An Integrated Robot System Architecture

RUSSELL H. TAYLOR, MEMBER, IEEE, AND DAVID D. GROSSMAN, MEMBER, IEEE

Invited Paper

Abstract—**This paper describes the architecture of a robot system designed both to work in an integrated manufacturing environment and to support continuing research in programmable automation. Major system components include a controller, robot and sensor hardware, operator's pendant, and system software. A new high-level interactive language, AML, allows the user to combine manipulation, sensing, computational, and data processing functions provided by the system. Important aspects of the system design objectives, major functional components, and the AML language are described, and examples drawn from an actual production application are used to illustrate the interrelationship of the topics discussed.**

I. INTRODUCTION

THE WIDESPREAD introduction of a new generation of intelligent automation systems, in which industrial robots operate under control of modern digital computers, could have a significant effect on industrial productivity within the next decade. These automation systems combine several fundamental capabilities:

Manipulation: The ability to move physical objects about a work station.

Sensing: The ability to obtain information from the physical world through the use of sensors.

Intelligence: The ability to use information to modify system behavior in preprogrammed ways.

Data Processing: The ability to interact with data bases to keep records, generate reports, and control activity at the work station.

The flexibility inherent in this combination offers important advantages over current fixed automation. Viewed as a stand-along device, a robot can use sensing and software to compensate for positional misalignments, reducing the requirement for high accuracy and expensive fixturing. Viewed as a part of a horizontally integrated manufacturing system, a robot can accommodate varying workpieces, spreading the capital costs over more products. It can interleave fabrication, assembly, and inspection operations, and it can allow more efficient production scheduling. Viewed as part of a vertically integrated design and manufacturing system, a robot can allow the automated manufacturing of unique customized products.

At IBM, the Automation Research project has been developing programmable automation systems since 1972 [1]-[10]. After five years of work, we had built a working experimental assembly robot and had accumulated considerable experience in programming it to perform a variety of demonstration tasks.

This experience led in 1978 to creation of a second-generation research robot system. It featured enhanced functional capabilities and a considerably improved programming interface with a new programming language, AML. This system became the prototype for the IBM 7565 Manufacturing System, which is in use in a number of manufacturing and laboratory sites around the world.

Since 1978, the system has evolved through many development cycles requiring cooperation between the Research group at Yorktown Heights and IBM's Advanced Manufacturing Systems group in Boca Raton. This paper describes the current Research version of the system architecture. Although this system is very similar to the 7565 version, it contains several experimental enhancements to support robotics research activities.

Manuscript received January 18, 1983; revised January 31, 1983.

The authors are with the Manufacturing Research Department, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

II. DESIGN OBJECTIVES

A. Experience with Previous System

Prior to the design of a new robot system, we accumulated considerable experience in applying out first-generation system [1]. As early as 1974, we used it to demonstrate automatic assembly of 22 out of 25 pieces that constituted the "rail support" of a typewriter. As part of the demonstration, the robot picked up and used both hand and power tools. The assembly program made extensive use of sensory feedback for precise alignment of parts.

In another exercise, the robot assembled 5 pieces of a magnetic deflection yoke from an IBM display terminal, using sensory feedback to fit the parts onto an assembly fixture. We also programmed the robot to prepare kits from palletized parts, adjust the feel of a typewriter keyboard, insert integrated circuit modules and terminating resistors on a circuit board, place cylindrical heat sinks into an array of holes in a plate, plug electronic cards into a tester, etc.

Our earliest application experiments required many months to program, and they executed slowly and unreliably. Later experiments showed considerable progress in all these areas. The improvement was due to increased experience, coupled with system enhancements that embodied much of this experience. In particular, strategies were developed for sensory calibration, and new commands were provided to support these strategies. Similarly, enhancements were made to allow the language to be used not just for end application programming, but also for generating software systems to simplify the programming and debugging process. The importance of teleprocessing and data-base management became clear.

In 1977, at the same time that we were designing our new hardware and software, we became closely involved in installing a two-armed robot system for testing computer mainframe backplane wiring at an IBM factory in Poughkeepsie, NY. In this application, the robot first uses sensors to feel for the location of 20 circuit boards on a large frame. Next, it probes

0018-9219/83/0700-0842$01.00

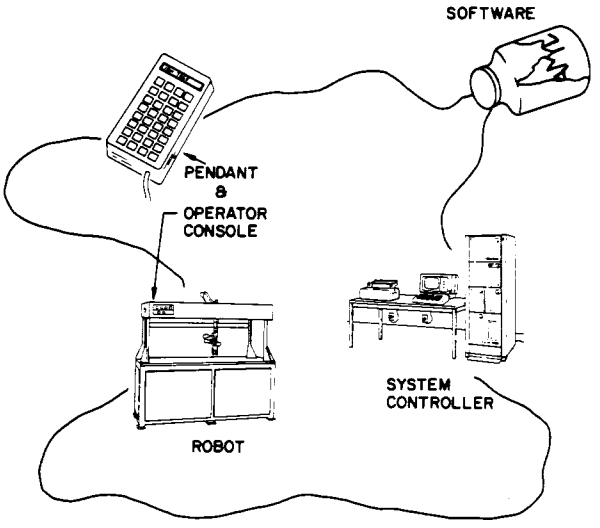

Fig. 1. Major system components.

about 4000 pairs of pins corresponding to installed wires to verify electrical continuity and the absence of short circuits to ground. A manufacturing control data base provides the nominal locations of the installed wires at the correct engineering change level for the particular computer being tested.

This application combined many features of early applications with some surprisingly tough performance requirements; our experiences with it strongly influenced the design objectives of the new robot system. Particularly important characteristics included the need to link to a factory control system, the use of multiple arms, the necessity of sensing, the need for good operator interfaces, and the importance of a powerful programming language with suitable debugging tools.

B. Key Problem: System Integration

When we started doing demonstrations with our first-generation robot system, we believed that robotics was mainly concerned with motion. From our application experience, we came to realize that the key problem was actually the integration of a broad spectrum of capabilities. Addressing this problem required a control computer and powerful software, providing three main classes of function:

First is the system configurability. Since no two robotic applications are exactly alike, there must be provision for the user to specify what the workstation will consist of. This specification includes the number of robot arms, their degrees of freedom, as well as sensing and other instrumentation. It includes interfaces to other equipment for material flow, such as conveyors, feeders and fixtures, and for data flow. It also includes data interfaces to other computers, allowing integration with factory design, inventory, and control systems.

Second is the user interface. The broad spectrum of users who interact with applications imposes difficult and partially conflicting design objectives. An easy to learn language subset is needed for novice users, but expert programmers need the power and flexibility of a system programming language for highly sophisticated applications. The broad spectrum of potential robotics applications requires a powerful general-purpose, modern programming language, suitably enhanced to provide real-time control of robots and other devices. It also needs interactive programming and debugging tools that provide editing, tracing, breakpointing, etc.

Third is the system reliability. The system must continually monitor its own performance and shut down in the event of any malfunction. Furthermore, there should be diagnostic sensors and software to allow defects to be isolated for rapid field repair.

C. Related Objectives

While our primary interest has been on developing the technology of practical assembly robots, we have tried to make our approach compatible with some long-range research topics being pursued primarily in academic laboratories.

The first of these is "artificial intelligence," a term generally used to describe machines that exhibit human-like "intelligent" behavior. Since a significant component of human intelligence is our ability to sense and manipulate the physical world, a number of artificial intelligence laboratories have traditionally supported "hand-eye" research projects [11]-[13], etc. As industrial robots have become more important, similar research has spread to other laboratories as well [14]-[16], etc. Current hand-eye research topics include the design of dextrous hands, force control, and picking disordered parts from bins. A design objective of our robot system was to provide an experimental facility suitable for performing subsequent hand-eye research.

A second related topic is automatic programming. In the context of robotics, the problem is to produce a system that can translate a description of the manipulatory task to be performed into a specification of the robot and sensing actions required to do it. For example, one can imagine a system that understands such statements as "place interlock on bracket such that interlock base contacts bracket top," rather than the dozens of robot commands needed to accomplish this task. Research in this vein has been done in our project [4] and is ongoing at several universities [17]-[20], but a practical system of this type still seems unlikely in the near future. A design objective of our robot system was to provide a target to allow research on task level systems that generate robot programs.

III. System Overview

The major functional components of the system are illustrated in Fig. 1. The central element is a system controller, which is responsible for coordination of all activity at the robot workstation. Essentially, it consists of a minicomputer, data pro-

cessing peripherals, and electronic interfaces to workstation devices, including robot joints, encoders, and sensors.

One component of the system controller software provides an interactive programming environment for a high-level programming language, AML [9]–[21], which is used for all application programming. A second software component performs trajectory planning, motion coordination, sensor monitoring, and other real-time activity in response to commands from the AML interpreter. A third component supplies standard supervisor services, such as file and terminal I/O.

A pushbutton control box known as a pendant is attached to the controller. Although it is used primarily for manually "teaching" robot locations, the buttons are programmable and may be used to augment the basic operator control panel, which has a small number of fixed-function switches and lights.

Digital and analog input/outputs allow a variety of devices to be attached to the system controller. Chief among them are the robot joint actuators, encoders, and sensors.

The robot arm is a seven degree-of-freedom, hydraulic manipulator, whose joints are controlled by analog position servos in the system controller. The arm geometry is basically Cartesian, with three linear joints. Three revolute wrist joints are used to position the gripper in roll, pitch, and yaw coordinates, and a seventh "joint" is used by the gripper. The controller is capable of running up to 14 joints, thus permitting two full arms or a greater number of reduced arms.

Sensors typically include force transducers and a light beam presence sensor mounted in the fingers, several solid-state television cameras, and miscellaneous application-specific sensors, such as empty indicators on feeders.

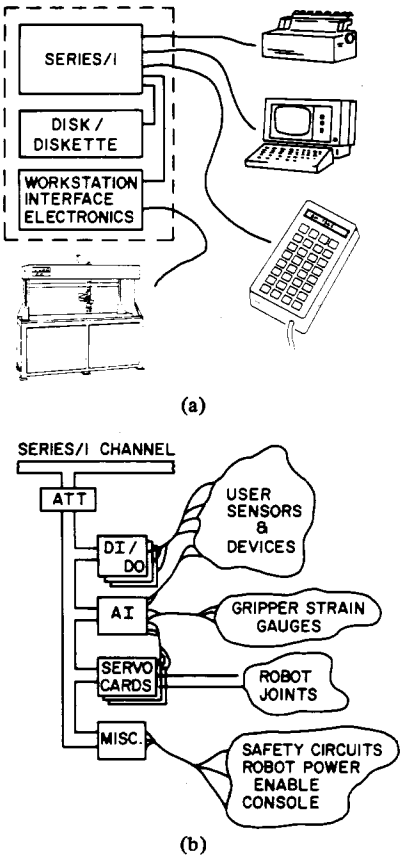

Fig. 2. System controller. (a) Overview. (b) Workstation interface electronics.

IV. System Controller Hardware

The system controller, selected in 1977, is an IBM Series/1 minicomputer. It has at least 256K bytes of main memory, relocation hardware to allow extended addressing beyond 64K bytes, floating-point hardware, and several sensor-based digital and analog input/output cards. Data processing peripherals vary somewhat according to the particular application requirements but normally include such items as keyboard, display terminal, printer, diskette drives, hard disk, and teleprocessing attachments to other computer systems.

The workstation interface electronics are illustrated in Fig. 2. They perform a number of functions, including positional control of robot joints, safety interlocks and robot power controls, input of sensor values, and output of control signals to miscellaneous devices at the workstation.

A. Joint Control Electronics

Position control of robot joints is accomplished by dedicated electronics cards packaged inside the system controller. This choice was a critical one for the system architecture and was a compromise based on several competing considerations:

System simplicity argues for all computation being performed in a single central processor. On the other hand, the limited computational power of our central processor makes some degree of parallelism mandatory. The simplest form of parallelism is to have outboard dedicated servo controllers, designed to provide a clean interface between the joints and the system software. In this context, "clean" means that widely different types of joints should look essentially the same to the central processor.

For the sake of simplicity, two assumptions were made. First, robot joint actuators are assumed to be dynamically independent. For anthropomorphic manipulators, this assumption would be very poor, but it is quite a good approximation for Cartesian kinematic structures like the 7565. The second assumption was that it is always safe to stop the robot. Thus if an error or unexpected condition arises or if the central processor needs time to analyze a situation, then the robot can be stopped for an indefinite period of time. This assumption is poor in some situations. For example, it is unwise to hold a part stationary for a long time in an induction heater or on a moving conveyor belt. For assembly workbench applications, however, the assumption is reasonable.

The central processor can issue the following commands to the robot joint control cards:

Set Position Goal. A 16-bit two's complement integer is transferred from the Series/1 to the servo-attachment, which interprets the value as a position command for that joint. Controlled acceleration, velocity, and deceleration are achieved by real-time software running in the Series/1, which generates new position goals every 20 ms. Coordinated motion of several joints is achieved in the same manner. Control software is also responsible for translating engineering units (such as inches or degrees) into the device units required by the interface.

Read Joint Position Error. A 16-bit two's complement integer is transferred to the Series/1. This number is interpreted by the real-time software as the difference between the commanded and actual joint positions. This value is used by the real-time software to verify correct functioning of the joint and for final nulling at the end of motions. Scaling of the reported value back into engineering units is performed by soft-

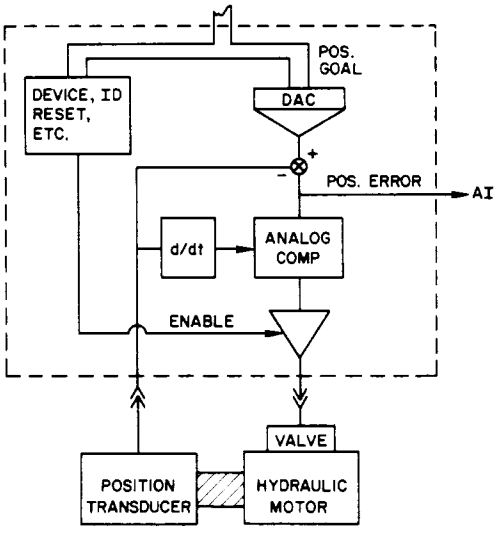

Fig. 3. Joint servo attachment.

ware; the scale factor need not be identical to that used for the position goal.

Read Device Identifier. A 16-bit code is transferred from the attachment to the Series/1. This code is required by the Series/1 architecture; it contains information identifying the atachment type and hardware configuration. It is routinely checked as part of the software intialization procedure and by various hardware maintenance programs.

The flexibility of this interface has allowed us to attach a variety of position-controlled actuators to the system controller and configure them as joints with only minor changes in software tables. This has permitted us to experiment with a number of different kinematic configurations, making it especially easy to design special-purpose robots for particular applications, while retaining the advantages of common software and programming support.

In the 7565, analog transducers are used to provide joint position feedback, and the actual servo control is performed by analog "PD" loops, as shown in Fig. 3. This scheme is a considerably enhanced version of that adopted in 1973 in the first-generation IBM system [1] and has proved to be both simple and reliable.

The principal drawbacks come from the inherent inflexibility of using an analog servo controller with a fixed design. The most obvious disadvantage is that changes in actuator characteristics usually require adjustments or design changes to the servo card, complicating both maintenance and interfacing of new joint types. A second drawback is that tuning a fixed design to operate well under widely varying conditions of speed and load is quite difficult. It should be noted, however, that the system architecture is fundamentally compatible with the use of more flexible programmable servocontrollers.

B. Miscellaneous Workstation Controls

Wherever possible, the responsibility for supervision of the robot workstation is left to software. The necessary signal interfaces are provided by an attachment card in the system controller, which is also responsible for certain safety functions that cannot be delegated to the joint control cards. For example, certain external inputs automatically cause the robot to be shut down.

Commands supported include: read and write data, read status, read device identifier, device reset, and set safety monitor. The last command restarts a safety timeout circuit on the card. If this circuit is not restarted at least every 25 ms, the robot power supply is turned off and all servo drive outputs are disabled. This timeout is only reset after the software has verified that all joint position errors are within prescribed tolerances.

C. Sensor I/O Interfaces

Standard analog and digital input cards provide low-level sensory input. Digital inputs are two's complement binary numbers from 1 to 16 bits long, with optional sign propagation. System software permits the configuration to be specified and associates one or more "logical" sensors with each physical sensor. A software command that reads a logical sensor performs automatic conversion from device units to engineering units.

There is also provision for attaching a solid-state video camera, using either directly programmed or cycle-steal (DMA) input/output. System software permits the configuration to be specified for a small number of cameras. Although the architecture is quite general, CPU speed and logical address size effectively limit the system to black/white images with a resolution of 128 by 128 pixels. Software commands are provided that set the black/white threshold and input images either in binary or run coded format [10].

V. Robot and Sensor Hardware

The key design objectives for both the robot and sensor hardware were modularity and configurability.

A. Robot Structure

The robots's kinematic structure is illustrated in Fig. 4. It consists of a rectangular frame supporting a hydraulically driven arm assembly, which, in turn, consists of three linear actuators, a wrist consisting of three rotary actuators, and a gripper with linearly actuated fingers. The three linear actuators are arranged orthogonally to carry the wrist while providing independent motion in X, Y, and Z directions. The three rotary actuators of the wrist control the angular orientation of the gripper in roll, pitch, and yaw. These actuators are arranged so that all three axes of rotation intersect in a single point and so that the yaw–pitch and pitch–roll axes are orthogonal.

For assembly applications, the orthogonal structure has several advantages over anthropomorphic style manipulators. Assembly applications frequently involve many short motions, the durations of which are limited primarily by robot accelerations. A Cartesian box frame robot can be made quite rigid, allowing it to be operated with high accelerations.

The rigidity also improves the robot's precision, which is important in assembly. Furthermore, the orthogonal structure makes the precision essentially independent of the robot's position. In addition, it decouples the dynamics of the X, Y, and Z axes, allowing them to be controlled largely independently. The orthogonal structure also allows considerable modularity in constructing robots of different sizes and shapes. The same linear joint is used on each axis, and the axes can be built to different lengths.

In many assembly applications, only three translational degrees of freedom are required. With an orthogonal structure, three linear joints can provide this freedom, whereas a robot with only revolute joints needs at least five joints to provide the same freedom. Furthermore, in applications requiring only three degrees of freedom, Cartesian coordinates are identical

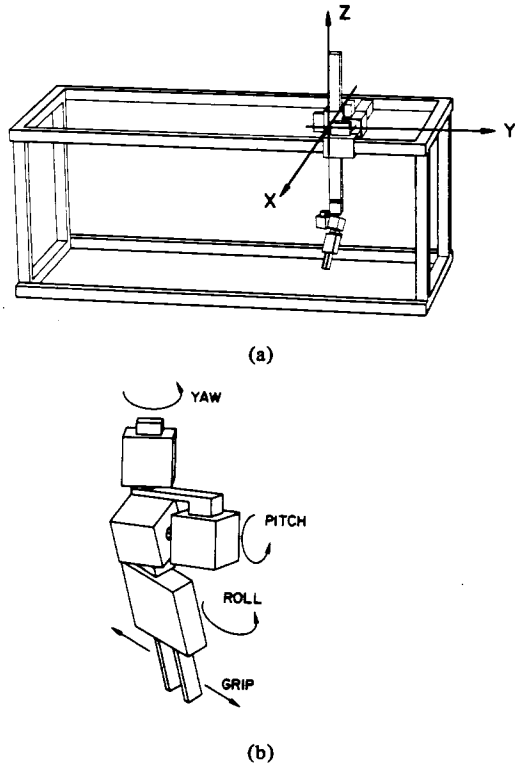

Fig. 4. Kinematic structure. (a) Rectilinear axes. (b) Wrist and gripper.

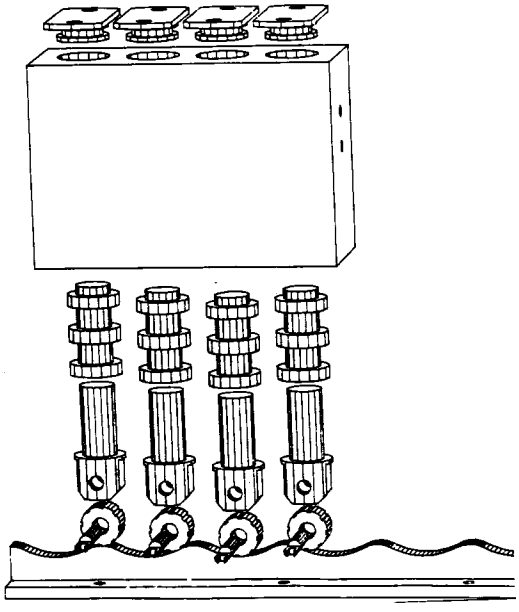

Fig. 5. Linear hydraulic motor. This figure was generated using the geometric modeling system described in [5].

to "joint angle" coordinates, eliminating the need for subroutines that perform coordinate transformation computations.

When multiple arms must work in coordination at the same workstation, collision avoidance is an important issue. With an orthogonal structure, multiple arms can share a linear axis, reducing the cost and simplifying the collision avoidance computation to a single degree of freedom.

B. Actuators

1) The linear axes of the robot are driven by a novel hydraulic actuator invented by H. Panissidi [22] and shown in Fig. 5. A multilobe linear cam extends along the full length of each driven linear axis. Sliding along each axis is a motor block with piston-driven rollers at quarter pitch locations along the cam.

The pistons have integral spool valves that actuate adjoining pistons cyclically, providing self-commutation. As each piston passes through the midpoint of its stroke, its spool valve directs oil under pressure to the next piston. The slope of the cam translates the piston thrust to a side force that displaces the motor block relative to the axis beam. A standard servo valve controls the flow of oil through the motor block and, hence, controls the motion of the actuator. Absolute position feedback to the servo controller is provided by a linear magnetostrictive encoder.

2) The rotary actuators are simple hydraulic vane motors controlled by standard servo valves. Position feedback is provided by a potentiometer.

3) Robot arms are generally thought of as general-purpose devices for positioning special-purpose tools designed for particular tasks. In this context, a gripper is simply a combination of one or more joints, sensors, and miscellaneous devices that happen to be mounted on the end of the arm. The hardware and software interfaces are defined accordingly in terms of a combination of relevant attachments.

The standard gripper on our robot has a single actuator that operates two opposing fingers through a linkage that keeps them parallel and moves them in a straight line. A small linear transducer provides absolute position feedback.

Each finger contains a miniature Cartesian compliance structure that permits parallel displacements of a few thousandths of an inch in response to applied forces. Strain gauges mounted on the compliance structure measure applied forces in three orthogonal directions in the coordinate frame of the finger, with very little mechanical cross-coupling. There is also provision to sense the presence of opaque objects between the fingers by having them break a modulated infrared light beam.

Although a truly general-purpose gripper is impossible, the gripper described above is reasonably general. It can grasp parts and tools, and it has a limited sensing capability. The user may modify the fingers if necessary for more specialized purposes.

Alternatively, the user may choose to replace the gripper completely by one of his own design. In this context, it is clear that current research on improving the dexterity of arms and grippers will ultimately result in entirely new designs.

VI. Pendant and Operator's Console

The form of operator interaction with a running robotic application is usually highly dependent on the particular application. For this reason, the architecture provides general-purpose input/output devices and system software, while relying on application-specific software to manage these devices.

There are three types of operator interaction devices. First is a small set of switches and lights providing a basic on/off capability. Two switches allow the operator to start and stop the hydraulic power supply. Two more, FREEZE and RUN, allow the operator to produce program interrupts requesting that motion be suspended or allowed. The application programmer can modify the AML subroutines used to service these interrupts, in order to meet special application requirements. However, the FREEZE interrupt request cannot be totally ignored: if the AML *FREEZE* command is not called within a specified time after the button is pushed, the system automatically shuts off hydraulic power to the robot.

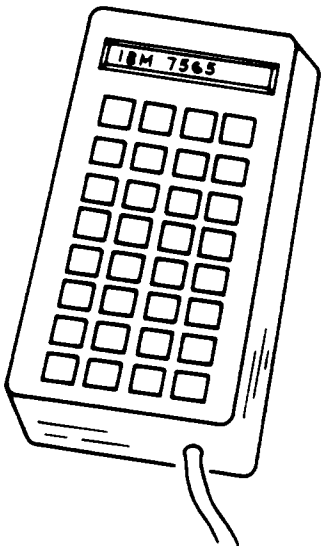

Fig. 6. Programmable pendant.

Second is a push-button control box known as a pendant, which is attached to the controller. This device has 32 binary switches, eight with indicator lights, and a 12-character alphanumeric display, as shown in Fig. 6.

Invocation of the system *GUIDE* command enables the pendant to be used for manual teaching. While in GUIDE mode, pairs of buttons control opposing motions. As long as a button is pushed, the corresponding joint undergoes uniform acceleration in the appropriate direction. This convention helps to avoid accidental high-speed crashes caused by pushing the wrong button. Manual teaching normally continues until an END key is depressed.

Although the pendant is used primarily for manually teaching robot locations, its buttons are programmable and may be used to augment the basic operator control panel. Implementation of such an interface depends on application software, using standard system commands for sensory input/output and event handling.

The third operator interaction device is a conventional data terminal, consisting of a keyboard and display. Typically, the application program displays menus, to which the operator responds with single key entries. Occasionally, applications may require more elaborate operator input, such as part numbers or other alphanumeric input.

VII. System Software

A. Overview

The structure of the system controller software is illustrated in Fig. 7. Major system components include the programming system, supervisor services, and workstation interface services.

B. Programming System

This component provides a language interpreter, interactive programming environment, and computational facilities for a high-level programming language, AML, which is used for all application programming. AML is described briefly in a subsequent section of this paper.

1) Requirements: Experience with earlier systems [1], [23], [24] convinced us of the critical importance of interactive progamming at the robot workstation. Debugging and application tuning often dominate the difficulty of installing new

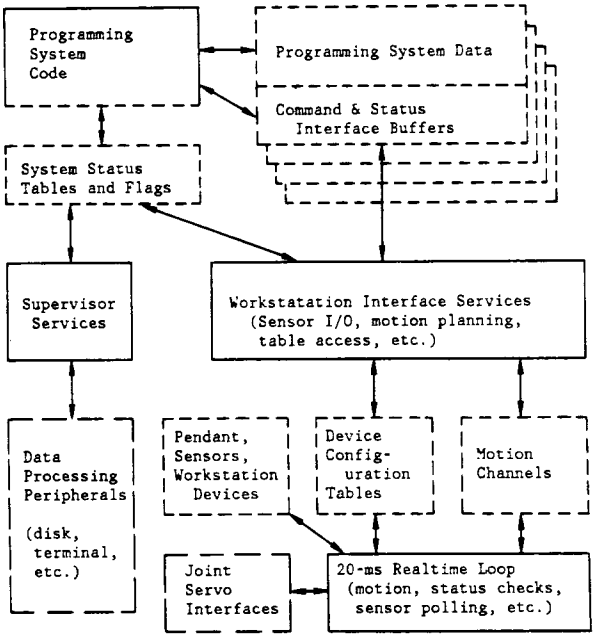

Fig. 7. System software overview.

robot applications. These activities seldom can be carried out completely off line, since workpiece locations, part tolerances, and sensor values are seldom completely predictable [25].

Our application experience with the backplane wiring tester, which has an overall execution time of nearly 5 hours convinced us that it is often utterly impractical to restart a robot program from the beginning every time a change must be evaluated or a problem diagnosed. Furthermore, the robot itself is an important tool in defining critical data points used by an application program and as a geometric aid in program debugging [6], [7], [26].

The requirement for a highly interactive programming environment led us to choose an interpreter-based system for AML. Since the robot is much slower than the computer, the computational overhead associated with language interpretation has not proved to be a significant problem for most applications. Where needed, computationally intensive functions such as kinematic joint solutions and binary image region analysis are performed by built-in subroutines coded in the implementation language of the interpreter.

2) Programming System Structure: Execution within the programming system is single threaded. Up to four data partitions, one associated with each AML "task" supported by the experimental version of the programming system, are used to hold AML programs, variables, temporary storage and command buffers for communication with other system components, as shown in Fig. 8. Most of each data partition is private, except for a small area used for global system status and a configurable area used to hold AML *SHARED* variables.

All code is placed in a shared 64K byte read-only partition, and standard multitasking facilities within the system supervisor are used to support multiple instantiations of the programming system code, each with a different data partition. The 7565 software has a similar structure. The principal differences are that only a single AML data partition is required and fewer built-in subroutines are included in the code segment.

3) AML Interpreter: The language interpreter itself is straightforward. AML programs are stored as sequences of lexical tokens which have been annotated to facilitate efficient

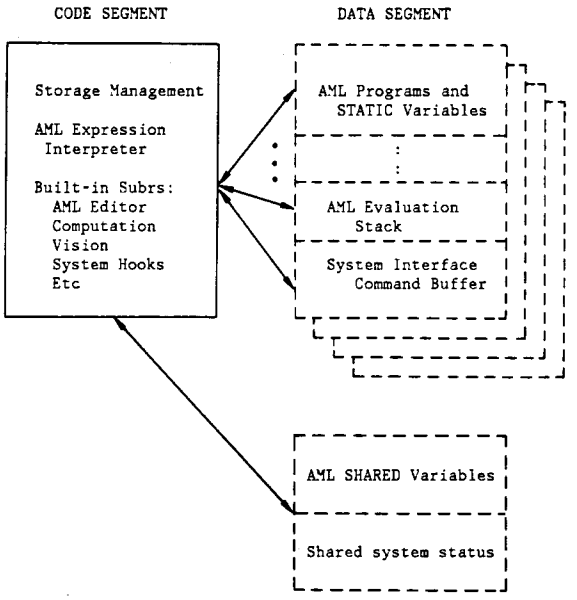

Fig. 8. Programming system structure.

interpretation and regeneration of program text from the internal form. A modified operator precedence scheme is used for expression evaluation [27] and standard shallow binding is used to implement the dynamic name scoping rules of AML.

4) Built-In Subroutines: Experience with a number of "robot" languages led us to believe that AML should be a general-purpose language whose design tradeoffs made it well adapted to robot programming [9]. Two major design objectives were that all access to robot or system functions should be through subroutines and that functional transparancy between built-in and user-written subroutines should be rigorously maintained. Adhering to this practice has given us considerable flexibility in implementaing functional hierarchies and in adapting the system to changing laboratory requirements.

Built-in subroutines now form a very substantial fraction of the whole programming system. A few, such as *RETURN* and *BRANCH*, are essentially part of the AML language definition. A much larger number perform computationally intensive numerical computation or manipulate the interactive programming environment. Others, such as *DISPLAY* and *MOVE*, call functions in other components of the system software. In all cases, the AML interpreter pushes parameters onto an evaluation stack and branches to the same system subroutine that interprets user subroutines. This code then notices that the thing being called is built-in and branches to the appropriate place. Return values are pushed on the evaluation stack and otherwise treated exactly like those produced by interpreted AML code.

5) Interactive Programming Environment: The top level of the AML programming system is a Read-Evaluate-Print loop similar to that found in most interactive programming systems. Built-in subroutines (*BREAK* and *LOAD*) are provided to invoke this top level recursively. AML expressions and subroutine definitions are entered either from the terminal (*BREAK*) or read from a file (*LOAD*), internalized, and evaluated. The results are then printed on the appropriate output device. A number of commands are provided to modify this sequence for tracing, singlestepping, etc.

Any inactive AML subroutine or unaccessed variable may be redefined by reentering it either from the terminal or from a file. If a syntax error is detected while an AML object is being internalized, an error subroutine is called, and the previous definition is retained.

The system editor is essentially line oriented, but supports full screen displays. AML programs may be edited either as text files or as subroutines in the interpreter's active memory. All editing functions are callable both from a special editing command interpreter and as built-in AML subroutines. This permits users to use the normal facilities of the language to construct their own edit macros. A more important reason for making editing functions AML callable, however, is that it facilitates the construction of specialized packages that use the robot hardware to help automate the production of application programs [6], [7], [26].

The initial source of AML input is a file selected by a configuration utility. For application debugging, this file usually contains standard user subroutines and packages. For production, it usually also includes an explicit call to the "main" application subroutine, so that machine operators are not required to interact with the AML interpreter. Instead, they usually interact with explicit interfaces (pushbuttons, display menus, etc) specified by application subroutines.

6) Exceptions and Interrupts: A number of mechanisms are provided for exception handling and simple interrupts. Users can specify the name of a subroutine to be called in the event of an exception (e.g., floating overflow, undefined identifier). Similarly, the interpreter permits a user-specified subroutine to be called whenever an interrupt event, such as a button being pushed or a monitored sensor value changing, is detected by other parts of the system. A typical use of this facility is to provide a user "attention" button by associating the *BREAK* function with one of the terminal keys. Another typical use is to monitor a parts feeder sensor. A "feeder low" indicator triggers code that instructs the operator to refill the feeder before the application is starved for parts.

C. Supervisor Services

The supervisor services consist primarily of device support for data processing peripherals, such as disks, diskettes, terminals, and teleprocessing attachments. All access from AML to these facilities is through "command" subroutines built into the programming system. Typically, these subroutines examine parameter values on the AML evaluation stack, set up commands in a standard interface area, and then invoke the appropriate supervisor function. When the supervisor function returns, the programming system routine checks for errors and then pushes an appropriate return value onto the AML evaluation stack.

D. Workstation Interface Services

This software component is responsible for all functions related to the robot workstation, including robot motion, sensor input and monitoring, device interface output, and routine safety checking. Structurally, it consists of a number of background functions called from the AML interpreter, a real-time task that runs every 20 ms, and a number of shared data tables that are used to keep track of device configuration and status information.

1) Data Structures: Data tables are used to control much of the workstation interface code. The decision to rely on a table-driven implementation and the structure of the tables themselves were motivated by a conscious desire to emphasize system simplicity and configurability as primary design objectives. We have found that, with careful design, a table-driven robot system can be made to perform quite efficiently and is

quite easy to maintain. A few of the principal tables used to control robot and other workstation interface functions are listed below:

Physical I/O Table contains device control blocks required to transfer 16-bit words between the Series/1 and I/O interface registers.

Logical I/O Table defines "logical" input/output devices as contiguous subfields of physical I/O devices. Also, contains information for automatic scaling between integer device units and floating-point engineering units.

Joint Table contains pointers to joint position command outputs and position feedback inputs, scaling and joint limit values, acceptable position error tolerances, and similar information. Also contains state information including the most recent commanded and measured positions, present position error, etc.

Motion Channel Table contains information required to describe a coordinated motion of several joints.

Monitor Table contains information required to monitor sensor threshold values.

System Log Table contains accumulated statistics (total joint travel, powered-on time, etc.) used by maintenance utilities to help assure system reliability and serviceability.

2) Motion Control: We chose coordinated "joint-space" kinematic control similar to that described in [28] as the primitive form of motion supported by the real-time system. Experience with the first IBM system had convinced us that uncoordinated "point-to-point" motions were often unpredictable and tended to make application debugging difficult. On the other hand, we wanted to keep the real-time system as simple as possible and wanted to avoid introducing unnecessary dependencies on particular kinematic configurations. Finally, we had found that the higher levels required in our application experiments, such as "Cartesian" motions, were easily defined in terms of joint space primitives.

Normal motions: The AML command

MOVE(joints,goals)

is used to move one or more joints to the specified final positions. For example

MOVE(<1,2,4>,<5.2,20,45>)

specifies that joints 1, 2, and 4 are to be moved to absolute displacements 5.2 in, 20 in, and 45°, respectively.

Motion is coordinated so that, at any time, all joints have moved the same fraction of the total displacement commanded by the *MOVE*. A typical motion trajectory is illustrated in Fig. 9, and consists of an acceleration phase, a constant-speed travel phase, a deceleration phase, and a motion-settling phase. Optional additional parameters supplied to the *MOVE* subroutine control the relative speed, the magnitude of acceleration and deceleration, and whether motion settling is to be ignored.

Execution of a *MOVE* proceeds roughly as follows. First the AML interpreter evaluates the *MOVE* actual parameters and calls the built-in *MOVE* subroutine, which then calls the motion planning function in the workstation interface services. This (background) function sets up a "motion channel" with various parameters required for efficient interpolation of intermediate goals along the motion trajectory. A command is then issued to the real-time system telling it to start the motion, and the background routine is suspended. When the motion is complete,

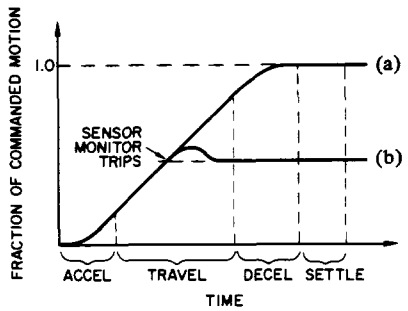

Fig. 9. Motion control. Normal (a) and interrupted (b) trajectories are shown.

the background routine is reactivated and passes appropriate completion information back to the AML *MOVE* routine, which returns to the user.

Guarded motions: The AML subroutine call

monlist = MONITOR(sensors,tests,threshold_parms, intervals)

instructs the real-time system to begin monitoring the specified set of sensors at the specified intervals. An AML aggregate containing the table indices corresponding to the specified activities is returned and stored in *monlist*.

Monitor numbers may be passed to the *MOVE* subroutine as optional parameters. If, during the course of the motion, the sensor value test associated with one of these monitors succeeds, the motion trajectory is interrupted at the last computed intermediate goal, and the motion settling phase is initiated.

Guarded moves are frequently used in calibration sequences, in which a sensor is used to detect the relationship between the robot's gripper and a reference surface or object. Other common uses include error detection, such as sensing when an unwanted collision has occurred, and for detecting completion of an assembly step, such as a peg being fully inserted into a hole [17].

Asynchronous motions: The AML subroutine call

AMOVE(joints,goals)

functions exactly like the comparable *MOVE* subroutine, except that control returns to the calling AML subroutine as soon as the motion is started. A subsequent call to the built-in subroutine

WAITMOVE

causes the AML program to wait for motion to be completed.

3) Background Workstation Interface Functions: These functions are called by subroutines built into the programming system and are called in the same way as supervisor service routines. They bridge the gap between the programming system, on the one hand, and the real-time system and workstation devices, on the other. Typical functions include motion planning, initiation and termination of sensor monitors, input from and output to workstation sensors and devices, and interrogation and modification of table entries.

4) Real-Time Structure: All real-time activity within the system is controlled by a single task, which is executed every 20 ms. This structure was chosen for its simplicity and reliability. A simplified summary of the major steps executed follows:

Step 1) Input status bits.
Read the operator's pendant and miscellaneous workstation status indicators (safety interlocks, arm power, etc.).

Step 2) Execute explicit commands.
Read the background-to-real-time command buffers and take appropriate actions. There is one such buffer for each AML background task. Typical commands include turning robot power on or off, starting or stopping motions, waiting for robot motion to complete, etc.

Step 3) Check safety conditions.
If robot power is on, check to see if it should remain on. Except when special diagnostic modes are enabled, this check requires that all joint position errors be within special tolerances and that no explicit "power off" signal has been asserted. If all safety checks are passed, issue the "set safety" command to the workstation interface card in order to restart the 25-ms hardware timeout. If this command is not issued, the timeout will automatically shut down the robot.

Step 4) Monitor sensor values.
Examine each entry in the monitor table. If the "happened" indicator for the entry has not already been set and a prespecified time has passed since the last time the entry was checked, then read the sensor value and make the specified test. For example, test to see if the value falls outside prespecified threshold values. If the test succeeds, set the "happened" indicator and wake up any sleeping background task that may have an AML interrupt routine associated with the monitor entry. Otherwise, simply go on the next monitor table entry.

Step 5) Interpret motion channel commands.
Examine each entry in the motion channel table for active motions. Perform the appropriate trajectory interpolation calculations and settling checks, as required.

Step 6) Output updated joint goals.
Examine the joint table for joints whose position goals have been changed. For each such joint, convert the new goal value from engineering units to device units, perform some safety checks, and output the value.

Step 7) Clean up and go to sleep.
Log appropriate historical data on total joint movement, etc. Then suspend processing until the start of the next 20-ms interval. When the new interval starts, go back to step 1).

VIII. AML Language

This section provides a summary description of the AML language. A more extensive discussion of the language and of some of the design tradeoffs that went into it may be found in [9]. An exhaustive description of the 7565 version may be found in the reference manual [21].

A. Language Design Objectives

The acronym, AML, stands for "A Manufacturing Language." After considering several approaches, however, we decided that the language itself should be a fully general-purpose programming language, but one whose design tradeoffs made it well adapted to programmable automation. Thus the base language provides computational primitives and algebraic operations, such as the rules for manipulating vectors and other "aggregate" objects, which are naturally required to describe robot behavior, but no specifically "robotic" constructs.

We identified several classes of user who must interact with automation systems, including machine operators, maintenance personnel, users of application customizers, application programmers with a wide range of experience, and system programmers.

The language had to have a simple, easily understood subset that would not scare off a beginner. At the same time, it had to be powerful enough to allow more experienced programmers to control sophisticated applications and to create packages that others could use. Consequently, a major design point of the language was that its rules should be as consistent as possible, with no special case exceptions that would make it difficult to learn more of the language as the need arose.

Functional transparency, in the sense that AML subroutines can be written and then used exactly like built-in system commands, was yet another design objective and has already been discussed.

B. Base Language

1) Data Objects: AML has constants and variables, just like any other programming language.

Scalar objects: AML supports *INT*eger, *REAL*, and *STRING* as primitive types. These types are all commonly found in user application programs. The language also supports a number of data types, such as identifiers, references, labels, etc., that are required for production of more sophisticated application subroutine libraries, editing and debugging packages, and other AML system code.

Aggregate objects: AML aggregates are ordered lists of scalar and/or aggregate AML objects. Aggregates are bracked by the symbols, < and >, and elements are separated by commas. For example,

```
<1,2,3>
<'hi there,'<1,2,3>,4.5,<>>.
```

2) Variables: Variables are typed and are declared by binding an identifier to the value produced by evaluating an expression. For example,

```
baz: NEW 2+2;
```

would bind the identifier *baz* to an integer data object whose initial value is 4.

The Research version of AML supports three kinds of variable declaration:

```
identifier: NEW expression;
identifier: STATIC expression;
identifier: SHARED expression;
```

NEW causes *expression* to be evaluated every time the subroutine in which the declaration appears is entered. *STATIC* and *SHARED* declarations only evaluate the expression on the first entry to the subroutine. On subsequent calls, the identifier is simply rebound to the same storage. The difference is that variables declared by *SHARED* may be accessed by several active AML programs, while those declared by *STATIC* are private.

3) Expressions: AML is an expression-oriented language, in the sense that every legal construct in the language produces a value which may be used as a part of some other expression. Expression evaluation is left-to-right, and the grouping of ex-

pressions is determined by operator precedence and parentheses as in most common programming languages.

Simple expressions: AML provides the conventional arithmetic operations of addition, subtraction, multiplication, division, and integer division. It also supports the usual comparison operators (*GT*, *GE*, etc.) between scalars and logical operators (*AND*, *OR*, *XOR*, etc.). Other binary operators include assignment (=), datatype coercion (*IS*), and string concatenation (|). Unary operators include negation, data type extraction (?), referencing (&), dereferencing (!), and a number of special quoting operators used in system subroutine packages.

Aggregate expressions: AML uses a uniform set of "mapping rules" to extend all scalar operators to aggregate objects. Essentially, these rules specify that an operator maps a scalar element-wise with an aggregate, and maps two aggregates element-wise with each other. For example, the value of *<1,2,3>*10* is *<10,20,30>*, and the value of *<1,2,3>*<4,5,6>* is *<4,10,18>*. These rules are applied recursively, so that *<1,<2,3>>*<<4,5>,6>* produces *<<4,5>,<12,18>>*. AML indexing permits references to individual elements or substructures within aggregates. For example, if *AA* represents the aggregate

<'hi,'<2,3>,<4,<5,6>>>

then *AA(1)* would refer to *'hi,'* *AA(2,1)* would refer to *2*, and *AA(<2,3>,2)* would refer to *<3,<5,6>>*.

Other aggregate operators include aggregate concatenation (#), replication (*OF*), assignment, and parallel assignment.

Control Expressions: AML provides the normal constructs of structured programming to direct the flow of program execution:

IF e1 THEN e2 ELSE e3
WHILE e1 DO e2
REPEAT e1 UNTIL e2
BEGIN e1; ...; en END.

All produce values. For example, the value of a *BEGIN ... END* block is the value of the last expression in the block.

4) Subroutines:

Subroutine definitions: AML subroutine definitions have the general form

```
subr_name: SUBR (formal_1, ..., formal_n);
    statement_1;
    statement_2;      --Comments are preceeded
        :             --by a "-," as this
        .             --example indicates.
    statement_k;
    END;
```

where *subr_name* is the name of the subroutine, *formal_1* through *formal_n* are formal parameters, and *statement_1* through *statement_k* may be executable statements (i.e., AML expressions) preceeded by optional labels, variable declarations, or subroutine definitions.

Subroutine calls: All AML subroutine calls have the general form

subr_name (expression_1, ..., expression_n)

where *subr_name* is the name of the subroutine and the *expression_i* are AML expressions whose values are to be passed as actual parameters to the subroutine. The value of the subroutine call expression is the value passed to a *RETURN* command executed by the subroutine.

Subroutine execution: After all actual parameters have been evaluated, all formal parameters are bound left-to-right. The language includes constructs that permit users to specify both call-by-value and call-by-reference binding. In the former case, the formal parameter is always bound to a copy of any variable passed as an actual parameter expression. In the latter case, the formal parameter is bound to the same data object as the variable identifier.

Next, all local variable declarations, statement labels, and local subroutine definitions are processed in the order they appear in the subroutine body. Finally, the AML expression interpreter is called to evaluate successive statements as they appear in the subroutine definition.

When the subroutine exits, all local bindings are undone, any temporary storage allocated for local *NEW* variables is released, and control returns to the calling subroutine or, in the case of a nonlocal *BRANCH*, to the target expression.

The language includes a *CLEANUP* construct that allows a user to specify that a named subroutine is to be called whenever the present AML subroutine exits. This call is made before any local name bindings are undone, and is often used to undo temporary changes made to the global execution environment. For example, a high-level search subroutine would typically use a *CLEANUP* to turn off any sensor monitoring that it had set up in the course of the search.

C. Commands

AML commands are simply system-provided subroutines that define the semantic functions for robotics, mathematical calculation, input/output, etc. No syntactic distinction is made between these routines and any other subroutine in the system. These functions may exist as either system-defined primitives written in the implementation language of the system or as AML subroutines that are almost always defined as part of the base system.

The semantic environment in which a user's AML program executes is thus defined by the semantics of these routines, together with any additional AML libraries that may have been selected by the user. Once an AML subroutine is loaded into the user workspace, it effectively becomes part of the system. Thus the *GRASP* subroutine described in the next section could be used by a naive user in exactly the same manner as the more primitive *MOVE* command. This transparency provides a natural growth path for extensions to the system through the use of functional subroutine libraries.

Commands can be classified, roughly, into several categories:

Fundamental subroutines are an essential part of the AML language definition. Generally, they affect the flow of control within the interpreter. For example, *RETURN* causes return from a subroutine, and *BRANCH* causes transfer of control to a label. Other examples include *CLEANUP*, which is used to set subroutine exit traps, *APPLY*, which is used to call a subroutine with a program-generated aggregate of arguments, and *MAP*, which is used to apply a subroutine element-wise to aggregates.

Calculational subroutines perform data manipulations on their arguments. Typical examples include *SORT*, which computes square roots, *LENGTH*, which computes the length of strings, and *DOT*, which computes generalized matrix and vector inner products.

Workstation interface subroutines provide the motion and sensing functions that specialize the system for control of a robot workstation. Typical examples include *MOVE*, which moves one or more robot joints, *SENSIO*, which reads sensor values

or outputs control values, and *MONITOR*, which initiates regularly scheduled monitoring of a sensor value.

Vision subroutines are a special class of workstation interface and calculational subroutines that implement a simple binary vision system [10]. Examples include *READVI*, which takes a picture, *BIXOR*, which performs a bit-per-pixel exclusive-*OR* of two images, and *RC2REGIONS*, which performs region analysis on a runcoded image.

Editing and debugging subroutines provide interactive programming services. Examples include *LOAD*, which reads AML programs from a text file into program memory, *SETBREAK*, which tells the interpreter to call the *BREAK* subroutine when it encounters a particular statement, and *EFILE*, which edits a text file.

Supervisor service subroutines mostly provide access to data processing peripherals. Examples include *OPEN*, which opens a file for reading or writing, *DISPLAY*, which displays data on the terminal, and *PREFILL*, which prefills the terminal input buffer with anticipated input text.

Multitasking subroutines are a special class of experimental subroutines that create, terminate, and synchronize multiple AML programs. Examples include *FORK*, which is used to create a new task, *ENDFORK*, which is used to terminate one, and *TESTSEM*, which is used to test a semaphore.

IX. Typical Application Example

Since many of the design principles of the overall system were influenced, in part, by the requirements for testing computer backplanes, the successful installation of this application was to be expected. Numerous subsequent applications, however, were entirely different, and their success constituted validation of the system design. Typical of these applications was a robot to assemble print chains, installed in IBM's plant in Endicott, NY, in 1981.

A print chain is a mechanism from a high-speed mechanical printer. It contains 80 type slugs, each with three alphabetic or numeric characters. When the printer operates, the chain of type slugs rotates rapidly above the ribbon and the paper. Small hammers under the paper are precisely timed to bang the paper against a character on the chain at precisely the right time to produce an image of that character in the correct place. The performance requirements of the printer, therefore, require that the chain be assembled with considerable precision. Since the process of inserting type slugs manually is tedious and error prone, the assembly was a natural candidate for automation.

A. Summary of Job Cycle

In this robotic application, the operator begins by loading the empty cartridge onto a passive transport mechanism. The robot grasps the mechanism, pulls it into the workstation, and verifies that the cartridge is empty.

The operator pushes a single key to indicate which style chain is desired. The system looks in a disk file to find the particular sequence of slugs required for this style chain. The robot then takes type slugs in the proper sequency from 42 gravity feeders and inserts them into the cartridge. At the appropriate points in the assembly, the robot removes plastic gear spacers from the ends of the cartridge. Each time a slug is inserted, a special-purpose linear actuator advances all the slugs so the insertion position is again empty. The final type slug must be inserted into a space that is precisely the right size. The robot uses sensors to feel for the space and nestles the type slug into place.

This application is characterized by extensive use of sensory feedback for error detection and recovery. For example, if the linear actuator fails to advance the slugs, the robot pauses while the actuator retries its motion. If a feeder fails to feed properly, the robot tries again. If an error persists, the robot pushes the transport mechanism out of the workspace and signals for operator intervention.

Each time the robot grasps a type slug, it uses sensors to center on it. If the type slug is found to be appreciably off-center, tables are updated appropriately. Furthermore, the daily operation cycle begins with a completely automatic self-calibration procedure. The robot uses sensors and a sequence of rocking motions to find the zero point of the pitch, yaw, and roll motions. It feels to find when the fingers are closed exactly. And it senses the position of a post to calibrate its X, Y, and Z joints. Then it feels to find the 1st and 42nd feeders and interpolates to infer the positions of the other 40 feeders.

B. Application Program

It is impractical to discuss the entire application program in the short space available here. Instead, we have chosen to concentrate on several key elements in order to give illustrate typical uses of system capabilities.

The programming examples presented here are based on the programs that actually run the print chain assembly robot. For pedagogical reasons, however, we have chosen to rewrite the programs substantially for ease of explanation. Additionally, our modifications embody some of the experience gained from this and other applications. Thus the examples given here are more representative of present AML programming practice than was the original application code.

1) High-Level Motion Subroutines: The AML code examples given in subsequent sections include calls to experimental AML motion subroutines used by one of the authors (Taylor) to study robot application programming techniques. The following very brief discussion is intended to provide information required to follow the examples and to illustrate the functions in a typical AML subroutine library.

Position goal representation: Robot position goals are represented as

<position,orientation,gripper>

triples. *Position* is an aggregate of three real numbers giving the desired position of the finger tips. The optional element *orientation* is three real numbers corresponding to an Euler angle representation of the desired hand orientation. *Gripper* is optional and corresponds to a desired gripper opening.

The AML subroutine call

goal = HAND_GOAL(joint_goals)

translates the joint goal values in *joint_goals* into AML goal aggregate of the above form. If *joint_goals* is omitted, then the current robot joint goals are used.

Cartesian frame motion commands: The AML subroutine call

CMOVE(goal,tests,cntl)

causes the robot to move to the position, orientation, and gripper opening specified by *goal*. *Tests* is an optional parameter describing sensory monitoring to be performed during the motion. If any of the specified sensor thresholds are exceeded, the motion is terminated. *Cntl* is an optional parameter giving speed and acceleration parameters to be used in the motion. If it is not given, the currently set default values are used.

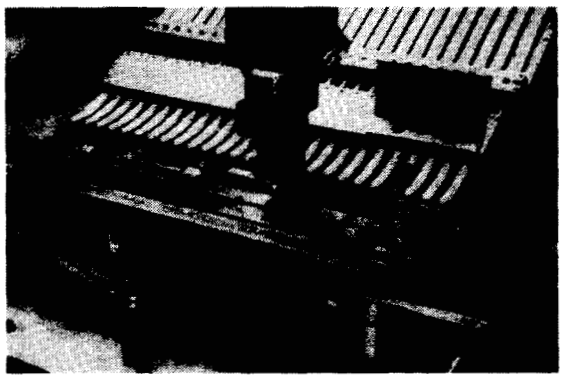

Fig. 10. Print chain assembly application.

If *tests* is specified, then *CMOVE* returns an aggregate of *TRUE* and *FALSE* values specifying which of the corresponding sensor thresholds was exceeded during the motion. Otherwise, it returns the null aggregate.

Similarly, the AML subroutine call

DCMOVE(offset,tests,cntl)

causes the robot to be displaced relative to its current Cartesian position by an amount *offset*. *Tests* and *cntl* have the same meaning as for *CMOVE*.

The AML subroutine call

GRASP(size,tolerances,force)

causes the robot to center the gripper jaws on an object of the specified size and grasp it with the specified force. If the object grasped is within specified tolerances of the nominal size, the subroutine returns the AML string *'OK.'* Otherwise, it returns *'too big'* or *'too small,'* as appropriate.

An AML implementation of *GRASP* may be found in [9].

2) Fetching a Slug from a Feeder:

Feeder locations: The hand orientation required to pick up a type slug is the same for all 42 feeders. Consequently, an AML aggregate

feeder_loc: STATIC 42 OF<REAL,REAL,REAL>;

is used to store the translational positions. *Feeder_loc(i)* corresponds to a fingertip position about 0.75 in above the grasping position for the *i*th slug.

Recalibrating the feeder array: The type slug feeders are arranged in a row parallel to the *Y*-axis of the robot, as shown in Fig. 10. Consequently, the nominal locations of each feeder are all the same except for their second coordinate. In fact, imprecision in the robot and in fabrication of the feeder array requires that all three coordinates be slightly different.

A special-purpose AML subroutine

locate_feeder(nominal_location)

was written to determine the actual location of a feeder, given an approximate nominal location. This subroutine moves the robot's gripper to the nominal approach location for the feeder. It then uses a sequence of guarded *CMOVE*'s and *DCMOVE*'s to locate known surfaces on the fixture by touch. It then computes the actual approach location, moves the robot's gripper there, and returns the approach location as the value of the subroutine.

During application development, this subroutine was used to determine the locations of all 42 feeders. These values were written onto a file, and are read into *feeder_loc* as part of program initialization. Unfortunately, temperature variations

```
calibrate_feeder_array: SUBR;

 f1_loc: NEW locate_feeder(feeder_loc(1));
 f42_loc: NEW locate_feeder(feeder_loc(42));

 f1_err: NEW f1_loc - feeder_loc(1);
 f42_err: NEW f42_loc - feeder_loc(42);

 weights: NEW (feeder_loc(1OTA(42),2)-feeder_loc(1,2))
                                    /(feeder_loc(42,2)-feeder_loc(1,2));

 feeder_loc = feeder_loc + weights*(42 OF f42_err)
            + (1-weights)*(42 OF f1_err);
 END;
```

Fig. 11. AML subroutine for feeder array calibration.

```
pick_up_slug: SUBR(fdr,tries);
 cc: NEW STRING(8);
 t: NEW 0;

 step_1:                                   -- Move to grasping position.
  CMOVE(<feeder_loc(fdr),feeder_orient,.5>);
  IF DCMOVE(<<0,0,-.75>,ANY_FORCE(2*OZS),<.5>) THEN
    BEGIN                                  -- Hit something on way in
    DCMOVE(<<0,0,2>>);                     -- Back out
    RETURN('jammed');                      -- Return error
    END;

 step_2:                                   -- Attempt to grasp slug.
  cc = GRASP(0.1,<-.04,.04>,PINCH_FORCE(1*LBS));
  IF cc NE 'ok' THEN
    BEGIN                                  -- Hit something on way in
    MOVE(GRIPPER,.5);                      -- Readjust gripper
    DCMOVE(<<0,0,1>>);                     -- Back out
    RETURN(IF cc EQ 'toosmall' THEN 'empty'
         ELSE if cc EQ 'toobig' THEN 'jammed'
         ELSE cc);
    END;

 step_3:                                   -- Update feed location.
  fy = HAND_GOAL()(1,2);                   -- "Y" position at grasp
  IF ABS(fy-feeder_loc(fdr,2)) GT 0.04 THEN
    feeder_loc(f,2) = fy;

 step_4:                                   -- Pull out slug and reverify hold.
  DCMOVE(<0,0,1>);
  IF ABS(SENSIO(<LPINCH,RPINCH>)) LT 15*OZ THEN
    BEGIN                                  -- Dropped part.
    IF tries GE t = t+1 THEN               -- If not too many,
      BRANCH(step_1);                      -- then try again.
    ELSE                                   -- Otherwise,
      RETURN('dropped');                   -- give up.
    END;

 RETURN('ok');
 END;
```

Fig. 12. AML subroutine to pick up a type slug.

and other effects can cause the robot's coordinate system to drift slightly over time relative to the feeder array. One way to correct for this would be to repeat the entire feeder location procedure at regular intervals. This procedure, however, takes several minutes that could better be spent doing productive work.

Instead, the application program exploits the fact that long-term drifts in robot calibration are likely to be systematic. It uses the AML subroutine shown in Fig. 11 to locate the end feeders and then interpolate the other 40 corrections. This procedure is quite quick and is performed routinely every 4 hours.

Code to pick up a slug from a feeder: Fig. 12 shows a simplified AML subroutine for picking up a type slug from a specified feeder. The actual application program provides for many more error contingencies than are shown here and does much more bookkeeping. The subroutine shown requires two parameters: *fdr*, which gives the feeder number, and *tries* which specifies how many times the robot is to attempt to pick a slug from this particular feeder before giving up. The algorithm is roughly as follows:

Step 1) Move the robot gripper to the approach position for the specified feeder and open the gripper to one

half inch. Then make a half-speed guarded move down to the grasping position. If the gripper hits anything, assume that there is a part jammed in the feeder. Back the gripper out and return an error indication to the caller.

Step 2) Center the gripper on the type slug and grasp it with 1 lb of pinching force. If the *GRASP* subroutine is unsuccessful, reopen the gripper, back out, and return an appropriate error code.

Step 3) If the centering perfomed in Step 2) moved the gripper more than 0.04 in, update the Y coordinate of the feeder location. This action tends to improve efficiency by reducing the amount of time required on the next feed cycle. In effect, the robot tracks drifts in its own calibration and in how parts are presented from the feeder.

Step 4) Pull the slug out from the feeder. Then, test the pinch force sensors to verify that it is still being held. If the part has been dropped, check to see if the retry limit has been exhausted. If not, go back to Step 1). Otherwise, return an error code.

3) Assembly Sequence:

Data Structures: The order in which type slugs are to be assembled is kept in an AML aggregate variable

```
print_train_sequence: NEW 80 OF INT;
```

which specifies what kind of slug should be used for each position in the train. These values are read from a disk file each time the operator specifies a new model number.

In order to allow the assembly program to continue without interruption while an empty feeder is refilled, the application program permits more than one feeder to be assigned to the same kind of type slug. The program uses two AML aggregate variables

```
current_feeder: NEW 42 OF INT;
next_feeder: NEW 42 OF INT;
```

to maintain a circular list of feeders associated with each kind of slug. These values of these variables are read in during the initial job setup and are updated whenever the assignment of type slugs to feeders are changed.

Populating the print chain cartridge: Fig. 13 shows an AML subroutine for transferring type slugs from the feeders to a type slug cartridge. This subroutine is called after an empty print cartridge has been placed on the transport mechanism and brought into the assembly station. Essentially, it consists of a big loop, which is repeated once for each type slug. The algorithm for each iteration is roughly as follows:

Step 1) If this iteration requires that one of the spacers be removed, do so. The code for this subtask is not shown, but is very straightforward. The robot grasps the spacer, pulls it out of the cartridge, and drops it into a box. Error recovery procedures are provided for dropped spacers, etc.

Step 2) Compute which feeder is to be used to fetch the next type slug and call *pick_up_slug* to go get it. If *pick_up_slug* reports an error, notify the operator, update *current_feeder* to select the next feeder for this type slug, and repeat the step. Otherwise, go on to Step 3).

```
populate_chain: SUBR;
 n: NEW 0;                                  -- index in print chain
 k: NEW 0;                                  -- kind of slug
 f: NEW 0;                                  -- feeder number

 WHILE 80 GE n=n+1 DO
  BEGIN

 step_1:                                    -- Do spacers, if need to
  IF n EQ n_first_spacer
    THEN withdraw_first_spacer
  ELSE IF n EQ n_second_spacer
    THEN withdraw_second_spacer;
  END;

 step_2:                                    -- Get slug from feeder.
  k = print_train_sequence(n);
  cc = pick_up_slug( f=current_feeder(k), 3);

  IF cc NE 'ok' THEN
    BEGIN
    IF cc EQ 'empty' THEN
      tell_operator('refill ',f,' with ',k)
    ELSE IF cc EQ 'jammed' THEN
      tell_operator('possible jam in feeder ',f)
    ELSE IF cc EQ 'dropped' THEN
      tell_operator('keep dropping at feeder ',f);
    current_feeder(k) = next_feeder(f);
    BRANCH(step_2);                         -- Go retry slug
    END;

 step_3:                                    -- put slug into cartridge
  cc = put_down_slug;                       -- Subroutine not shown
  IF cc EQ 'dropped' THEN
    BRANCH(step_2);                         -- Go retry slug
  END;
 END;
```

Fig. 13. AML subroutine to populate printer chain.

Step 3) Put the slug into the cartridge. This code, which is not shown, involves a number of refinements to compensate for minor alignment problems and to operate an auxiliary actuator efficiently. A typical error recovery action is shown: if the type slug is dripped, go back to Step 2) to get another one.

As with the pickup subroutine, the code shown is greatly simplified and ignores many error checks performed by the actual application program. For example, the actual program may decide that a particular print cartridge is hopelessly jammed. In this case, the transport mechanism is operated to exchange the print cartridge for an empty one. The robot proceeds with the new cartridge while the operator fixes the old one.

C. Lessons Learned

The extensive use of sensing was found to be necessary to keep the application running reliably in a manufacturing environment. In addition to sensing, the application program reads disk files, displays messages on the terminal, reads key buttons, and interacts with an auxiliary actuator. This particular application did not require teleprocessing, multitasking, or vision, but other running applications have used these features.

The application tooling and program were developed over a period of several months by members of an IBM manufacturing engineering group at Endicott, with occasional consultation from the Research group. We were pleased to observe that the interactive nature of the system made it quite easy for engineers with little prior robotics or programming experience to experiment with new techniques. In time, the Endicott group developed considerable programming skill, in part by following the coding style of expertly written subroutine libraries. Initially, they tended to use these libraries as black-box extensions of the system but soon developed the confidence to modify and enhance them to suit their own purposes.

The implementation team had several other responsibilities, and installation took slightly longer than expected. However, the assembly runs about 25 percent faster than originally anticipated, and the system has an excellent financial return on investment. The application has now been running for about 2 years, and the people involved have gone on to work on many other successful robotic applications.

X. Experience and Conclusions

This paper has described the system architecture of a second-generation robot system now in use at IBM T. J. Watson Research Center. Although the system described is very similar to the IBM 7565 Manufacturing System, it has been enhanced to support our present research activities. Since these enhancements are expreimental, they may never be part of any IBM product.

The system has been operational for four years at the IBM Research Center and at selected IBM manufacturing sites around the world. More recently, the product version of the system has been installed in a number of non-IBM sites as well. Application experience has tended to confirm the original decision to make system integration the focus of the design.

The system has been used successfully in a wide variety of applications, including assembly, inspection and test, light fabrication, and intelligent materials handling. These applications have generally made extensive use of sensing, calibration, and programmed error recovery in order to function reliably and efficiently in a manufacturing environment. Many have used data files to customize production on each job cycle, while others have used the system's flexibility to accommodate especially "tricky" steps in hard automation lines.

The variety of application requirements has confirmed the critical importance of good general-purpose programming language capabilities to the success of an integrated robot system. AML has been used successfully by a variety of people including researchers, professional programmers, manufacturing engineers and system maintenance engineers. The interactive nature of the language implementation has proved to be of critical importance for application development and debugging. It has also proved an invaluable aid to learning both the language and new robot techniques. Simple programs often can be implemented in a few hours. More sophisticated applications require periods ranging from a few days to several months.

AML has proved to be well adapted for implementating a wide variety of operator interfaces. The design of such interfaces is an important part of any application development effort, and designs chosen have ranged from simple on-off controls to relatively sophisticated interactions with menus and textual prompts shown on a display terminal.

In some cases, the kinematic arrangement of the joints has been modified to allow the robot to perform specialized functions. The modular structure of the workstation interface has permitted us to accommodate such changes easily, without special-purpose programming or other changes to the system design.

The system has proved to be a reasonably flexible tool for robotics research. We have used it in our laboratory to control a number of different robots, with a variety of actuator and sensing technologies. Recently, we have begun investigating application of nonlinear and optimal control methods to robots by replacing our analog servo cards with very powerful microprocessor attachments interfaced to the Series/1.

The AML language has also proved to be well adapted to research purposes, especially as a vehicle for prototyping experimental functions and for investigating robot programming techniques. The principal limitation has been the lack of a compiler. Experiments requiring fast real-time response often require hand-coding once the algorithm has been worked out, although the interpreter has proved adequate in a surprising number of cases.

The system would provide an excellent execution environment for automatically generated robot programs, and we hope to see it used for that purpose some time in the future.

Acknowledgment

The system described in this paper is truly the work of many people. Unfortunately, the number of colleagues at Yorktown, Boca Raton, and elsewhere who have contributed significantly to the system has now risen to the point that it is impractical to name them all. At the risk of some injustice, however, we would like to name a few individuals. P. Will was for many years the leader of the Research effort and made an incalculable contribution to the overall system. S. Buckley was responsible for many significant refinements to the AML language and to the interactive programming system. Other individuals who should perhaps be singled out for special mention include H. Aviles, A. Brennemann, E. Collins, M. Condon, S. Davis, G. Ellsworth, G. Faurot, J. Fink, R. French, H. Fritz, S. Glass, P. Gothier, D. Heikkinen, S. Hutcheson, S. Kupka, M. Lavin, L. Lieberman, J. Meyer, H. Panissidi, E. Pole, J. Roberts, P. Summers, D. Tolchin, P. VanDyke, C. Wadsworth, H. Wattanbarger, and many, many more.

References

[1] P. Will and D. Grossman, "An experimental system for computer controlled mechanical assembly," *IEEE Trans. Comput.*, vol. C-24, p. 879, 1975.

[2] R. Evans *et al.*, "Software system for a computer controlled manipulator," IBM Res., Yorktown, Heights, NY, Rep. RC-6210, 1977.

[3] M. Blasgen and J. Darringer, "MAPLE: A high level language for research in mechanical assembly," IBM Res., Yorktown Heights, NY, Rep. RC-5606, 1977.

[4] L. Lieberman and M. Wesley, "AUTOPASS: An automatic programming system for computer controlled mechanical assembly," *IBM J. Res. Devel.*, vol. 21, p. 321, 1977.

[5] M. Wesley *et al.*, "A geometric modeling system for automated mechanical assembly," *IBM J. Res. Devel.*, vol. 24, no. 1, p. 64, 1980.

[6] D. Grossman, "Programming a computer controlled manipulator by guiding through the motions," IBM Res., Yorktown Heights, NY, Rep. RC-6393, 1977.

[7] P. Summers and D. Grossman, "XPROBE: An experimental system for programming robots by example," IBM Res., Yorktown Heights, NY, Rep. RC-9082, 1981.

[8] J. Meyer, "An emulation system for programmable sensory robots," *IBM J. Res. Devel.*, vol. 25, p. 955, 1981.

[9] R. Taylor, P. Summers, and J. Meyer, "AML: A manufacturing language," *Int. J. Robotics Res.*, vol. 1, no. 3, 1982.

[10] M. Lavin and L. Lieberman, "AML/V: An industrial machine vision programming system," *Int. J. Robotics Res.*, vol. 1, no. 3, 1982.

[11] T. O. Binford *et al.*, "Exploratory study of computer integrated assembly systems," Stanford Computer Sci. Dep., Stanford, CA, Memo AIM-285 and STAN-CS-76-568, 1976.

[12] P. Winston, Ed. "Progress in vision and robotics," MIT AI Lab Memo AI-TR-281, 1973.

[13] ——, "Robotics institute technical bibliography," Carnegie Mellon Univ., Pittsburgh, PA, 1983.

[14] R. Paul, J. Luh *et al.*, "Advanced industrial robot and control systems," Purdue University, Lafayette, IN, Tech. Rep. TR-EE 78-25, 1978.

[15] J. Birk, R. Kelley *et al.*, "General methods to enable robots with vision to acquire, orient and transport workpieces," University of Rhode Island, Kingston, Rep., 1980.

[16] J. Birk and R. Kelley, "Workshop on research needed to advance the state of the art in robotics," NSF Workshop Rep., University

of Rhode Island, Kingston, 1980.
[17] R. Taylor, "A synthesis of manipulator control programs from task-level specifications," Ph.D. dissertation, Memo AIM-282, Artificial Intelligence Lab., Stanford Univ., Stanford, CA, 1976.
[18] T. Lozano-Perez and P. Winston, "LAMA: A language for automatic mechanical assembly," in *Proc. 5th Int. Joint Conf. on Artificial Intelligence* (MIT, Cambridge, MA, 1977).
[19] R. Popplestone, P. Ambler, and I. Bellos, "RAPT: A language for describing assemblies," *Indust. Robot*, vol. 5, p. 131, 1978.
[20] R. Brooks, "Symbolic error analysis and robot planning," MIT Artificial Intelligence Lab. Memo, 1982.
[21] —, *IBM Manufacturing System: A Manufacturing Language Reference Manual.* IBM Corporation, Publ. no. 8509015, 1983.
[22] J. Alvarez and H. Panissidi, "Analysis of a linear self-commutating actuator for robotic systems," in *Robotics Research and Advanced Applications* (ASME Book H00236). New York: ASME, 1982.
[23] R. Finkel, R. Taylor, R. Bolles, and J. Feldman, "AL—A programming language for automation," Stanford Artificial Intelligence Lab. Memo AIM-243, Stanford University, Stanford, CA, 1974.
[24] S. Mujtaba and R. Goldman, "AL user's manual," Artificial Intelligence Laboratory, Memo AIM-323, Stanford Univ., Stanford, CA, 1979.
[25] A. Bloch, *Murphy's Law and Other Reasons Why Things Go Gnorw.* Los Angeles, CA: Price/Stern/Sloan, 1977.
[26] D. Grossman and R. Taylor, "Interactive generation of object models with a manipulator," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-8, p. 667, 1978.
[27] V. Pratt, "Top down operator precedence," manuscript, ca. 1974.
[28] R. Taylor, "Planning and execution of straight line manipulator trajectories," *IBM J. Res. Devel.*, vol. 23, p. 424, 1979.

Sensor-Based Robotic Assembly Systems: Research and Applications in Electronic Manufacturing

ARTHUR C. SANDERSON, MEMBER, IEEE, AND GEORGE PERRY

Invited Paper

Abstract—The analysis, design, and evaluation of robotic systems for manufacturing applications require understanding of sensing, representation, and manipulation as well as the integration and configuration of computer processors for control of the system. Robotic assembly presents particularly complex research issues due to the physical interactions among workpieces, precise positioning, and complex geometry of the manipulations. This paper presents an overview of research in the Flexible Assembly Laboratory at Carnegie-Mellon University and describes results in force, tactile, and visual sensing, sensor-based control, gripper design, and the configuration of a testbed multirobot integrated system for experiments in assembly of electronic components, using vision for the acquisition and orientation of parts and tactile sensing for assistance in insertion tasks. A joint research program between CMU and Westinghouse Electric Corporation has resulted in the development of the Standard Electronic Assembly Station (SEAS) which integrates a variety of research concepts into a production prototype for circuit board assembly.

I. Introduction

ROBOTIC SYSTEMS offer tremendous promise for the flexible automation of many manufacturing tasks. While assembly has been identified as a task where potential productivity gains could be achieved, the development of automated assembly techniques still requires progress on a variety of challenging research issues. This paper examines a number of these issues, reviews results of research in the Flexible Assembly Laboratory at CMU, and describes the evolution of a sensor-based assembly system for electronics manufacturing developed through a joint research program between Westinghouse Electric Corporation and CMU.

Assembly is one step in a manufacturing process which typically includes: forming of parts, presentation of parts, assembly of parts, and joining or bonding of parts after assembly. The assembly system must successfully interface to parts presentation and storage systems as well as to subsequent joining operations. The performance of such a system is evaluated in terms of its speed, throughput, reliability, and resulting product yield, as well as its impact on other parts of the manufacturing process including inventory, parts handling, work in progress, product design, and flexibility for changeover among designs. The rationale for automating an assembly operation may involve some combination of these performance parameters. Throughput alone is not always the primary motivation, and often throughput may be sacrificed for economic advantages incurred by increased reliability, product yield, or flexibility. Such decisions are dependent on the specific assembly task, manufacturing environment, and production goals.

In this paper, we will focus on the research issues which must be addressed in order to design and build automated systems for assembly applications. In particular, we will stress the functional requirements and performance tradeoffs among mechanical dexterity, adaptive control, and multisensory inte-

Manuscript received November 15, 1982; revised March 9, 1983. This work was funded in part by the Westinghouse Electric Corporation and in part, at CMU, by the National Science Foundation under Grant ECS-7923893 and The Robotics Institute.

A. C. Sanderson is with the Department of Electrical Engineering and The Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA 15213.

G. Perry is with the Manufacturing Science and Technology Center, Westinghouse Electric Corporation, Columbia, MD 21203.

0018-9219/83/0700-0856$01.00

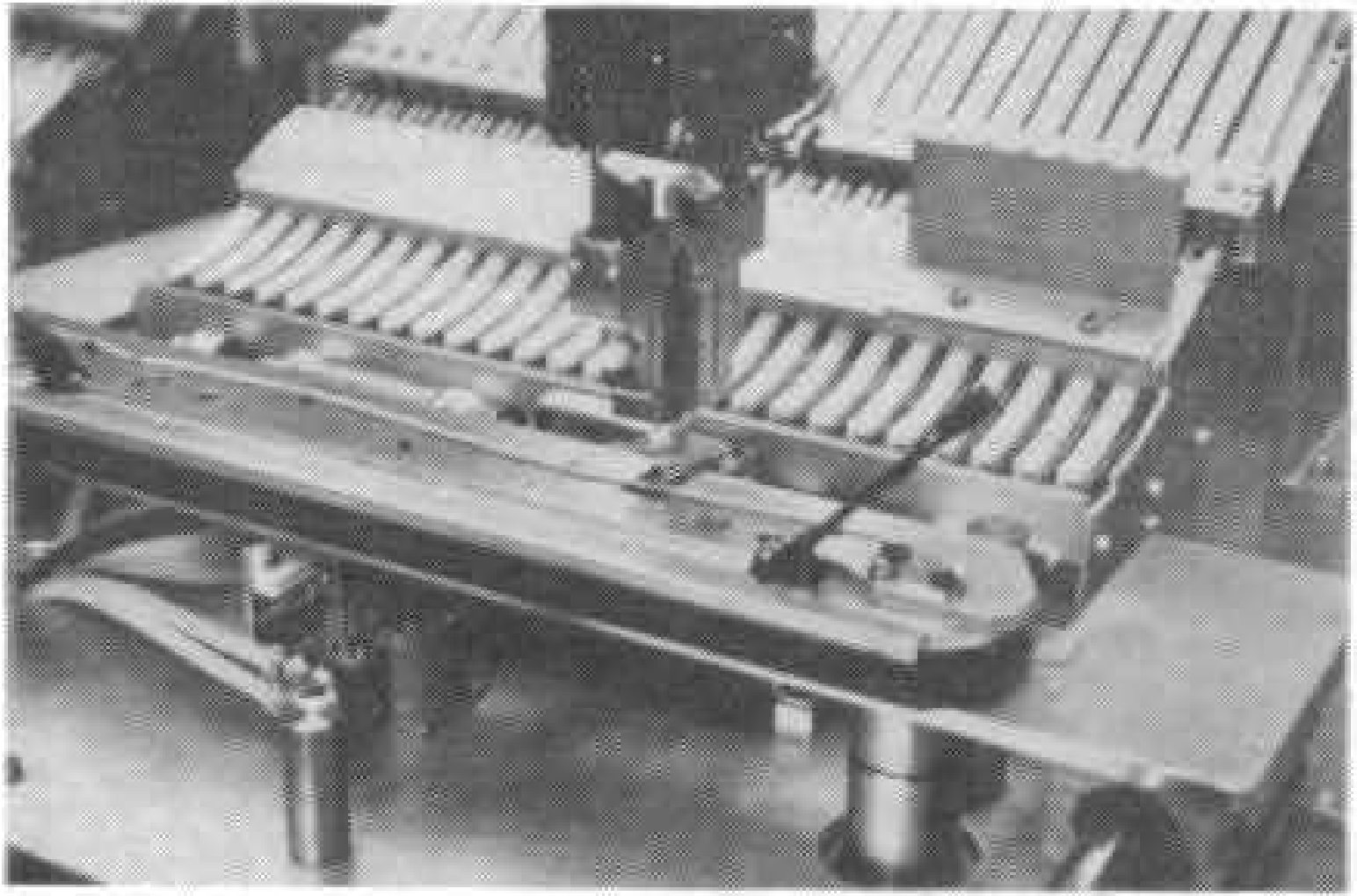

Fig. 10. Print chain assembly application.