0 = 0x0
->

-> sp stroke, 45.0, 0.0, 10.0, 0
sp stroke, 45.0, 0.0, 10.0, 0
task spawned: id = 0x387b88, name = t40
value = 3701640 = 0x387b88
->
>> Track: velocities and accelerations

TIME = 10.000

DEFAULTACCELERATION = 50.000

>> Track: preparing data to send
>> MCC prepare data: 18 points
>> MCC prepare data:  0  0 X< .36,  .12> Y<-1.03,  .20>
>> incr2edges: i = .367, e = 104
>> time2edges: t = .121, msec = 121
>> incr2edges: i = -1.039, e = -1892
>> time2edges: t = .203, msec = 203
>> MCC prepare data:  1  1 X<17.74, 2.98> Y<-29.42, 2.98>
>> incr2edges: i = 8.690, e = 2471
>> time2edges: t = 1.433, msec = 1433
>> incr2edges: i = -14.194, e = -25841
>> time2edges: t = 1.392, msec = 1392
>> incr2edges: i = 8.690, e = 2471
>> time2edges: t = 1.433, msec = 1433
>> incr2edges: i = -14.194, e = -25841
>> time2edges: t = 1.392, msec = 1392
>> MCC prepare data:  2  3 X<17.88, 3.00> Y<-29.63, 3.00>
>> incr2edges: i = .138, e = 39
>> time2edges: t = 0.020, msec = 20
```

```
P_CMD:          kinematics added to system

P_CMD:              queuer added to system

O_CMD:                 on added to system; parse string:%A;#Aa:

O_CMD:                off added to system; parse string:%A;#Aa:

O_CMD:                pos added to system; parse string:%A;#Aa:

O_CMD:         axis_status added to system; parse string:%A;#Ax:

O_CMD:               coord added to system; parse string::

O_CMD:               abort added to system; parse string:%A;#Aa:

O_CMD:                tune added to system; parse string:%A%c:

O_CMD:          joint_move added to system; parse string:%f%f%f%i:

O_CMD:        joint22_move added to system; parse string:%f%f%f:

O_CMD:      cartesian_move added to system; parse string:%f%f%f:

O_CMD:                home added to system; parse string:%A;#Aa:

O_CMD:    joint_trajectory added to system; parse string::

O_CMD:              stroke added to system; parse string:%f%f%f%i:

O_CMD:                zero added to system; parse string::

I_CMD:                  at added to system

Commands installed.

***** Error handling task: alive and well...
***** Dispatch task: alive and well...
***** Scheduler task: alive and well...

MOSAIC OPEN-ARCHITECTURE CONTROLLER 3.0 INSTALLED

Controller configured for:     EDUCATION_ROBOT
... machine tool initialization begins

value = 0 = 0x0
creonics_init(1, 0xffffff00)
value = 0 = 0x0
# creonics_init(2, 0xfffffe00) # still missing
mcc_axis_init("theta1_axis")

>> Creonics board slot:  1, base address: 0xFFFFFF00
>> mcc_driver.o: Driver initialization OK
value = 10 = 0xa
mcc_axis_init("theta2_axis")
>> Starting mcc_axis_driver
>> pos_conv_factor = 284.444444
>> vel_conv_factor = 18641.351111
>> acc_conv_factor = 18.641351
>> Axis  0:              theta_1 driver:Alive and well
Axis  0: Is a ROTARY axis
value = 10 = 0xa
```

```
VMEbus table created.
Slot  0:         Null_Driver at 0X  3cf694:status = 0x80

Slot  1:        mcc_driver.o at 0X  3b1014:status = 0x80

VMEbus table Filled.

Address Book created.
Mail for Error at: 0
Ordered   mail at: 1
Immediate mail at: 2
Scheduled mail at: 3
Address Book initialized.

Machine Tool Structure Created.
>> add_axis: status = 0

Axis number  0 (x):             theta_1
    Status:           0x80
    Control Type:     0
    Negative Limit: -180.000
    Positive Limit: 180.000
    Home Offset:    .0000000
>> add_axis: status = 0

Axis number  1 (y):             theta_2
    Status:           0x80
    Control Type:     0
    Negative Limit: -180.000
    Positive Limit: 180.000
    Home Offset:    .0000000

Sensor table created.
Sensor table filled.

MACHINE:     EDUCATION_ROBOT
       2 Axis
       0 Sensors

Mode:              0x 0
Feed:               .000 ipm
Speed:              .000 rpm
Force:              .000 (if in force mode)
Clearance Plane:    .000
Tolerance:          .000 inch
Dwell Time:         .000 sec
Rapid_Feed:         .000 ipm

Machine:     EDUCATION_ROBOT Mode:
Move Type:    ABSOLUTE
Force Mode:   OFF


O_Cmd table created.
I_Cmd table created.
P_CMD:       creonics_init added to system

P_CMD:     mcc_axis_driver added to system

P_CMD:       mcc_axis_init added to system

P_CMD:         queuer_util added to system
```

```
      result[i] = ks[i].x;
      break;
    case tbd_y:
      result[i] = ks[i].y;
      break;
    default:
      printf(">> ERROR: unreconized knot structure component %d\n", comp);
    }
  }
}
/* extract_knot_component -- */


#endif /* #define straight_line_i */

/* end of file -- straight_line.c */
```

A.2. Sample Execution

```
ed1@robocop 11> rlogin grope

-> cd "mosaic:/cadman.a/robots/ed1/src"
cd "mosaic:/cadman.a/robots/ed1/src"
value = 0 = 0x0
-> < MOSAIC.EDI
< MOSAIC.EDI
shellLock 1
value = 0 = 0x0
# cd "mosaic:/robust.b/robots/ed1/ed1"
ld < vme_util.o
value = 0 = 0x0
ld < mail_util.o
value = 0 = 0x0
ld < machine_util.o
value = 0 = 0x0
ld < command_util.o
value = 0 = 0x0
ld < mosaic_init.o
value = 0 = 0x0
ld < vel-acc.o
value = 0 = 0x0
ld < track-move.o
value = 0 = 0x0
# ########################################## #
#                                            #
#    Version 3.0 Of MOSAIC use caution       #
#                  -for-                     #
#             EDUCATION ROBOT                #
#                                            #
# ########################################## #
mosaic_init("ed1.cfg")


MOSAIC OPEN-ARCHITECTURE CONTROLLER INITIALIZATION:
==================================================
```

```
{
  int i;

  for (i = 0; i < MAX_KNOTS_NUMBER; i++) {
    ks->x = ks->y = 0.0;
    ks->filled = 0;
    ks++;
  }
}


/* compact_knots_sequence -- Compacts the sequence of knots 'ks', by
 * removing the 'unfilled' slots.
 */

int compact_knots_sequence(ks1)
     knot_struct ks1[];
{
  int i;
  int  k1, k2;
  knot_struct  ks2[MAX_KNOTS_NUMBER];

  init_knots_sequence(ks2);

  for (k1 = 0, k2 = 0; k1 < MAX_KNOTS_NUMBER; k1++) {
    if (ks1[k1].filled) {
      ks2[k2] = ks1[k1];
      k2++;
    }
  }

  /* Now copy back ks2 into ks1 */

  for (i = 0; i < k2; i++)
    ks1[i] = ks2[i];

  return k2;
}
/* compact_knots_sequence -- */


/* extract_knot_component -- */

void extract_knot_component(ks, comp, kcount, result)
     knot_struct ks[];
     tbd_components comp;
     int kcount;
     double result[];
{
  int i;

  for (i = 0; i < kcount; i++) {
    switch (comp) {
    case tbd_x:
```

```
                          mid_index, index2);

    if (ret_code == ERROR) return ERROR;
  } else
    return OK;
}
/* taylor_bdev -- */


/* compute_deviation --
 * It computes how much the point M = <mid_x, mid_y> is distant from
 * the line between A = <init_x, init_y> and B = <final_x, final_y>.
 */

double compute_deviation(init_x, init_y,
                         mid_x, mid_y,
                         final_x, final_y)
     double init_x, init_y, mid_x, mid_y, final_x, final_y;
{

  /* To visualize the computation, just call
   * (init_x, init_y) = A
   * (mid_x, mid_y) = M
   * (final_x, final_y) = B
   * (M_proj_x, M_proj_y) = C
   *
   * Where A, C, B are on the same line and M is the point of which we
   * want to know how much it deviates.
   */

  double AB_x = init_x - final_x;
  double AB_y = init_y - final_y;
  double AM_x = init_x - mid_x;
  double AM_y = init_y - mid_y;

  /* Take some inner products */

  double AB_sqr = SQR(AB_x) + SQR(AB_y);
  double AM_sqr = SQR(AM_x) + SQR(AM_y);
  double AM_AB  = (AM_x * AB_x) + (AM_y * AB_y);

  double AC_len = AM_AB / sqrt(AB_sqr);

  return sqrt(AM_sqr - SQR(AC_len));
}
/* compute_deviation -- */


/*=========================================================================
 * knots manipulation routines.
 */

void init_knots_sequence(ks)
     knot_struct *ks;
```

```
  }

  ret_code = arm_forward_kinematics(final_theta1, final_theta2, &final_x, &final_y);
  if (ret_code == ERROR) {
    printf(">> ERROR: taylor_bdev: error from forward kinematics.\n");
    return ERROR;
  }

  dev = compute_deviation(init_x, init_y,
                          mid_x, mid_y,
                          final_x, final_y);

  if (dev >= TOLERATED_DEVIATION) {
    /* The followings need to be done: */
    /* a - compute the Cartesian Space 'intermediate' point between */
    /*     'init' and 'final'. */
    /* b - compute the inverse kinematic for its position in Joint Space. */
    /* c - set the point in Joint space found in the previous step as */
    /*     the real knot. */

    double intermediate_x = (init_x + final_x) / 2.0;
    double intermediate_y = (init_y + final_y) / 2.0;

    mid_index = (index1 + index2) / 2;

    if ((mid_index == index1) || (mid_index == index2)) {
      printf(">> ERROR: insufficient knot_sequence array space\n");
      return ERROR;
    }

    ret_code = arm_inverse_kinematics(&mid_theta1, &mid_theta2, intermediate_x,
intermediate_y);

    /* Attention! I changed the values of the mid_theta's. */

    if (ret_code == ERROR) {
      printf(">> ERROR: taylor_bdev: error from inverse kinematics.\n");
      return ERROR;
    }

    knots_seq[mid_index].filled = 1;
    knots_seq[mid_index].x = mid_theta1;
    knots_seq[mid_index].y = mid_theta2;

    ret_code = taylor_bdev(init_theta1, init_theta2,
                           mid_theta1, mid_theta2,
                           knots_seq,
                           index1, mid_index);

    if (ret_code == ERROR) return ERROR;

    ret_code = taylor_bdev(mid_theta1, mid_theta2,
                           final_theta1, final_theta2,
                           knots_seq,
```

```
  /* Massage the data. */
  extract_knot_component(knot_sequence, tbd_x, knot_count, th1s);
  extract_knot_component(knot_sequence, tbd_y, knot_count, th2s);

  if (track_move(th1s, th2s, knot_count, time, acc) == ERROR) {
    printf(">> ERROR: stroke: tracking motion failed\n");
    return ERROR;
  }

  /* Unsafe! If an error occurs during the move, then there is no */
  /* guarantee that the position obtained is the desired one. */
  Axis[0].Last_Pos_In_Joint = Axis[0].Target_Pos_In_Joint;
  Axis[1].Last_Pos_In_Joint = Axis[1].Target_Pos_In_Joint;

  return OK;
}
/* stroke -- */


/* taylor_bdev --
 * Function that computes the sequence of 'knots' to be followed in
 * order to approximate a straight line in cartesian space.
 * References: P. J. McKerrow, Robotics, pages 523--532, Addison
 * Wesley, 1991.
 */

int taylor_bdev(init_theta1, iniit_theta2,
                final_theta1, final_theta2,
                knots_seq,
                index1, index2)
     double      init_theta1, init_theta2, final_theta1, final_theta2;
     knot_struct *knots_seq;
     int         index1, index2;
{
  double mid_theta1 = (final_theta1 + init_theta1) / 2.0;
  double mid_theta2 = (final_theta2 + init_theta2) / 2.0;
  double mid_x, mid_y,  init_x, init_y,  final_x, final_y;

  int mid_index; /* Index where the knot will be stored. */
  double dev = 0.0; /* The deviation from the desired */
/* trajectory. */
  int ret_code = OK;

  ret_code = arm_forward_kinematics(mid_theta1, mid_theta2, &mid_x, &mid_y);

  if (ret_code == ERROR) {
    printf(">> ERROR: taylor_bdev: error from forward kinematics.\n");
    return ERROR;
  }

  ret_code = arm_forward_kinematics(init_theta1, init_theta2, &init_x, &init_y);
  if (ret_code == ERROR) {
    printf(">> ERROR: taylor_bdev: error from forward kinematics.\n");
    return ERROR;
```

```
double compute_deviation();
void init_knots_sequence();
int compact_knots_sequence();
void extract_knot_component();

/* Compute joint space configuration of final point, assuming that  */
/* the original point is already contained in the 'Last_Pos' fields */
/* of the Axes. Given this assumption there is no need to compute the */
/* joint space configuration for the starting point. */

if (mode == CARTESIAN_SPACE) {
  inv_ret_code = arm_inverse_kinematics(&final_theta1, &final_theta2, x, y);
  if (inv_ret_code == ERROR) {
    printf(">> ERROR: stroke: pos. <X Y> = <%4.2f %4.2f> is unreachable\n", x, y);
    return ERROR;
  }
} else {
  final_theta1 = x;
  final_theta2 = y;
}

Axis[0].Target_Pos_In_Joint = final_theta1;
Axis[1].Target_Pos_In_Joint = final_theta2;

/* Now compute the sequence of knots needed to */
/* approximate the straight line in cartesian space. */

init_knots_sequence(knot_sequence);

knot_sequence[0].x = init_theta1;
knot_sequence[0].y = init_theta2;
knot_sequence[0].filled = 1;

knot_sequence[MAX_KNOTS_NUMBER - 1].x = final_theta1;
knot_sequence[MAX_KNOTS_NUMBER - 1].y = final_theta2;
knot_sequence[MAX_KNOTS_NUMBER - 1].filled = 1;

inv_ret_code = taylor_bdev(init_theta1,
                           init_theta2,
                           final_theta1,
                           final_theta2,
                           knot_sequence,
                           0,
                           MAX_KNOTS_NUMBER - 1);

if (inv_ret_code == ERROR) {
  printf("ERROR: stroke: unrechable intermediate point\n");
  return ERROR;
}

knot_count = compact_knots_sequence(knot_sequence);

/* Trying the tracking move. */
```

```
 */

/* knot_struct -- A structure for computing the knots of the Taylor
 * Bounded Deviation Algorithm in a bidimensional joint space.
 *
 * Note:
 * Actually the 'x' and 'y' should be renamed to something like 'th1'
 * and 'th2'.
 */
typedef struct _tbd_struct {
  double x, y;
  int filled;
} knot_struct;

typedef enum {tbd_x, tbd_y} tbd_components;


/*=========================================================================
 * Functions
 */

/* stroke --
 * Moves the manipulator from the current position (x-start, y-start)
 * to the position x, y in the specified amount of time.
 *
 * This function is actually the driver for Taylor's bounded deviation
 * algorithm (pg. 527 of McKerrow).
 */

int stroke(x, y, time, acc, mode)
     double x;
     double y;
     double time;
     double acc;
     int    mode; /* joint space or cartesian space */
{
  int  i;
  int  axis;

  double  init_theta1 = Axis[0].Last_Pos_In_Joint;
  double  init_theta2 = Axis[1].Last_Pos_In_Joint;
  /* Hardcoded axis numbers! */

  double  final_theta1 = 0.0; /* Dummy final values */
  double  final_theta2 = 0.0; /* Dummy final values */

  int  inv_ret_code = OK;

  double  delta1, delta2;
  int knot_count;
  knot_struct  knot_sequence[MAX_KNOTS_NUMBER];
  double th1s[MAX_KNOTS_NUMBER];
  double th2s[MAX_KNOTS_NUMBER];
```

Appendix: A
An Example Program: the `stroke` Routine

A.1. Program Listing

```
/* -*- Mode: C -*- */

/* straight_line --
 * External routines for the first try of the straight line
 *
 * Authors: Louie Pavlakos, Marco Antoniotti, Ajay Rajkumar
 * Address: Robotics Laboratory
 *          Courant Institute of Mathematical Science
 *          New York University
 *          New York, NY, 10012
 *
 * Copyright (c) 1992 Elias Pavlakos
 *  1993 Marco Antoniotti, Ajay Rajkumar.
 *               All rights reserved.
 *
 * Version (RCS format):
 * $Id: straight_line.c,v 3.9 1993/05/28 15:51:21 ed1 Exp ed1 $
 */

#ifndef straight_line_i
#define straight_line_i

#include <math.h>
#include <semLib.h>
#include "machine.h"
#include "mail.h"
#include "creonics.h"
#include "VMEbus.h"


#include "straight_line.h"

/* GLOBAL ROUTINES */

/* EXTERNAL ROUTINES */

extern int joint_move();
extern int track_move();

/* EXTERNAL DATA */

extern AXIS_STRUCT Axis[];


/*============================================================================
 * Data structures
```

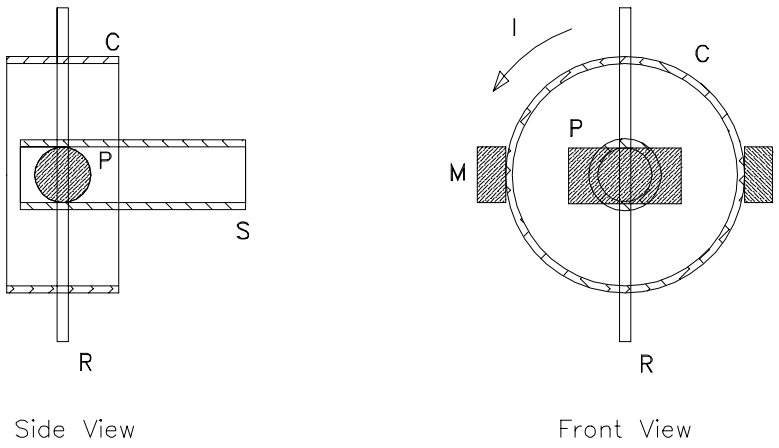


Figure 15: **Principle of direct drive: The motor stator holds a coil C and a rotor bearing R. The rotor consists of a permanent magnet P and a shaft S. Current I in the coil results in motion around the axis R in a direction given by the sign of I. An optional second stator magnet M provides a reference field from which the rotor may be deflected or pointed.**

fingers or miniature arms, grippers and multi-fingered hands from simple parts), they are safe and they are easy to program. Several student projects based on these robots have been carried out under the guidance of Richard Wallace.

The key components are very low cost direct drive DC motor actuators. The motors are based on Nd-Fe-B rare earth permanent magnets and controlled by low cost microcontrollers. The motors have low friction, small size, high speed, low construction cost, no gear backlash, operate safely without limit switches, have limited self-braking, and generate moderate torque. Significantly, one motor can generate enough torque to lift a second motor of about the same size against the force of gravity, at a distance approximately equal to the size of the motor, without resorting to the use of a counterweight.

We developed a prototype three link finger direct drive which can exert a force of up to 1.0 Nt at the finger tip (see Figure 14). The finger joints are primitive permanent magnet DC motors consisting of a single stator coil and a single rotor magnet, which provide a means of direct joint torque control by varying current through the coil. The joints of the DD finger contain no gears or transmission mechanism (see Figure 15), which results in very low friction and eliminates gearbox control problems, and facilitates force control strategies such as compliance.

The very low cost (about $10) of miniature DD motors makes them attractive for education as well. Students in our robotics lab, including secondary school interns, build mini DD motors as a lab assignment. This gives the students practical first-hand experience with the principles of electromagnetic actuation and control theory. In particular, some students built a small two link direct drive manipulator. They then related its kinematics, statics and dynamics to the simple two link planar manipulator theory (see e.g. [7,25]) they learn in class.

In future, we plan to incorporate these robots into the regular class experiments and report our experience.

6. Concluding Remarks

Following materials have been developed under this project:

- A prototype "Educational Robot:" ED I.
- Manuals: Laboratory Manual and Developer's Manual. Manuals for the `VxWorks`^{tm} based real-time system.
- An undergraduate robotics textbook authored by Li, Murray and Sastry.

The primary benefit to our institution has been through the establishment of a permanent robotics course sequence that is accessible to undergraduates from both New York University and Stevens Institute of Technology. Also, the creation of a multifunctional laboratory has helped us to form a coherent sequence of experiments for many different courses that the student can get involved in with a minimal overhead. Secondarily, it has also helped our research work in robotics and manufacturing, by sharing a common software development platform and by acting as a testbed for some of the miniature actuators developed in an unrelated project.

The primary contribution to the computer science community is through a body of well-trained computer scientists who have a strong hands-on understanding of robotics. Also, over the period of the project, we have invited several visitors from other computer science departments to guide and share our work. This dissemination effort is hoped to influence the way robotics and manufacturing classes are taught currently.

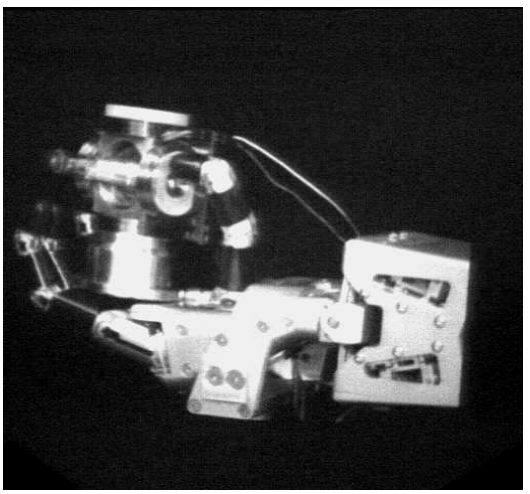


Figure 14: **A direct drive pointing device.**

6.1 The Future

In parallel with the ED I project, we have also explored alternate technologies that are suitable for teaching robotics. One approach has been developing miniature direct drive robots that are ideal for teaching as they are easy to configure (e.g., we have been able to build mobile robots, robot

Additional care is taken to ensure that shut down, accidental removal or variation of level in the power source does not result in a hazardous condition. For instance, the power failure does not cause release of the load from the gripper as the robot is equipped with a mechanical gripper.

The robot has two readily accessible emergency stop devices that can be used to turn the robot off as an emergency situation begins to develop.

The robot is physically encased within a lexan enclosure equipped with a "guard mechanism" such that the guard prevents the robot from computer-controlled operation mode if and when the guard is open. If the guard is opened during a computer-controlled operation the robot shuts itself off and does not start until the guard mechanism is closed and certain reinitialization operations are performed.

5. Safety Issues

The current design and implementation of ED I robot was examined by Dr. Anshin (a mechanical engineer and roboticists from Russia) and the design, construction, programming, operation and maintenance of the ED I robot was found to meet the ISO 10218 safety standard (Document: Manipulating Industrial Robots—Safety).

Several technical measures are employed in order to prevent accidents that may result in physical damage to the operator, bystanders, other mechanical and computational hardware or the robot itself. These measures are based upon two fundamental principles, as follows:

- The absence of humans in the safeguarded space during automatic operation;
- The elimination of hazards during intervention (e.g. repair or maintenance) in the safeguarded space.

5.1 General Design Requirements

The main strategy in avoiding accidents resulting from electrical connection was to follow the standard recommended practice (for instance, IEC 204-1) and the manufacturer's specifications with substantial margins of safety. The power supply, grounding (protective earth) and cabling are instances where special care was taken. The power source has been isolated in such a way that it poses no danger to humans and also has a lockout/tagout capability.

5.2 Design and Construction of the ED I Robot

5.2.1 Ergonomic Aspects

The design and installation of the robot was carried out in a manner such that for routine operations of the robot (as in carrying out an experiment) no direct human intervention is necessary. The robot in its encased surrounding is operated from a terminal at a distance of few feet. All the commands can be sent from the terminal and the sensor readings are directly displayed on the terminal. In case of an unexpected situation, the operator can turn the robot "off" by pressing an emergency stop "red button."

In few cases, where human intervention is necessary (i.e. certain direct data collection operations where the robot cannot be powered off), the design balances the robots strength, size and movements against a typical user.

5.2.2 Mechanical Aspects

The range of motion of the robot are mechanically limited by placing limit switches and mechanical stops on the primary axes. Thus, even a careless user is protected against operating the robot in an unintended manner.

5.2.3 Control Aspects

The robot is controlled in a safe manner: the commands to the robot are routinely checked for their potentials to create a dangerous situation. For instance, the speed of the primary axes of the robot is kept limited by both electronic and software means. The static and dynamic forces created by the load at the end-effector are kept within the load capacity and dynamic response of the robot.

Summary

1. Draw a diagram of the control law you used in part three.
2. The models used in this experiment are ideal models. How would friction affect the model?
3. In the control parts, what is the effect of the *b-compensation*?
4. What is the result of positive feedback?

(d) $E_g = K_E\,\omega$. Verify K_E as follows: Measure the voltage using a voltmeter. By turning the bar, estimate the angular velocity you are applying, and measure the voltage. Plot the voltage vs. the angular velocity.

2. Familiarization with the power amplifier and the decoder:

   (a) Send some signals through the power amplifier, and measure the output.

   (b) Rotate the bar at the end of the motor, and measure the output of the decoder

3. Implementation of a simplified PD control:

$$V = -K_p e\,.$$

   The program sets the desired theta to zero, and stays there.

   (a) Modify the program to change theta desired.

   (b) Modify the value of K_p. What effect does changing K_p have on the robot.

4. Implement the PD control law.

$$V = -K_v \dot{e} - K_p e\,.$$

   First, calculate $\dot{\theta}$:

$$\dot{\theta}(i+1) = \frac{\theta(i+1) - \theta(i)}{\Delta t}.$$

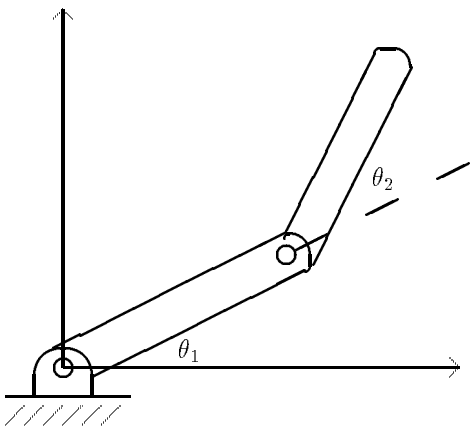


Figure 13: **ED I modeled as a planar arm with two direct drive joints**

In other words, it is the current position—the previous position, divided by the amount of time. In order to know what the value of Δt is, one can do a number of things: Either rewrite the program using the timer interrupts, and interrupt the CPU every 1 msec, at which time it should run the procedure. In this case $\Delta t = 1msec$. Or, since $\dot{\theta}$ is multiplied by K_v anyway, just assume Δt is part of K_v, and vary K_v.

For the motor used in this experiment, here are the exact specifications:

$$\begin{aligned} J &= 4.88 \times 10^{-4}\ \text{N.m.sec}^2, \\ K_T &= 0.346\ \text{N.m./amp}, \\ K_E &= 0.346\ \text{V/rad/sec}, \\ R &= 2.08\ \text{ohms}. \end{aligned}$$

Control Laws:

Model-Based Control:

Model-Based Control is used when the parameters of the system a, b are known. For set point regulation, a model-based control law might be:

$$V = a(-K_v\dot{\theta} - K_p(\theta - \theta^d)) + b\dot{\theta},$$

where θ is the actual θ, and θ^d is the desired θ.

For Trajectory tracking, we have:

$$V = a(\ddot{\theta}^d - K_v\dot{e} - K_p e) + b\dot{\theta},$$

where $e = \theta - \theta^d$, and $\dot{e} = \dot{\theta} - \dot{\theta}^d$.

PD Control

PD control is used when there is no model of the system, i.e. the parameters of the system are not known a priori. In this case, a simpler control scheme is adopted by industrial manipulators, which does not assume any knowledge of the system. For example (for trajectory tracking):

$$V = -K_v\dot{e} - K_p e.$$

- **Procedure:**

1. In this first part, verify the parameters of the motor: K_E, K_T, R, J and a, b.

   (a) Measure the terminal resistance, R.

   (b) Measure the current, I.

   (c) $T_g = K_T\ I$. Verify K_T in the following manner: Attach a bar to the edge of the motor, and rest the other end of the bar on a scale.

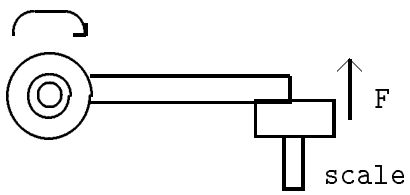


Figure 12: **Experimental setup for the D.C. Motor experiment**

By supplying a current to the motor, the bar will exert a force on the scale. Measure the force. The torque produced is equal to the force multiplied by the length of the bar. Vary the current from 0 to the maximum, and plot the computed torque vs. the current.

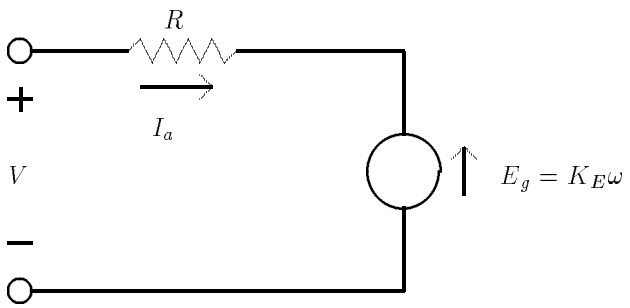


Figure 11: **A lumped-circuit model of a D.C. Motor**

Here, R (ohms) is called the *terminal resistance*, which can be measured by hooking up an *Ohmeter* across the motor wires with the motor at stall. E_g is the back EMF voltage, and is proportional to the rotor velocity, $\omega = \dfrac{d\theta}{dt}$, with proportionality constant K_E (known as the back EMF constant). One way to estimate K_E is by rotating the rotor at a known fixed velocity, and then measuring the voltage generated at the terminal ends. V is the applied voltage, and I_a is the motor current.

The electrical equations for the above figure is given by:

$$V = R\, I_a + K_E\, \omega = R\, I_a + K_E \frac{d\theta}{dt}.$$

In the presence of motor current, I, a torque proportional to I is produced:

$$T_g = K_T\ I_a\,,$$

where K_T is called the torque constant.

The mechanical equation relating T_g to the rotor motion is given by

$$T_g = J \cdot \frac{d^2\theta}{dt^2},$$

where J $(N.m.s^2)$ is the rotor inertia. Note that all parameters (J, K_T, R, K_E) can be found from the data sheet for the motor.

Combining the two equations above for T_g yields the equation relating applied voltage, V, to joint motion $\theta(t)$:

$$V = a\frac{d^2\theta}{dt^2} + b\frac{d\theta}{dt},$$

where

$$a = \frac{JR}{K_T} \quad \text{and} \quad b = K_E.$$

In other words, the control input, V affects the joint motion, $\theta(t)$, through a second-order linear differential equation. The coefficients (a, b) of the differential equation depend on the motor construction only.

■ Experiment 4: D.C. Motors

• **Objective:** The purpose of this experiment is to familiarize the students with the basic operation of D.C. motors. For this experiment, the students use simple position control of one joint.

• **Principles of Operation:**

Direct-Current (D.C.) Motors: The last revolute joint of the NYU ED-1 is driven by a D.C. motor manufactured by Inland Inc. A rudimentary D.C. motor is comprised of three main parts:

1. A *stator* with a permanent magnet providing a magnetic field for energy conversion,
2. a *rotor* (or *armature*) with current carrying conductors, and
3. *carbon brushes* and *commutator* for introducing current to the moving conductors.

In order to have a computer controlling either velocity, joint point position, joint velocity, or torque of a D.C. motor, a *power amplifier* and an *optical encoder* are required. The power amplifier takes a commanding signal from the computer and outputs a driving signal to the D.C. motor. An optical encoder and a decoder device together transform joint position information to the computer. A computer controlled D.C. motor system is shown below:

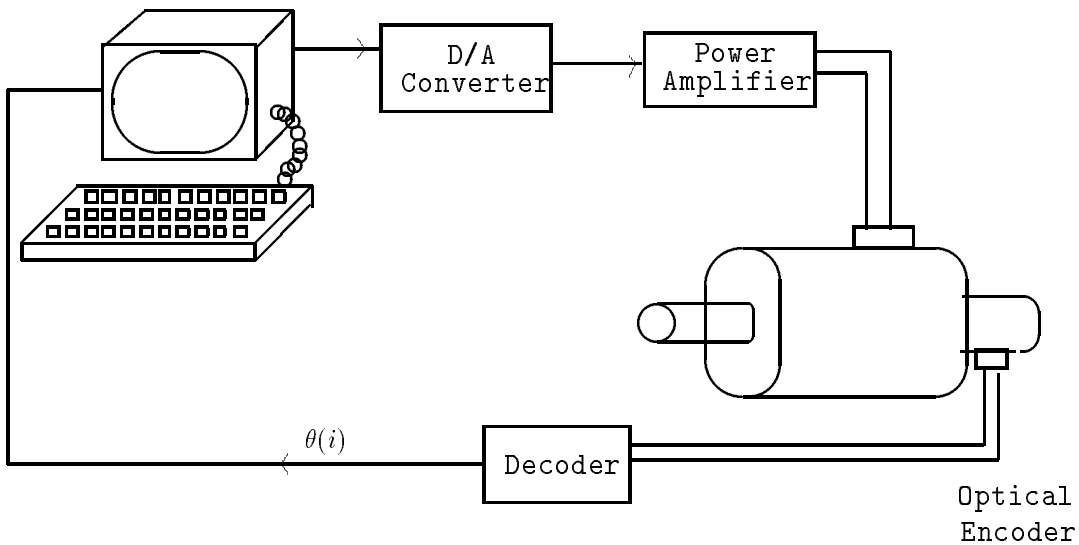


Figure 10: **A computer controlled D.C. Motor**

Note that a Digital-to-Analog (D/A) Converter is used because a computer outputs only discrete signals.

Model of a D.C. Motor: To understand the control of a D.C. motor, we need to derive the equations describing the motor. A simplified equivalent electrical circuit of the motor is shown below: (This model is adequate for position control.)

```
15              forward(theta1, theta2, &trajx[k], &trajy[k]);
16              if (time > length-of-time-for-this-segment)
17                {
18                   time = 0;
19                   /* increment seg */
20                  }
21         }
22       }
```

4. Using the above procedures, write a program to compute a trajectory between two points. The program should accept up to five via-points.

   Choice of time step:

   (a) Ask the user to input the total time. Let the time for each segment be the total time divided by the number of segments.

   (b) Ask the user to input how much time each segment should take.

   (c) Any other method one may wish to choose.

   The program looks basically as follows:

```
1       main()
2       {
3          /* get the points and the time duration */
4          /* do inverse kinematics on the points to get the values of theta1
5               and theta2 for each point. */
6          /* compute the joint velocities of the via points
7               (at each point you need velocities for theta1 and theta2) */
8          /* compute the cubic coefficients
9               (in each segment, 4 coeffs for theta1 and 4 for theta2) */
10         /* compute the path. */
11      }
```

- **Procedure:**

1. Make sure the program works (compiled and debugged). See if the numbers look reasonable.

2. Run your program on the ED I robot. Run it at least three times. What points were chosen? What intermediate points were generated by the program? Describe what actually happened when the robot is commanded to follow the given trajectory.

3. Run the program using the five points on the grid from the *inverse kinematics* experiment. Choose one of the points to be both the initial point and the final point, and the other four points to be via points. How does the generated trajectory compare with the trajectory from the *inverse kinematics* experiment (compare speed, path, etc.)?

- **Summary:**

1. **Optional**: Plot the x, y coordinates generated by the program. How does this plot compare to the trajectory which the robot followed?

■ Experiment 3: Trajectory Generation

• **Objective:** The purpose of this experiment is to familiarize the students with the concept of generating a trajectory. They write a path planning program which uses the cubic spline method for joint space trajectory generation.

• **Principles of Operation:** The background required for this experiment can be found in chapter 7 of J. Craig's book([7]).

In the two previous experiments, the students performed some trajectory generation tasks, in that they were required to specify points through which the robot end effector must pass. In contrast, the students here become acquainted with one of the more standard methods of trajectory generation.

Preparation Below is a list of procedures that the students need to complete their program.

1. Write a procedure to compute the cubic coefficients of a cubic polynomial. The procedure should take as parameters: the initial theta, the final theta, the initial velocity (`thetadot0`) and the final velocity (`thetadotf`), and the amount of time. It should return the four coefficients.

```
1        cubic_coeff(th0, thf, thdot0, thdotf, tf, coefs)
2           float th0, thf;
3           float thdot0, thdotf;
4           float tf;
5           float *coefs;
```

2. Write a procedure to calculate the joint velocities at the via points. For each segment (for n via points, there are $n-1$ segments), the velocities are different. The students are encouraged to use any of the three methods described on pp. 235-236 of Craig[7]. The easiest to implement is choice 2: the system chooses the average of the slopes as the via velocity. In other words, for each segment i, the velocity of $\theta_1 = \frac{1}{2}((\theta_{1,i} - \theta_{1,i-1}) + (\theta_{1,i+1} - \theta_{1,i}))$. Similarly for the velocity of θ_2.

3. Write a procedure to compute θ_1 and θ_2. For each segment i, θ_i is computed by formula (7.3)[7]. The basic algorithm is as follows:

```
1       run_path()
2       {
3         time = 0;
4         seg = 0;
5         number-of-ticks =  totaltime / deltat;
6
7         for (k=0; k<number-of-ticks; k++)
8         {
9               /* increment time by deltat */
10              /* calculate theta1 using the coefficients for
11                          theta1 in seg */
12              /* calculate theta2 using the coefficients for
13                          theta2 in seg */
14
```

- **Procedure:** Followings are the main steps of the experimental task:

  1. This first part is to familiarize the students with the system. The lab instructor starts by demonstrating how to power up the system, how to use the editor, how to compile the programs, etc. Students run two sample programs: **TESTMOVE** and **SENSR** which reads the values of the position sensors.

     Turn on the system, and run the program read. Move the robot around a bit and write down the sensor values. (The purpose of this is to give students a chance to operate the system on their own).

  2. Measure the link lengths of each of the two links on the ED I robot. These lengths are needed in order to compute the forward kinematics.

  3. Construct the forward kinematics map for first two links of the ED I robot. Write a procedure which computes the forward kinematics. Input to the routine should be the values of *th1* and *th2*. The x and y values should be placed in *xtip* and *ytip*.

```
1       #define  LINK1  <value_measure_for_link1>
2       #define  LINK2  <value_measure_for_link2>
3
4       int
5       forward_kinematics(thet1, thet2, xtip, ytip)
6          short thet1, thet2;
7          float *xtip, *ytip;
```

  4. Now, test out the forward_kinematics procedure. Write a program which does the following:

     (a) Prompts for values for *thet1* and *thet2*.

     (b) Commands the robot to move to this desired configuration (i.e. *thet1* and *thet2*).

     (c) Reads the sensor values and prints them out.

     (d) Calls the procedure *forward_kinematics* using these values and prints out the cartesian coordinates of the end-effector.

  5. Run the program a few times, and write down the output. Use values in the range of $20° - 60°$. Using the grid on the table by the robot determine what the position of the end effector is. How closely does it compare with the values computed by the procedure?

  6. Run your program with the following values and note what happens:

     (a) 0 0

     (b) 0 90

     (c) 90 0

     (d) -20 0

     (e) 200 200

- **Summary:**

  1. What happens when the program is given values that are out of range?

  2. If the initial configuration is not really $\theta_1 = 0, \theta_2 = 0$, how will that affect the results of forward kinematics procedure?

• **Principles of Operation:** The *inverse kinematics map* is the inverse of the forward kinematic map. Given the position in cartesian space, it returns the values of the joint angles required to achieve that position.

Unlike forward kinematics, which produces only one solution, the inverse kinematics may produce multiple solutions, depending on the configuration of the robot. If the robot contains more than two degrees of freedom in the plane or more than three degrees of freedom in the space, multiple solutions may exist. Since in this experiment, only two joints of the ED I robot are used, the problem discussed above need not concern us immediately.

• **Preparation:** Students are asked to compute the inverse kinematics of the simplified ED I robot (the two joints) and write code that will take coordinates x and y and return the angles θ_1 and θ_2 that correspond to the configuration where the end effector is at x, y.

• **Procedure:** Following are the main steps of the programming task:

1. Write a procedure to compute the inverse kinematics of the ED I. Input to the routine should be the coordinates of the end-effector, and the procedure should return the values of the two joint angles.

```
1      int inverse_kinematics(xtip, ytip, thet1, thet2)
2         float xtip, ytip;
3         short *thet1, *thet2;
```

2. Write a program which does the following:

   (a) Prompts for values for x and y coordinates.

   (b) Calls the procedure *inverse_kinematics* using these values and prints out the values obtained for *thet1* and *thet2*.

   (c) Generates the cubic coefficients for each joint.

   (d) Moves the robot to this desired configuration using the code in Part C.

■ Experiment 2: System Familiarization and Robot Kinematics

• **Objective:** Introduction to the system, consisting of the robot, its actuators, its control units, the interface, and the computer system. The students are shown some demonstrations illustrating how to read feedback signals and to command joint movement of the robot. The students then perform some simple experiments and familiarize themselves with basic robot operation.

• **Principles of Operation:** The robot and the system the students use in this experiment is NYU ED I robot and is explained in §2 and §3. See the previous subsection for an explanation of how to command the ED I robot to move, and how to read the position sensors of the robot, etc.

• **Preparation:** No prior preparation is necessary.

• **Summary:** Following is a list of questions that were posed to the students in connection with the preceding experiment:

1. What is the danger of having a large frequency and a large magnitude for the sin function which controls the angle of the joint? (i.e. why would this translate to large torques on the joint?)
2. Try to come up with values of M, ω, and T to produce a trajectory which progresses smoothly from 0 to 90 degrees (0 to $\pi/2$ radians).

□ Part B: Generating Trajectories Using a Cubic Polynomial

• **Objective:** To generate a cubic polynomial based on an initial angle, an initial velocity, a final angle, a final velocity, and the amount of time to move from the initial configuration to the final.

• **Principles of Operation:** Since a cubic polynomial has four coefficients, if given four constraints, one can uniquely determine these coefficients. In this part of the lab, the students use the initial angle, the initial velocity, the angle at time T, and the velocity at time T as constraints to determine the cubic polynomial.

• **Preparation:** Students are asked to read a set of notes on cubic polynomials. As in the first part, they are expected to write the necessary code before they come to the lab.

• **Procedure:** Students are asked to write a procedure to take as input integers θ_0, θ_f, and T, where θ_0 is the initial angle of joint 1, θ_f is its final angle, and T is the total number of seconds to move from angle θ_0 to θ_f. They may assume initial and final joint velocities are zero. The procedure should return the coefficients of the cubic polynomial (a function of time) that fits θ_0, θ_f, and T, (i.e. the cubic polynomial P such that $P(0) = \theta_0$ $\dot{P}(0) = 0$, etc.)

Next, students are asked to write a procedure that takes the coefficients and number of seconds T as parameters, and calls MOVE_ROBOT as in the Experiment 1A. Note that the "do-loop" now iterates on steps—not seconds. To call MOVE_ROBOT, the students need to calculate the velocity, as well as the position.

□ Part C: Using two Joints

• **Objective:** To familiarize the students with moving two joints together.

• **Preparation:** Once the students have figured out the previous experiment, writing the code for this section should be straightforward. Thus no prior preparation is necessary.

• **Procedure:** Students are asked to write a procedure to take as input four angles and a time T. The first two angles are the initial and final angles for joint 1 and the final two angles are the initial and final angles for joint 2. Assuming that the initial and final velocities are 0, the procedure fits a cubic polynomial to each joint. Using these polynomials the program moves both joints (together) in time T.

□ Part D: Inverse Kinematics

• **Objective:** In this experiment the students familiarize themselves with inverse kinematics.

■ Experiment 1: Introduction to ED I

□ Part A: Trajectory of sin Function

• **Objective:** Introduction to the system, consisting of the robot, its actuators, its control units, the interface, and the computer system. The students are shown some demonstrations illustrating how to read feedback signals and to command joint movement of the robot. They then perform some simple experiments to familiarize themselves with basic robot operation.

• **Principles of Operation:** The robot and the system students use in this experiment is explained in a supplementary handout. (See sections §2 and §3.) This includes some description of the ED I robot, how to command the robot to move, and how to read the position sensors of the robot.

• **Preparation:** Students are asked to prepare the code described under the *Procedure* section ahead of time. In our experience, even if this code is not perfect, it still saves considerable amount of time in the lab if the students had to start from scratch.

• **Procedure:** Students are asked to write a procedure to take as input values for M, ω, and T, where M and ω are floats and T is an integer and plan a trajectory for the JOINT1 determined by the equation

$$\theta(t) = M \sin \omega t,$$

where t is the time which increases by units of size STEP and $\theta(t)$ is the value for the angle of JOINT1 at time t (see the sample code in the lab manual).

To generate the sin trajectory, they first convert from T seconds to `STEPS_PER_SEC` $*T$ steps. The function to move the joint takes a new angle approximately every 5/1000 seconds to ensure smooth motion. Next, they calculate the new angle at each step using the formula

$$\theta_i = M \sin \omega i.$$

The procedure must carefully check the values of M and ω. Before they call any functions to move the arm, the students were asked to run through their code and just print out the values of θ and $\dot{\theta}$ after each iteration of the loop. Consecutive values of θ should not differ by too much, and $\dot{\theta}$ should not be too large. This is to avoid putting large torques on the joints of the arm. Typically the students were asked to try $M = 0.8$ and $\omega = 0.3$ to get an idea of what types of values are safe. Note that as ω is the frequency and M the magnitude, when the value of ω is increased, the value of M must be decreased and vice-versa. Students were told to seek assistance from the instructor and lab assistants if they were unsure whether the chosen values are in the proper range.

In a "do-loop," they calculate a new value for θ at each iteration, using which they then calculate

$$\dot{\theta} = M\omega \cos \omega i.$$

Then, once their code is working correctly, the next version of the procedure calls MOVE_ROBOT(JOINT1, θ, $\dot{\theta}$) to actually move the arm. Also, the students are asked to store the actual position at each iteration in an array. The actual position is obtained by calling GET_POS(JOINT1). This returns the angle in radians.

```
7          while (1)
8          {
9             if(READ_SENSORS(&th1_act, &th2_act)
10              printf(``theta1: %hd  theta2: %hd\n'', th1_act, th2_act);
11            else{
12              printf(``Error in MOVE_ROBOT routine\n'');
13              break;
14            }
15         }
16      }
```

init_sensors()

Initialize the sensors. This sets the current configuration to the home position (i.e. $\theta_1 = 0, \theta_2 = 0$).

Editing and Compiling The `TURBOC` editor is called `tc`. It is menu driven, and very simple to use. The students can compile and run most of their programs from within the editor (unless, of course, their code contains inline assembly language). Or they may choose to compile and run outside the editor. There are also other editors available in the system: for instance, `vi`, `emacs` (which is `microemacs`), and `epsilon` (another `emacs`).

The command to compile C files is `TCC` (Turbo C Compiler), and to compile assembler files is `TASM`. If it is necessary to compile more than one file, (eg. `tcc file1 file2`), the executable will be called after the first file, with an `exe` extension (eg. `file1.exe`). The students may instead choose to use Makefiles. Since it is necessary to link together more than one file, it is a good idea to set up a makefile and take advantage of the incremental compiling features. There is a sample makefile in the `\tc\programs` subdirectory:

```
1          OBJECTS=..\procs\rdsnsr.obj ..\procs\robmove.obj
2
3          .c.obj:
4             tcc -c $<
5
6          .asm.obj:
7             tasm $*
8
9          sensr.exe: ..\procs\rdsnsr.obj readsens.obj
10            tcc -esensr ..\procs\rdsnsr.obj readsens.obj
11
12         testmove.exe: testmove.obj
13            tcc testmove.obj robot.lib
14
15         moveread.exe: moveread.obj $(OBJECTS)
16            tcc moveread.obj $(OBJECTS)
```

This **makefile** was constructed to show some of the capabilities of the makefile program. The library `robot.lib` contains all of the procedures declared in `robot.h`, so one only needs to specify `robot.lib` (as was done for `testmove.exe`).

```
1       #include <stdio.h>
2       #include <math.h>
3       #include <robot.h>
4
5       sintraj(w, M, T)
6              float w, M;
7              short T;
8       {
9         int i;
10        float vel, pos;
11        short *th;
12        th = (short *) malloc((T/STEPSIZE) * sizeof(short));
13        for(i=0;i<(T/STEPSIZE);i++)
14        {
15          th[i]= (short) (M * sin(w * (float)i));
16          vel = (short) (M * w * cos(w * (float)i));
17          if(!MOVE_ROBOT(JOINT1, th[i], vel)){
18            printf(''Error in MOVE_ROBOT routine\n'');
19            break;
20          }
21        }
22        pos = GETPOS(JOINT1);
23        printf(''Final Position: %f \n'',pos);
24      }
```

This program computes $T/STEPSIZE$ values for the angle of JOINT1. JOINT1 is following the following trajectory in space:

$$\theta(t) = Msin(\omega t) \quad t \in [0, T]sec.$$

MOVEROBOT requires an angle and a velocity so we also compute *vel*, the desired velocity of the joint, which is equal to the derivative of the desired position. We then call MOVEROBOT each time through the loop. After we complete the loop, we return the current angle of JOINT1. We store the desired angles in *th*, although we do not make any use of this array this time.

READ_SENSORS (sens1, sens2)
float *sens1, *sens2;

This procedure reads the position sensors to obtain the actual values of the two joints. It is located in the file rdsnsr.c. The procedure **read_sensors** takes two parameters, both of which are pointers to **short**, and it returns the actual values of θ_1 and θ_2 into the variables. Below is an example of a program which continuously outputs the sensor values:

```
1       #include <robot.h>
2
3       main()
4       {
5          float th1_act, th2_act;
6
```

4.1.5 Commanding the Robot

Most of the C code is used to generate desired positions. Much of the low level `VxWorks`tm code has already been written for the students and is fairly stable at this point. The students write their procedures in C, which calls the low level routines. In order to use the robot procedures already written, the students simply include the file `robot.h` along with the standard include files (e.g. `stdio.h`, `math.h`, etc.). Following is a standard header:

```
1       #define DEG_TO_RAD (M_PI / 180)
2       #define RAD_TO_DEG (180 / M_PI)
3
4       extern void MOVE_ROBOT();
5       extern void INIT_SENSORS();
6       extern void READ_SENSORS();
```

The first two lines contain constant values needed to convert from degrees to radians and back again and are needed to compute forward and inverse kinematics (for instance, **sin**, **cos**, and other trigonometric functions expect their values in radians).

The three procedures declared are described below:

MOVE_ROBOT (joint, pos, vel)
short pos, vel;
short joint;

The procedure MOVE_ROBOT is a low level routine which takes three parameters, all three are of type **short**. *pos* is the desired *theta* for the joint and *vel* is the desired velocity. This is the procedure that sends the motion commands to the robot.

The MOVE_ROBOT program can be written using the joint_move routine of §2 as follows:

```
1       #define JOINT1 1
2       #define JOINT2 2
3       #define JOINT3 3
4       #define JOINT4 4
5
6       int MOVE_ROBOT(joint, pos, vel)
7               short pos, vel;
8               short joint;
9       {
10         double cur_pos, time;
11
12         cur_pos = GETPOS(joint);
13         time = (cur_pos - pos)/vel;
14
15         if(joint == JOINT1)
16            return joint_move(pos, 0, time, 0);
17         else if(joint == JOINT2)
18            return joint_move(0, pos, time, 0);
19         /* ETC. FOR OTHER JOINTS */
20      }
```

Below is an example of a C program which calls the MOVE_ROBOT procedure.

The Computer: In generating robot motion, one commands the robot to go to a *desired* point or trajectory within the workspace. The desired motion may either be specified directly by the user, or may be generated by a computer program. The computer program is written in the C high-level language. The computer then commands the robot to move.

In addition, the computer also monitors the robot motion and adjusts for any deviations from the desired position. By checking the values of sensors incorporated into the joints of the robot, the computer "knows" the *actual position* or the robot. It compares the actual position with the desired position to determine the error, and then adjusts accordingly. This forms the so-called "feedback control loop" for the robot.

The Interface: The interface is basically the connection between the computer, which is controlling the robot, and the robot itself. It allows the computer read the sensor values and to send motion commands to the robot.

The interface is composed of Analog-to-Digital conversion devices (A/D's) and Digital-to-Analog conversion devices (D/A's) which communicate with the computer and robot through a bus. When the computer wishes to send a motion command to the robot, it must first convert the motion to an *encoder* value, and then sends the value to the D/A which converts the digital value to a current or voltage, since the motor can receive only analog signals. When the computer wishes to read the sensor values (to determine the actual position), it reads the value from the A/D. The sensor uses encoder values: it transmits a number of pulses proportional to the angle of the joint. The signal is then converted to a digital value by the A/D and it can then be read by the computer.

The Servo-pack: The *servo-pack* is a power amplifier which is needed to amplify the signal produced by the interface in order to send it to the motor. This is somewhat similar to the way the accelerator pad in a car sends signals to the engine.

The Sensors: There are two types of sensors used in this system. One type, mentioned above, is the *position sensor* which is used to measure the actual position of the joint. Often, it is also called the *feedback sensor* since it sends information about the environment back to the computer. The other type is the *safety* and *home* sensors. The safety sensors determine when the robot has overrun its limit in the workspace. When the robot approaches the edge of its workspace, the sensor is triggered, and the output signal will automatically cut off the motor power, and the robot will stop. The home sensor is the device used to measure the motion origination position.

4.1.4 System Software

The students write all their software for ED I using TURBO C. The TURBO C directory is called `\TC`. This directory contains the compilers and editors, as well as additional procedures that can be called from students programs. Within the `\TC` directory, there are two important subdirectories: One is called `procs`, and it contains procedures which the students call from their own programs. The other is `programs`, and it contains several example programs. The low level details of maintaining the real time nature of the control is hidden from the students

The program for controlling the robot is written in a combination of C and calls to `VxWorks`tm real time system which controls the *creonics* motor control boards. The students are expected to follow a given set of programming guidelines to ensure that their programs do not lead to a hazardous situation. (See the box on page 36).

Safe Programming Practices (Contd.)

- If a function is returning values via arguments (i.e. by using pointers), you should set these return values to some meaningful default value if the function fails (i.e. if anything goes wrong). This is just another safety feature to incorporate into your code.
- Now suppose you need to check several conditions for potential failure inside a function. Your function should be coded such that all successes are handled first, and you return after a success. Then all failures can be handled together at the end of the function. The rationale behind this is that there may be a bunch of things that you need to take care of in the event of failure, and the easiest thing to do is handle all the failures together.
- Any non-trivial sequence of operations that is used several times in your code should be replaced by a function or a macro. In general, use a macro where you can express the operation in one line. For example, the function to square a number should be implemented as a macro. When using macros, keep in mind that your code will be put in-line as-is. This can cause problems if you haven't parenthesized your expression properly.
- Use structures and arrays where applicable to avoid having to handle several related variables separately. For example, if the robot has 3 joints, you may want to pass the associated joint angles around in a 3 element array. This also helps avoid the pitfall of confusing two variables such as `JOINT1` and `JOINT2` by a typo.
- Use data abstraction where applicable. Basically, data abstraction consists of building data structures to hide the underlying structure of your data, and then manipulating these data structures only through a limited number of functions.
- Try to make your functions as general as possible. It is a little tempting to go overboard with this, so as a rule of thumb, this should be something that makes your life easier—not more difficult. In other words it should allow you to reuse code—it shouldn't force you to right more code than you otherwise would.
- Either initialize all variables when you define them, or *make sure* they have values assigned to them before they are used.
- Test your code out before trying it out on the robot.
- But, finaly remember that none of these rules are etched in stone (except where safety issues are involved), so use your judgment when programming.

BOX 4.2. Instructions regarding safe programming style provided to the students.

Safe Programming Practices

Here are some things to consider when writing code for the ED I robot (actually, lots of this stuff applies to programming in general):

- Every function should return a value indicating whether or not it has succeeded. A failure should return 0 and in most cases, success should return 1. Then, when you call a function (say, a function called `foo` with 3 arguments), your call should be of the form

```
if(foo(var1, var2, var3))
  /* do something */
else
  /* something went wrong - handle it */
```

  This is to insure that if any function fails, you will be aware of it and not try to make the robot do something "bad".
- You shouldn't have any numeric constants hard-coded into your program. All constants should be `#defined` at the beginning of your program. Furthermore, anything you `#define` should be all caps.
- Your code must be well commented. Here are some guidelines:
  - There should be a comment associated with each `#define`.
  - Each function should be proceeded by a paragraph of comments explaining what its arguments are and what the return value is.
  - Comments should be embedded within a function where needed to explain anything that is not obvious from the code. For example, suppose somewhere in your code you have `y = 2 * x;`. A comment of the form `/* y gets twice x */` is not very helpful, but some comments about why you are doubling `x` might be in order.
  - A comment should be associated with each variable declaration unless the name of the variable itself entirely makes clear its purpose. For example, a variable called flag is explanatory to some degree, but you should include a comment to describe what it is "flagging".
  - Just keep in mind that you may have to look at the code in the future. Anything that would be helpful in understanding the code should be included in the comments.
- Choose meaningful names for variables. Not only should the names be descriptive, but they should also be such that you won't easily confuse two variables by making a typo, for example. Confusing two variables can really be a safety hazard!

For the purpose of the first experiment, the students operate only the first two joints. In later experiments, they use the full functionality of all the joints. So for the first lab, the manipulator can be considered a two-link planar manipulator. A simplified picture of the system controlling one joint (say θ_1) is as follows. The origin of the base frame of the robot is at the θ_1 joint.

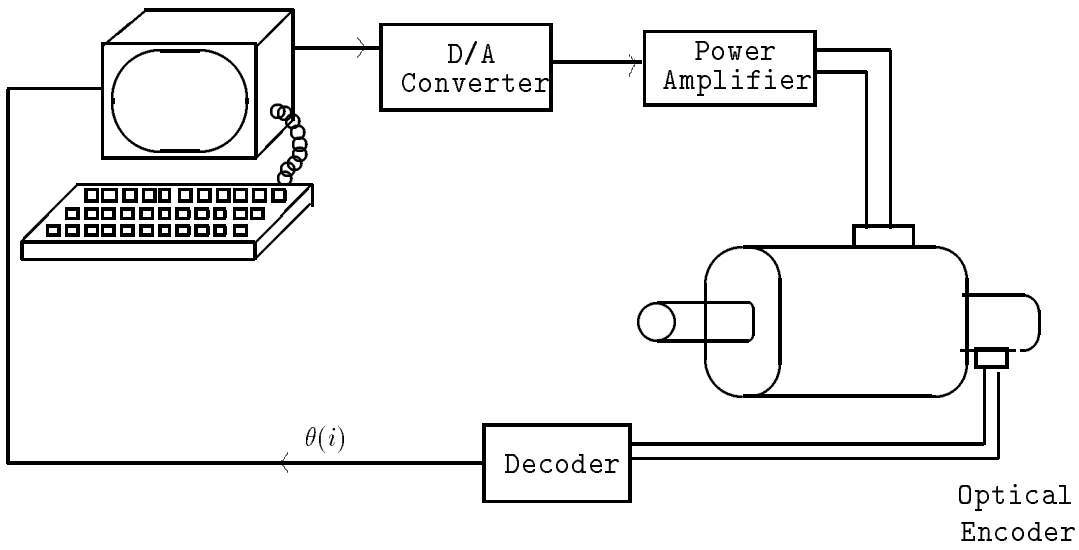


Figure 8: **A simplified picture of the system controlling one joint**

4.1.3 The controller

The robot is controlled by a digital computer, an interface, a servo-pack and sensors. A simplified diagram of the system is shown below.

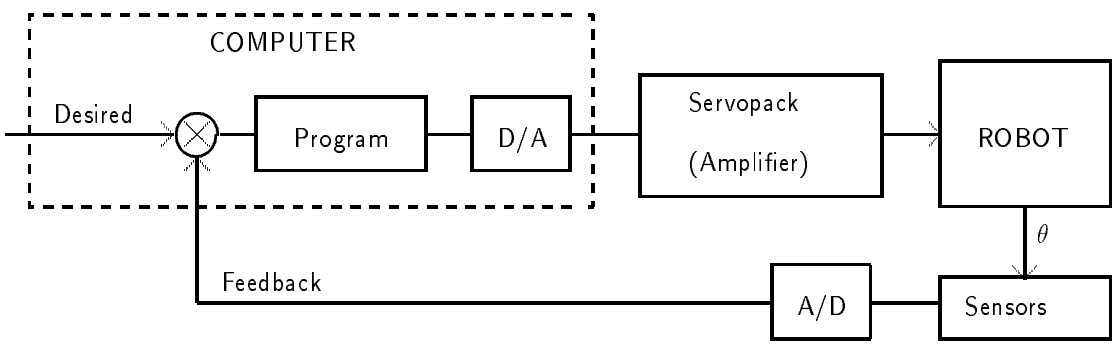


Figure 9: **A simplified picture of the controller**

Safety Instructions

Safety precautions should always be observed, since unpredictable failures can occur during operation. If at any time, you think the system is not functioning properly, alert the lab instructor at once. Additionally, the following practices should be observed:

- The first movement commanded by a program should be at no more than 20% of the full speed.
- Before instructing the robot to move, be sure *nobody* is in the workspace of the robot. In general, unless you are positioning something for the robot to use, you should never stand in the work area of a robot.
- No program should allow the robot to remain stationary for a long period of time, say, more than 20 seconds, and then move without warning. Warning can be given by:
  - Requiring an operator to enter a response at the console;
  - Moving at very low speed for a few seconds before increasing speed to normal;
  - Sounding a beep for 3 to 5 seconds before motion begins.
- In an emergency, use the **STOP** button on the control panel to stop the manipulator.
- Debugging should be done with very careful attention to the **motion** of the robot. The robot is very predicatable; it executes the commands given by the user programs. This, however, is not always what the user intends. If the robot does even the smallest thing you don't understand, it is probably a bug, and you should investigate it.
- Bear in mind that the robot can damage itself relatively easily, so run programs in their early stages of development slowly and with careful and constant attention. Avoid any actions which may endanger any people (including yourself!) in any way while using the robot.

Box 4.1. Safety instructions provided to the students.

4.1.2 The manipulator

The robot used by the students in this course is the NYU Direct Drive ED I robot. The manipulator is a two-jointed arm structure with four degrees of freedom arranged in the well known SCARA arrangement. The joints of the arm, θ_1 and θ_2 provide two degrees of freedom. The end-of-arm rotation, provides two more degrees of freedom: one through rotation, ρ, and one prismatic along the z-axis.

4. Experiments

4.1 Basic Robotics

4.1.1 General Information

This laboratory section for the "*Introduction to Robotics*" course is designed to assist the students in learning the fundamental robotics concepts. While an accompanying manual is provided to guide them in performing the experiments listed here, the students were asked frequently to also try things not specifically mentioned in the manual.

Students were also encouraged to consult with the instructor, the lab assistant and among themselves, when they were unsure of what is expected in the manual. However, it was expected that students will satisfy their curiousity and creativity, while adhering to the standard safe laboratory practices. (See the box on page 34.)

Each student is expected to carry a *Laboratory Notebook*, for the purpose of writing down everything that they do in the course of each experiment. They are asked to record all observations and data in the notebook and to label every observation taken in the laboratory.

The students are divided into several groups with 3 to 4 students per group. Each group is required to submit a 5–10 pages laboratory report one week after the completion of the experiment. (Most of the experiments take about two weeks). The report is expected to be either typed or written up in neat, clear handwriting with every diagram and graph/plot clearly labeled.

The report submitted by the students consists of the following sections:

1. **Introduction** (1–2 paragraphs).This is a statement of the purpose of the experiment, i.e. what the student set out to do.

2. **Theory of Investigation** (1 page).The basic theory underlying the experiment.

3. **Method of Investigation and Results**.This is the main body of the report. In this section the students explain in detail what it is that they did, and what the results of each step were.

4. **Analysis of Results** (1 page).At the end of each experimental description, the students are provided a *Summary* section, with questions which they are expected to answer in this section of the report. They relate to interpreting the results: what the students learned in performing this experiment, why the experimental results may not be what was expected, etc.

The experimental system differed from the developmental system in one fundamental way: The student experiments were conducted using an IBM PC and a TURBO C package running on it. Thus the system used by the students is significantly different from the development system using a Unix workstations (with M68020 cross-compiler) over the ethernet and described in the preceding section. The main advantage of the IBM PC and the TURBO C system was that the students were familiar with these systems and could take advantage of the NYU's academic computing centers to develop their programs outside the robotics lab. The hardware architecture of our system is such that the switch from one configuration to another remains relatively easy.

The interaction of the queues and of the semaphores is best represented graphicallly. See the figure above.

The lines in the schema show the flow of commands as they are queued by the listener (`queuer` and where they may be stopped by the semaphores.

The synchronization among the various components and commands is done by the three message queues (each one for the three kinds of commands) and by a set of semaphores. Currently there are the following semaphores[12].

- A **dispatch semaphore** used by the dispatcher mainly as a wait primitive until the next command is in. Also the commands (and the task executing them) will have to wait if this semaphore is not green.
- A **axis semaphore** for each axes. The axis driver is not allowed to execute more than one command at a time.
- A **card semaphore** for each card on the VME Bus. Since there may be more than one tasks requesting the services of the card, this semaphore is used to regulate the access. The number of concurrent users is declarable in the configuration files.

The primary responsibility of the `VxWorks`tm is thus to dispatch the commands from the generic command queue (coming from the frontend) to the axes driver while ensuring that the semantics associated with *immediate*, *ordered* and *privileged* commands are correctly obeyed. The figure 6 graphically describes how this is achieved by semaphores and queues together with a simple scheduler.

[12] Some are binary MUTEX; some are counting semaphores.

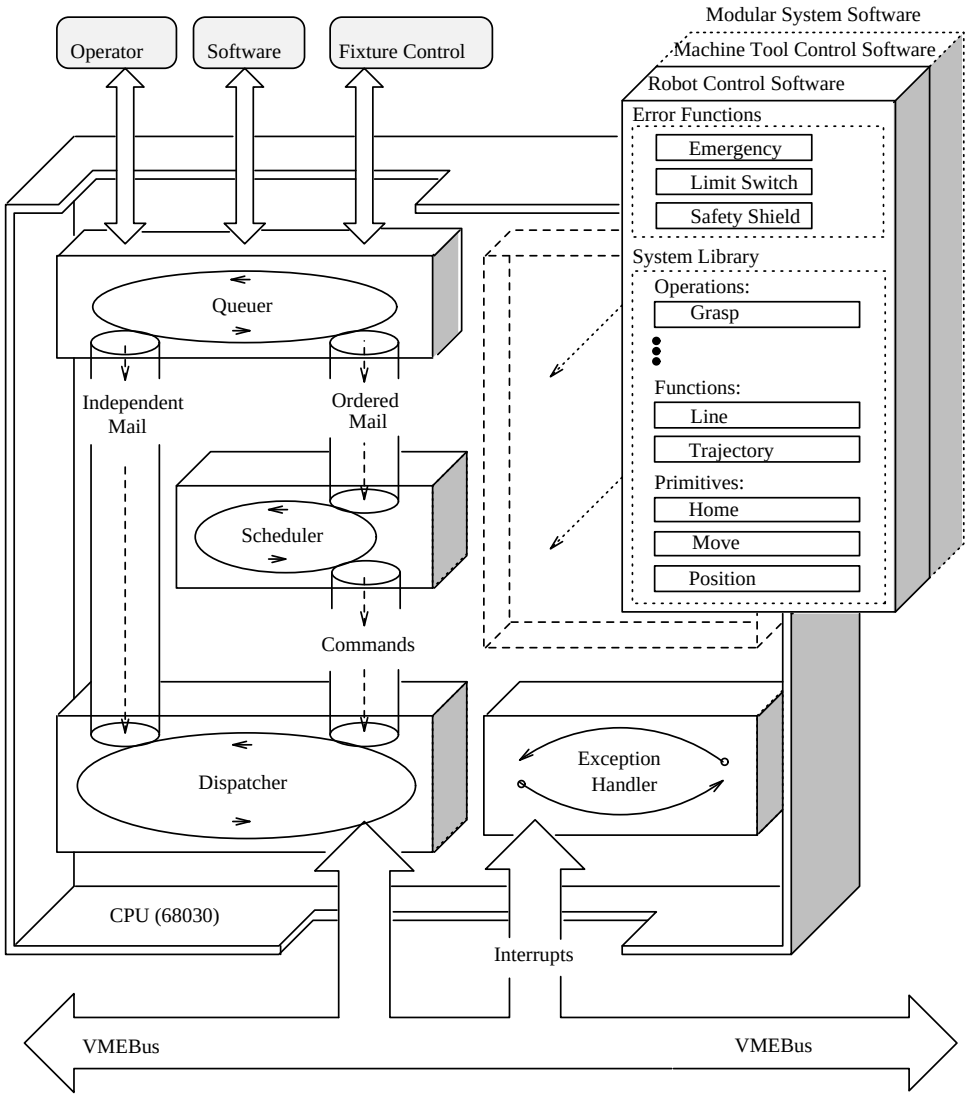


Figure 7: **The software organization of the MOSAIC/ED I system**

• **The Internal Queues and Semaphores:** The system relies its correct functioning on four queues and one "general" semaphore, plus a semaphore for each card and each axes. Both queues and semaphores serve to synchronize access to 'critical' resources. Moreover there is one basic assumption that must be made explicit:

> *The speed of the cards and of the* `VxWorks`tm *real time operating system, coupled with a* careful hand made *serialization of the tasks, is sufficient to avoid race conditions and disastrous events.*
>
> *Moreover, the actions the robot (or a machine tool) can perform are usually not constrained by* deadlines *as in a more "standard" real time scenario.*

This is taken for granted in the implementation of the system.

Each of the following lines is read and queues up in the immediate commands queue for direct interpretation by the card (via the `dispatcher` routine). The meaning of the first such line is (as interpreted from the Creonics manual [8]).

Code	*Description*
`9`	Number of data entries on this line (each to be read and later interpreted as a single byte)
`0x11`	Card axis identifier
`0x07`	Creonics code for *direct commands*
`0x00`	Code for setting the *servo loops gains*
`0x2d 0x00`	Proportional Gain
`0x00 0x00`	Integral Gain
`0x00 0x05`	Velocity Gain

The second line is similar. It is just used to set the initial values of the *home position* of the axis (codes `0x07` and `0x01`. The format of this entry is

Direct Command Code	Command Type	Four Bytes (units in edges [8])
`0x07`	`0x01`	`0x27 0xfa 0xff 0xff`

The remaining commands in the axis initialization file set the following parameters:

- *Home* and *Overtravel* mode,
- *Maximum Travel* limits (two lines),
- *Maximum Velocity*, *Acceleration* and *Deceleration*,
- *Position Error Tolerance*,
- *Deadband Compensation*,
- *Feedforward Gain*,
- *Encoder Mode*,
- *Axis/Drive type Specification*,
- *In-Position Tolerance*,
- *Velocity Error Tolerance*,
- *Unwind Constant*,
- *Servo Output* limit and *PAC Constants*.

A line containing only a `0` marks the end of the *usable* section of the file.

At this point all the "immediate" and "ordered" commands declared in the configuration file can be used.

3.2.4 The Internal Organization

The commands in the initialization script cooperate in getting the system up and running. Here is a brief description of what it is done by them.

- **Behavior of `mosaic_init`:** After reading the configuration file, the procedure `mosaic_init` starts three tasks that actually take care of the internal workings of the system.
  - *The* `scheduler` *task,*
  - *The* `dispatcher` *task, and*
  - *The* `error` *task.*

  The `scheduler` task is in charge of taking commands from the *command message queue*, checking that they fall in the "ordered" category, parsing them and "executing" them[10].

  The `dispatcher` task is in charge of actually communicating with the cards on the VME bus. Its main loop checks whether there are commands of the different kinds and executes them accordingly, provided that semaphores conditions do not prevent it to do so.

  The `error` task is presently only a message dumping routine that waits for messages on a specialized queue.

- **Behavior of `creonics_init`:** The `creonics_init` routine is in charge of initializing the Creonics card with some hardware dependent parameters. The whole process is achieved by sending an 'internal' immediate command to the dispatcher over the appropriate queue. The meaning of the command is to *initialize the driver*.

- **Behavior of `mcc_axis_init`:** The routine `mcc_axis_init` does two things. First it reads an initialization file for the axis it is initializing and then it spawns a task that executes the specialized driver for the axis itself, accordingly with the declaration contained in the configuration file.

  The initialization file for the axis contain data that sets some of the default parameters of the driver (see [8] for a complete reference). For example, the file `theta1_axis` contains the following: (Two of these files are loaded; one for each axis.)

```
1       ttheta1_axis 0 1 0x11 102400
2         9 0x11 0x07 0x00 0x2d 0x00 0x00 0x00 0x00 0x05
3         7 0x11 0x07 0x01 0x27 0xfa 0xff 0xff
4                          ...
5         0
```

  The line `1` contains the name to be given to the task in charge of driving the axis[11], followed by the axis number, the slot/card number, the `0x11` Creonics internal axis selector (...) and a parameter denoting the *initial position* of the axis.

  The name, the axis identifier and the card identifier will be eventually used by `mcc_axis_init` to spawn the correct axis driver (which was declared in the configuration file: that is the routine `mcc_axis_driver`).

[10] Each of the commands actually must be aware of some conventions that are used in the system. E.g. the commands operating on the axes should always start their actions with the functions calls `axis_cmd` and `axis_open`, etc.

[11] The double `t` is a `VxWorks`tm convention.

```
21      ORD coord :
22      ORD abort %A;#Aa:
23      ORD tune %A%c:
24      ORD joint_move %f%f%f:
25      ORD joint22_move %f%f%f:
26      ORD cartesian_move %f%f%f:
27      ORD home A;#Aa:
28      ORD joint_trajectory :
29      #
30      # Immediate commands
31      #
32      IMD at  taborttune 30
33      END
```

3.2.3 Starting the System

The whole system must now be started from the **VxWorks**tm *shell*[8] This is done in two steps[9]. First a script is loaded into the shell and afterward a specialized command interpreter is started (the **queuer** prioritized command of the previous section). The initialization file is listed hereafter.

```
ld < vme_util.o
ld < mail_util.o
ld < machine_util.o
ld < command_util.o
ld < mosaic_init.o
####################################
# Version 3.0 Of MOSAIC use caution #
#      for EDUCATION ROBOT 1        #
####################################
mosaic_init("ed1.cfg")
creonics_init(1,0xffffff00)
mcc_axis_init("theta1_axis")
mcc_axis_init("theta2_axis")
```

The lines containing with `ld <` ⟨*object file*⟩ tell the **VxWorks**tm linker/loader to place the exported symbols in the systemwide symbol table.

The last four lines in the script file launch different tasks that run concurrently in the **VxWorks**tm environment (which is composed of different cards on the VME bus; each of the cards potentially running concurrently a different task).

Each of the routines spawns a task that sets up various pieces of the whole environment, starts the "interpreter loops" and, most importantly, creates the queues and the semaphores used for synchronization.

• **Starting the Command Interpreter:** The commands can be given to the system by starting a simple command interpreter (as in the configuration file) by hand. The "programs" **queuer_util** and **queuer** must be started by typing directly to the **VxWorks**tm shell:

```
-> queuer
```

[8] The **VxWorks**tm "shell" act almost as a C language interpreter. Refer to the manuals for its inner working.

[9] All the steps are done by the **VxWorks**tm; no work is required on the part of UNIX.

• **Command Declaration Section:** The last section of the configuration file contains a declaration of the commands "acceptable" by the system. The commands are divided into three categories:

- *Prioritized commands* (or *privileged*)
- *Ordered commands*
- *Immediate commands*

The distinction among the different kinds of commands is made depending on the particular roles they play within the real time environment.

Prioritized commands: are commands that must be actually executed at the highest priority. They usually include the initialization routines for the various drivers running on the cards of the VME bus. Moreover, they also declare "generic" library routines that have to be loaded in the `VxWorks`tm kernel in order to run the system[7].

Ordered commands: are those commands whose execution should be terminated (if no error occurs) before other ordered commands are started. "Move" commands are typically in this category. The bottom line is that this kind of commands is *queued* for execution.

Immediate commands: are commands that can be executed immediately by the real time kernel upon "completion" of the current task. I.e. they can bypass the usual queue. Information reading commands can usually be run in this way.

The standard format for an entry in this section is:

command type	*command name*	*arguments*

The *command type* field can be one of `PRI`, `ORD`, `IMD` or `END`. The actual form of this section is listed hereafter.

```
1       #
2       # Acceptable Commands declaration section
3       #
4       # Truly "prioritized" commands
5       #
6       PRI creonics_init
7       PRI mcc_axis_driver
8       PRI mcc_axis_init
9       #
10      # Actually libraries dynamic loading
11      #
12      PRI kinematics
13      PRI queuer_util
14      PRI queuer
15      #
16      # Ordered commands
17      #
18      ORD on  %A;#Aa:
19      ORD off  %A;#Aa:
20      ORD pos  %A;#Aa:
```

[7] This is actually a trick in the current state of the implementation. A new category (possibly **library commands**) should be introduced.

• **Machine/Axes Declaration Section:** The *machine and axes section* contains a line containing the "name" of the machine followed by the declaration of the axes. The declaration of the axes present on the machine is done on lines having the following format:

Field	**Description**
initial status	A bit-vector integer used to encode various state informations regarding the card
control	
position limit	The *positive* limit reachable by the axis (expressed in degrees)
negative pos. limit	As above but in the "negative direction".
velocity	Velocity
acceleration	Acceleration
deceleration	Deceleration
last position	Initialized to a "home position" of `0.0`
target position	As above
axis coordinate	A single character denoting the actual axis coordinate
identifier	The name actually given to the axis (used mainly for printouts)

The actual entries in our sample configuration file is

```
1       #
2       # Axis declaration/definition section
3       # EDUCATION_ROBOT
4       00 0 180.0 -180.0 180.00 150.0 150.0 0.000 0.000 x theta_1
5       00 0 180.0 -180.0 180.00 150.0 150.0 0.000 0.000 y theta_2
6       00 0 0.00 0.00 0.00 0.00 0.00 0.000 0.000 - END
```

Our particular configuration file contains two axes called `theta_1` and `theta_2` (lines `4` and `5`). The last line (`6`) with a fictitious axis named `END` signals the end of the section (unfortunately all the previous entries must be filled in).

• **Sensor Declaration Section:** This section describes each sensor in the system. Its entries are of the following form:

initial status	*sensor identifier*

The actual section is

```
1       #
2       # Sensor Declaration Section
3       #
4       00 END
```

I.e. no sensors are currently attached to the ED I robot. The dummy sensor identifier `END` signals the end of the section. Thus at present the user has access to the joint position informations only.

3.2.2 The Configuration File

We begin with the description of the initialization process, i.e., how the software starts up and builds for itself a *representation* of the diverse components. The initialization routine (and the entry point for the whole "program") begins by reading a *configuration file* that contains a description of various components of the system and creates their representation. At the end of this process, the software is ready to operate in its normal mode and in a safe state.

The configuration file, `ed1.cfg`, for the ED I robot system is divided into the following four sections:

- *Card Declaration-definition Section.*
- *Machine/axes Declaration-definition Section,*
- *Sensors Declaration Section.*
- *Commands Declaration Section.*

• **Card Declaration Section:** The *card section* has entries (one per line) denoting all the card that are actually lined up on the VME bus. Each entry has the following format:

card–number	*accesses–number*	*SW driver name*

The actual content of this section is

```
1       #
2       # Card declaration/definition section
3       #
4       0 99 Null_Driver
5       1 1 mcc_driver.o
6       -1 99 Null_Driver
```

The card having *card–number* equal to `0` denotes the 'main processor card' (line `4`); in our case a card containing a MC68020 running the `VxWorks`tm kernel. The name `Null_Driver` tells us that no software driver is actually running on the card (since this card actually runs the OS). This particular card has also an *accesses–number* equal to `99`, telling us that the driver associated (none in this case) will accept up to `99` concurrent "accesses" by different "tasks" (in the `VxWorks`tm terminology).

The line `5` denotes the card numbered `1` on the VME rack and it is actually used to run the axes of the machine. The *accesses–number* of `1` tells us that only one task will be performed at once by the card. The "performer" will actually be the routine `mcc_driver`, which is in charge of the very low-level details of communicating directly with the hardware.

The last line(`6`) is just a place holder. Its *card–number* equal to `-1` tells the initialization routine that the card declaration section is finished. As long as the *accesses–number* and the *driver name* "fields" are filled consistently (i.e. the reader routine does not signal an error) their content is nevertheless ignored. If the reading of the configuration file fails then the system does not start.

This function generates a sequence of *via points* (see [7]) in order to move the end effector of the manipulator on a straight line in the XY plane. The movement is done in a duration specified by `time`. If the `mode` parameter is equal to `CARTESIAN_SPACE` (1) then the values of `pt1` and `pt2` are intended as a point in the physical space XY plane, otherwise (`mode` is equal to `JOINT_SPACE`) they are intended as a final position in joint space. Kinematics computations are executed accordingly.

The value returned is `OK` if all the computations terminated correctly, `ERROR` otherwise. Once again, if `ERROR` is returned, most likely the end point $\langle$`pt1`, `pt2`$\rangle$ or one of the generated via points is outside the manipulator workspace and the kinematics routines failed to complete. See the error message printed on the console for more information.

▶ home routine:

```
void home(char c)
```

This function performs the *homing sequence* specified at a very low level by the motion control hardware[5]. The argument `c` is used to specify for which axis the homing sequence is to be performed. No value is returned.

3.2 Low Level Software Architecture: MOSAIC/ED I

The ED I system is based on the architecture of the MOSAIC open architecture manufacturing control system. As explained earlier, its main components are a UNIX workstation and a `VxWorks`tm Real-Time subsystem, which are directly in charge of the special Creonics hardware cards that control the robot axes. The UNIX workstation is used mainly for cross-development, but it could also be used as a *client* for a `VxWorks`tm *server*.

The `VxWorks`tm real time system loads and runs (see [37]) a set of concurrent routines, which are synchronized by means of traditional (i) *semaphores* and (ii) *message passing*. The real-time nature of the system is obtained by means of a prioritization scheme (i.e. privileges) and by supporting a wide variety of styles of command executions. We will defer the discussion of how the MOSAIC/ED I system takes advantage of these structures, while keeping the details hidden from the user. We begin by looking at how the complete MOSAIC/ED I system is put together: namely, its main components, their software representation and how these components interact within the `VxWorks`tm kernel.

3.2.1 Main Hardware Components

The ED I system is composed of the following abstractions of hardware components:

$$\textbf{A Machine} \implies \begin{cases} \textbf{A Set of Axes} \\ \textbf{A Set of Sensors} \end{cases}$$

Axes and sensors must be controlled by special *hardware cards*[6], which run special *software drivers*. Therefore we can see a machine from another viewpoint as

$$\textbf{A Machine} \implies \textbf{A Set of Cards} \implies \textbf{A Set of Software Drivers}$$

Using the above model, we can now explain the architecture of the ED I system as it is implemented on the `VxWorks`tm platform. Our exposition also closely follows the actual programming structure and can be viewed as a *code-walk*.

[5] Basically the axis is moved until it hits the hardware limit switches and it is slowly moved off them before stopping. This functionality and its various settings are provided by the Creonics Motion Control Card.

[6] The current set up has two Creonics cards capable of controlling two axes; the first controls the x and y axes, while the second will control the z and the ρ (rotational) axes.

▶ `coord` routine:

```
void coord()
```

This function resets the origin of the coordinate frame to the current position in Joint Space of the manipulator joints.

▶ `arm_forward_kinematics` routine:

```
int arm_forward_kinematics(double theta1,
                           double theta2,
                           double* x,
                           double* y)
```

Given the point in Joint Space $\langle$`theta1`, `theta2`$\rangle$, the function returns in `x` and `y` the coordinates of corresponding point in Cartesian Space.

▶ `arm_inverse_kinematics` routine:

```
int arm_inverse_kinematics(double* theta1,
                           double* theta2,
                           double x,
                           double y)
```

Given the point in Cartesian Space $\langle$`x`, `y`$\rangle$, the function returns in `theta1` and `theta2` the coordinates of (one of) the corresponding point(s) in Cartesian Space.

`ERROR` is returned if the computation did not terminated correctly, i.e. the point was not in the workspace of the manipulator. Otherwise `OK` is returned.

▶ `joint_move` routine:

```
int joint_move(double theta1,
               double theta2,
               double time,
               int mode)
```

This functions moves the two main joints (θ_1 and θ_2) from their current positions to the one specified by the parameters `theta1` and `theta2` in time `time`. If the last integer parameter (`mode`) is equal to `ABSOLUTE_MOVE` (0), then the two joints will move to the position specified with respect to the current frame. If the `mode` parameter is `RELATIVE_MOVE` (1) then the values of `theta1` and `theta2` are used as increments to the current position.

The value returned is a code that specifies whether the operation completed correctly or not. A value equal to `OK` indicates correct termination. Otherwise `ERROR` is returned.

A return value of `ERROR` usually indicates that the kinematics computations could not be completed, i.e. the selected point is outside the manipulator workspace.

▶ `stroke` routine:

```
int stroke(double pt1,
           double pt2,
           double time,
           int mode)
```

Functions available to the programmer	
`on`	Turns on the power for the ED-I motors
`off`	Turns off the power for the motors
`pos`	Returns the position of a specified joint
`axis_status`	Prints out a summary of various values for a specified axis
`coord`	Reset the Coordinate frame to the current position as origin
`arm_forward_kinematics`	Computes the forward kinematics for the two main joints
`arm_inverse_kinematics`	Computes the inverse kinematics for the two main joints
`joint_move`	Move the two main joints
`stroke`	Moves the joints from the current position to a specified one interpolating a straight line
`home`	Performs the predefined *homing sequence*

Table 10: **Functions available for programming**

• **Available Functions:** In this section we provide a quick reference to all the functions currently available to a programmer. Table 3.1.3 contains a list of such functions. Each function will be described in more detail in the following. Argument types will be given in an ANSI C style.

▶ `on` routine:

```
void on(char c)
```

This function turns on the power for the motor of the axis specified by the mnemonic character passed to it.

▶ `off` routine:

```
void off(char c)
```

This function turns off the power for the motor of the axis specified by the mnemonic character passed to it (as in case of `on`).

▶ `pos` routine:

```
void pos(char c)
```

This function works by side effect. It prints on the console the actual position (in encoder edges with respect to the origin of the current coordinate frame) of the axis specified by the mnemonic character. Moreover, it makes sure that such value is recorded in the field `Target_Pos_In_Joint` of the relevant `AXIS_STRUCT`.

▶ `axis_status` routine:

```
void axis\_status(char c)
```

This function prints on the console in a human readable format the contents of the `AXIS_STRUCT` relative to the mnemonic character passed to it.

VxWorks^{tm} RT OS Features	
Manufacturer	Wind River System
Version	5.02b
Memory Model	shared space
RT Feature	Multitasking Preemptive Priority Scheduling and Priority Inversion
IPC Features	Semaphores, Message Queues
Connectivity	TCP/IP

Table 9: **VxWorks^{tm} real time operating system features**

Alternatively they can be invoked directly from the VxWorkstm shell (see [37] for details). This turns out to be very useful for debugging purposes.

• **The VxWorks^{tm} Real Time Operating System:** The core of the MOSAIC/ED-I architecture is the VxWorks^{tm} real time operating system manufactured and marketed by Wind River System [37].

Its characteristics as a RT OS are listed in table 3.1.3. An advantage of VxWorkstm is its tight integration with a UNIX workstation. In our case VxWorkstm runs on a Motorola MVME147s board and communicates with the UNIX host through a standard Ethernet connection. Programs are developed and compiled (but *not* linked) on the UNIX host and linked/loaded into the VxWorkstm memory through the network.

The VxWorkstm uses a shared memory model. The basic unit of execution is the *task*. Each task runs a C function as a separate entity in a shared addressing space. Synchronization primitives as semaphores and message queues are provided for basic IPC (Inter Process Communication) programming.

When VxWorkstm is first loaded (i.e. at the end of its boot phase) it starts a special session of its own *shell* on the system console or on a remote terminal[4] (most likely an `xterm` on the UNIX host running X-Windows).

The VxWorkstm shell acts almost like a C 'interpreter.' The object files produced in the cross-compilation phase must contain the symbol table for them. Once these object files are loaded into the VxWorkstm workspace, a function can be simply called by typing its name and arguments at the shell. VxWorkstm spawns a task for the function that executes to completion or until an error occurs. Of course the function may call the OS primitives to spawn other tasks (e.g. `taskSpawn`).

As an example, the function `stroke` defined in the beginning of this section could be invoked as

```
-> stroke(45.0, 0.0, 5.0, 0)
```

where `->` is the shell prompt.

Built in commands are available for directory inspection and, above all for debugging purposes. It is in fact possible to inspect the execution stack of the running tasks and to actually block them via breakpoints. As such the VxWorkstm shell also works as a limited debugging environment.

[4] There are ways to bypass this step and launch directly an application using the EPROM chips on the CPU board running VxWorkstm.

```
30
31   arm_inverse_kinematics(&final_theta1, &final_theta2, x, y);
32
33
34   /* Now compute the sequence of knots needed to */
35   /* approximate the straight line in cartesian space. */
36
37   /* Initialize the values for the points already obtained. */
38   /* ...omitted... */
39
40   taylor_bdev(init_theta1, init_theta2,
41               final_theta1, final_theta2,
42               knot_sequence,
43               0,
44               MAX_KNOTS_NUMBER - 1);
45
46   knot_count = count_knots(knot_sequence);
47   new_time   = time / knot_count;
48   for (i = 0; i < knot_count; i++) {
49     joint_move(knot_sequence[i].x, knot_sequence[i].y, new_time,
50                ABSOLUTE_MOVE);
51     taskDelay(iround((time / knot_count) * TICKS_SEC * 1.5));
52     /* Delay to be sure that move has ended! The 1.5 factor is just */
53     /* a safety measure. Note, 'taskDelay' is a VxWorks primitive */
54   }
55   return OK;
56  }
```

The `stroke` routine is straightforward C code[2]. The only routines that need some explanations are `arm_inverse_kinematics` and `taylor_bdev`. The routine `count_knots` is obvious.

The routine `arm_inverse_kinematics` computes the Joint Space position of the manipulator, given two values in Cartesian Space. The values are returned in the first two arguments, which are passed 'by-reference.'

The routine `taylor_bdev` computes the via points in Joint Space (by calling also `arm_inverse_kinematics`) using the Taylor's algorithm, mentioned earlier. The resulting via points are stored in the array `knot_sequence` and are eventually fed to `joint_move`.

This is just a very simple example which produces a "stop'n'go" movement of the manipulator. The better version also produces values for velocities and accelerations and needs to set up a special communication link with the low level hardware.

3.1.3 How Programs Get Executed

Once the programs are written and compiled into an object file[3], they can be loaded into the `VxWorks`tm address space. A simple machinery (explained in the next section) links them to a parser that can be used to invoke them.

[2] The comments of the form `/* ...omitted... */` indicate omitted code and is done to avoid clutter. E.g. most error checking code is left out from what is shown here.

[3] In reality, the programs are cross-compiled for the 68020 architecture.

will move the joints to move to the (Joint Space position) $\langle 90.0, 15.0 \rangle$.

There is an identical procedure for the other two distal joints, called `last_joint_move`.

- `on` routine: is used to power on a specified axis.
- `off` routine: is used to power off a specified axis.

The `on` and `off` routines are therefore viable means to operate the robot safely under program control.

3.1.2 An Example Program: the `stroke` Routine

In this subsection, we illustrate the natural style of C programming for the ED I robot, written using the library routines discussed earlier. The example, we have chosen for this purpose, is a rather simple routine that moves the end effector in a *straight line* in the physical space. The full listing appears in the appendix A.

The routine uses the `joint_move` routine and produces a simple "stop-and-go" movement through the *via points* (see [7] for terminology). The via points are generated by a version of Taylor's *bounded deviation* algorithm (see [25] for details).

```
1   /* stroke --
2    * Moves the manipulator from the current position (x-start, y-start)
3    * to the position x, y in the specified amount of time.
4    */
5
6   int stroke(x, y, time)
7        double x;
8        double y;
9        double time;
10  {
11   int i = 0;
12
13   double init_theta1 = Axis[0].Last_Pos_In_Joint;
14   double init_theta2 = Axis[1].Last_Pos_In_Joint;
15
16   double final_theta1 = 0.0; /* Dummy final values */
17   double final_theta2 = 0.0; /* Dummy final values */
18   /* ...omitted... */
19   int knot_count = 0;
20   knot_struct knot_sequence[MAX_KNOTS_NUMBER]; /* knot_struct, defined
elsewhere */
21
22   /* Functions used */
23   double compute_deviation();
24   double taylor_bdev();
25   int    count_knots();
26   int    joint_move();
27   /* ...omitted... */
28
29   /* First compute the Joint Space position of the final point. */
```

```
21        char Note[MAX_MSG_SIZE];  /* Used for communication with driver */
22      } AXIS_STRUCT;
```

One important field of the data structure above is the `Note` field, which serves as a buffer where low level commands are placed before being fed to the Creonics motion control card driver and where results are read from. How exactly this is accomplished by the `VxWorks`tm will be explained further in the next subsection.

Two other functions that manipulate the `AXIS_STRUCT` are `axis_cmd` and `axis_open`. They mask the operations on the semaphore `Busy`. The first one does a V operation on the semaphore, allowing other commands to "jump in", while the second one does a P operation, eventually blocking out other commands on the axis.

Since the axes of the robot are stored in a table a third function is useful, `find_axis`, which retrieves an "axis identifier" given a character mnemonic for it.

• **Other Major Routines:** In the current implementation there are a number of routines that can be used to write 'higher level' commands, or that can serve as a minimal protocol for the functioning of ED I robot arm. Following is a list of a few of the currently available low-level routines:

Routine	*Classification*
`joint_move`	Movements related
`home`	
`on`	System control
`off`	

► `home` routine: takes an axis identifier (a single character) and performs the *homing sequence* for it. After the homing sequence is completed (i.e. the axis has hit its limit switches), we have a firm reference point starting from which the axis movements can be accurately computed.

► `joint_move` routine: This routine takes four parameters

1. A (double) float value that specifies the "new location" of the first joint of the robotic arm (θ_1),
2. Another double that specifies the new location of the second joint (θ_2),
3. A (double) float that specifies how much "time" must be used by the movements,
4. A flag that specifies whether the movements must be *absolute* or *relative*.

As an example, if the robotic arm is in Joint Space position $\langle \theta_1, \theta_2 \rangle = \langle 0.0, 0.0 \rangle$ then the call

```
joint_move(45.0, 30.0, 5.0, 0);
```

will cause the joints to move to the (absolute) position $\langle 45.0, 30.0 \rangle$ in the current coordinate reference in Joint Space, in 5 seconds. The subsequent call

```
joint_move(45.0, -15.0, 5.0, 1);
```

We must warn that the above picture of the software organization is occasionally violated inside the intermediate level of `VxWorks`tm real time systems, primarily for the sake of occasional efficiency improvement. However, for the present document, we shall ignore these details in order to avoid confusion.

3.1.1 The Library Routines

Currently, executing a high-level robotic task on ED I (e.g. pick-and-place, tracking or peg-in-the-hole) involves simply writing C code sprinkled with library calls. Thus, the user accomplishes all of the numerical computation, user interface, error handling, data structure organization and temporal ordering of the actuation and sensing using the primitives provided by the C language, a set of standardized data structures, and some related libraries (e.g. `math.h`). The compiled and linked code then automatically runs the robot in the intended manner.

In this subsection, we describe in substantial details, the library facilities available and illustrate their usage by a practical example. The example is taken from a student-project written at a preliminary stage of the software development. (See appendix A, for the full listing.) Complete details of the library will be provided in a separate Programmer's manual and will also be available on-line.

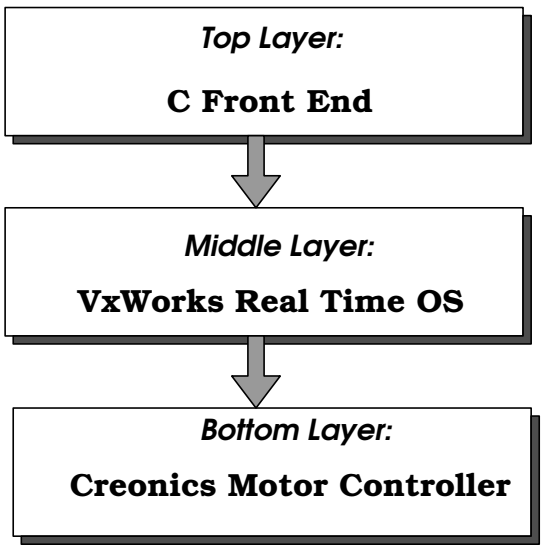
• **The `AXIS_STRUCT` Data Structure and Related Routines:** The most important data structure used is the one defining the characteristics of an *axis*. Its definition, shown below, is logically divided into four parts:

1. *Identification and general axis information*;
2. *Limits and hardwired constants for the axis structure* (These are read from configuration files);
3. *Physical status information* (e.g. current position, velocity, etc.);
4. *Software status information* (e.g. semaphores, flags etc.).

```
1       typedef struct axis_struct {
2         /* Identification and general info. */
3         char Id[IDENT_LENGTH];    /* The full name of the axis */
4         char Name;                /* A single character identifier for */
5                                   /* the same purpose as Id[]*/
6
7
8         /* Limits and constants */
9         /* ...omitted... */
10
11        /* Physical status */
12        /* ...omitted... */
13        double Current_Origin;    /* Origin of the current coordinate */
14                                  /* system, with respect to the 'original'. */
15        double Current_Pos_Limit;
16        double Current_Neg_Limit;
17
18        /* Software status */
19        SEM_ID Busy;              /* True if axis is ready for a command */
20        char Status;              /* Byte encoding of various info */
```

3. Software

In this section we briefly describe the structure of the software for the ED I robot, and explain some of the basic principles underlying the MOSAIC/ED I system. The system is composed of two components: (a) **frontend**: comprising of a C programming system with access to C library routines that perform various actuation and sensing operations; and (b) **backend**: a `VxWorks`^{tm} based real-time system that implements the actuation and sensing operations. In the future, we plan to implement a graphical interface that will operate above the current frontend and make the man/machine interface substantially friendlier.



The software for the ED I robot is organized in a layered manner. It consists of a top layer of C environment, visible to the users, where the programming of ED I is done by calls to routines in a C library. The middle level translates these calls to low-level commands for the Creonics controllers and is accomplieshed by the VxWorks real time systems. The lowest level executes these commands. All but the top layer is completely invisible to everyone except the systems developers.

Figure 6: **The software organization of the ED I system**

3.1 Programming the MOSAIC/ED1 System

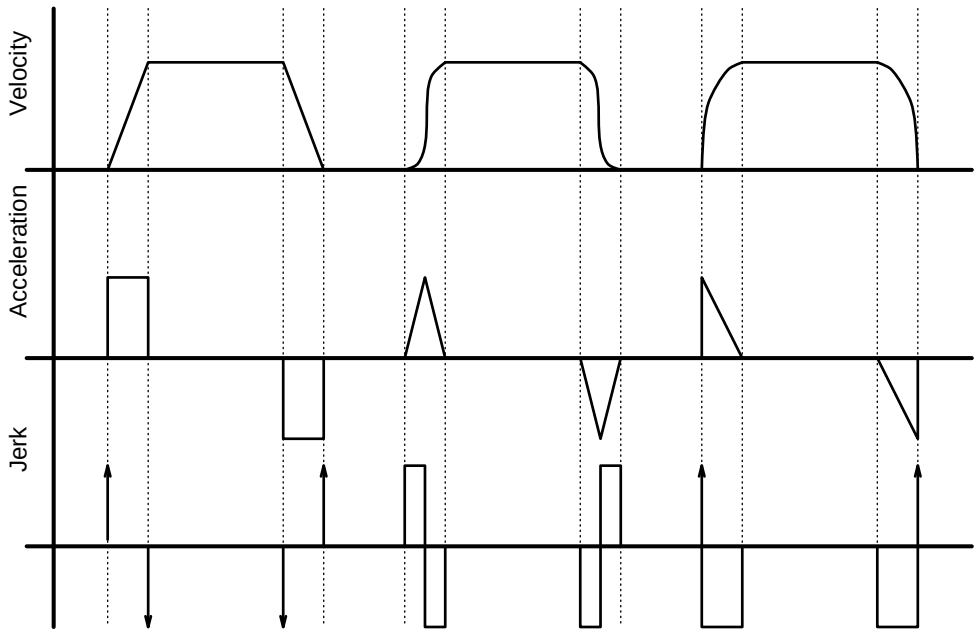
The MOSAIC/ED1 system is usually programmed in a high level language such as C or Pascal, where the low-level actuation and sensing operations are accomplished by a set of "library" calls. In this way, the intermediate level details of real time constraints and the very low-level servo parameters remain completely hidden from the novice users. The real time operating system `VxWorks`^{tm} handles the synchronization of and mutual exclusion among various low-level operations via internal queues and semaphores and thus provides the user the illusion of smooth concurrent operations of the axis actuators and the sensors; additionally, it ensures that these operations are executed in a timely manner. A detailed discussion of the intermediate level is discussed in details later in this section. The *low level axes driver interpreter* (see figure 3.2.4) then translates the commands dispatched by the `VxWorks`stm into still lower-level commands that are executed by the Creonics motor controller card (see the preceding section for more details).

The Interpolated Track command, like the Track command, allows direct control of the designated axis by the host processor, where the axis positions are controlled by incremental position values and the move time. Unlike the previous command, there is no direct control over the velocity/acceleration profiles; instead, the MCC interpolates intermediate points in order to accomplish a smooth motion over the given time interval.

A double buffering scheme is used so that the host can generate and update the next incremental position value even before the previous track motion is complete.

- **Abort Motion Command**

Immediately stops any motion in progress without disabling feedback—useful for safe execution of basic operations.



	Trapezoidal	*S-Curve*	*Parabolic*
Motion Stress	Fastest	2x slower	2x slower
Motor Stress	Worst	Better	Best
Mechanical Stress	Worst	Best	Better

Figure 5: **Profiles for Move command**

For most of the applications, the Trapezoidal profile is used, as it provides the fastest move for a given acceleration and deceleration limit. However, often S-curve profile is used, at the expense of the optimal speed, in order to minimize the stress on the mechanical system. Parabolic profile has the advantage as it matches the torque-speed characteristics of most motors and thus optimizes the motor stress.

The Move command can either specify the *distance* to be moved (relative/incremental) or the *endpoint* to move to (absolute) at a designated speed, acceleration and deceleration.

- **Home Axis Command**

Initializes the designated axis by enabling feedback and performing the predefined homing sequence.

- **Track Command**

The Track command allows direct control of the designated axis by the host frontend processor. In this case, the host generates the move profiles by controlling velocity and acceleration dynamically.

- **Interpolated Track Command**

Motion Control Card	
Manufacturer	Creonics
Model	VMEbus MCC
Microprocessor	Intel 80C186 @@ 12.5 MHz
Pulse Width Modulation Interface	Custom Motion Control IC
Control	Digital Feedback control with nested Proportional, Integral, Velocity and Feedforward servo loop
Sample and Update	1KHz for position and velocity loops
Positioning	32 bits $\pm 2.147 \times 10^9$ encoder edges
Velocity	32 bits, 0.015 to 1 million edges per second
Acceleration	32 bits, 15.259 to 3.28×10^{10} edges per second square
Encoder Edge Rate	1 MHz
Command Execution	2 msec or less
Data Transfer Buffer Access Time	460 nsec
Interrupt Acknowledgement	510 nsec
Power	1.5 amps @@ 5V, 200mA @@ $\pm$ 12V from VMEBus

Table 8: Creonics Motion Control Cards Characteristic

- Creonics provides an *exclusive automatic servo setup* which performs a quick and easy servo tuning, thus making the system ideal for the novice users.
- Its *direct tracking* capability allows complex motion profiles (for velocity and acceleration) to be computed and loaded by master CPU program in real time.
- Real time variable gain; four programmable software timers; packet type command and communication structure, etc.

The MCC's provide memory-mapped I/O by allowing indirect access to the 256 byte Command/Response Buffer that appear as eight registers to the VMEbus master. Additional access is also provided to various control and status registers. All *commands* and *requests* are transmitted via these registers.

Some of the frequently used *direct commands* that generate motion are discussed below briefly:

- **Move Command**

The Move command causes a motion of the designated axis using one of the following three profile types: *Trapezoidal Motion Profile*, *S-Curve Motion Profile* and *Parabolic Motion Profile*.

2.2 Electronic Hardware

The robot is operated by a digital controller which consists of a CPU (Motorola MVME147S), and two motion control cards (Creonics 2-axis MCC), one for each pair of motors (i.e. the one for θ_1 and θ_2 axis-pair and the other for z and *spindle* axis-pair). The CPU and the MCC's communicate via a high bandwidth (40 Mbit/sec) VME bus. (See figure 2.) The digital controller receives commands from a workstation running the interface (a real time operating system and a C/UNIX programming environment) and generates appropriate commands for the actuators (the joint motors). In addition to the actuators that the motion control cards control, we also have *servo amplifiers for the actuators*, *digital encoders* and *encoder and drive power supplies* in order to run a closed-loop servo control scheme on the motion control cards.

Clearly, the heart of the digital control of the robot resides in the Creonics motion control cards. The VME bus only facilitates transfer of data between the master CPU and the multiple "slaves" (MCC's) without contention by means of a central arbiter. The master communicates with the MCC's in order either (1) to cause motion, set and modify parameters, etc., using *direct commands* or (2) to request status information using *request for reply commands*. We shall not delve into a lengthy discussion of the VME bus or the Motorola master CPU here as additional informations can be found in [36] and [29].

Bus	
Manufacturer	Motorola
Model	VME
Bandwidth	40 Mbit/sec

Table 6: **VME Bus Characteristics**

CPU Board	
Manufacturer	Motorola
Model	MVME147S
Microprocessor	Motorola 68020 @ 20 MHz
RAM	4Mb
Connectivity	Ethernet and SCSI through a MVME718 module

Table 7: CPU Board Characteristics

2.2.1 Creonics MCC

The Creonics VMEbus Motion Control Card (MCC) is a microcomputer-based, 2-axis digital motion controller, designed to work in conjunction with the VMEbus standards. The Creonics MCC is ideal for ED I control applications as it provides for each axis a closed-loop control for point-to-point positioning with *velocity* and *acceleration* control. The closed-loop control uses a PID (proportional, integral and derivative) feedback control mechanism, thus providing enormous flexibility in choosing the dynamic behavior of the controlled system. The MCC interface also allows for incremental optical encoders that provide both position and velocity informations.

The additional advantages provided by the Creonics MCC for our applications are as follows:

4. Hall-Effect Sensor: for magnetic measurement purposes.

5. Camera: for computer vision and graphics applications.

• **Safety** In addition to the safety mechanisms necessary to protect the users, we also had to provide several mechanisms to protect the unit from self-inflicted damage. First of all, all the motors have been kinematically constrained via *limit switches* so that none of the motors (except the spindle) can make more than a full revolution. Among other things, these constraints protect the electrical connections. The only self-collision that is possible (assuming that the robot goes past the limit switches) in ED I is between the spindle motor and the first link. In order to cushion against such an accidental self-collision, we have installed hydraulic shock absorbers on the first link. Other safety issues are discussed in detail in Section 5.

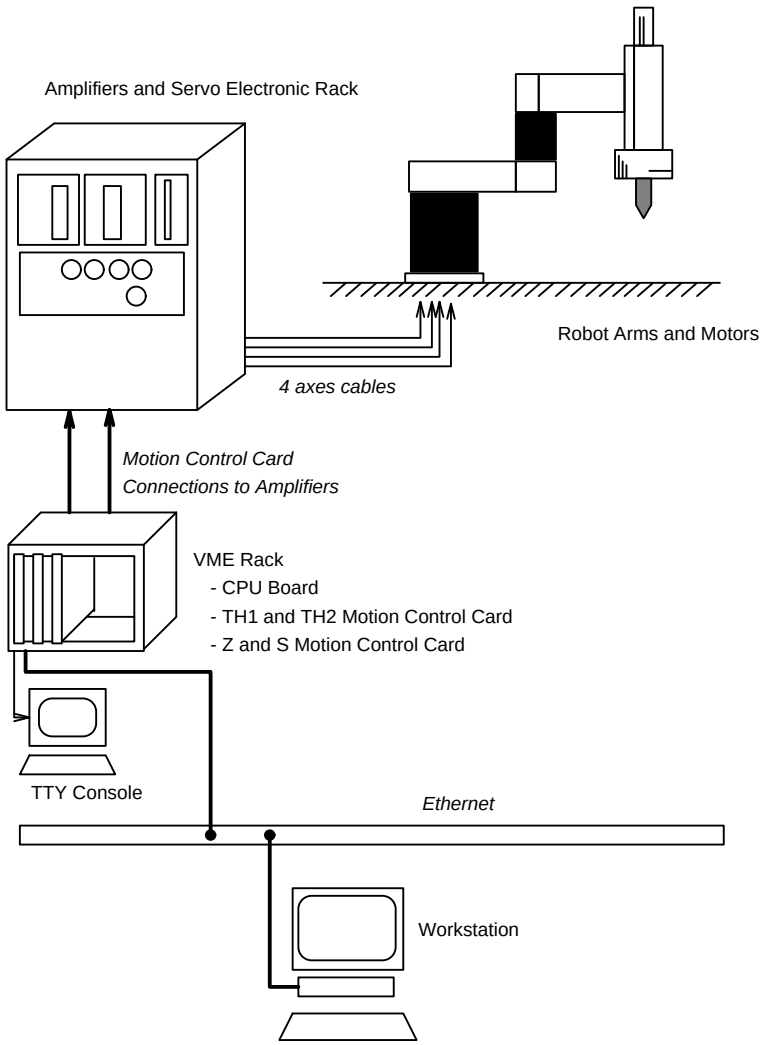


Figure 4: **Lay out of the ED I Robot System.**

z **Axis Motor Specification**	
Manufacturer	Yasukawa
Model	USAREM 01 BE 2 K
Rated Output	100W
Rated Torque	0.318 N·m
Continuous Max. Torque	0.367 N·m
Peak Torque	0.955 N·m
Rated Current	1.7 A
Rated Speed	3000 rpm
Max. Speed	4000 rpm
Torque Constant	0.198 N·m/A
Inertia J_M	0.125 kg·cm^2
Power Rate	8.09 kW/s
Related Servo Amplifier Values	
Model	CACR-SR 01 AB 2 ER
Optical Encoder Feedback	1500 pulses/rev

Table 4: z Axis Motor Characteristics

ρ **Axis Motor Specification**	
Manufacturer	Inland Motor Kollmorgen Corp.
Model	QT 3801 E
Rated Output	187 W
Rated Torque	1.75 N·m
Peak Torque	1.75 N·m
Rated Current	9.41 A
Rated Speed	$\approx$ 21600 rpm
Max. Speed	$\approx$ 21600 rpm
Torque Constant	0.185 N·m/A
Inertia J_M	0.202 kg·cm^2

Table 5: ρ Axis Motor Characteristics

• **The End-Effector** The linear ball-spline member of the linear axis is threaded to accept various end-effectors. Following is a short list of the devices that have been used as end-effectors:

1. 3-Jaw Chucks: for manufacturing purposes.
2. Parallel-Jaw Grippers and Pinch-Type Grippers: for robotics pick-and place tasks.
3. Zebra Force-Sensing Wrists.

θ_1 **Axis Motor Specification**	
Manufacturer	Yokogawa
Model	DR1400A
Rated Output	200W
Rated Torque	400 N·m
Rated Speed	30 rpm (0.5 rpsec)
Inertia J_M	400×10^{-3} kg·m^2
Related Servo Amplifier Values	
Model	SR1400A-2SN
Encoder Feedback	$102,400 \times 8$ pulses/rev

Table 2: θ_1 Axis Motor Characteristics

θ_2 **Axis Motor Specification**	
Manufacturer	Yokogawa
Model	DM1060B
Rated Output	60W
Rated Torque	48 N·m
Peak Torque	60 N·m
Rated Speed	90 rpm (1.5 rpsec)
Torque Constant	N·m/A
Inertia J_M	23×10^{-3} kg·m^2
Related Servo Amplifier Values	
Model	SD1060B
Encoder Feedback	655,360 pulses/rev

Table 3: θ_2 Axis Motor Characteristics

• z **and** ρ **Axes** The structure of the z and the spindle (ρ) axes of the ED I is some what unique in many ways. The z axis is a linear actuator that moves the end-effector vertically by a ball-screw mechanism connected to a Yasukawa (USAREM 01 BE 2 K) motor housed at the top of the z axis. There are four members that guide the linear axis. Two of them are *precision ground shafts* with linear bearings that provide significant amount of rigidity to the spindle axis, while letting the entire mechanism to travel up or down with ease and precision. Third member is a *precision ball-screw drive* that transforms the rotary motion of the Yasukawa motor to the linear motion of the z axis. The fourth linear guiding member is the spindle itself and is composed of a *precision linear ball-spline*, holds the end-effector device, and is actuated by the spindle motor (Inland Motor—Model QT 3801 E) using direct-drive technology. The spindle motor rotates the ball-spline freely thus orienting the end-effector at any desired angle.

electronics are based on off-the-shelf components.

2.1.1 ED I DD Arm

ED I is 4 degree-of-freedom robot arm with SCARA like architecture. The main links—shoulder, θ_1 axis, and elbow, θ_2 axis—form a planar kinematic chain. At the end of the second link, the z axis provides a linear actuator to move the end-effector in the vertical direction and the distal joint, ρ axis allows the end-effector to be oriented at any angle. The workspace of the ED I robot is shown below.

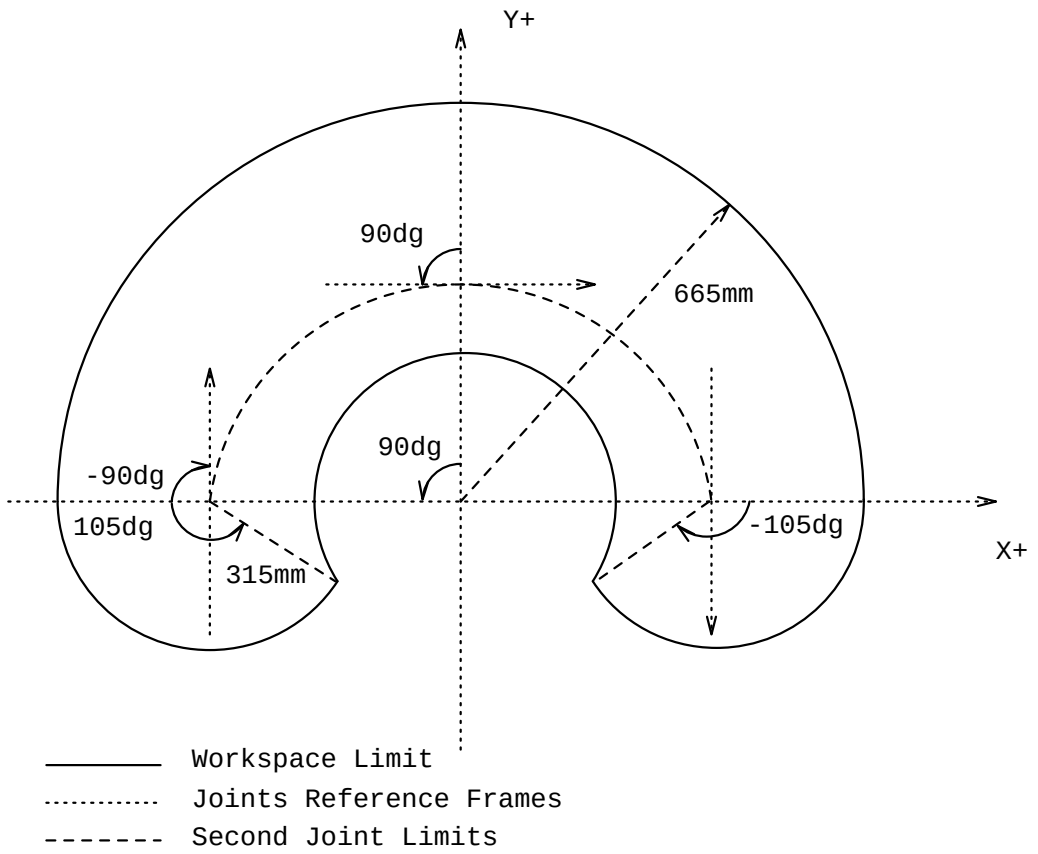


Figure 3: **Workspace of ED I.**

• θ_1 **and** θ_2 **Axes** The θ_1 and θ_2 joints are actuated by two Yokogawa direct-drive AC servo motors with very high torque as well as high accuracy capabilities. The motor housing is connected to one end of the preceding link (or base) and the armature is directly connected to one end of the succeeding link. There is no additional coupling or transmission mechanism. The motors for the joints have built-in encoders which provide the positional sensor-values directly. The mechanical specifications of the θ_1 (Yokogawa DR1400A) and θ_2 (Yokogawa DM1060B) are given below.

Last but not least, the robot system was designed to be safe for even novice users. This goal was to be achieved without sacrificing the system's ability to be used in heavy industrial applications. Thus, the safety was achieved not by down-scaling or under-powering the entire system, but by providing layers of protective mechanisms. This issue is addressed in detail in Section 5.

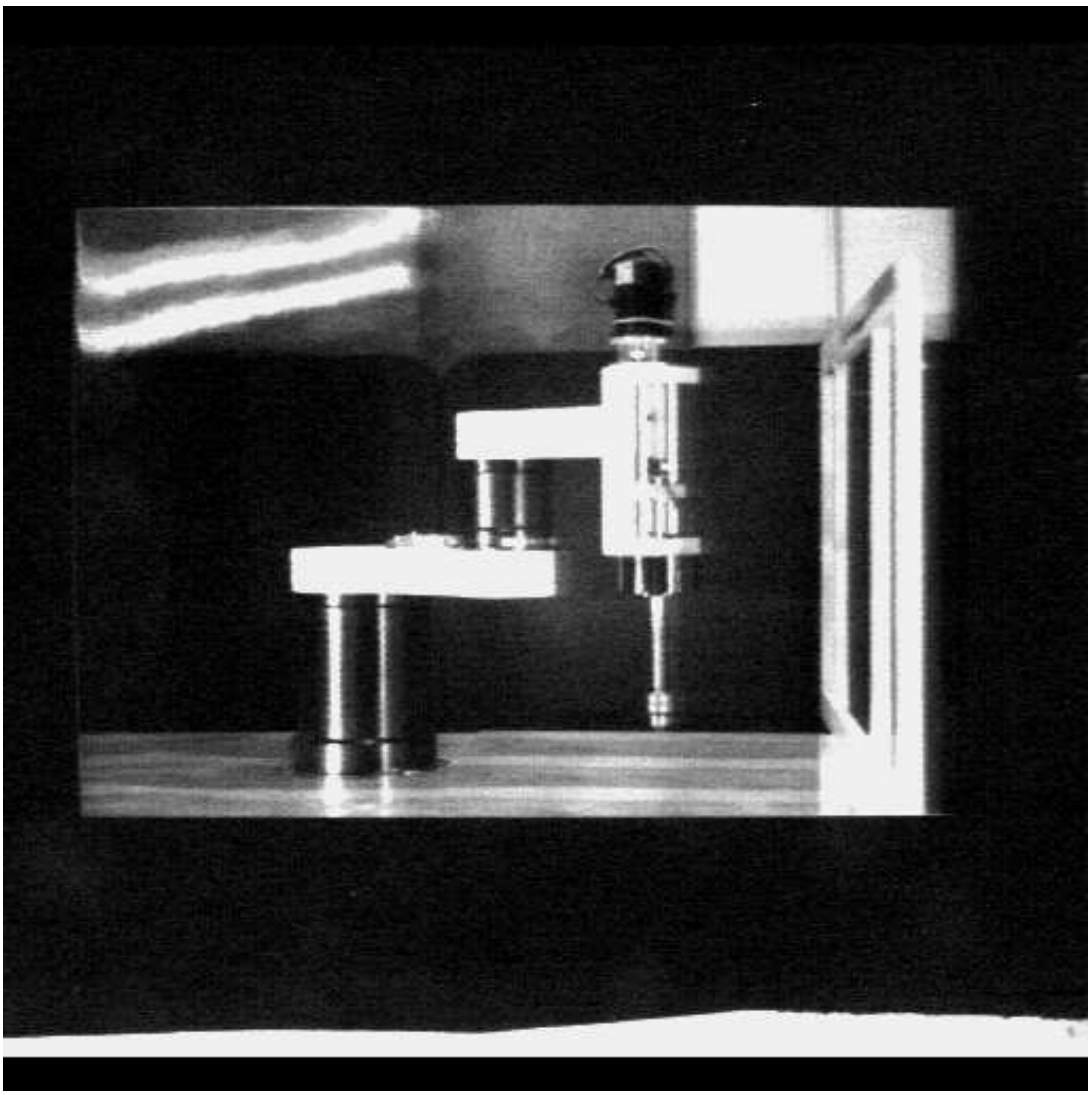


Figure 2: **ED I robot. June 1993.**.

The ED I arm was constructed with the facilities and technical skills available within the Robotics and Manufacturing Laboratory of NYU. The construction required only simple standard machine shop methods. The construction can be speeded up considerably if one uses the CNC machines to produce a small batch of the components. No special manufacturing practices would be necessary for either methods of construction.

Most of the components are commercially available. The main-links that required in-house construction are simple aluminum ($3 \times 5''$) box tubings and are rather easy to construct. Some of the components of the z axis required some skilled machining. The actuators, the sensors and the

2. Hardware

2.1 Mechanical Hardware

The main work-horse of the NYU educational project was chosen to be a multi-functional ED I robot system. This system was designed with several goals in mind: Firstly, the system was required to be simple and inexpensive in order that it could be reproduced and disseminated easily—with the cost going down with each reproduction process.

The simplicity of the system was achieved by chosing the arm to be based on the *direct-drive technology*. Each joint (altogether four) itself is a motor without any complicated transmission or coupling mechanism. Each of the main links (altogether two) is simply an aluminum hollow rectangular tubing with the ends being motor mounts that provide adaptive connections. Also, the rectangular tubing can be interchanged with another of a different size, thus providing an easy way of modifying the robots kinematic (as well as dynamic) parameters, if a particular application requires it.

The kinematic structure is based on the standard SCARA architecture, which simplifies the kinematics and dynamics considerably as the main component of the arm is just a two-link planar kinematic chain. The actuators are also so chosen that the control laws are not too complicated. The only sensors that are currently available are the position sensors (encoders) already integrated with the motors and easily accessible via the motion control hardware.

ED I Mechanical Features	
θ_1 *Axis Range*	0–300 degrees
θ_2 *Axis Range*	$\pm$105 degrees
z *Axis Range*	0–200 mm
ρ *Axis Range*	∞
Work Coordinate System Selection	YES
Absolute/Incremental Programming	YES
Emergency Stop	YES (Hardware and Software)

Table 1: ED I Features

The main-link motors (Yokogawa DR 1400A and DM 1060B) are hollow. As the links themselves are made of hollow tubings the cabling of the system is rather simple. As a result, the entire robot system can be easily assembled and disassembled with little difficulty and thus, amenable to experimentations requiring reconfiguration and modifications.

Another important design goal was to make the robot system flexible (i.e., multi-functional) by allowing for several attachments and integration of other devices. For instance, the system can be easily modified to act as a manufacturing cell with addition of chucks, fixtures and x-y table (with a pan-tilt mechanism). Also, cameras can be easily attached to the arm itself for visual servoing, or in a stationary position just to provide some amount of visual information. Some of the experiments using the arm include: measurement of magnetic fields of various components of NYU miniature DD actuators, filming of models to be incorporated with computer generated animation graphics, controlling a reactive gripper, etc.

was made in the summer of 1992. A new team was created to redesign the laboratory; the new team was headed by Bud Mishra and also included Fred Hansen (an experienced engineer) and Louis Pavlakos (a software engineer).

• **Courses:** A series of robotics related courses using the laboratory have been successfully taught at NYU. The courses include: Introduction to Robotics (twice), Advanced Robotics, CAD/CAM and Computer Vision. The first-level robotics course has been accepted by the university as a regular course for the computer science major and is now listed as a 4 credit course in the university handbook. The maturity and the preparation of the students in these courses have varied over a wide range: high school students (from Dalton School), engineering students (from Stevens Institute of Technology) and computer and mathematics students (largely from New York University). Some undergraduate students have also used the lab facilities to carry out independent study programs and projects.

• **Evaluation:** The first implementation of the robot was evaluated by a team headed by Bud Mishra and Israel Greenfeld (currently a research engineer in Israel). The precision provided by the robot as well as various safety measures were considered severely lacking. The software structure was deemed not easily portable. A completely new hardware and software design was made. The implementation of the new system was started in July 1992 and a first implementation was completed by mid September.

In order to improve the precision (specially for CAD/CAM experiments), the z axis was completely redesigned and reimplemented. The other two main links were remachined to rectify certain design errors. The hardware safety was improved by running the motors at lower power levels; incorporating shock absorbers, limit switches and by enclosing the entire assembly by a lexan enclosure. The motor controllers were redesigned using Phoenix and Creonics boards, which are capable of stopping the robot (and eventually shutting the power off) in case of a serious exceptional situation. A manual emergency "red" button has been made available for the teaching assistant in order that a potentially dangerous situation could be averted.

The software was redesigned under the "MOSAIC" system (developed independently by the NYU manufacturing group[11]) and on top of `VxWorks`tm real time systems. The MOSAIC system has been recognized by and gained acceptance in the manufacturing community as a leading control software system. By founding on a well-tested system, we could improve the system reliability and the development time considerably. However, if there is a need, the MOSAIC/`VxWorks`tm system can be easily interchanged with another real time system.

1.4 The Organization of the Report

The report is organized as follows: In Sections 2 and 3, we describe the hardware and the software architecture of ED I, respectively. In Section 4, we discuss primarily the undergraduate "Introduction to Robotics" course (taught by B. Mishra and R. Even), and list the experiments conducted with ED I, in the context of this undergraduate course. This section by itself may be of interest to other instructors who may want to adapt these experiments to a different robot system. Section 5 describes the safety measures taken with respect to ED I and finally, Section 6 concludes the report with some thoughts on our contributions and on the future of robotics education. We have also provided two appendices: one containing some example programs for and operation of the ED I robot, and the other containing a description of a sequence of courses, where ED I has already been used or is planned to be used.

design can be augmented by machine tools, x-y table with pan/tilt, vision system, various sensory tools etc. The amount of time required to switch from one configuration appropriate for one course to another configuration for quite a different course was constrained to be a matter of less than couple of hours. Thus, in principle, the same system can be used by two different courses running in parallel.

Figure 1: **NYU Undergraduate Teaching Laboratory. ED I, in the background, is being operated by an experimenter.**

1.3 History

The construction of the laboratory was started in 1990 with a team consisting of Z. Li, A. Cen (an electronic engineer), J. Li (a mechanical engineer), C. Li, N. Silver (graduate students) and L. Gurvits (a control theory researcher). A first course was taught with existing IBM SCARA robots. The first educational robot was completed by the end of 1991. The first evaluation of the laboratory

1. Introduction

1.1 Initial Goal

In the spring of 1990, we initiated an educational project at NYU, whose primary goal was to create a disseminable, multi-functional[1] and inexpensive laboratory course sequence, aimed at improving the practical skills of undergraduate students specializing in robotics, vision, AI and manufacturing disciplines.

Our motivation to build such a laboratory arose primarily from our desire to provide students with the practical experience to complement their understanding of the principles involved in robotics, automation, AI and manufacturing sciences. In the past, most of the research in robotics was motivated by simple applications using industrial robot arms suitable for routine welding, spray-painting and part-assembly operations. At the time we initiated this project, there were few universities with regular robotics courses aimed at undergraduates; where a standard introductory course existed, students typically regarded a robot as an open kinematic chain, having a straightforward kinematic and dynamic structure. The accompanying laboratory training typically involved only the programming of a commercial industrial robot to perform simple 'pick-and-place' tasks.

Thus, we felt that there was an acute need for robots suitable for the more extensive educational purpose. To achieve this goal, we proposed to build an 'educational robot' with a structure, which is simple and yet easily modifiable for the multiple functions it may have to serve.

We have designed and built a 4-DOF direct-drive arm, ED I, which can be easily augmented with tactile and force sensors as well as manufacturing tools, and which can be controlled from a simple computer system. This robotic system has been successfully used in two introductory robotics courses (one undergraduate, the other graduate), a CAD/CAM course and to some extent, in a vision course.

In this document, we shall describe the robot and our experience with the undergraduate course taught by B. Mishra and R. Even (teaching assistant).

1.2 The Role of ED I Robot

The ED I robot plays a key role in the entire project. Thus much thoughts and efforts went into careful design, periodic evaluation and modification, and augmentation of the functionalities of the ED I robot system.

The final design was chosen to incorporate direct drive technology in order to keep the system modeling and control simple. The actuators were chosen to be rather powerful so that if necessary they can be used at their full power; for the normal experiments they were used at a lower power level with a smaller gain. As the system was planned to be used for certain manufacturing applications, the robot was designed to operate with a high level of accuracy. Finally, as the system was to be used by novice users, it was assumed that the system will be used occasionally in inappropriate manners and precautions were taken to improve the safety of the system at various levels. As the ED I robot began to be used by the students, the design-team monitored the students usage and rectified any flaw—especially, in the system safety—that became apparent.

In order to make the system multifunctional, the design of the system (particularly, workspace, payload limits, kinematics and dynamics) was kept extremely simple and flexible. The ultimate

[1] The proposed laboratory is multi-functional in the sense that the same laboratory may be used for more than one area: AI, vision, robotics, real-time systems, manufacturing, for instance.

CONTENT

§1
Introduction
3

§2
Hardware
6

§3
Software
16

§4
Experiments
33

§5
Safety Issues
53

§6
Concluding Remarks
55

Appendix: A
An Example Program: the `stroke` Routine
57

Appendix: B
A Proposed Course Sequence
73

•
References
80

ABSTRACT

The primary goal of the NYU educational robot project is to create a disseminable, multi-functional and inexpensive laboratory course sequence, aimed at improving the practical skills of undergraduate students specializing in robotics, vision, AI and manufacturing disciplines.

The main work-horse of the NYU educational project was chosen to be a multi-functional ED I robot system, consisting of a 4 DOF DD arm and several auxiliary devices. The system was designed to be simple, inexpensive, flexible and safe.

In this report, we describe the history, design, structure and evaluation of this robot system. We also describe several robotics and related course sequence that can use the ED I system effectively. We also provide some example experiments that have been run on ED I successfully.

This report has benefited from the labor, contribution, discussions, advice and criticisms of several people on the ED I project team and the credit for the final product goes to the entire team.

ED I Project Team:

M. Antoniotti, A-B. Cen, R. Even, I. Greenfeld, L. Gurvits, F. Hansen, A. Rajkumar, C. Li, J. Li, Z-X. Li, B. Mishra, S. Mallat, E. Pavlakos, J. Schwartz, N. Silver and R. Wallace

Supported by NSF Grant #CDA-9018673.

Address: Robotics Research Laboratory, Courant Institute of Mathematical Sciences, New York University, 715/719 Broadway, 12th Floor, New York, NY-10003.

ED I:
NYU Educational Robot
Design and Evaluation

Bud Mishra
&
Marco Antoniotti

With contributions from:
F. Hansen, N. Silver, R. Wallace and R. Even

Robotics & Manufacturing Research Lab
Courant Institute, New York University