A MANIPULATOR CONTROL TESTBED: IMPLEMENTATION AND APPLICATIONS*

John Fiala, Albert Wavering, Ronald Lumia†

An implementation of the lower levels of the NASA/NIST Standard Reference Model (NASREM) Telerobot Control System Architecture has been developed at NIST. The implementation includes manipulator servo control, rate teleoperation, autonomous trajectory generation, and visual sensing. This paper describes how the system is designed to be a testbed for manipulator control via generic interfaces and a modular Ada software architecture. The multiprocessor hardware architecture which supports the software architecture for real-time operation is also described. The paper presents applications of the testbed system to specific manipulator control problems, including some example comparisons of different strategies for servo control and trajectory generation.

INTRODUCTION

In order to compare and evaluate different methods of manipulator servo control and trajectory generation, it is desirable to have a manipulator control system which can serve as a testbed. Such a testbed system must have a sufficiently generic structure to easily accommodate a broad variety of different approaches. The control system architecture must be general enough to support a large number of different algorithms at one time without modification. However, since the system must undergo continual maintenance, i.e. to test new control laws and add new capabilities, it is not sufficient to design the system on an ad hoc basis. The system must maintain a high degree of modularity to allow individual components to be independently developed and modified. This is to be achieved while obtaining the real-time performance required for advanced control. The computational requirements for model-based control on a seven degree-of-freedom manipulator, for example, motivate a multiprocessor design. The control system thus needs modular, easily modifiable multiprocessing with multiprocessor execution.

---

† Robot Systems Division, MEL, Technology Administration, U.S. Dept. of Commerce, National Institute of Standards and Technology, Bldg. 220, Rm. B127, Gaithersburg, Maryland 20899.

In the Intelligent Controls Lab of the National Institute of Standards and Technology (NIST), a manipulator control testbed has been implemented on a Robotics Research Corporation (RRC) K-1607 dextrous robot arm*. In addition to the robot, a DFVLR sensorball, Telerobotics, Inc. (TRI) electric parallel-jaw gripper, and JR3 force/torque sensor are integrated into the testbed system. This testbed system is modular, easily modifiable, and uses multiprocessing on a multiple processor platform to achieve real-time performance. The system design is based on the NASA/NIST Standard Reference Model (NASREM) Telerobot Control System Architecture[1,2], as described below.

ARCHITECTURE

The NASREM model is a description of a hierarchical, multiprocessing architecture for a telerobot control system. The model defines, among other things, *computational pipelines* of functions which allow the control computations to be distributed to a multiprocessor computing architecture. Only the rudiments of the model are presented here as an introduction to a testbed implementation for a robotic manipulator. For a more detailed description of various aspects of NASREM, see Ref. 2.

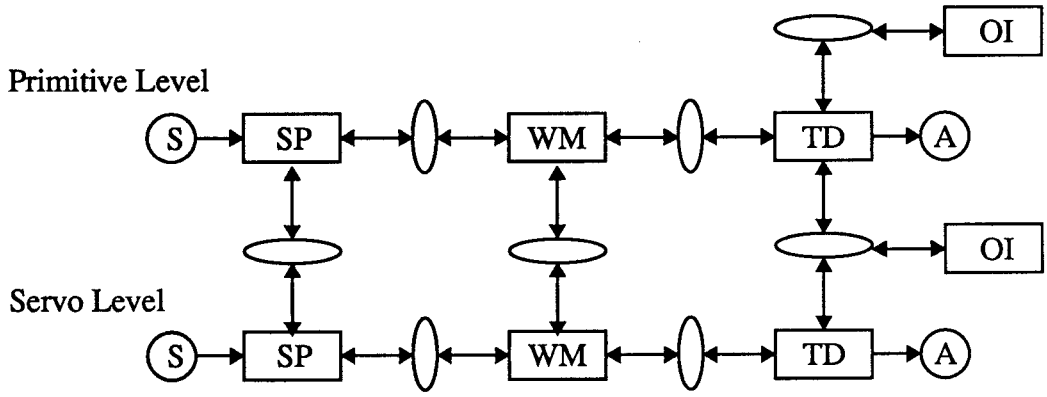

Fig 1 Basic Pipelines of the NASREM Architecture

A basic structure of the model is depicted in Fig 1. Each rectangular box in the figure represents a function, or a set of functions, of the system. The ovals represent the data communicated between functions. *Horizontal pipelines* of concurrent computation are created by the SP-WM-TD construct. That is, data arriving at a sensor S would enter through an SP (Sensory Processing) function, proceed, if necessary, through WM (World Modeling) functions, arriving at the TD (Task Decomposition) functions for use in computing an output to an actuator A. *Vertical pipelines* of concurrent computation between *levels* of SP-WM-TD constructs, also called *nodes*, allow operator-friendly, high-level commands to be decomposed into the actuator commands ultimately required. The operator may input commands at the various levels as indicated by the OI (Operator Interface)

* Certain commercial equipment is identified in this article to describe the work adequately. Such identification does not imply recommendation or endorsement by NIST, nor does it imply that the equipment is necessarily the best available for the purpose.

function. By definition of the vertical and horizontal pipelines, NASREM specifies the concurrency inherent in the robot control problem.

In developing an implementation of the NASREM model, the basic architectural framework must be detailed sufficiently to serve as blueprint for the system. An analogy can be drawn to a house plan. Even after the basic components are established—a roof, one-or-more floors, exterior walls, interior walls, rooms, etc.—a detailed architecture must be designed to support construction of a specific house. The detailed architectural design of the testbed implementation and some of the principles used to obtain it have been described previously[3-11]. The ideas are summarized in the following.

The implementation design begins by establishing the nodes (or levels) needed to provide appropriate operator/programmer interfaces. In the manipulator testbed system the two lowest levels of the hierarchy have been implemented. A Servo Level provides a servo reference command interface and a Primitive Level provides a path command interface[10]. The implementation of these two levels will be discussed in later sections.

The conceptual boxes of SP, WM, and TD are decomposed into *atomic units*[9]. Atomic units express the ultimate concurrency of the system—they do not contain concurrent elements. For example, the TD box within a node decomposes into three atomic units, a Job Assigner (JA), Planner (PL), and Executor (EX)[2,4,5]. These units form a vertical pipeline within a node.

To each atomic unit is ultimately assigned a software entity called a *process*. A process has the following properties:

1) Concurrent with other processes
2) Interfaces with other processes through a multiprocess data system
3) Read-compute-write execution cycle
4) Continuous cyclic execution
5) Inactivation

Note that an atomic unit is not a function call or a case statement alternative since these are sequential programming elements. It is implemented by an independent software syntactic structure called a *process model*. Such a structure can be implemented in any language. In Ada, it is convenient to do this using packages and tasks, as given in Fig. 2.

In the Fig. 2 model, data is communicated between processes using *send* and *receive* functions. The NASREM Architecture specifies that interfaces between processes be multiprocess accessible. This so called "global data" concept does not mean that every process accesses every interface. In fact, it is desirable to limit processes to accessing the minimum number of interfaces appropriate for their pipelines. However, it may be necessary to change the set of pipelines to which a process belongs during execution[2]. Thus, the multiprocess accessibility concept requires that the data of the interface exist outside of the communication action itself. This rules out parameter passing as an option. Communication send and receive functions can provide the necessary interfaces while maintaining software modularity principles.

The final architecture design consists of the atomic units along with their functional

```
package YURBOX_PROCESS is              -- process model consists of init procedure
                                       -- and process body task which performs
                                       -- read-compute-write execution cycle

        procedure YURBOX_INITIALIZATION;

        task YURBOX_TASK is
                entry CYCLE;
        end YURBOX_TASK;
        procedure YURBOX renames YURBOX_TASK.CYCLE;
end;

with YURBOX_COMPUTE;
with INPUT_DATA_STRUCT, OUTPUT_DATA_STRUCT, READER_WRITER;

package body YURBOX_PROCESS is

        YURBOX_IN : INPUT_DATA_STRUCT.YURBOX_INPUT_TYPE;
        YURBOX_OUT: OUTPUT_DATA_STRUCT.YURBOX_OUTPUT_TYPE;

        procedure YURBOX_INITIALIZATION is separate;

        task body YURBOX_TASK is               -- main body of the process
        begin

                loop
                accept CYCLE do                -- optional entry, allows invocation
                                               -- control for inactivation, etc.

                    READER_WRITER.RECEIVE( YURBOX_IN );

                    YURBOX_COMPUTE( YURBOX_IN, YURBOX_OUT );

                    READER_WRITER.SEND( YURBOX_OUT );

                end CYCLE;
                end loop;

        end YURBOX_TASK;

end YURBOX_PROCESS;
```

Fig 2 Ada Process Model for Implementing an Atomic Unit

description and the definition of the interfaces between them. Once this is obtained, development of software is straightforward. Data structures for the interfaces are defined. Each atomic unit is implemented as a process using the data structures to define the inputs and outputs. Then the functionality of each unit is implemented by a "compute" procedure. In the final implementation phase, the software processes are allocated to hardware processors. Since the software design is independent of the hardware, there is a great deal of flexibility in meeting the real-time performance requirements, and the system can easily be modified to account for a new processor configuration or new requirements[11].

TESTBED SERVO IMPLEMENTATION

The design for the Servo Level of the manipulator testbed system is depicted in Fig 3. This implementation is based on the NASREM architecture as described above. The

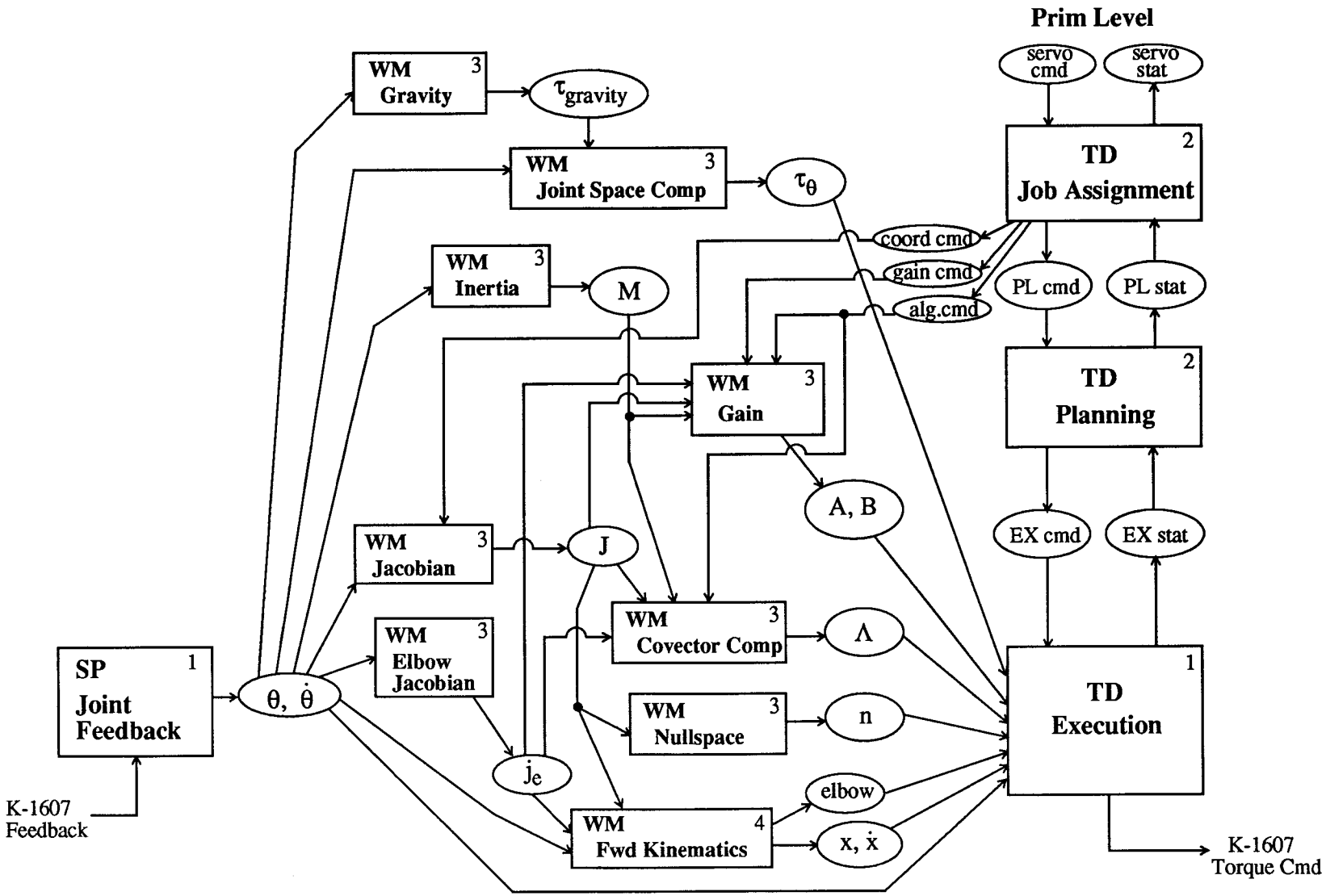

Fig 3 Testbed Servo Level Architecture

Servo Level consists of numerous horizontal pipelines which flow from Joint Feedback to Execution through World Modeling processes. Joint Feedback filters the velocity feedback and converts data to SI units. The World Modeling processes compute model-based quantities, such as the inertia tensor, for use in the control law. The Execution process computes the control law based on commands received through the vertical pipeline from the Primitive Level. Execution and Joint Feedback communicate with actuators and sensors by way of the RRC K-1607 controller. The RRC controller provides an interface which allows joint torque commands to be issued to the actuator torque control loops, and which makes joint sensor feedback available, every 2.5 ms.

Complete details on the functions and interfaces of the individual units of Fig. 3 can be found in Refs. 3, 4, 7 and 12. It is of note, however, that a generic Task Decomposition interface is defined for the Servo Level to support the testbed nature of the system. The servo command interface as specified in Ada is given in Fig. 4. This interface consists of a generic set of command reference signals, such as desired position, velocity, and force, as well as a generic set of gain parameters for the corresponding servo loops. Since the inter-

```
type coordinate_spec is          -- coordinate system specifier record
record
        coord_sys_type:          coordinate_name;                -- (joint, endeff, world)
        aux_endeff_transform:    array(1..7) of short_float;     -- 3D tranform consists
        aux_world_transform:     array(1..7) of short_float;     -- position & quaternion
end;

type elbow_cmd_type is           -- redundancy resolution parameter
record
        position:                short_float;                    -- 32-bit floating point
        velocity:                short_float;
end;

type time_stamp_type is          -- time stamp parameter to determine time of execution
record
        set_flag:                boolean;                        -- synchronization flag
        Tp:                      short_float;                    -- time cmd will start
        Tp_final:                short_float;                    -- time cmd is over
end;

type servo_command_type is       -- servo command interface from Prim to Servo
record
        algorithm:               algorithm_name;                 -- (PID, stiffness, etc.)
        ref_frame:               coordinate_spec;
        position:                array(1..7) of short_float;     -- desired position
        velocity:                array(1..7) of short_float;     -- desired velocity
        acceleration:            array(1..7) of short_float;     -- desired acceleration
        force:                   array(1..7) of short_float;     -- desired force
        elbow_cmd:               elbow_cmd_type;
        Kp                       array(1..7) of short_float;     -- position gain
        Kv:                      array(1..7) of short_float;     -- vel. gain (1 per dof)
        Ki:                      array(1..7) of short_float;     -- integral gain
        Kpf:                     array(1..7) of short_float;     -- force gain
        Kvf:                     array(1..7) of short_float;     -- force/velocity gain
        Kif:                     array(1..7) of short_float;     -- integral force gain
        S:                       array(1..7) of short_float;     -- hybrid coord selection
        time_stamp:              time_stamp_type;
end;
```

Fig 4 Ada Implementation of Servo Command Interface

face supports a number different control algorithms, an algorithm name parameter is used to select different modes. The interface supports servo control in Cartesian or joint space. The coordinate specification allows Cartesian servo coordinates to be placed arbitrarily in the robot workspace[12]. Note that the interface is only slightly modified from that proposed in Ref. 4. The highest order reference signals have been removed since they are not required to satisfy the testbed's goals. Also, the redundancy resolution parameter has been reinstated[3] as reference signals on elbow position, although a more general redundancy resolution command may ultimately be needed.

Since there is a one-to-one correspondence between the boxes in Fig 3 and the software processes which implement them, the diagram also serves as a specification of the software design. The testbed hardware to which these processes are allocated consists of three MC68030 and four MC68020 processor boards connected via a VME bus as depicted in Fig. 5. The numbering of processes in Fig. 3 identifies the processor to which each is allocated. Processes assigned to processors 1, 2 and 4, each execute once every 2.5 ms. This gives joint-space control an effective loop rate of 200 Hz. For Cartesian control the pipeline with forward kinematics is used, giving a loop rate of about 133 Hz. The remaining WM processes are all allocated to a single processor board. Since different sets

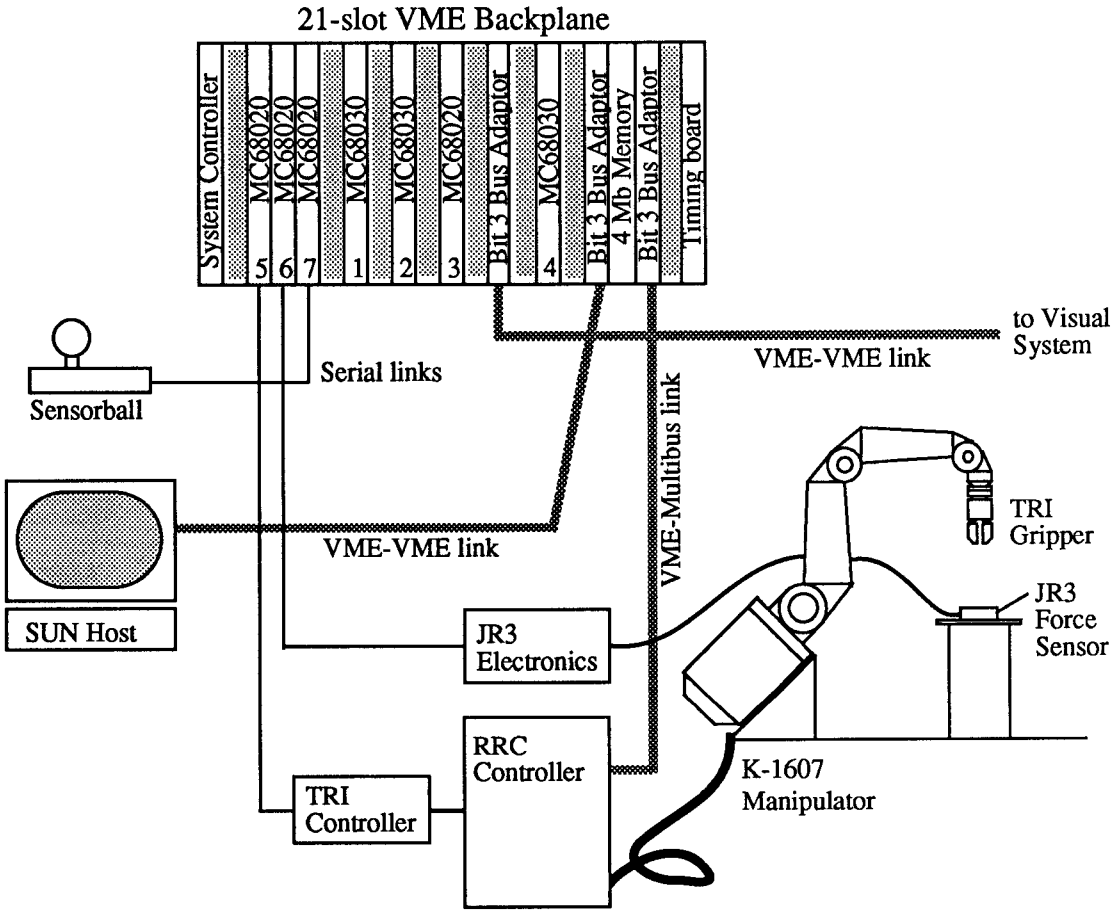

Fig 5 Testbed System Hardware

of these processes may be active for different control algorithms, the cycle time for these processes varies from 5 to 100 ms. This is not critical however, since these pipelines are not within feedback control loops.

TESTBED PRIMITIVE IMPLEMENTATION

The Primitive Level testbed architecture is depicted in Fig. 6. Again, the process numbering indicates allocation of processes to processors (Fig. 5). As with the Servo Level, the processes consist of Sensory Processing, World Modeling, and Task Decomposition functions. In addition, there are Operator Interface (OI) processes which allow the user to enter commands to the system, perform teleoperated control, and log real-time data during operation. Each of the processes shown in Fig. 6 is briefly described below. Additional information on the interfaces and operation of the processes shown in Fig. 6 may be found in Refs. 5, 6, and 8.

The User Interface process for Primitive task decomposition handles the user interaction required to send motion commands to the robot and gripper. Commands may be entered via the keyboard, or in the form of previously-prepared command files which contain parameters for a number of motions. Command file motion sequences may be repeated an arbitrary number of times. In addition, the User Interface allows robot Cartesian poses to be recorded, and includes a facility for defining coordinate frames by teaching origin and X- and Y-axis points. The User Interface also coordinates switching between teleoperated and autonomous control. When the teleoperation mode is selected, the entry of new autonomous commands is suspended until the user indicates that teleoperation is finished. When this occurs, a command to move to a safe position is automatically executed and the user is presented with the options of the main menu.

The Job Assignment process maintains a queue of input commands to the Planning process. When a new command is received from the User Interface, it is added to the queue. When a command is completed, the oldest command in the queue is removed. In the current implementation, the oldest command in the queue is the command previous to the one that is being executed. The Job Assignment module writes out the two oldest commands in the queue (i.e., the "previous" command and the "current" command) to the Planning process. This is why two command buffers are shown between Job Assignment and Planning in Fig. 6.

The Planning process plans the dynamic trajectory to perform a commanded motion. Typically, this involves determining the motion start and goal states in a specified coordinate system, and planning trajectory equation coefficients or parameters. Also, other planning data, such as gains, evaluation interval, and servo algorithm are obtained from planning data files which contain information for specific trajectory algorithms. In addition to the input and output command and status buffers, the Planning process reads from several manipulator feedback interfaces which are updated by Servo World Modeling processes. These interfaces provide joint position and velocity, end effector Cartesian position and velocity, and elbow angle position and velocity.

The Execution process executes the trajectory function supplied by the Planning process. The trajectory function is evaluated for multiples of the specified time interval (typi-

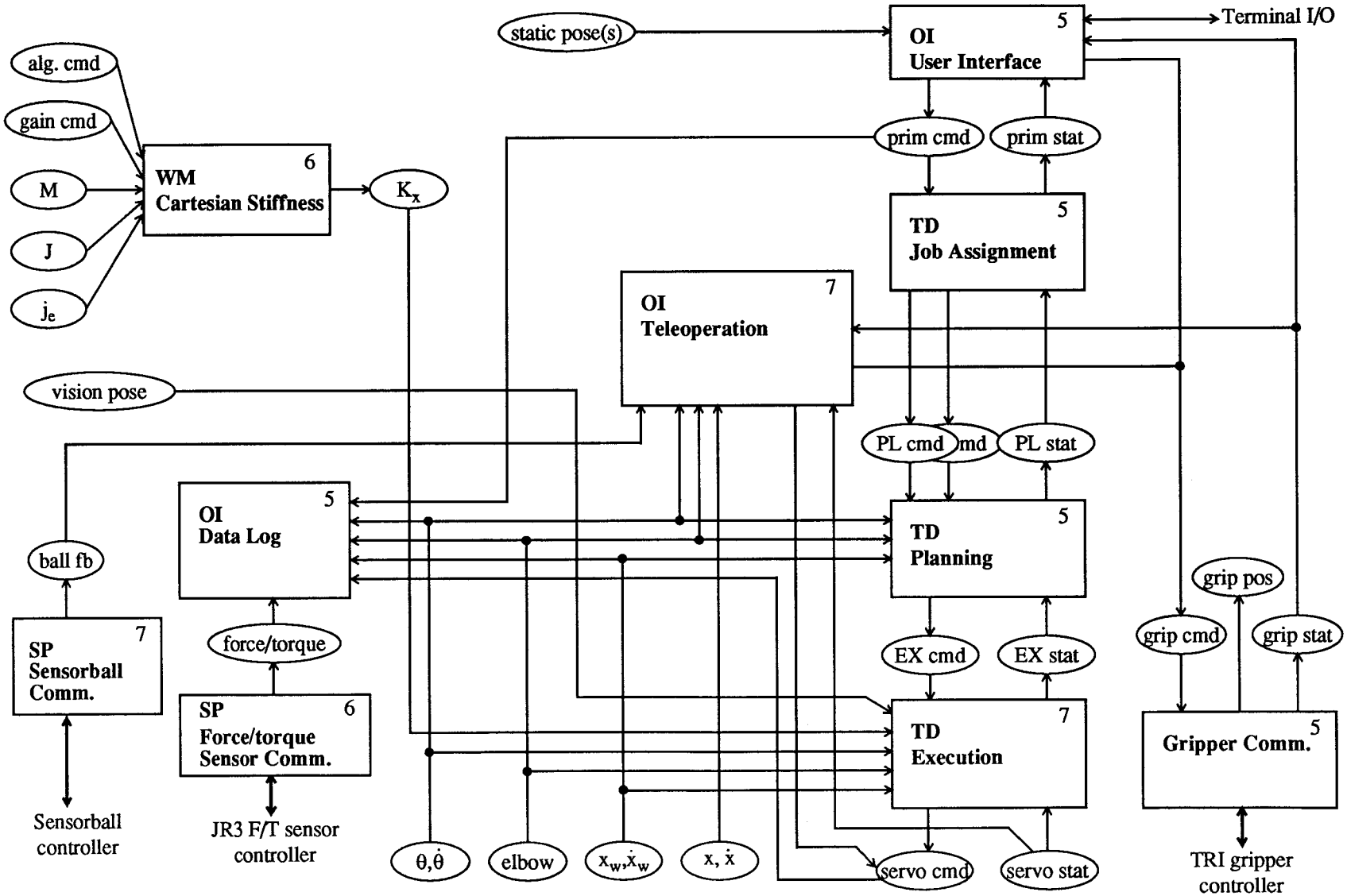

Fig 6 Testbed Primitive Level Architecture

cally 2.5-25 ms). The loop time of the Execution process is determined by a local loop timer. The loop timer is a procedure within the read-execute-write cycle (process body) which reads a clock and delays for the amount of time required to achieve the desired loop time as specified by the evaluation interval. The Primitive Execution process thus runs completely asynchronously from all Servo processes.

The Execution process reads the same joint, Cartesian, and elbow angle feedback buffers that are used by the Planning process. The Execution process also reads a buffer that contains the equivalent Cartesian stiffness for a given servo algorithm, gain matrix, and robot configuration. This information, computed by the Cartesian Stiffness process, is used by some trajectory execution algorithms for Cartesian force limiting. The Execution process also has access to object pose data supplied from the vision SP/WM processes. This enables sensory-interactive trajectories, such as tracking and catching a moving object, to be performed[13].

The Data Log process reads information from various interface buffers and provides facilities to record the data they contain. The process reads from Cartesian, elbow angle, and joint feedback buffers, as well as the Servo command buffer (for desired positions, velocities, etc.), and the feedback buffer from the force/torque sensor. The Data Log process also reads the command input to the Job Assignment process to see when a new trajectory is about to be executed. Time stamps for the recorded data are obtained from a global clock.

The two processes required for teleoperation are the Teleoperation process and the Sensorball Communications process. Together, these processes take the place of the autonomous Primitive trajectory generation processes to generate the sequence of goal states for the Servo processes. The Teleoperation process performs rate teleoperation based on the sensorball input data from the operator. The operator can teleoperate in any valid servo coordinate system. As shown in the figure, teleoperation uses elbow angle, Cartesian position, and joint feedback, and the elbow angle and arm Jacobians. Other parameters, such as the servo algorithm, coordinate specification and gains, are read in from a data file during initialization. The Teleoperation process writes commands to and receives status from both the Servo Job Assignment process and the gripper communications process. Note that the Teleoperation process uses exactly the same Servo and gripper interfaces as used for autonomous motions.

The Sensorball Communications process performs serial communications with the sensorball. It converts ball forces and switch settings to the desired format and writes this information to an interface buffer. The Teleoperation and Sensorball Communications processes share a board with the Primitive Execution process. At any given time, either the Execution process is running, or the Teleoperation and Sensorball Communications processes are running. The processes are activated and deactivated by the User Interface process. During teleoperation mode, the User Interface process waits for user input to indicate when teleoperation is finished, as described above.

In addition to the Sensorball Communications process, there are two other processes which perform serial communication with external devices. These are the Force/torque Sensor Communications process and the Gripper Communications process. These pro-

cesses serve to provide "ideal" multiprocess interfaces to equipment connected to commercial controllers. In doing this, they also perform conversion to/from SI units.

The current implementation of the Primitive Level provides the capability to perform joint space and Cartesian straight-line trajectories with quintic polynomial and trapezoidal velocity profiles, and visually-guided Cartesian motions. Sinusoidal and square wave motions for both joint and Cartesian degrees of freedom are also available. Trajectory generation for Cartesian motions includes a self-motion trajectory, using one of several redundancy resolution techniques.

APPLICATIONS

The initial objective of the testbed implementation was to validate the architectural ideas discussed above. In addition, the testbed system has been used for a number of different applications. For example, it has been used to compare schemes for manipulator stiffness control[12]. The system has been used to evaluate techniques for manipulator metrology[14], and to study issues related to contact stability[15], impact dynamics, and force limiting in contact. A number of studies were conducted to support the needs of the NASA Flight Telerobotic Servicer (FTS) project. With the generic testbed implementation it was often possible to perform these studies quickly with little system modification. Equally important, when additions and modifications were required, they were easily performed.

The static and dynamic performance of Cartesian stiffness control algorithms that do not use explicit force feedback was investigated, as described in Ref. 12. A number of Cartesian servo algorithms were implemented on the testbed system and compared. Implementing the new algorithms was straightforward because the command and status interfaces did not change, the necessary WM processes could just be added to the (slower-rate) WM processor board, and many SP/WM processes were common to several different algorithms. All that was required to implement a new algorithm in many cases was to add the new EX procedure which processed the WM information in a different way. It was found that Jacobian-transpose servo algorithms provided translational stiffnesses typically within 10% of the desired value (rotational stiffnesses were 60-80% of the desired value). In some cases, significant coupling terms were evident. Dynamic response tests, which used sinusoidal and square-wave trajectories, indicated that a 2 Hz bandwidth could be achieved with such algorithms on this robot, and demonstrated the advantages of using dynamic damping to achieve constant time response automatically for changes in end effector stiffness (position gains) and robot configuration.

In Ref. 14 manipulator metrology tests were developed and proposed for the first Demonstration Test Flight (DTF-1) of the FTS project. The test procedures were based on American National Standards Institute/Robotic Industries Association (ANSI/RIA) and International Standards Organization (ISO) standard procedures. Modifications to the standard test procedures were made to accommodate particular DTF-1 constraints, the primary one being that only a single measurement position was to be used. The proposed positioning accuracy and repeatability tests were then performed and evaluated using the testbed system. An individual joint PID servo algorithm with gravity compensation was used for all motions. The tests involved repeated motions from a variety of start locations

to a single "sensor nest" test position using Cartesian and joint space quintic polynomial trajectories. Cartesian endpoint positions were recorded using a Automated Precision, Inc. (API) Smart 310 laser tracker metrology system. The resulting data were analyzed to determine the accuracy and repeatability of the manipulator. The Cartesian position computed by the forward kinematics process was also recorded at the end of each test motion and compared with the laser tracker position transformed to the same coordinate system. It was determined that, although using only a single measurement position is less than ideal, a substantial amount of useful manipulator metrology information could be obtained with such a setup.

Another application of the testbed system was to study the force dynamics of the impact created when the manipulator tip initially contacts the environment. As shown by the example force plot in Fig. 7, the study found that the impact produces a force spike, the magnitude of which is dependent on the impact velocity and the materials at the interface. The manipulator may bounce on impact, as in Fig. 7, where the initial spike is followed by subsequent spikes separated by periods in which the manipulator tip is not in contact with the environment. Note that this impact phenomenon of bouncing is separate from the issue of contact stability. A controller may be unstable in contact with the environment when there is too much delay in the velocity feedback loop[15]. In a separate study on the testbed system, it was determined that if the loop rate of the servo control was decreased sufficiently, contact instability would occur. A contact instability is characterized by an oscillation growing without bound once contact is established. In the presence of stiction, however, only an oscillation of large enough amplitude will be sustained and grow. Since large amplitude oscillations may involve impacts, the contact stability problem can be confused with normal impact bouncing, but the two phenomena are quite distinct. In Fig. 7 the manipulator exhibits only stable contacts even though the impact causes bouncing. Bouncing due to impact occurs even when the manipulator is not powered.

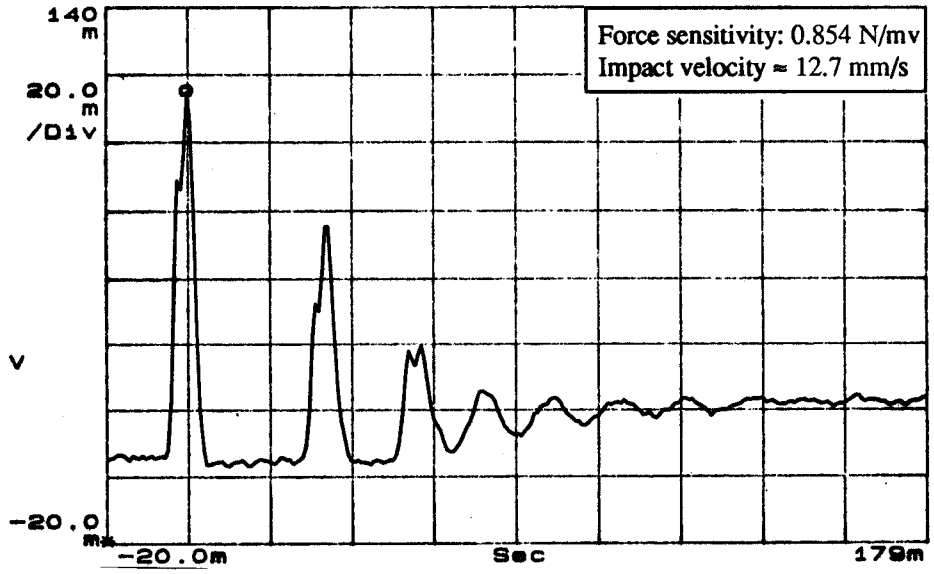

Fig 7 Force Recording for Impact of Manipulator Tip on Steel Sensor

The evaluation of different trajectory modification strategies for limiting interaction forces is the final testbed application to be described. For DTF-1 of the FTS project, it was desired to investigate techniques which could limit interaction forces without using actual end effector force or joint torque measurements. One way to do this is to use an estimate of the effective Cartesian stiffness in combination with the current end effector position to estimate the interaction forces which would occur if the next position command was sent without modification. If the estimated force exceeds the limit, then the position command increment can be scaled by the ratio of the force limit to the estimated force. The servo torques which are generated should then result in forces which are within the desired range. This type of force limiting technique was studied to investigate its effectiveness when used with both Cartesian stiffness (Jacobian transpose) and inertia-decoupled joint servo algorithms. The use of a single worst-case scale factor for all position directions was also compared with scaling back individual directions according to the force in those directions.

The force limiting strategies were tested by commanding a motion into a rigid fixed pedestal and measuring the actual resultant forces (see Fig. 5). As an example, Fig. 8 shows the command and feedback positions recorded during a force-limited motion where

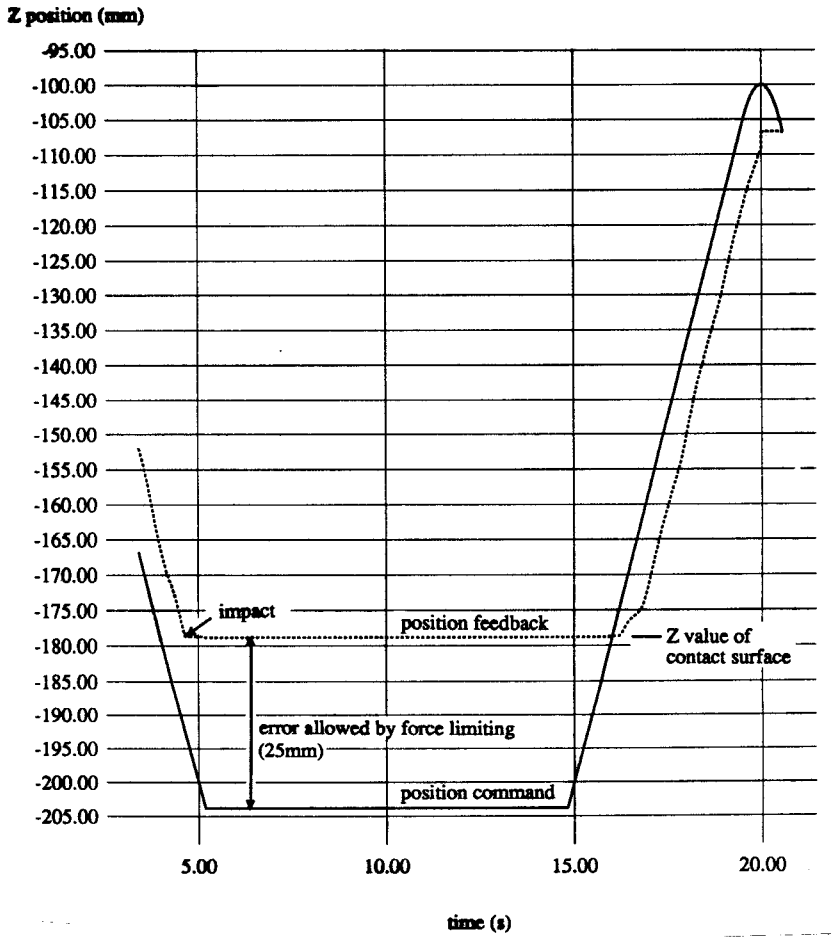

Fig 8 Recorded Positions for Motion to Z = -290mm (Cartesian Stiffness Algorithm)

it was desired to move down to Z = -290 mm at t = 10 s, and then move back up. The contact surface is at Z = -178 mm. For this motion, the commanded stiffness in the Z-direction was 2000 N/m, and the force limit was set at +/-50 N. The velocity of the motion was about 21 mm/s, and a Cartesian stiffness servo algorithm was used. As seen in Fig. 8, after the contact surface is reached, the position error increases until the error times the stiffness equals the force limit. This happens when the error is 25 mm. At that point, the position command remains unchanged (due to the force limiting) until the position trajectory moves back up and the error drops below 25 mm. Eventually the trajectory pulls the arm up off the contact surface. A plot of the actual force in the Z direction for this motion is shown in Fig. 9. The force increases quite rapidly after impact, and then stays between -40 and -45 N for the most part during the portion of the trajectory where force limiting takes place.

This work yielded several insights regarding command position-based force limiting. First, such limiting can be an effective way to safeguard against excessive interaction forces. However, the accuracy of the force limiting is related directly to the accuracy of the estimate of the Cartesian stiffness. It is important, therefore to have an idea of how well the manipulator produces an ideal stiffness. Secondly, this type of force limiting

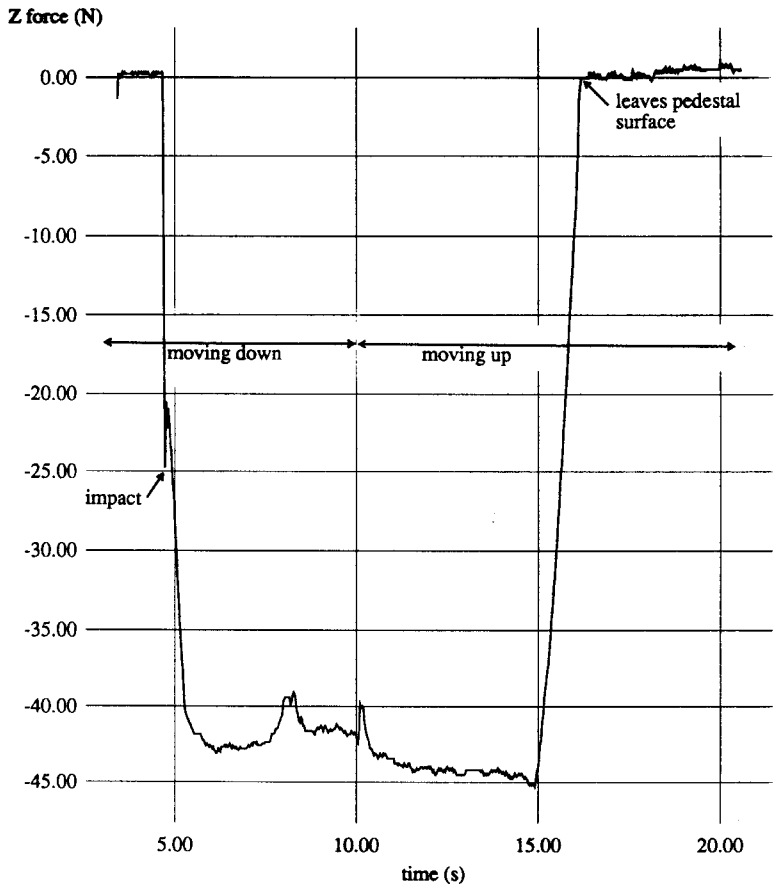

Fig 9 Recorded Z Force for Motion to Z = -290mm (Cartesian Stiffness Algorithm)

works better with Cartesian stiffness servo algorithms than with joint space algorithms. Since the Cartesian stiffness matrix is usually highly coupled in the joint servo case, deflections in the constrained direction(s) can result in undesired motion along unconstrained directions. It was also found that scaling back position directions on an individual basis allows better performance of the desired motions in the unconstrained directions, as they are not affected by the scaling. It is also important to keep in mind that command position-based force limiting can undesirably limit free space motion performance. This is because the position command will be scaled back whenever the position error gets too large, whether the error is caused by contact or just insufficient tracking. This was not a problem for the DTF-1 case, because quite slow motion speeds were to be used.

CONCLUSIONS

The NIST testbed system has demonstrated that the NASREM Architecture can successfully form the basis of an advanced, real-time controller for a robot manipulator. Proper use of the architecture promotes modular system design and independence of software and hardware. In the system as currently implemented, it takes very little time to realize a modification of the hardware. A new processor board can be incorporated within a day's time, as it is simply a matter of redistributing processes to processors. The approach could ultimately be generalized to use an on-line model of system hardware to dynamically allocate processes to processors. As processors continue to improve in performance and decline in cost, this type of flexible hardware approach becomes more attractive.

The addition of new algorithms and processes to support them is also greatly simplified by the modular design of the system, and by the properties of the atomic units upon which the system is built. The testbed system therefore provides an expandable platform for the construction of more advanced intelligent control systems, such as described in Ref. 16. For example, the generic Servo Level interface described in Fig. 4 can form the basis of the *command frame*, i.e., the set of *task frame* parameters required for execution, to the lowest level of a variety of controlled systems[16].

A good example of the generality of the testbed implementation may be found in the recent application of the system's design to a new robot. The same Primitive and Servo architecture is used to control a four degree-of-freedom stereo camera pointing device. Although this device is significantly different than the manipulator arm of the initial application, the generic functions and interfaces in the testbed allowed much of the original system to be transferred directly to the new device. This significantly reduced the development time for this new application.

REFERENCES

(1) Albus, J. S., Lumia, R., McCain, H. G., "A Control System Architecture for the Space Station Flight Telerobot Servicer," Space Telerobotics Workshop, Pasadena, CA, January, 1987.

(2) Albus, J. S., McCain, H. G., Lumia, R., "NASA/NBS Standard Reference Model Telerobot Control System Architecture (NASREM)," NIST Tech. Note 1235,

NIST, Gaithersburg, MD, July, 1987.

(3) Fiala, J. C., Lumia, R., Albus, J. S., "Servo Level Algorithms for the NASREM Telerobot Control System Architecture," SPIE Conf. - Space Station Automation III, Cambridge, MA, November, 1987.

(4) Fiala, J., "Manipulator Servo Level Task Decomposition," NIST Technical Note 1255, NIST, Gaithersburg, MD, October, 1988.

(5) Wavering, A., "Manipulator Primitive Level Task Decomposition," NIST Technical Note 1256, NIST, Gaithersburg, MD, October, 1988.

(6) Wavering, A., Lumia, R., "Task Decomposition Module for Telerobot Trajectory Generation," SPIE Conf. -- Space Station Automation IV, Boston, November, 1988.

(7) Kelmar, L., "Manipulator Servo Level World Modeling," NIST Tech. Note 1258, NIST, Gaithersburg, MD, December, 1989.

(8) Kelmar, L., "Manipulator Primitive Level World Modeling," NIST Tech. Note 1273, NIST, Gaithersburg, MD, December, 1989.

(9) Fiala, J., "Note on NASREM Implementation," NIST Internal Report 89-4215, NIST, Gaithersburg, MD, December, 1989.

(10) Lumia, R., Fiala, J., Wavering, A., "The NASREM Robot Control System and Testbed," Intl. Jour. Robotics & Automation, Vol. 5, no. 1, 1990.

(11) Fiala, J., Lumia, R., "An Approach to Telerobot Computing Architecture," NIST Internal Report 4357, NIST, Gaithersburg, MD, June, 1990.

(12) Fiala, J., Wavering, A. J., "Experimental Evaluation of Cartesian Stiffness Control on a Seven Degree-of-Freedom Robot Arm," Jour. Intelligent & Robotic Systems, to appear in 1992.

(13) Kelmar, L., Lumia, R, "World Modeling for Sensory Interactive Trajectory Generation," Third Intl. Symp. on Robotics in Manufacturing, Vancouver, July, 1990.

(14) Dagalakis, N., Wavering, A.J., Spidaliere, P., "Recommended Fine Positioning Test for the Development Test Flight (DTF-1) of the NASA Flight Telerobotic Servicer (FTS)", NIST Internal Report 4478, NIST, Gaithersburg, MD, February, 1991.

(15) Fiala, J., Lumia, R., "The Effect of Time-Delay and Discrete Control on the Contact Stability of Simple Position Controllers," submitted for publication.

(16) Albus, J. S., "Outline for a Theory of Intelligence," IEEE Trans. Sys.,Man, & Cybern., Vol. 21, No. 3, May/June, 1991.