Robotics & Computer-Integrated Manufacturing, Vol. 1, No. 1, pp. 81–95, 1984
Printed in the U.S.A. All rights reserved
0736-5845/84 $3.00 + 0.00
Pergamon Press Ltd

• Technical Papers

AN APPLICATION EXAMPLE OF THE NBS ROBOT CONTROL SYSTEM

L. S. HAYNES, A. J. BARBERA, J. S. ALBUS, M. L. FITZGERALD, and H. G. McCAIN
Industrial Systems Division, National Bureau of Standards†

The National Bureau of Standards, Industrial Systems Division, has designed The Robot Control System where high level goals are decomposed through a succession of levels, each producing strings of simpler commands to the next lower level. The bottom level generates the drive signals to the robot, gripper, and other actuators. Each control level is a separate process with a limited scope of responsibility, independent of the details at other levels, thus providing a foundation for future modular, "plug compatible" hardware and software for robotics and other real-time sensory interactive control applications.

To aid in specifying the required task decomposition and task processing, a programming language and program developing environment have been implemented. Programs at each control level are expressed as state tables, and the programming environment permits the generation, editing, emulation, and evaluation of these state tables. The control system is completely interactive, allowing the system to run freely, or be single-stepped to any level of detail.

This paper describes the first application of The NBS Robot Control System in a realistic factory environment, fully integrated with a workstation control system, database system, safety computer, gripper control system, vision system, and network.

1. THE PROBLEM

Because of the large capital investment required to install state of the art automated manufacturing equipment, it is essential that it be possible to incrementally upgrade the equipment within a factory, adding a new robot, new machine tool, vision system, etc., as funds become available and as new equipment becomes more cost effective than old equipment. What are needed, therefore, are plug-compatible systems which can be expanded and/or upgraded in much the same way as stereo systems can today.

The National Bureau of Standards, Center for Manufacturing Engineering, is implementing an experimental factory called the Automated Manufacturing Research Facility (AMRF) to provide a testbed for studying interface standards which will permit interchangeability of components.

In January 1984 the AMRF was completed to the extent that a user could select one of a number of prespecified parts, and have the AMRF manufacture that part without human intervention. The major systems in the AMRF (shown in Fig. 1) are the cell control, data administration, network, material handling control, robot cart controller, horizontal workstation control, horizontal machine tool controller, turning workstation control, turning machine tool controller, two robot control systems, two robots, a gripper control system, and watchdog safety system (not shown in figure). As of January 1984 the AMRF contained 33 different interfaces, all implemented using a single, unified approach to system interface.

Reference 8 is a detailed description of the objectives and overall philosophy of the AMRF. Reference 7 describes the overall architecture of the AMRF. Reference 6 describes the virtual cell concept, and Ref. 3 describes the vision system.

One of the most important systems within the AMRF is the robot control system, designed by Dr. Anthony Barbera, Dr. James Albus, and others at NBS. References [1,2,4,5] document the theory underlying the design of the NBS robot control system. This paper describes the way in which the control system was used for the January 1984 run of the AMRF, and is the first report detailing an actual application of the NBS Robot Control System (RCS).

2. INTRODUCTION TO THE NBS ROBOT CONTROL SYSTEM

The following section gives a very brief description of

†The National Bureau of Standards Automated Manufacturing Research Facility is partially supported by funding from the Navy Manufacturing Technology Program.

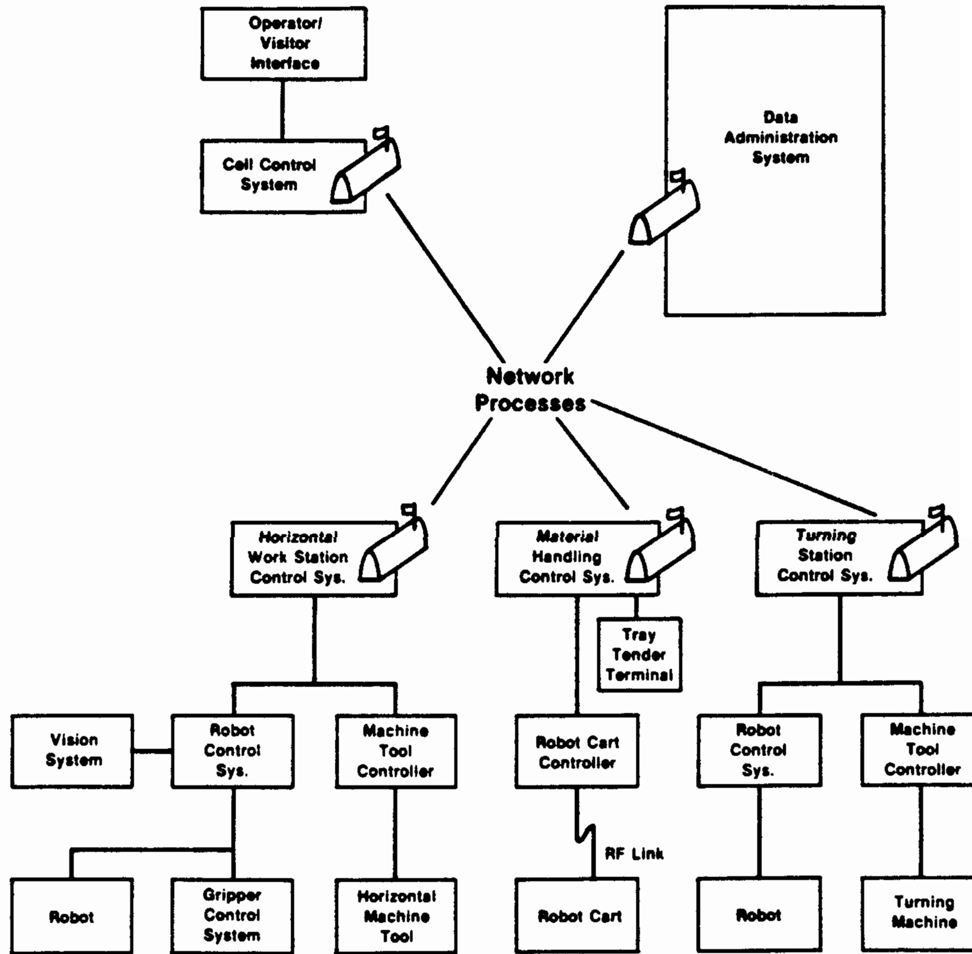

Fig. 1. Conceptual diagram of AMRF.

the structure of the *Robot Control System* from the point of view of a user.

The atomic unit within the control system is a functionally bounded module, as shown in Fig. 2. This module consists of inputs, a process, and outputs. Each functionally bounded module is given a name, and the module can be executed at any time by typing that name on the terminal. Input variables can be changed by typing <variable name=new value> and the value of any variable can be interrogated by typing <name ?>. Functionally bounded modules are executed in fully compiled machine code form; hence, the interactive capability does not severely impair efficiency.

Functionally bounded modules can be assembled into

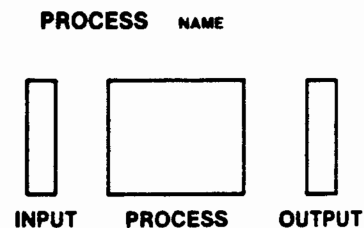

Fig. 2. Functionary bounded, named module.

an executing owner with its own name, as shown in Fig. 3. Typing the name of the owner causes the functionally bounded modules to be executed in the order listed.

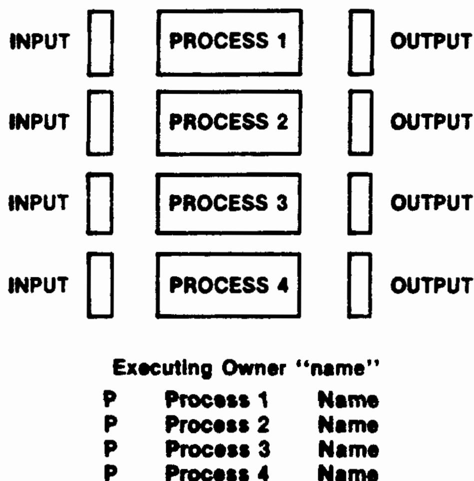

Fig. 3. Executing owner.

Executing owners are assembled into control levels. The structure of a control level, shown in Fig. 4, is generic and every level has the identical structure. A control level consists of a preprocess function, a state table function, and a postprocess function. In general, control levels are executed every cycle (40 msec for the January 1984 run, but this could have been much shorter). The preprocess functions are used for converting incoming data in a format suitable for the rest of the level. The postprocessing functions convert output data into formats required for other systems. The Kernel of a control level is a state table[4]. As shown in Fig. 4, the left side of the table is a list of conditions which test the relevant state of the world as determined from the input variables and internal machine state. Exactly one line of this table must match with the relevant state of the world. In this case, the output procedures listed on the right side of that state table line are executed. (The algorithms which execute state tables are implemented within the system as an executing owner, as shown in Fig. 4.) State tables will be described in more detail later. For the present, we note several characteristics of state tables which are not present in procedural software programs.

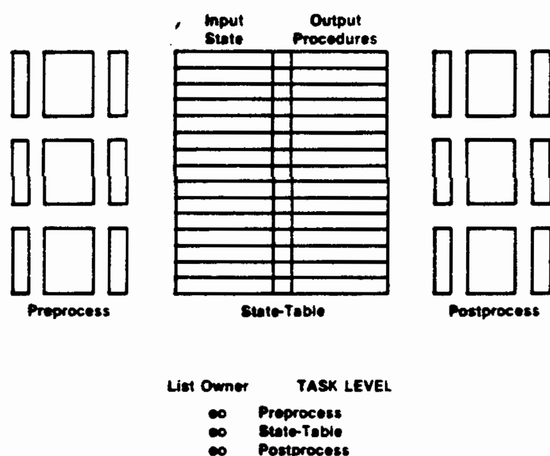

Fig. 4. Structure of a control decision level.

(a) The NBS Robot Control System does not require interrupts. Every cycle the state table inputs are examined. If a particular input condition requires immediate attention, this can be reflected in the state table, and because the state table is examined every cycle it is guaranteed that whatever action is required will be initiated within a maximum of 1 cycle time. It would of course be possible for a user to write conventional software which produced the same results, also without interrupts, but the NBS Control System structure hides much of the necessary detail from the user, and ensures consistency in complex application code.

(b) A large percentage of the code for robot applications will effect recovery from various undesirable events. Using state tables organized into control levels as described, additional situations can be handled by adding lines to the state table. With conventional software a user must first figure out where in the code this particular situation may occur, and changes and/or patches may have to be made in many places in the code. Stated differently, state tables isolate what operations are required for each specific situation, and let the application designer think in this high level manner — what to do in each case, independent of if, then, loop, or other software control structures.

(c) State tables separate the issue of when you perform a function from the function itself. This attribute of state tables is synergistic with the interactive state table mechanization which lets you execute single functionally bounded modules or lists of functionally bounded modules independently. One can debug the functions and the state tables which call the functions independently. Again, it is possible to design conventional software such that the control flow portion of the algorithms call procedures to perform the computation. One can then develop the control flow code, leaving stubs for all the procedures, etc.

achieving some of advantages of the NBS control system. Again RCS hides those details from the user, and ensures consistency.

(d) On the negative side, state tables tend to hide the sequence of execution. State transition graphs can help alleviate this shortfall as described in Section 8.2.

Figure 5 shows the combination of control levels to form more complex hierachical structures. In general, control levels execute their preprocessing, state table, and post processing functions every cycle. Each control level decomposes input commands it receives into output subcommands which it sends to the level below, while reporting status back to the level above. The interface between each level is structurally identical, hopefully providing a foundation for future modular, "plug compatible" hardware and software for robotics and other real-time sensory interactive control applications. Figure 6 shows the TASK, ELEMENTAL MOVE (E-MOVE), and PRIMITIVE levels as they were implemented in January 1984 for the AMRF. The commands and data to each level and to the vision system are shown, as is the status and data back from each level.

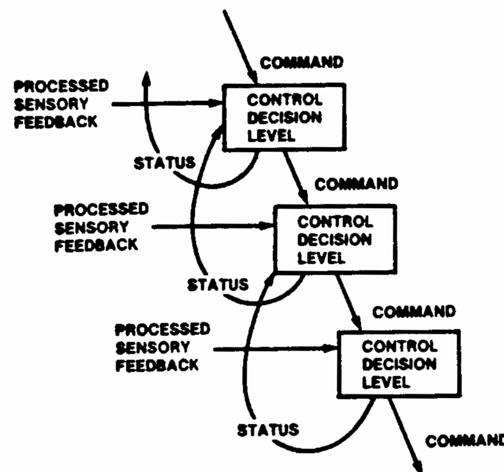

Fig. 5. Hierarchical control levels.

3. TASK LEVEL

Commands from the WORKSTATION to the TASK LEVEL are:

3.1 ACQUIRE (*OBJ* [*at* A])

From its current position, the robot will go to position *A*, (SOURCE), and grasp the named object, (OBJ). If no location *A* is specified, then the object will be acquired from the robot's current location.

3.2 MOVE ([*OBJ from* A] *to* B)

The robot will acquire the named object from location *A*, (SOURCE), and will move it to *B*, (DEST), but will not release the object. If (OBJ from *A*) is not specified, the robot will move from its current position to *B*.

3.3 RELEASE [*end-at* A]

The robot will release the object it is holding at its present position and move to position *A*. If *A* is not specified, the robot will move to the "SAFE" position associated with its current location (e.g., if the robot is at the fixture it will move to the fixture safe location a few centimeters above the fixture).

3.4 TRANSFER ([*OBJ from* A] *to* B [*end-at* C])

The robot will acquire the named object from location *A*, (SOURCE), release it at position *B*, (DEST), and move to *C*, (END-AT). If (OBJ from *A*) is not specified, and if the robot is already holding an object, that object will be moved to *B*. If (end-at *C*) is not specified the safe position for *B* will be used as the end-at location.

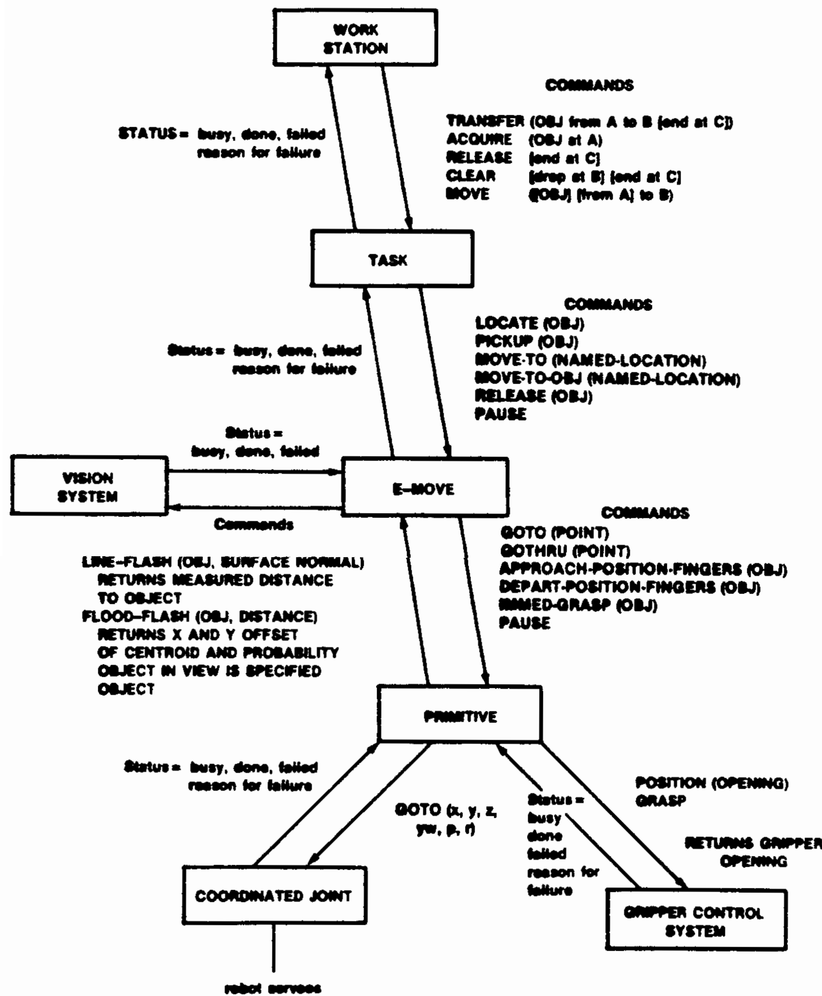

Fig. 6. Commands to each level of robot control system.

3.5 CLEAR (*drop-at* A [*end-at* B])

The robot will go to position *A* and release the part it is currently holding. Then it will end-at *B*. If *B* is unspecified the safe position of *A* is used.

Figure 7 shows the sequence of the above commands issued by the WORKSTATION to the TASK Level of the RCS to manufacture a box, one of the sample parts used in the January 1984 Run of the AMRF.

One of the primary objectives of the NBS control system is that, to the maximum extent possible, commands are data driven. There is a database system that provides data to each level in the control hierarchy to permit it to carry out its tasks. Whenever possible the need for application-specific data (such as machine locations, part geometries, etc.) is deferred to the primitive level; however, this cannot always be done. The TASK level, for example, must reference the database to determine if vision is required to acquire a part or if the exact position of the part is already known, as it would be if the part were in a fixture. For the January 1984 Run of the AMRF, every named point has the coordinates of that point (reference pose) stored in the database. If the named point is a collection or array of locations (called a pallet) then the database will include an array of offsets from the reference pose.

4. *E*-MOVE LEVEL

The TASK level issues commands to the *E*-MOVE level to implement the commands from the WORKSTATION. The commands from the TASK level to the *E*-MOVE level, shown in Fig. 6, are:

4.1 LOCATE (*OBJ*)

First, the gripper is opened fully to allow the robot

MAKE BOXT

COMMAND	PART	SOURCE	DEST	END AT
VISE-OPEN RIGHT, CLOSE LEFT				
MOVE	BOXT (2)	BUFFER	FIX-MAC1	
VISE-CLOSE RIGHT				
RELEASE				FIX-MAC2
MOVE			FIX-MAC3	
VISE-OPEN RIGHT				
MOVE			FIX-MAC4	
VISE-CLOSE RIGHT				
MOVE			FIX-SAFE	
ENGRAVE PART				
ACQUIRE	BOXT (2)	FIX-MAC5		
VISE-OPEN RIGHT, OPEN LEFT				
TRANSFER	BOXT (2)		V-BLK1	V-BLK2
TRANSFER	BOXT (3)	V-BLK3	V-BLK4	V-BLK2
VISE-CLOSE LEFT				
MOVE	BOXT (2)	V-BLK5	FIX-MAC1	
VISE-CLOSE RIGHT				
RELEASE				FIX-MAC2
MOVE	"CLOSE-GRIP"		FIX-MAC3	
VISE-OPEN RIGHT				
MOVE			FIX-SAFE	
MACHINE DOVE TAIL				
ACQUIRE	BOXT (2)	FIX-MAC5		
VISE-OPEN RIGHT, OPEN LEFT				
TRANSFER	BOXT (2)		STATION-x(y)	WS-SAFE

Fig. 7. Commands from WORKSTATION to make a box top.

vision an unobstructed view of the scene. Then the robot will analyze the scene currently in its view. Using the vision system, it will identify the specific object and determine its orientation. The robot will then close the grippers to the approach opening for that object. Finally the robot is positioned such that closing the grippers will grasp the object. The database provides the physical characteristics of the named OBJ to the vision system. The details of the recognition algorithm are defined in the vision system. If the specified object cannot be identified, a locate failure is returned to the TASK level. The interface to the vision system is discussed in Section 5.

4.2 PICK-UP (OBJ)

The gripper is closed until the gripping force on the object reaches the pre-specified value. The lower level will return a grasp failure if the final gripper opening is not within the required tolerance. The object size is acquired from the database and the gripper opening is part of the status data passed up from the Gripper Controller.

4.3 MOVE-TO (LOCATION)

The robot is moved from the current location to the named destination location. The data structures supported by the NBS control system provide many options for defining approach and departure paths for any combination of source point-destination point moves. The details of these data structures are beyond the scope of this paper. In January 1984 Run of the AMRF, almost all moves were of the simplest type. In this case the database provides pointers to a set of points defining a departure path from the initial robot location, and a set of points defining an approach path to the destination. The robot will follow a straight path between the end of the departure and beginning of the approach path. In the case that the source or destination, or both, are pallets, the move table will be adjusted to offset the approach and/or departure paths to account for the multiple sectors. The MOVE-TO command successively accesses the points along the desired trajectory, commanding the PRIMITIVE level to GOTHRU each point until the last point, at which time a GOTO command is issued to GOTO that last point.

In addition to the approach and departure trajectories a set of motion parameters (such as maximum speed, acceleration and deceleration) are set from the database for each point.

If the physical layout of the machine or geometry of the parts are changed, only the data stored in the data

structure will change — none of the program needs to be modified.

4.4 MOVE-TO-OBJ (OBJ, NAMED-LOCATION)

MOVE-TO-OBJ is exactly the same as MOVE-TO except that the approach opening for the named object is accessed from the database and before moving, the grippers are opened to the approach opening of the specified object.

4.5 RELEASE (OBJ)

The RELEASE command opens the gripper to the release opening specified in the database for the specific OBJ at its current location.

4.6 PAUSE

Each level has a pause command. When received, it passes a pause down, and a done back up to the level above.

5. VISION SYSTEM INTERFACE

The *E*-MOVE level is the only level that was interfaced with the vision system during the January 1984 test run. The vision system is capable of many complex operations including visual servoing at frame rates, and comparison of the current image with the expected image as determined from a world model. Reference [3] gives more information on the NBS Vision System. During the January 84 test run only two vision commands were used:

5.1 FLOOD-FLASH (OBJ, EXPECTED RANGE)

The name of the OBJ and the expected range of the object are passed to the vision system. The vision system accesses the database to acquire the feature values of the object, and compares these features to the object or objects in its field of view. It locates the image of the correct object and returns the x and y offset in mm in camera coordinates of the object and the roll angle of the object. It also returns the range of the object based upon an assumed surface orientation.

5.2 LINE-FLASH (OBJ, EXPECTED RANGE, EXPECTED SURFACE-NORMAL-VECTOR)

Upon receiving this command, the vision system takes a 2-line structured light picture, as described in Ref. 3. This picture reveals the range and surface orientation of the observed object. If the measured surface orientation and range do not match with the expected surface orientation and range, then the vision system returns a line-flash fail. Otherwise it returns the measured range. Several of the parts manufactured during the January 1984 Run had identical top surface dimensions and could only be distinguished by their thickness. The line flash command was used to verify that the correct part was being observed. Ranges could be resolved to 0.2%. This yields, for the viewing distance we were using, approximately 1 mm resolution. Surface orientation could be determined to approximately 2%. The vision system could function without problem in all of the ambient light conditions experienced, and contains hardware and algorithms to help alleviate problems usually encountered in dealing with highly reflective parts.

6. PRIMITIVE

The PRIMITIVE level is the level which interfaces with the robot and gripper. The NBS control system fully supports having multiple systems (the robot and the gripper controller for example) functioning simultaneously. However, the January run of the AMRF did not exploit this capability. The commands implemented by the PRIMITIVE level presently control either the robot or the end-effector but not both simultaneously.

6.1 GOTO (POINT)

The GOTO command will cause the robot to move in a straight line to the desired point. As it nears that point it decelerates and stops at the destination point. A pointer to the maximum deceleration, speed, and other motion parameters for the specific move being made are passed to the PRIMITIVE level from the E-MOVE level.

6.2 GOTHRU (POINT)

The GOTHRU command is identical to the GOTO command except that the robot does not decelerate as it approaches the commanded point. A done is returned to the *E*-MOVE level when the robot comes within a breakpoint distance of the specified destination. The *E*-MOVE level then sends the next point while robot continues to move.

6.3 APPROACH-POSITION-FINGERS (END-EFFECTOR-PARA-PTR)

The system database includes an end-effector parameter table which specifies the approach opening, departure opening, and grip opening for each part. The *E*-MOVE level passes a pointer to this table to the PRIMITIVE level. The APPROACH-POSITION-FINGERS command accesses the actual data pointed to by the END-EFFECTOR-PARA-PTR for the approach opening and commands the gripper to open to that amount. If the gripper, for whatever reason, cannot carry out that command, a GRIP failure is reported to the E-MOVE level.

6.4 DEPARTURE-POSITION-FINGERS (END-EFFECTOR-PARA-PTR)

DEPARTURE-POSITION-FINGERS is the same as the APPROACH-POSITION-FINGERS except that the departure-opening is passed to the gripper control system

	STATE TABLE ACQUIRE				STATE TABLE ACQUIRE		
INPUT	LINE 1	LINE 2	LINE 3	LINE 4	LINE 5	LINE 6	LINE 7	LINE 8
new-command	(EQ) true	(EQ) true	(EQ) true	(EQ) true	(EQ) false	(EQ) false	(EQ) false	(EQ) false
ref-command-in	(EQ) pause.	(EQ) pause.	(EQ) pause.	(EQ) pause.	(EQ) move-to-o	(EQ) move-to	(EQ) locate.	(EQ) pickup
status-in	(EQ) done	(EQ) done	(EQ) done	(EQ) done	(EQ) done	(EQ) done	(EQ) done	(EQ) done
cur-state					(EQ) 4-state	(EQ) 2-state	(EQ) 3-state	(EQ) 5-state
loc-of-obj	(NE) vis-reqd.			(EQ) vis-reqd.				
obj-acquired	(EQ) false	(EQ) true	(EQ) false	(EQ) false				
source-spec.	(EQ) true		(EQ) false	(EQ) true				
OUTPUT								
	SET SOURCE	CMD = >FAILURE	CMD = >FAILURE	VUPOS-SETUP	5-NEXTSTATE	3-NEXTSTATE	5-NEXTSTATE	D-NEXTSTATE
	4-NEXTSTATE	F-NEXTSTATE	F-NEXTSTATE	2-NEXTSTATE				
command-out	MOVE-TO-OBJ	PAUSE.	PAUSE.	MOVE-TO	PICKUP	LOCATE.	PICKUP	PAUSE.
status-out	EXECUTING	FAIL	FAIL	EXECUTING	EXECUTING	EXECUTING	EXECUTING	DONE

	STATE TABLE ACQUIRE				STATE TABLE ACQUIRE		
INPUT	LINE 9	LINE 10	LINE 11	LINE 12	LINE 13	LINE 14	LINE 15	LINE 16
new-command	(EQ) false	(EQ) false	(EQ) false	(EQ) false	(EQ) false	(EQ) false		DEFAULT:
ref-command-in	(EQ) pause.	(EQ) move-to	(EQ) move-to-o	(EQ) locate.	(EQ) pickup			
status-in	(EQ) done	(EQ) failed	(EQ) failed	(EQ) failed	(EQ) failed		(EQ) executing	
cur-state	(EQ) d-state					(EQ) f-state		
loc-of-obj								
obj-acquired								
source-spec.								
OUTPUT								
		MOVE = >FAILURE	MOVE = >FAILURE	LOC = >FAILURE	PIC = >FAILURE			STATE = >FAILURE
		F-NEXTSTATE	F-NEXTSTATE	F-NEXTSTATE	F-NEXTSTATE			F-NEXTSTATE
command-out	PAUSE.	PAUSE.	PAUSE.	PAUSE.	PAUSE.	PAUSE.		PAUSE.
status-out	DONE	FAIL	FAIL	FAIL	FAIL	FAIL	EXECUTING	FAIL

Fig. 8. State table for ACQUIRE.

instead of the approach-opening.

6.5 IMMED-GRASP (END-EFFECTOR-PARA-PTR)

This command directs the gripper controller to close the gripper until the default force of 100 lb is encountered on the fingers. Then the END-EFFECTOR-PARA-PTR is used to access the grip-opening of the part and this is checked against the current finger spacing of the gripper. If the two values do not match within tolerance, the PRIMITIVE level returns a GRIP failure to the E-MOVE level.

6.6 PAUSE

Reports done to the level above, and directs the lower level to PAUSE.

7. BELOW THE PRIMITIVE LEVEL

7.1 Robot interface

The PRIMITIVE level provides the robot three x, y, z values defining the location of the center, orientation, and rotation of the wrist plate. These nine values define the exact pose of the robot wrist plate. This interface with the robot may be sufficiently generic that it (or a similar interface) can provide the basis for a standard at some time in the future. The coordination of joint motion to achieve the required pose is robot dependent and is performed by the robot controller.

7.2 Gripper interfaces

In the January 1984 test run only two commands were used.

7.2.1. POSITION (GRIPPER-OPENING, FORCE, SPEED). This command will servo the gripper to the specified opening at the specified speed. If the force on the fingers exceeds the specified force before the grippers are within tolerance of the specified opening, a GRIP failure is returned.

7.2.2. GRASP (GRIPPER-OPENING, FORCE, SPEED). The grippers are closed at the specified speed until the force on the fingers exceeds the specified force. The gripper controller continually returns the current gripper-opening.

8. APPLICATION CODING

At each level of the control system, incoming commands are decomposed into output commands which are performed by the next lower level. The decomposition of tasks is defined through state tables and procedures called by the state tables.

Each command has a state table which is executed

whenever that command is received from the level above. (There are a total of 20 state tables, one for each command.) Here we will describe the state table for ACQUIRE (OBJ, SOURCE), as shown in Fig. 8.

8.1 Acquire state table

Looking at Fig. 8, the input variables in the upper left of the state table are the "table test variables" which represent feedback variables from the lower level and from sensory systems, as well as the values of internal variables. Every cycle of the control level, these variables are compared with the test conditions in each and every line of the state table. A line is said to match the table test variable if every input value in that line matches the table test variable's value. If no lines or more than one line matches, a STATE-TABLE error has occurred. If exactly one line matches then the procedures in the output section of that line are executed. These procedures are totally arbitrary — there are no restrictions as to what they must or must not do. In the January run these procedures are coded in a language called SMACRO; however, any language would be sufficient.

We must emphasize that there is no interest at this time in defining a standard language for programming such procedures. In fact, given today's state-of-the-art, we believe any such effort would fail. Instead we are pursuing standard interfaces, and a standard structure based on hierarchical decomposition and communicating levels of the hierarchy. The language used for specific computation seems to be of secondary importance. Because the use of SMACRO is not a primary issue, we will not present the syntax or features of SMACRO. Some SMACRO procedures will be shown, but for the purposes of this paper, they should be considered pseudo-code.

The table test variables for the state table ACQUIRE are:

- new command.
  True if and only if a new command has been received from the level above. When any command is sent, a command number is incremented and sent along with the command. This incremental-command-number is what a level uses to tell if a command is new. A pre-processing routine which executes each and every cycle sets the new-command flag if a new incremental command number is received.
- ref.-command-in.
  The command received by a lower level is echoed back.
- status-in
  The status of the lower level: Executing, Done, or Failed.
- cur-state
  An internal variable used for sequencing through the states of a state table. Specific state variables are not required. Several of the state tables used in the January Run did not use a state variable for sequencing.
- loc-of-obj
  The ACQUIRE command gives the location of the object to be acquired. The database is accessed to determine if vision is required to locate the part at that location. If vision is required, table test variable loc-of-obj = vision-reqd, otherwise loc-of-obj does not equal vision-reqd.
- obj-acquired
  This flag is an internal flag which will be true if there is currently an object in the robot gripper.
- Source-spec
  This flag will be false if the SOURCE location has not been specified.

When the ACQUIRE command is first received, table-test-variable new-command will be true, ref-command-in will be pause, and status-in will be done. Only lines 1,2,3, or 4 of state table ACQUIRE could possibly match. If either an object has already been specified or the source is not specified, lines 2 or 3 will match because the command is illegal. Procedure CMD = > FAILURE is executed. Figure 9 is a listing of most of the procedures used by ACQUIRE. As can be seen, CMD=>FAILURE simply prints a message, and sets the word "command" into a variable "failure" which will be returned to the WORKSTATION to indicate the type of failure which occurred. Procedure F-NEXTSTATE sets variable cur-state to a value indicating a failure. In this failure case, line 14 will match on the next cycle, commanding the lower level to PAUSE and returning a fail to the level above. Line 14 will continue to be executed until a new command is received.

Assuming the ACQUIRE command was legal, either line 1 or 4 will match depending on loc-of-obj. If loc-of-obj equals vis-reqd then vision is required. Line 4 matches, procedure VUPOS is executed, and command MOVE-TO is issued to the E-MOVE level. (Note that MOVE-TO is itself a procedure but its effect is simply to issue the MOVE-TO command.) Procedure VUPOS computes the robot pose for the particular sector of the unload station. The sector-number is sent from the WORKSTATION as part of the ACQUIRE command. Procedure 2-NEXTSTATE sets cur-state to 2. While the robot is moving, status-in = executing and line 15 will match. Eventually the robot will reach the specified goal point, the E-MOVE level will report status-in = done, and line 6 will match. The variable cur-state will be set to 3, and procedure LOCATE commands the E-MOVE level to locate the specified object using vision. LOCATE returns the x and y offsets and roll angle which will place the robot gripper directly over the object, aligned with the axis of the object. While LOCATE is

```
( OUTPUT-COMMAND-PROCEDURE SETUP-SOURCE. 8/31/83 )
TASK DEFINITIONS

## SET-SOURCE

    source-locpt-ptr-in => locpt-ptr-out
    obj-ptr-in => obj-ptr-out
    PRINT" SET-SOURCE "
    source-locpt-ptr-in => remember-current-loc
    false => apoint-pos-flag

end-routine

( OUTPUT-COMMAND-PROCEDURE MOVE-TO 9/16/83 )
TASK DEFINITIONS

## MOVE-TO
    obj-ptr-in => obj-ptr-out
    move-to => output-command
    PRINT" MOVE-TO "

end-routine

## MOVE-TO-OBJ
    obj-ptr-in => obj-ptr-out
    move-to-o => output-command
    PRINT" MOVE-TO-OBJ "

end-routine

( OUTPUT-COMMAND-PROCEDURE PICKUP 9/16/83 )
TASK DEFINITIONS

## PICKUP
    obj-ptr-in => obj-ptr-out
    pickup => output-command
    PRINT" PICKUP "
    true => obj-acquired
    obj-ptr-in => obj-now-in-gripper
    obj-now-in-gripper => gripped-obj-ptr-out-to-emove
    call REMOVE-FROM-BUFFER-DB

end-routine

( VUPOS-SETUP          9/13/83 )

## VUPOS-SETUP sector-in => sector-out
    station-VUPOS => locpt-ptr-out
    obj-ptr-in => obj-ptr-out
    PRINT" VUPOS-SETUP "
    source-locpt-ptr-in => remember-current-loc

end-routine

( OUTPUT-COMMAND-PROCEDURE LOCATE 8/31/83 )
TASK DEFINITIONS

## LOCATE

    obj-ptr-in => obj-ptr-out
    source-locpt-ptr-in => locpt-ptr-out
    locate => output-command
    PRINT" LOCATE "

end-routine

( REMOVE-FROM-BUFFER-DB     9/15/83 )
TASK DEFINITIONS

## REMOVE-FROM-BUFFER-DB
  if remember-current-loc (EQ) buffer
  then    if old-output-command (NE) pickup
    then
      case obj-ptr-in
        case: tbox
          if tbuffer-boxt (LE) 0-integer
            then f-state => cur-state     buf => failure
            else tbuffer-boxt (-) 1-integer => tbuffer-boxt
          endif  break:
        case:  bbox
          if bbuffer-boxb (LE) 0-integer
            then f-state => cur-state buf => failure

( REMOVE-FROM-BUFFER-DB continued )

            else bbuffer-boxb (-) 1-integer => bbuffer-boxb
          endif  break:
    case:  flag
      if fbuffer-flag (LE) 0-integer
        then f-state => cur-state buf => failure
        else fbuffer-flag (-) 1-integer => fbuffer-flag
      endif break:
    case:  hous
      if hbuffer-hous (LE) 0-integer
        then f-state => cur-state buf => failure
        else hbuffer-hous (-) 1-integer => hbuffer-hous
      endif break:  default:  break:
    end-case endif  endif
end-routine
```

Fig. 9. SMACRO procedures used by ACQUIRE.

executing, the TASK level state table for ACQUIRE will again match on line 15. When LOCATE completes, status-in will equal done, and line 7 will match. The procedures executed in line 7 will set the NEXTSTATE to 5, and will execute procedure PICK-UP. This procedure, shown in Fig. 9, does a number of things.

1. Passes the object-pointer down to the *E*-MOVE level so the PICKUP command can use the database to determine the expected size of the object to be grasped and compared it with the actual finger spacing after the grasp operation.
2. Sets the value PICKUP into the output command variable. This variable determines which state table will be executed at the E-MOVE level.
3. Sets true into object-acquired because as soon as the pickup completes, an object will be in the gripper. If the pickup fails to pickup anything (fingers close to 0 spacing) this flag will be reset to false.
4. Sets the object pointer into variable obj-now-in-gripper. This variable always tells what is currently in the gripper. If the E-MOVE level returns a fail because the grasped object was the wrong size, this value will be set to "unknown-object".

```
( OUTPUT-STATUS-PROCEDURE EXECUTING)
TASK DEFINITIONS

## EXECUTING

    executing => status-report
    PRINT'' executing=>status-report ''

end-routine


( OUTPUT-STATUS-PROCEDURE DONE 4/8/83 )
TASK DEFINITIONS

## DONE

    done => status-report
    PRINT'' done=>status-report ''

end-routine


( OUTPUT-COMMAND-PROCEDURE PIC=>FAILURE 8/31/83 )
TASK DEFINITIONS

## PIC => FAILURE

    pic=>failure
    PRINT'' PIC=>FAILURE ''
    false => obj-acquired
    null-obj => obj-now-in-gripper
    null-obj => gripped-obj-ptr-out-to-emove

end-routine


( OUTPUT-COMMAND-PROCEDURE CMD=>FAILURE 8/15/83 )
TASK DEFINITIONS

## CMD=>FAILURE

    cmd => failure
    PRINT'' CMD=>FAILURE ''

end-routine


( OUTPUT-COMMAND-PROCEDURE MOVE=>FAILURE 8/31/83 )
TASK DEFINITIONS

## MOVE=>FAILURE

    move => failure
    PRINT'' MOVE=>FAILURE ''
    null-loc. => remember-current-loc

end-routine
```

Fig. 9 (continued).

5. Calls REMOVE-FROM-BUFFER-DB. It is the robot control system's responsibility to maintain how many of each parttype are currently on the buffer table. Procedure REMOVE-FROM-BUFFER-DB keeps track of the number of each part on the buffer, and will return a failure if there are no parts of that type currently available for this PICKUP operation.

When the PICKUP command is done line 8 of the ACQUIRE state table matches and the ACQUIRE command is complete. A PAUSE is sent down to the next lower level, a DONE is reported back to the WORKSTATION, and the next state is set to the done-state. From then on, line 9 will match until a new command is received.

In the case that the loc-of-obj is not equal to vis-reqd, then vision is not required. Line *1* will match, and the MOVE-TO-OBJ command will result in the gripper being opened to the approach-opening of the named object, and the robot moving to the source location. When the MOVE-TO-OBJ is done, line 5 matches and from there execution proceeds as already described for the case where vision is required.

Lines 10 – 13 handle the several types of failure which may occur. In these specific cases, no error recovery was attempted. The type of error is simply reported to the level above via the variable "failure" passed to the WORKSTATION.

The other state table and procedures all look and function similar to the ACQUIRE.

8.2 State graphs

Alternatively, state tables can be written as state graphs. Figure 10 is a state graph of the ACQUIRE command. The two formats are nearly isomorphic, and we are working on a system to convert from either format to the other. The circled numbers of Fig. 10 correspond to the lines of the ACQUIRE state table shown in Fig. 8. State tables emphasize the conditions under which an operation is performed but tend to hide the sequence of execution. The converse is true for state graphs, so each have their place.

8.3 Pre and post processing

The particular application being reported on in this paper did not require any application specific pre or post processor routines. However, there are several pre and post processor routines which are required by each control level. These routines (identical in every control level) test whether the new commandflag should be set true, or whether a new output command has been issued and the incremental request number should be incremented.

9. COMMUNICATIONS

The actual transfer of command and status buffers normally occurs every cycle, and the details of this transfer are invisible to the application state tables and procedures. The names of buffers to be moved and the name of the destination buffer are entered into the

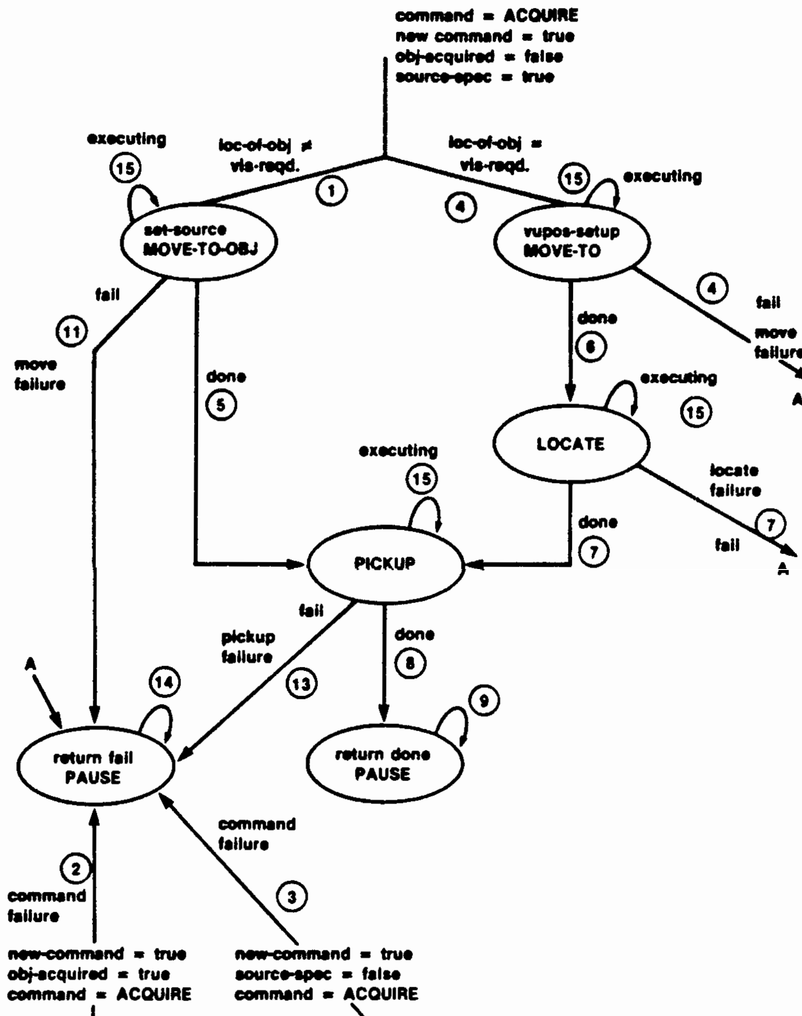

Fig. 10. State transition diagram for ACQUIRE.

communication process and the rest is automatic. This is true regardless of whether the levels reside on the same or separate processors, or whether they are buffers to or from control systems levels or external systems such as the gripper control system or vision system. Figure 11 shows all but two of the command and status buffers used by the robot control system in the January 1984 run of the AMRF. The fact that these buffers are the only mechanism for communication between levels and external subsystems greatly simplifies integrating large systems together. At the hardware level, the communications processor transfers the output buffers of all processors in a common memory area and updates any input buffers appropriately from that. The communications processor has a 2K byte common memory dedicated for this purpose. The diagnostic sytem can access all variables and record the complete state of the system at every cycle. Those recordings can then be studied or displayed in real time or off-line.

Although the NBS Control System cycles as defined above, this does not mean that any level must necessarily respond in one cycle. From the point of view of the state tables and procedures, each level is fully asynchronous. When a command is issued to a lower level, an incremental command number is incremented and passed down with that command, as can be seen in Fig. 11. The issuing level's new command flag will not drop to false until the new incremental command number is echoed back. Likewise, when a state table line is matched, and a new state is to be set, the current state variable (cur-state in Fig. 8), will not be updated until the lower level has acknowledged receipt of its command by

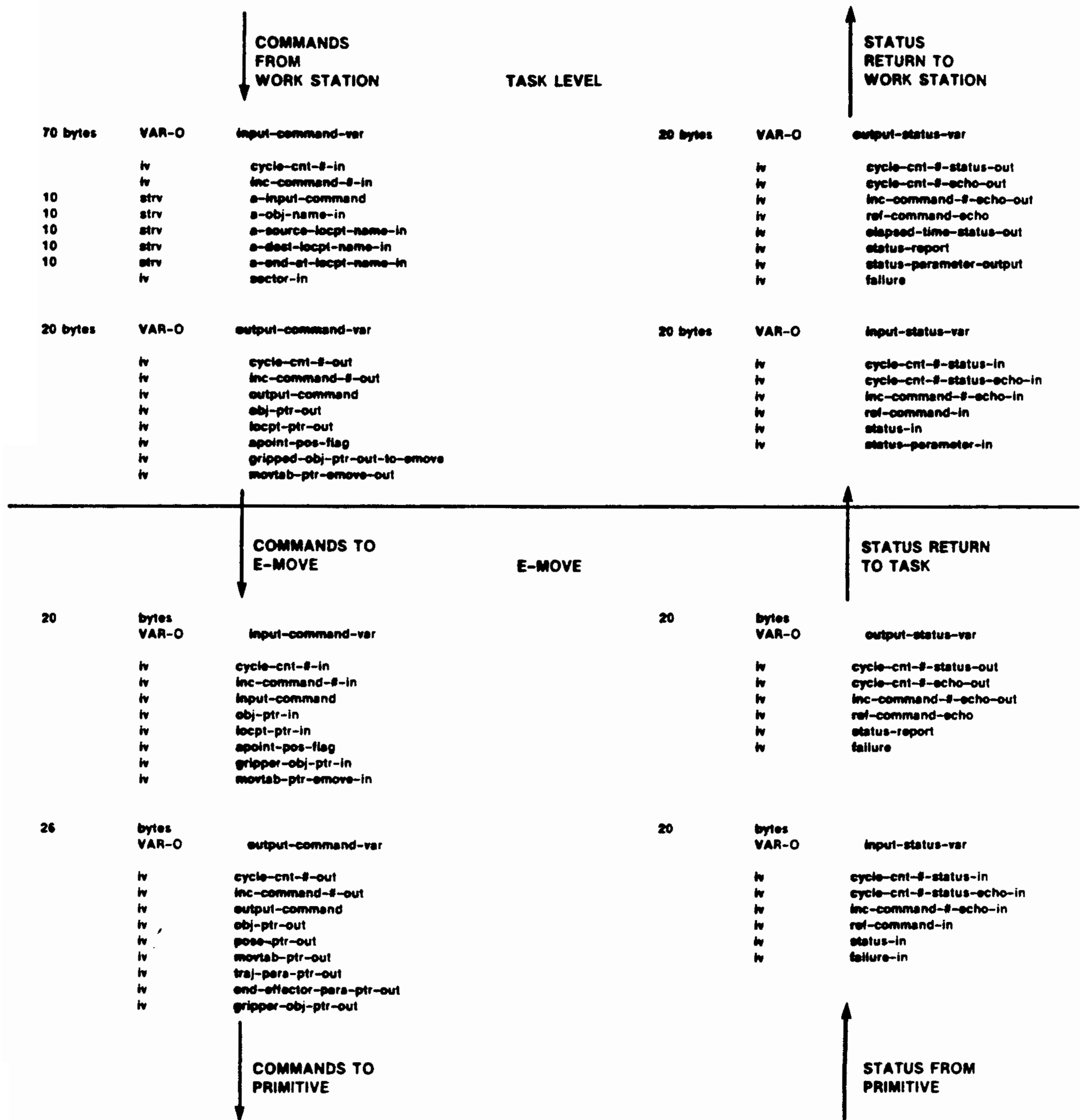

Fig. 11. Communications.

echoing back the new incremental command number. To understand why this is necessary, consider Fig. 8, line 5. The command PICKUP is to be issued to the *E*-MOVE level, and the current state value is to be set to 5. When eventually the PICKUP is done, line 8 will match. When line 5 sets the next state value to 5, and issues the PICK-UP command, if the lower level takes several cycles to respond, there will be cycles where ref-command-in is still move-to-obj, and status-in is still done, yet the state variable cur-state would now be 5. In the above case there

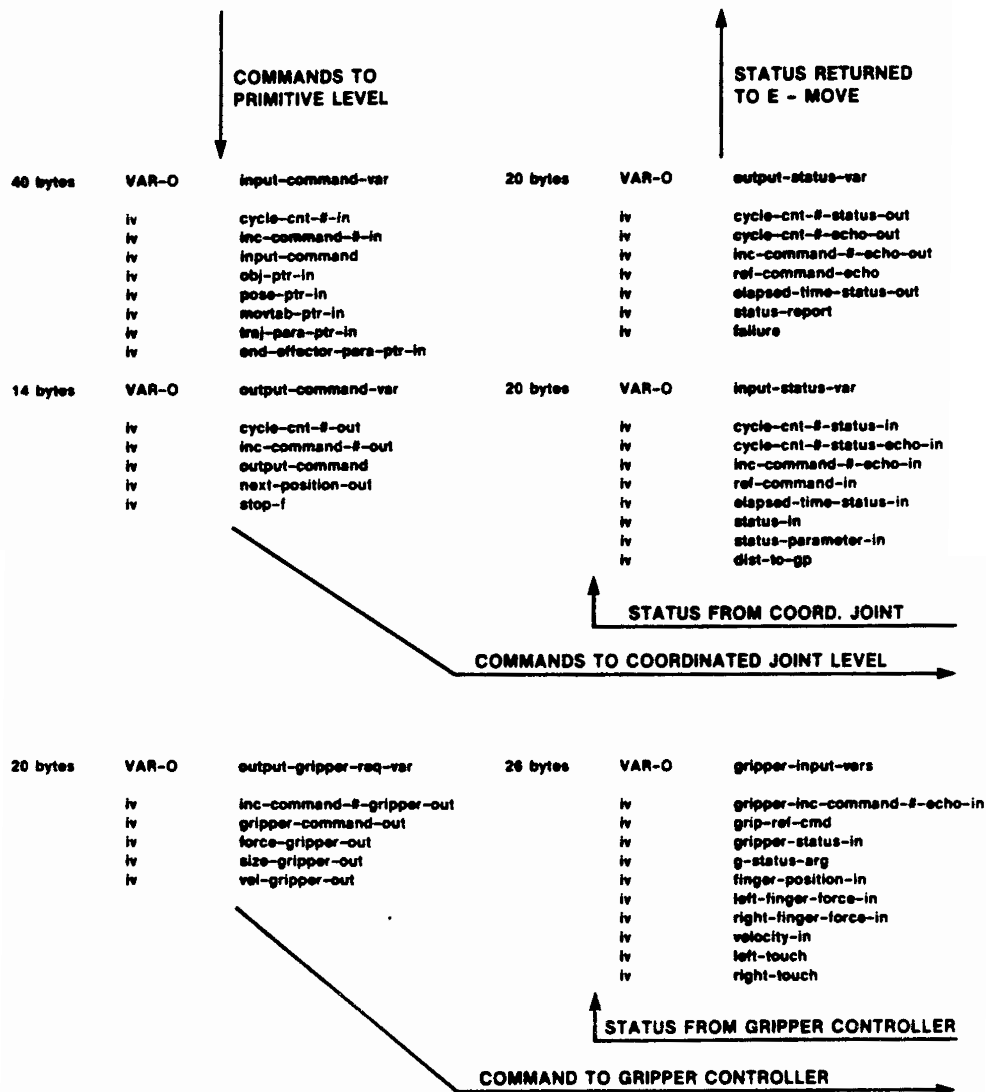

Fig. 11 (continued). Primitive level communication.

would be no matches in the state table. This problem could be solved by adding additional lines in the state table to handle these cases but it would nearly double the size of the table. Instead, the procedures NEXTSTATE do not actually change the value of cur-state. They store the new state value in a temporary location, and a preprocessing procedure stores that value into cur-state every cycle where the incremental-command-number-echo-in equals the incremental-command-number-out. At the primitive level, if we are issuing commands to two different controllers, both command-number-echo-in values must match their corresponding command-out values.

10. USER INTERFACE

The basic emphasis of FORTH in providing an interactive user environment was expanded and further structured. This language was then used as a basic tool to design and implement the multiprocessor operating system, the procedural language SMACRO, various editors, file managers, data dictionaries, and graphics support operators.

The control system is completely interactive, allowing the user (for diagnostic purposes) to single-step the system at any level of detail. For example, any procedure may be run by typing its name. To cycle the entire TASK level, one types <TASK-LEVEL>. The preprocessing

routines may be run by typing <PREPROCESS>. The state table can be cycled by entering <STATE-TABLE>. Variables may be read out and/or modified by name at any time. Likewise procedures may be changed using the editor, the modified procedure(s) reloaded, and the system will execute with the new definitions.

For the January run of the AMRF separate software subsystems were coded to simulate upper and lower levels including workstation controller, the coordinated joint level, the gripper, the sensory systems and all of the levels of the robot control system itself. These simulations, along with the interactive capabilities of the system, greatly facilitated debugging by permitting portions of the system to execute as if the remainder of the system were running.

An extensive diagnostic system has also been implemented. Because of the structure of the communication system, every cycle the relevant machine state of all processes is available in the common memory. The diagnostic processor has access to the common memory every cycle. A log of the machine state is made, permitting the NBS Robot Control System to be stepped so the machine state at any time can be observed. This greatly facilities debugging sensory interactive real time applications where events happen at unpredictable times and there is usually no way to recreate the exact sequence of events which caused a particular problem.

In addition to the log described above, a menu driven system allows the user to select a wide variety of real time graphical outputs including multiple *xy* graphs using any variables in the system. For example, one could display a graph of tool tip velocity versus time, or the orthographic projections (*xy*, *yz*, and *xz*) of the tool tip position. This allows a graphic display of robot position even when the robot is not available to the application designer. The values which are displayed can also be controlled by the state tables being executed so that different displays will appear depending on exactly what the robot is doing at that instant.

In addition to graphs and plots of variables, a stick figure robot can be displayed, and will move as commanded by the output of the PRIMITIVE level in exactly the same way as the real robot. This has proved extremely useful for debugging software, alleviating the problems of attempting to run a real robot under control of untested software.

This article was prepared by United State Government employees as part of their official duties and is therefore not subject to copyright.

Identification of commercial software does not imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the software is necessarily the best available for the purpose.

REFERENCES

1. Albus, J.S., Barbera, A.J., Fitzgerald, M.L.: Programming a hierarchical robot control system. 12th International Symposium on Industrial Robots, Paris, France. June 1982.
2. Albus, J.S., Barbera, A.J., Nagel, R.N.: Theory and practice of hierarchical control. Twenty-third IEEE Computer Society International Conference. Sept. 1981.
3. Albus, J.S., Kent, E., Nashman, M., Mansbach, P., Palombo, L., Schneier, M.: Six-Dimensional Vision System. *SPIE* **336**: Robot Vision, 1982.
4. Albus, J.S., McLean, C.R., Barbera, A.J., Fitzgerald, M.L.: Hierarchical control for robot in an automated factory. 13th ISIR/Robots 7 Symposium Proceedings, Chicago, Illinois. April 1983.
5. Barbera, A.J., Fitzgerald, M.L., Albus, J.S.: Concepts for real-time sensory-interactive control system architecture. Proceedings of the Fourteenth Southeastern Symposium On System Theory. April 1982.
6. McLean, C.R., Bloom, H.M., Hopp, T.H.: *The virtual manufacturing cell*. Information Control Problems in Manufacturing Technology. Washington, D.C., McGregor & Werner. 1982.
7. McLean, C.R., Mitchell, M., Barkmeyer, E.: A computer architecture for small-batch manufacturing. *IEEE Spectrum* 59–64, 1983.
8. Simpson, J. Hocken, R., Albus, J.S.: The automated manufacturing research facility of the National Bureau of Standards. *J. Manufacturing Systems* **1**: 1982.