Adaptation of the IBM 7535 Robot for Use as a Slave in a Hierarchical Master/Slave Relationship

M. Ward and Allen C. Eberhardt
Integrated Manufacturing Systems Engineering Institute, School of Engineering, North Carolina State University, Raleigh, North Carolina 27695
Received February 28, 1986; accepted March 12, 1986

A new programming environment for the IBM 7535 robot is introduced. AML/E is replaced by a hierarchical control scheme involving a host and a subservient robot. Application programs are developed on the host, an IBM-PC XT, in an extended version of the "C" programming language. These extensions allow an application program to be composed of several concurrent processes, one of which is responsible for commanding the robot. To configure the robot as a slave in this relationship, a new operating system was developed for its controller. Written in the same language employed at the host level, it is structured as four concurrent processes managing the robot's resources. An interprocess message-passing scheme provides a path for master/slave communication. The resultant environment is believed to be superior to the original language AML/E for the following reasons: Concurrency, along with the data and control structures of "C," is made available to the application programmer. Communication is well structured. The robot's operating system is documented, written in a high-level language, and open to the user for modification. Due to the significance of integrating the robot into more complex applications involving sensors, these characteristics are felt to be essential in future robots.

本論文では、IBM7535 型ロボット用の新しいプログラミング環境を紹介する。ALM/E はホスト・コンピュータとロボットを含む階層的制御方式に置き換えられている。プログラムは拡張C 言語を用い、ホスト・コンピュータであるIBM-PC/XT 上で開発された。ここに提案する拡張により応用ソフトの並列処理を可能とした。ロボットをスレーブと位置づけ、新しいオペレーティング・システム(OS)の開発を行なった。ホストで用いられている言語と同一のものを用い、 4つの並列処理系がロボットを共有する形をとっている。
プロセス間のメッセージ・パスによりマスター／スレーブ間の通信を行なう。ここに提案する方式は従来のAML/E 言語より数段性能が良いと思われる。それは次のような理由による。C 言語のデータおよび制御構造のみばかりでなく、並列処理にもプログラマがアクセスできる。通信が非常に良く構造化されている。ロボットのオペレーティング・システムが文書化されており、さらには高級言語で書かれていてユーザによる追加変更が可能である。ロボットをセンサ等を含むより複雑なシステムに組込むためには、以上に述べたような特徴は必要不可欠と思われる。

I. INTRODUCTION

IBM's 7535 manipulator is an example of a SCARA robot. Often used in electronic assembly, its characteristics make it suitable for tasks where low cost, accuracy, and less flexibility is required. An example of such an application is the insertion of electronic components onto printed circuit boards.

Although mechanical performance is excellent, users of the IBM 7535 series have uncovered shortcomings in the manner in which this particular robot is intended to interact with other equipment in the workcell. That is, when one attempts to integrate several of them into a complex system where they are required to look outside of their own controller for commands, there is little support in the application language, AML/E. A popular solution has been to utilize the 7535's parallel I/O as a means of communicating between the external environment and the robot's application control program. While limited success has been achieved using this technique, the authors' intent is to introduce a more elegant, although ambitious, approach.

Given that the cause of the problem is the 7535's rigid implementation as an autonomous teach-and-repeat robot, a concept which hopefully will be demonstrated within this article, it is suggested that the situation could be improved by configuring the manipulator as a generic peripheral device which looks to a host for commands in real time (not unlike a printer!). Then, rather than attempting to utilize the robot's hardware as both a cell and robot controller, we adopt a hierarchical view of the workcell, as shown in Figure 1, in which the robot is directed by a higher level of intelligence. A more suitable computer could then be used as a cell controller and it would perhaps play host to more than one manipulator.

To justify discarding the programmer's view of the robot afforded by AML/E, an examination of recent trends in robot programming languages will be presented. The findings, when compared with the 7535's language, demonstrate a lack of support for concepts believed to be of significance in future applications.

To remedy this situation, a concurrent programming system is offered which configures the IBM-PC XT and the 7535 as a hierarchical, master/slave pair. Employing an extended version of the "C" programming language, the new system allows concurrent programs to be developed and executed on the PC. The personal computer thus becomes a master controller with one program dedicated to robot control, while

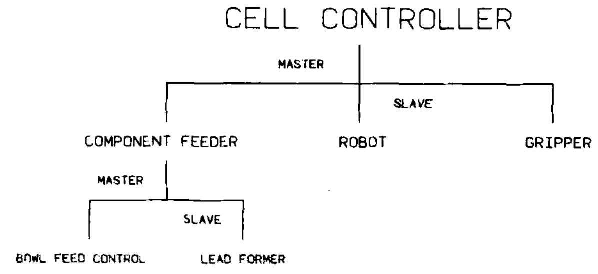

Figure 1. Hierarchical view of workcell.

others manage peripheral equipment. Support for communication is provided to allow the behavior of these programs to be interdependent.

As the slave in the relationship, the robot controller waits for commands to arrive from the master, the PC, over a serial data link. Upon receiving a command, it carries out the request and transmits a completion code to the host. To implement the slave interface on the robot controller, a ROM-based operating system was developed using the same concurrent programming techniques applied at the master level. This results in a uniform approach to programming in the hierarchical environment.

While most application programming will be done at the master level, should it be necessary to modify the behavior of the robot, its personality is defined by programs written in the same high-level language used in application programs. Thus, demanding applications may be approached by first using the level of robot control offered at the master controller. Then, should this prove to be insufficient, the robot's personality may be optimized. This is unique in comparison to today's commercial robots in that personalities are cast in proprietary, undocumented operating systems which force the programmer to work only at the application programming level. As applications become more complex, this approach grows significantly less appealing.

II. CURRENT TRENDS

A significant emphasis in robotic-control research lies in the area of second-generation programming languages. The second-generation language may be defined by contrasting the goals of its developers with the products which have been commercially available over the past few years. Current languages generally rely on a teach mode to define a sequence of points and then utilize sequential programming constructs to move the machine through these positions. Second-generation languages will emphasize the need for real-time trajectory control, decision-making capability, and asynchronous interaction with other machines and sensors. The aim is thus to provide application programmers with the flexibility to create adaptive control programs which respond rapidly enough to allow the robot's path to be modified "on the fly."

These language characteristics are necessitated by the following fact: Because manufacturing systems are rapidly becoming more advanced, robots which have the ability to interact with other computer-controlled equipment will be required in increasing numbers. Primarily because early robots were quite inflexible, most current applications are characterized as being insensitive to tolerances and as requiring little or no interaction with other intelligent machines. It is not surprising then that market surveys reveal that a high percentage (80%) of sales fall into the following applications: spot welding, spray painting, machine loading, and material handling. The hope therefore is that more powerful programming and control methodologies will increase the number of potential applications.

To gain insight into the nature of this second movement in languages, a brief examination of several articles will be presented. These works show that the development effort extends from the university research lab to a commercially available product.

In 1984, Adept Technology presented VAL-II at the IEEE Computer Society Robotic

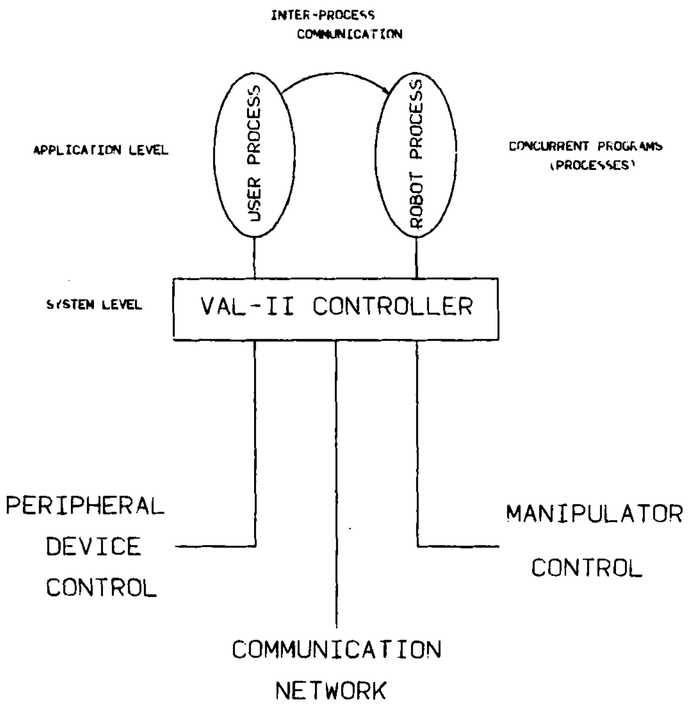

Figure 2. VAL-II architecture.

Conference.[1] This language was the result of five years of experience with the original Unimation language VAL. Adept Technology's view of the second-generation approach to programming may be understood by first examining the rationale behind VAL-II's emergence and then by discussing the capabilities of the new language.

The primary objective of VAL was to improve the motion-control capabilities of the robot by taking advantage of newer, more cost-effective, and powerful computer technology. The resulting language was a compromise between the teach pendant robot and the mainframe-based controllers used by research institutions. This compromise led to the following situation: Users accustomed to teach-and-repeat systems found VAL to be inefficient, while more sophisticated users were dissatisfied with VAL's inability to handle complex problems.

Faced with the choice of having to prioritize the needs of one of the two groups of users, Adept Technology chose the latter in designing VAL-II. By providing the new language with sophisticated programming tools, the application programmer is able to solve each of the above problems. That is, given a flexible language, the most complex application may be addressed. That application may be viewed as either the programming of the robot or the construction of a user interface for the less-sophisticated programmer.

With reference to Figure 2, VAL-II attempts to create a rich programming environment by addressing the following areas: mathematical description of robot motion, concurrent programming, and communications support. Description of robot motion in mathematical terms is a necessary ingredient for the generation of a robot-control program which performs path modification. To change the motion, one must at the very least be able to model it.

The implementation of concurrency allows the robot controller to perform double duty as both a robot-control computer and a cell controller. This is limited somewhat in that only two concurrent processes are allowed. One is responsible for directly controlling robot motion while another provides an interface for sensors and peripheral devices. Using provisions for interprocess communication, it is possible for the robot to be dependent upon the process-control algorithm.

The communication support is further extended to allow complex information to be conveyed between the two concurrent programs and other computers in the work environment. A network design is used which is reported to be specialized enough to address the needs of robotics while not rendering it incompatible with other schemes such as DECNET and ETHERNET.

To date, VAL-II's characteristics are unique to industrial robots. Many of its features are more commonly found on research-oriented systems. One such system, developed at Purdue, is examined next.

Through the language "RCCL," a somewhat different approach to language implementation is offered by Paul[2] of Purdue University. Paul's view of the second-generation programming problem is dominated by a concern for the relationship between various sensors and one or more computer-controlled robotic arms. In attempts to control this situation, it is maintained that programmability is the key to success. Consequently, robots running under the control of second-generation languages will, in Paul's opinion, emphasize the programmed aspects of motion control and minimize the teaching aspects of the first-generation languages. In addition, the need for asynchronous monitoring of sensors is necessary for path control to incorporate data gathered from the environment into a real-time decision-making process. The cumulative effect of these factors is a belief that a manipulator language should present similar features as a modern high-level computer-system programming language.

Two opposing philosophies exist among designers of programming languages. The first and perhaps more widely accepted philosophy of robotics is to dedicate the language to the application. That is, the syntax and often the architecture of the language will reflect the nature of the application at hand. This has the advantage of efficiency in describing robot motion. Disadvantages are as follows: A compiler must be written; programmers must be retrained; existing libraries are unusable. An alternative is to use an available high-level language to express robotic problems through extensions, if necessary to the language. If these extensions can be made through function cells, no compiler modification is required, and the language is implemented by linking to a library at the application-programming level. This is the approach chosen for the programming language "RCCL."

"RCCL" is based upon the "C" programming language. It is written entirely in "C" and runs under the UNIX operating system. Consisting of a library of user-callable functions and associated data structures which provide the "C" to robot interface, "RCCL" complies with the standard "C" implementation. "C" provides no syntactic support for input or output within the formal definition of the language. It merely states the characteristics of a standard I/O library.[3] Thus the aspects of the language which are machine dependent are hopefully localized to this library. In effect, "RCCL" extends the I/O capability of "C" to include the robot as a device which may be accessed (ropen'ed?) for I/O under the UNIX operating system.

"RCCL" appears to achieve its goal of demonstrating that a powerful and efficient manipulator language may be implemented through function calls. It has been interfaced to a Scheinman Stanford Arm and a Unimation Puma 600. Our interpretation of the resultant software-development process is shown in Figure 3. In contrast with VAL-II, "RCCL" does not utilize the robot's controller as a software-development station. Instead, the UNIX environment is utilized for both the development of programs and as an interface to the robot. Two positive consequences result from this: first, multiple users may work on the system simultaneously; second, more than two concurrent control processes may execute to comprise a control program.

Consequently, Paul's philosophy is not at all unlike that found at Adept Technology. Each provides the robot programmer with a more powerful language. Neither, however, addresses the manner in which the robot should interact with surrounding equipment in the workcell. Adept Technology implies, however, that the robot controller is the central element in the workcell-control scheme. This concept is challenged by some researchers.

Moving away from control of the robot in particular, the following example addresses the problem of controlling several machines which interact to perform a specific task. Alami[4] places the role of the robot controller in a somewhat different perspective.

In this work, a hierarchical view is taken of the manufacturing workcell environment. Referring now to Figure 4, a master module (MM) is responsible for directing the efforts of several subsystems. These subsystems, referred to as specialized modules, consist of the software and hardware involved in some "activity." The activity could be, for example, the control of a robot or a conveyor. It is the organization of the specialized module (SM) which is particularly interesting in the context of our work, for the manner in which the robot should interact with other equipment is explicitly stated.

The SM software is defined as a collection of primitive functions which are invoked

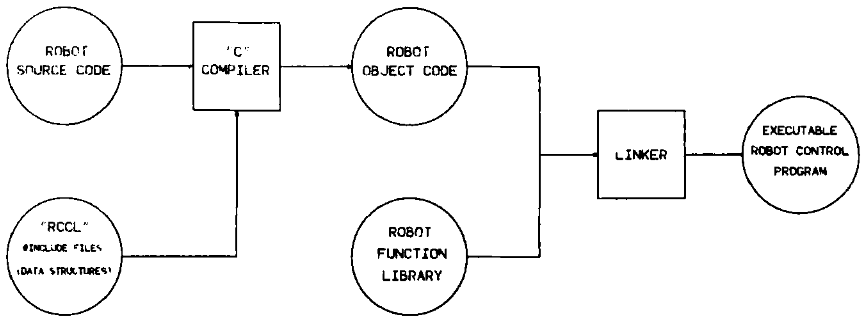

Figure 3. "RCCL" program development process.

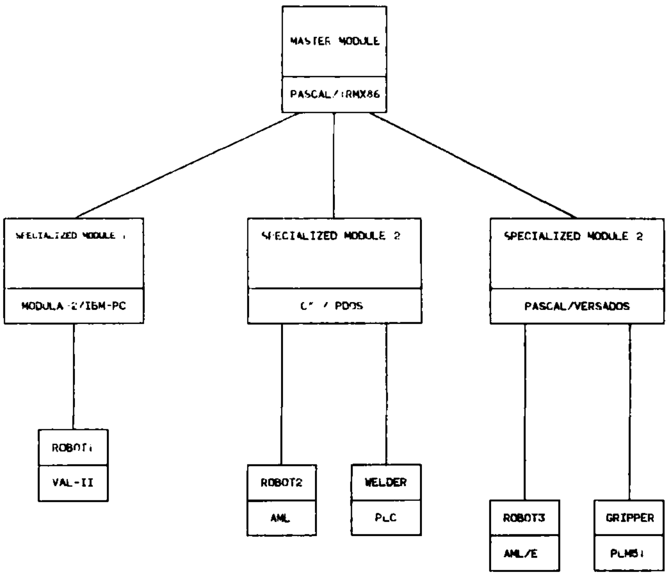

Figure 4. NNS hierarchical control structure.

by a command language interpreter. The implementation language used at the SM level is allowed to be hardware dependent. Internal operations of the SM are, however, required to be transparent at the MM level, with only the interface to the command language interpreter being visible.

To interact with the SM, and MM begins by issuing a service request. Upon receiving this request, the SM must interpret the data, invoke the corresponding function, and send a reply to the MM. At the master level, the programmer is shielded from the intricacies of these actions by the application-programming environment. To perform the programming tasks at the MM level, the Lisp language was chosen.

The NNS environment presently provides two versions of Lisp for application programming. The first, a "classical" Lisp, is sequential in nature. It does, however, provide a wait or nonwait type of request to the SM's. That is, a request to an SM may return immediately to the caller regardless of the status, or the return may be delayed until the SM issues a reply. The other Lisp implementation is an experimental multiprogrammed system which provides the concurrent mechanisms: processes, semaphores, and events. Using this version, the programmer may break a complex problem into several parallel processes, each of which has access to all SM primitives.

An interesting aspect of this approach is the view of the robot. The robot as an SM is responsible for performing some abstract function. While doing so, it must, however, be capable of responding to asynchronous events such as communication requests. Thus the overall scheme depends upon the availability of suitable peripheral devices whether they are conveyors or robots. This places a burden on the programming language employed at the SM level, for it must be used to construct a specific interface to the MM. Therefore, besides providing the capability of exercising the SM device, the language must provide low-level facilities suitable for adapting its communication interface for use in an environment perhaps not envisioned by its manufacturer.

This final point is significant with respect to the 7535. The problems with the robot seem to be associated with communication difficulties experienced when it is placed at a low level in a hierarchical environment. It is therefore suggested that the ability to modify a robot's communication protocol is highly desirable.

A final example is drawn from the Japanese Conference on Automation where two new languages were introduced.[5] While the previous examples were concerned with complex control problems, teaching many robots an identical, perhaps simple task is the issue here.

In a large manufacturing environment, a tremendous need exists for languages which will allow off-line programming. The solutions presented involve languages which have a generic robot as a target system but may be developed off line in a robot-independent environment. The two languages discussed, ARL from Hitachi and STROL from Tokyo University, take slightly different approaches to solving the problem.

ARL is a Pascal-based programming language. Robot independence is achieved by localizing robot-dependent code in libraries. As with "RCCL," an advantage to this approach is that the basic control structures will be robot independent. Consequently, programmers may move from robot to robot without retraining.

STROL has basically the same objectives as ARL, but the implementation is somewhat different. As opposed to Pascal, Forth is used as a model and the language is implemented with the use of an intermediate code known as STROLIC. STROLIC is the machine code for a virtual CPU. Robot independence is achieved by implementing the STROLIC virtual machine on various robot controllers. Then each robot will accept the same executable code from the programmer and the STROLIC interpreter will be left to insure that the desired results are obtained.

Emphasized in both efforts is the development environment used for application programming. Rather than utilize the robot's controller as a workstation, the editor and language compiler are moved to a larger computer system capable of serving many users. This has two benefits. First, it allows several programmers to work concurrently without duplicating hardware. Second, efficiency will improve for each individual through the use of the host system's utilities such as word processing, data sharing, and communication. An environment is envisioned in which code is written, simulated, and debugged without ever touching an actual robot. These issues are portrayed in Figure 5.

From the previous examples of robotic-control work, several concepts seem to be significant. Although the second-generation language may emerge as a unique creation, the possibility of it evolving instead from a more standardized programming language is certainly possible. In the likely event that many will emerge, they will share a few common characteristics. Perhaps most importantly, more attention will be given to the support of concurrent programming. The need to respond to asynchronous events will provide a driving force to ensure this. Also, new approaches will allow the robot to be programmed without the use of a teach mode. The teaching method of programming will be rendered obsolete because of the need for real-time path modification and the ability to download and execute a program without teaching points. Finally, the Jap-

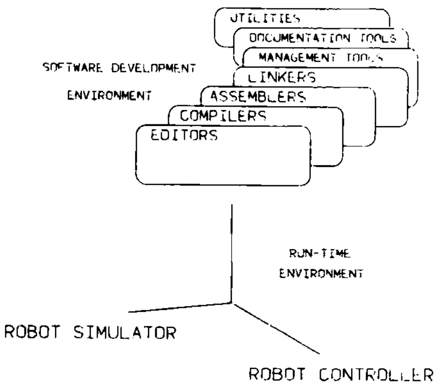

Figure 5. Ideal cross-development environment.

anese papers suggest that considerable emphasis will be placed on the development environment of robot-application programs. Rather than utilize the controller as a programming environment, cross-development tools will be employed to take advantage of superior programming environments. Then, only in the testing stage of development will the controller come into play. With these points in mind, the 7535's programming language AML/E will now be examined.

III. AML/E

AML/E is an outgrowth of IBM's original manipulator language AML. To fully appreciate AML/E, it is beneficial to first examine its parent. Only then is it possible to view the sibling with a proper perspective.

AML was a product of the T.J. Watson Research Center and was announced in 1982.[6] It emerged from the development of the RS/1 experimental manipulator which evolved into the commercial 7565 robotic system. Examination of its design philosophy reveals that AML is quite similar to the Unimation product VAL in nature. However, some of its objectives align it more with VAL-II.

In approaching the design, it was established that roughly 10% of the robot-control programs developed at the research center involved the explicit control of the robot's path. The other 90% dealt with providing an operator interface, system calibration, error recovery and data processing. Due to this, it was felt that the language's semantics should not overly emphasize the needs of robot motion. Given this constraint, it was next asked whether the new language should be developed from scratch or founded upon an existing language. A survey of available languages in 1977 led to the decision to develop a new language and to extend it to support robot control.

A significant feature of the new language was the recognition that there were several classes of robot users with considerable disparity in sophistication. The classes were organized as follows: machine operators, maintenance personnel, application programmers, and system programmers. The solution to this situation was to provide a powerful language with low-level capability which could satisfy the needs of the sophisticated user, and, in turn, could be used to implement high-level user interfaces for the other categories. The language itself supports a hierarchy of commands which is only accessible to users on a given privilege level.

The fundamental support provided by AML encompasses the areas of manipulation, sensing, intelligence, and data processing. Programming constructs along with data structures are implemented which allow the application programmer to interface with the underlying operating system. The hardware environment is the IBM Series/1 minicomputer. The complete software-development environment is implemented on this system; thus no cross-support tools are provided.

In terms of second-generation languages, AML is partially qualified for this description. While it does support a teach mode of programming, manipulator motions may be completely specified from within the language, and sophisticated vector manipulation functions are provided for modeling the robot's path. Furthermore, it supports synchronous communications with a host computer using IBM's bisynchronous communication protocol. AML's only shortcomings are perhaps its lack of support for concurrent programming and cross development.

AML/E[7] enters the picture with the 7535 SCARA robot which is manufactured by Sankyo/Seiki in Japan. AML/E is referred to as a subset of AML, but in fact it is quite different in terms of capabilities and implementation. While AML is executed by the Series/1 (interpreted), AML/E does not appear to directly execute on any machine. It instead consists of a compiler, running on the PC, that translates AML/E source code into another text file which is then downloaded to the 7535 controller. The controller, whose CPU is a Z-80 microprocessor, is believed to utilize the file as input to a command interpreter, which in turn parses the text and produces sequential robot motion.

In comparison with AML, AML/E has various deficiencies. It lacks the high-level mathematical operators of AML, it has no data-processing capabilities, and language features which would allow an operator interface to be constructed in software are missing. A shortcoming common to both is the lack of concurrent programming support. AML/E, in contrast with AML, is really only a manipulator-control language and has little capability to solve other problems in the workcell.

IV. ADOPTING THE HIERARCHICAL VIEW

Given this situation, an effort was begun to transform the 7535 into a more generic, reprogrammable machine. The hierarchical view of the robot as a specialized module receiving commands from a master module was accepted as an appropriate architecture. Support for concurrent programming, cross development of software, and a flexible approach to communications were set forth as objectives.

The resultant master-slave relationship between the master controller and the robot is diagramed in Figure 6. It shows are desire to have the programming task at each level broken down into concurrent processes. At the host level, this allows one process to manage the robot while others take care of peripheral equipment in the workcell. At the robot level, the interface presented by the robot reflects an emphasis on a primitive command structure. One process is dedicated to each of the robot's subsys-

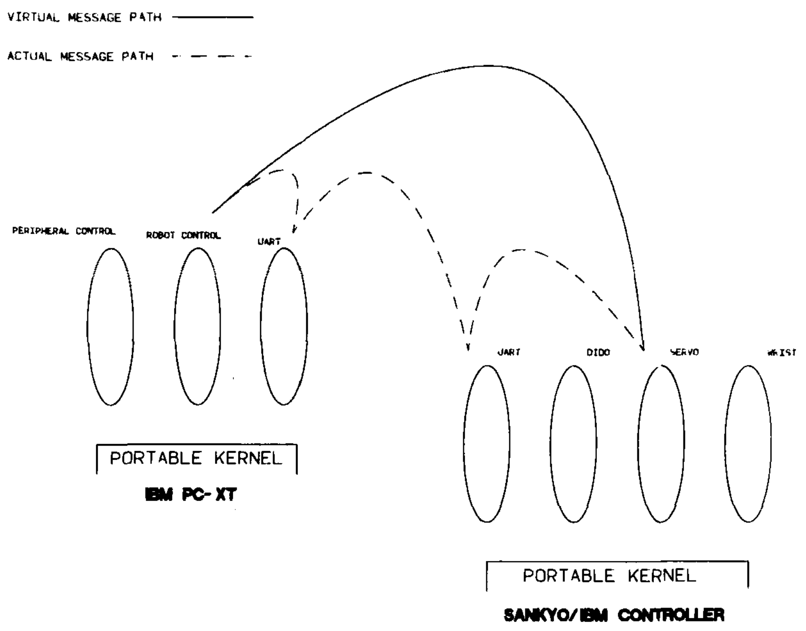

Figure 6. "Concurrent" relationship between IBM-PC XT and 7535.

tems, and they each interpret commands from the host. This results in a flexible programming interface to the robot. That is, by obeying a primitive set of commands, the robot allows its functionality to be defined more at the host level than from within its ROM-based, and hence more difficult to modify, operating system.

To implement this relationship, an IBM-PC XT was chosen as the host controller. This decision was heavily influenced by the fact that most 7535 users already own this hardware for use with AML/E. The next step was the selection of languages to be used. Required was a low-level language for systems programming, an application programming language with support for concurrency, and, perhaps most important, a development environment. Cost constraints, combined with the availability of both native and cross compilers, let to the use of a PC as a development system as well as the host. The language choice was then influenced by the requirement that both Z-80 and 8088 support be available for the PC. Due largely to this constraint, "C" won out over Modula-2 as both an implementation and application programming language. To allow for concurrent programming, a portable, real-time kernel was developed for use at both master and slave levels of the hierarchy.

For a closer look at the resultant environment, an examination of its constituent parts will now be presented. First an overview of the kernel is attempted. This should provide insight into the nature of concurrent processes. Then a brief look at the structure of the program now resident in the robot, along with the nature of host-level programming, will be taken.

One approach to implementing concurrency involves a software kernel. A kernel, or executive as it is sometimes called,[8] is actually nothing more than a program which

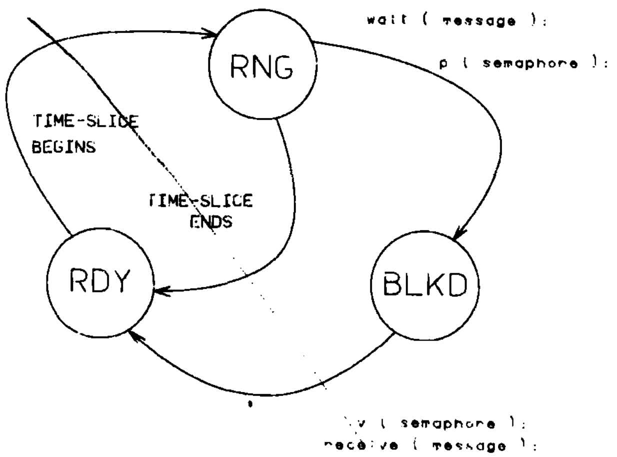

Figure 7. Process state transition diagram.

controls an application's access to the computer's hardware. To create a multitasking environment with a single processor, the kernel must create the illusion that there are multiple copies of the hardware for each program or process. The basic approach is to allow each process to run for a fixed amount of time, stop it, and allow another to run. By making the time interval small, and alternating between all the processes, it appears that they are running simultaneously. This concept is referred to as time slicing or time sharing. Other features required for true multiprogramming are provisions for process synchronization, interprocess communication, and I/O.

The result of all this is that a process may be thought of as being in one of three states READY (RDY), RUNNING (RNG), or BLOCKED (BLKD).[9] The transition from state to state is controlled by the kernel and is a function of the behavior of each process. A state transition diagram depicting this behavior is given in Figure 7.

State transitions are caused by various things. With simple time slicing, a process is cycling between the ready and running states. However, the use of semaphores allows the programmer to modify the process' schedule. For example, a process may p() upon a particular semaphore which was initialized to zero. This causes the process to enter the blocked state, and it will remain there until another process invokes the v() operation upon the same semaphore.[10]

The portable kernel, diagramed in Figure 8, was developed so that a new operating system for the 7535, along with the master-level application-programming environment, could benefit from a high-level language with support for structured concurrent programming. Starting at the lowest level of interaction between hardware and software, the interrupt mechanism, software layers were built around the systems' single-board computers. Each layer attempts to build upon the previous to present a more palatable environment to the programmer. At the outermost layer, concurrent processes are available for the programming of tasks within the systems which must execute in parallel.

Using a hardware timer found within both systems, processes are allowed to run for a fixed time interval, roughly one/thirtieth of a second. The time-slice algorithm provides each process with an equal opportunity to run, although provisions have been made to support different levels of priority.

To facilitate interprocess communication, four mechanisms are provided. The semaphore and message data type, along with shared memory between processes and the monitor structuring concept, are implemented. Using these abstractions, it is possible to modularize the concurrent program into processes which communicate with one another in a well-ordered and reliable fashion.

The semaphore data type consists of an integer count and a queue of processes which is initially empty. The kernel provides three functions which may operate on this type. Its count may be initialized and a process may either p() or v() upon a semaphore. The p(c) operation involves decrementing the count of semaphore c by 1 and then testing for count less than zero. If c.count is less than zero, the process which invoked the p() function is removed from the RUNNING state and is attached to the queue associated with the semaphore. Had the value of c.count been greater than or equal to zero, the process would have continued to run. The v(c) operation is the inverse of p(c). If c.count + 1 is less than or equal to zero, the RUNNING process is moved to the READY state, and a process is detached from the queue associated with the c semaphore. Had c.count been greater than zero, the calling process would have continued in the RUNNING state. When a process is attached to a semaphore queue, we will refer to the process as being in the BLOCKED state. Note that when an application requests a service from the kernel, that service is fulfilled with interrupts disabled. This is necessary to ensure mutually exclusive access to the kernel's resources. As an example, close inspection of the p() and v() algorithms reveal that one must ensure that the increment and test operations are performed without interrupt.

The semaphore is a primitive data type which may be used to build more advanced functions. One of these is a message-passing facility. This feature begins with the message data type. A message consists of a header containing routing information and an array of data. The header specifies the address of the sender and receiver process. This address may refer to either a process in the resident concurrent program or a process in a program running on another computer. A synchronous communication link between the two concurrent programs will route a message from a process in one system to a process in the other. This link is modeled after Intel's BITBUS interconnect

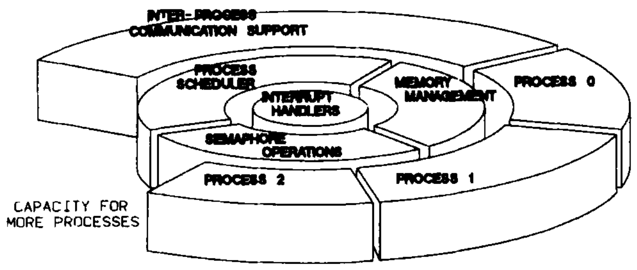

Figure 8. Structure of the portable kernel.

```
/*
**      declare CPU type and include kernel's data types
*/

#define MPU8088
#include <k.h>

/*
**      declare processes in concurrent program
*/

concurrent_program

        cobegin
                /*
                **      communication process
                */

                uart();

                /*
                **      operator interface
                */

                menu();

                /*
                **      move robot through sequence
                **      of concentric squares
                */

                square();

        coend


end_concurrent_program



```

Figure 9. Example concurrent program (only square process is shown).

protocol.[11] As with the semaphore, the kernel offers several functions which operate on the message data type. A process may build(), route(), send(), receivem(), and remove() messages. build() accepts a pointer to information to be placed in a message and returns a pointer to a dynamically allocated message containing the referenced information. route() is then used to specify the address of the source and destination process. send() acts on the message and provides the destination process with its own private copy. receivem() does the following: If a message is available, a pointer to it is returned. If not, the process is blocked. Consequently, the send() function must cooperate and unblock a process which had been waiting for a message, Finally, remove() allows a process to deallocate the storage associated with a message and receive() others.

The third feature is that of shared memory between processes. Variables may be declared to be shared between processes as opposed to local. This causes the variable to be global and another process may refer to it. The programmer is, however, responsible for enforcing access rights to the shared memory.

```
/*
**      define CPU type and include relevant data types
*/

#define MPU8088
#include <k.h>
#include <math.h>

/*
**      defines used to compute trajectory
**
*/

#define RATE 250

/*
**      declare process used to move robot through a series
**      of concentric squares
*/

square process

        /*
        **      variables used by process
        */

        double X1, X2, Y1, X, Y;
        double XSTART, YSTART, INCR;
        double a(), b();

        /*
        **      functions used by process
        */

        double a( x )
        double x;
        {
                /*
                **      return y as a function of a diagonal
                **      line across the square
                */

                return( YSTART + ( XSTART + INCR - x ) );
        }

```

To facilitate the enforcement of such rights, the monitor concept was added to the kernel in a later revision. The monitor abstraction allows a group of variables to be associated with functions which act upon them, known as monitor "entries".[12] The variables, which are important to surrounding processes, are not visible outside the monitor, while the entries are. Thus, to obtain knowledge of the variables, a process must access them by invoking a monitor entry. At this point, access rights are enforced in that only one process is allowed to enter a monitor at a given time. Once inside, however, condition variables which may be used for synchronization are available. Similar to a semaphore, the condition has two functions which operate on it: wait() and signal(). wait(c) causes a process within a monitor to be removed from the running state and blocked on c.queue. In that the process is blocked and no longer in the monitor, another process may now access the critical data. The inverse of wait(c), signal(c), results in the following: If a process is attached to the condition due to a

```
/*p*/

        double b( x )
        double x;
        {
                /*
                **      return y as a function of a diagonal
                **      line across the square
                */

                return( YSTART + ( x - XSTART ) );
        }

        move ( x, y )
        double x, y;
        {
                int error;

                /*
                **      move to x-y position
                */

                if ( (error = xy_move( x, y, RATE )) > 1 )
                {
                        printf ( "move error = %d : aborting program \n", error );
                        leave();
                }

                /*
                **      report new position at console
                */

                scr_rowcol( 13, 0 );
                printf ( "SQUARE DEMO. 1.1\n" );
                printf ( "-------------------\n" );
                printf ( "x = %3.2f\n", x );
                printf ( "y = %3.2f\n", y );

        }

```

previous wait() operation, the signaling process is blocked, and the waiting process reenters the monitor. It is unblocked once the process inside leaves the monitor. Had there been no process waiting on the condition, the signal() operation would have had no effect.

In many ways a monitor is analogous to a fence around critical data. Only one process is allowed inside the fence and any attempting to enter afterwards will be blocked on the monitor's queue of processes waiting to enter. Through the use of condition variables, processes may exchange the privilege of entrance in a well-defined manner.

To interface these structuring concepts to the "C" language, an assortment of macros and data structures have been developed, along with a library containing the necessary function definitions. The concept of a main() program is done away with, and a "concurrent" program is adopted instead. The concurrent program consists of a declaration of the processes to be executed concurrently by the kernel. Each process is then coded according to several rules. A process consists of a declaration of the process name, a declaration of the data elements contained within the process, a declaration

```
/*p*/

        /*
        **      initialization code
        */

        begin_init

                XSTART = 0, YSTART = 550, INCR = 50;
                X1 = XSTART, Y1 = b(XSTART);
                X2 = X1 + INCR;

        end_init

        /*
        **      move robot in concentric rectangular paths
        */

        begin_process

                if ( (X2 - X1) > 0 )
                {
                        /*
                        **      move to four endpoints of square
                        */

                        X = X1, Y = Y1;

                        X += INCR; move( X, Y );

                        Y = b(X);  move( X, Y );

                        X = X1;    move( X, Y );

                        Y = Y1;    move( X, Y );

                        /*
                        **      move to smaller square
                        */

                        XSTART += 5, YSTART += 5, INCR -= 10;
                        X1 = XSTART, Y1 = b(XSTART);
                        X2 = X1 + INCR;

                }
                else
                {
                        /*
                        **      restart from original square
                        */

                        XSTART = 0, YSTART = 550, INCR = 50;
                        X1 = XSTART, Y1 = b(XSTART);
                        X2 = X1 + INCR;
                }

        end_process
```

of statements to be executed at initialization time, and the statements to be executed concurrently. An example concurrent program is shown in Figure 9.

The operating system written to control the robot is divided into four processes: uart, dido, servo, and wrist. The uart process implements each end of the communication protocol (BITBUS) between MM and SM. The digital I/O is handled by the

```
#include <\pk\core\src\k.h>

concurrent_program

        cobegin
                uart();         /*  uart manager executes as proc.  0   */
                menu();         /*  user interface follows as proc. 1   */
                square();       /*  robot is managed by proc. 2         */

        coend

```

dido process, while two servo motors are managed by the servo process. To control the wrist roll axis, the wrist process manages a stepper motor.

Communication between master and slave is handled by the kernel's message-passing facility. In operation, each robot process waits for a message from the host. When a message arrives, it contains a command code and parameters relevant to the destination process. The robot process acts on the command, and when finished sends a reply message to the host. Note that because the processes are executing concurrently, a message could be sent to the servo process immediately followed by a digital I/O command. The result of this is that the two commands would be received, and the actions would occur, in parallel. The host's role in the scheme is simply to construct messages for the robot processes, send them, and then interpret the replies.

Turning now to the master module level of the hierarchy, preliminary work has been done using the portable kernel on the IBM-PC XT. As with the robot, host programming involves breaking the problem at hand down into concurrent processes. A debug program, for example, has one process to handle operator input, another for managing the serial communication link to the robot, and another for maintaining a screen display. Using this tool, the robot processes may be exercised, The communication link may be demonstrated by commanding the robot to make transitions between an on-line and off-line mode. The arm may be commanded to move from point to point, and its ability to accept a command, to stop in route, and then restart may be demonstrated.

While the above is not meant to suggest that a final solution to the 7535's control problem has been achieved, it does demonstrate that an environment conducive to such an attempt has been developed. In fact, of the total effort, perhaps only 10% of the time was devoted to writing the concurrent program which replaced the original ROMs in the robot controller. The vast majority of time went into the construction of the kernel and its associated library. What this implies is that given the proper tools, the personality of the robot may be easily changed. This is in sharp contrast to most commercial robots in use today.

V. CONCLUSION

A hierarchical robot-control system, employing the IBM-PC XT and the 7535, has been prototyped. In doing so, a programming methodology which allows concurrent processes to execute on both the PC and the robot controller was demonstrated. This

```

#include <\pk\core\src\k.h>

/*
**      defines used to compute trajectory
*/

#define RATE 50         /* always move the robot at this rate                   */
#define STEP 10         /* each square is STEP cm's narrower then the previous  */

square process

        local double  x_start, y_start, /* initial point in each square        */
                      width;            /* width of each square                */

        begin_init

                x_start = 0, y_start = 550;     /*  begin from here             */
                width = 50;                     /*  dimensions in centimeters   */

        end_init

        begin_process

                move( x_start, y_start );                       /* first point  */
                move( x_start + width, y_start );               /* then, second */
                move( x_start + width, y_start + width );       /* third        */
                move( x_start, y_start + width );               /* and fourth   */

                width = width - STEP;   /*  decrease size of each square        */

                if ( width == 0 )       /*  stop when size diminishes to nil    */
                        leave();
                else
                {
                        /*
                        **  otherwise, move to next smaller square !
                        */

                        x_start += STEP/2;
                        y_start += STEP/2;
                }

        end_process



```

allows the PC to assume the role of a master controller and the robot to function as one of its slaves in a hierarchical master/slave relationship. Much work remains to be done, however.

By introducing concurrency, a need arises for an improved language compiler. With concurrent programming, syntactically correct programs may result in unforeseen consequences due to the interaction of processes executing in parallel. Some errors of this sort could be detected by the compiler.

Another need involves communication. Our implementation utilized asychronous communication controllers at both master and slave levels. Data-link control was then realized in software following Intel's BITBUS specification. Significant improvements in performance could be realized by carrying out this scheme in hardware as it was intended to be. At the robot end in particular, the single Z-80 processor is severely burdened by an interrupt per character when messages are sent and received. The Intel

```

        /*
        **  Define a function that invokes the robot library routine
        **  xy_move() which in turn drives the robot's arm.
        */

        local int corner_counter,  /* running count of square's corners        */
                  square_counter;  /* running count of # of squares            */

        move ( x, y )
        double x, y;
        {
                int error;

#               ifndef DEBUG    /*  define DEBUG to ignore robot               */

                /*
                **  move to x-y position if not in debugging phase
                */

                if ( (error = xy_move( x, y, RATE )) > 1 )
                {
                        printf ( "move error = %d : aborting program \n", error );
                        leave();
                }

#               endif

                /*
                **  report new position at console
                **  whether debugging or running
                */

                if ( (corner_counter++ % 4) == 0 )
                        printf( "\n\nSquare #%d\n", square_counter++ );

                printf ( "\tx = %3.2f, y = %3.2f\n", x, y );

        }


```

8044 communication controller would offload the Z-80 until a complete message has been processed, allowing it to spend more time tending to arm motion.

Finally, the semantics of concurrently controlling several devices must be explored. Because the robot is now capable of accepting commands and acting in parallel with the host, an approach to programming must be devised which facilitates this way of thinking. An example of this situation is a simple command to move. A move command involves the construction, routing, and transmission of a message. The robot will receive the message, act, and then reply. The problem here involves deciding when the host will try to receive the reply. If an attempt is made immediately after transmission, the receiving process will be blocked, and it will be unable to request any action from the robot during the move. A concern here is the robot's ability to accept a command to stop during a move and then restart. If a process is blocked while waiting for a reply, it will certainly not be able to ask the arm to stop. Granted, this may not always be a concern. There could be situations, however, where the status of a sensor could be examined during the arm motion, and under certain circumstances the information would indicate that the arm should be stopped. What this seems to imply is that there should be two alternative approaches to moving the arm. One would be a simple move command in which the calling process would be blocked during the move. Another approach would involve a begin_move command in which only the command to move would occur immediately, with the attempt to receive a reply postponed until an end_move command. This would allow commands to be issued between begin move and end move. Another entirely different approach would be to have one process moving the arm and another watching the sensor. If the sensor process detected a problem, such as an operator in the work area, it could send a stop command to the servo process. This illustrates the flexibility of the communication link in that two processes could issue robot commands. Unfortunately, this power results in not one but many different ways to accomplish the task of robot programming.

It appears then that by configuring the robot as a slave, rather than a master controller, a significant amount of work has been generated for the future. Whether this is good or bad remains to be seen. However, it does allow the 7535 to accept a role in the hierarchical workcell and judging from trends in research, this is a more suitable role than that afforded by AML/E.

References

1. Bruce Shimano, Clifford C. Geschke and Charles H. Spalding, "VAL-II: A New Robot Control System for Automatic Manufacturing," *IEEE Computer Society Robotics Intl. Conf., Atlanta, March 13–15, 1984*, p. 278.
2. Richard P. Paul, "Robot Manipulator Control Using the "C" Language under UNIX," *IEEE Computer Society Robotics Intl. Conf., Atlanta, March 13–15, 1984*, p. 239.
3. Brian W. Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice-Hall Software Series, Prentice-Hall, Englewood Cliffs, NJ, 1978.
4. Rachid Alami, "NNS: a Lisp-Based Environment for the Integration and Operating of Complex Robotics Systems," *IEEE Computer Society Robotics Intl. Conf. Atlanta, March 13–15, 1984*, p. 349.
5. "Japanese Robot Languages," *Ind. Robot 11(1)*, March 1984.
6. R. H. Taylor, P. D. Summers and J. M. Meyer, "AML: A Manufacturing Language," *Int. J. Robotics Res.*, 1(3) (1982).
7. *7535 Manufacturing System User's Guide*, International Business Machines Corporation, 1982.
8. Walter S. Heath, "A System Executive for Real-Time Microcomputer Programs," IEEE Micro. 4(3), 20–29 (1984).
9. G. S. Graham, R. C. Holt, E. D. Lazowska and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, Reading, MA, 1978.
10. E. W. Dijkstra, "Hierarchical ordering of sequential processes," in: *Operating Systems Techniques*, Academic Press, New York, 1972.
11. *Distributed Control Modules Databook*, Intel Corporation, 1984.
12. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Commun. ACM*, 17(10), 549–557 (1974).