4

Intelligent Robot Programming

> Thus man is the most intelligent of all animals and so, also, hands are the instruments most suited to an intelligent animal. For it is not because he has hands that he is the most intelligent, as Anaxagoras says, but because he is the most intelligent that he has hands, as Aristotle says, judging most correctly.
>
> Galen, Greek physician
> (ca. 130–200), *On the Usefulness of the Parts of the Body* (Boorstin, 1983)

Industrial robots with smart sensors, such as vision, touch, and hearing, and machine intelligence promise to significantly increase the flexibility of robots within the next decade. According to the RIA *Worldwide Robotics Survey and Directory* (1983) there were 6301 robot installations in the United States at the end of 1982. Annual production of robots is expected to be 24,000 units by 1990. The total projected number of robots in use in the United States by 1990 is 100,000, and the Japanese expect to have 557,000 robots in use by then. Thus by 1990, several hundred thousand robots will be operating in the world, and there may well be several million by the year 2000. Many of these new installations will be using the new generation of intelligent robots that rely heavily on visual and other sensors to permit them to be intelligent in the sense that they can accommodate changes in their environment. A large number of machine and artificial intelligence techniques have been developed in the field of computer science that can now be applied to the manipulation of robots.

Intelligence is often demonstrated by game-playing skills. Chess, checkers, and Rubik's cube are just some of the games that can now be played by intelligent robots.

Sensors detect the environment. The environment in these games may be the locations of the chess or checkers pieces, or the color patterns on the cube. A computer then determines the best move, using a program based on artificial intelligence techniques, which lets it search through possible solutions. Then, the computer commands and controls the robot manipulator to carry out its series of motions to accomplish the action. A more practical intelligent application is in palletizing parcels of mixed size and weight, which requires a space-filling solution. Such tasks as part scheduling, complex assembly, and mobility also require intelligence.

4.1 Artificial Intelligence

Intelligence is a fundamental human characteristic and has many varying definitions, implied meanings, and levels of sophistication. Let us consider some of the ways intelligence may be applied to robots.

Robot intelligence permits a robot to adapt to various changes in its environment. The foundations of this ability are found in previous developments in the fields of artificial and machine intelligence. Since intelligence is difficult to quantify, let's start by reviewing the basic definition of human intelligence.

Intelligence is defined in Webster's dictionary several ways. We will consider the following:

1. The ability to learn or understand or to deal with new or trying situations
2. The ability to apply knowledge to manipulate one's environment

The first ability is much more difficult to accomplish than the second. In fact, we may distinguish between the studies in artificial intelligence (AI) and machine intelligence (MI) by these abilities. An attempt to implement the first ability into a machine may be called AI, and the implementation of the more modest second ability may be called MI. The goal of AI is to produce computer systems that imitate human performance in a wide variety of intelligent tasks. The goal of MI is to design a useful, adaptive, intelligent machine. These goals may be expanded.

The goals of AI include the following:

1. Finding new methods for extracting useful information from sensors
2. Developing methods for building, updating, and retaining information from a knowledge base
3. Inventing algorithms for utilizing the information stored in a knowledge base for making intelligent decisions
4. Finding improved methods for translating needs into a workable software system
5. Developing reusable software components that can expand toward an ultimate software system

AI systems require a knowledge base containing several types of information, such

as that about objects, processes, reasonable goals, and hard-to-represent concepts about time, space, and causality. Knowledge representation, therefore, forms the foundation upon which much AI activity is based. Many new questions about knowledge representation have been raised in the past few years. For instance, what is the best way to structure the knowledge for ease of access, storage, and use? How can a set of rules for manipulating the specific elements in a knowledge base be devised so that implicit as well as explicit knowledge can be inferred? How is incomplete knowledge to be handled? What methods can be used for acquiring new knowledge? What is the best way to extract knowledge from a human expert? How can queries about the data base be answered? These and many other questions form an exciting basis for research in AI.

Artificial intelligence is of interest not only in understanding the human but also in providing a basis for building intelligent machines. Some of the topics included under the umbrella of AI include knowledge representation, pattern recognition, computer vision, reasoning, natural language understanding, and cognition. The scientific goal of understanding the nature of intelligence is as fascinating as Einstein's involvement with understanding the fundamental principles of nature; however, it is looking inside ourselves rather than outward toward the galaxies. Attempting to build intelligent robots is a practical goal, not only permitting us to develop some very useful machines, but also leading us to very basic questions about intelligence that must be answered to understand ourselves.

The most basic act of intelligence is simply making a decision, such as to eat or not to eat. This level of intelligence is found in all the forms of animal life and is also the basic process that led to the development of computers and robots. In fact, this level is so well integrated in the human that it is considered a rather oversimplification of what we mean by machine intelligence. Today's researchers in AI are more interested in higher levels of intelligence, such as reasoning or "common sense." Common sense is vital to human survival and behavior. However, it is so difficult to define and involves so much complex reasoning that it is often cited as a quality that even some humans lack. Another interesting problem relates to how reasoning ability or judgment works. Human reasoning is very advanced. We often make decisions based upon incomplete or partial information about a situation. For example, investments in the stock market are regularly made on the basis of incomplete information. Another aspect of intelligence is expert reasoning, or the ability to make judgments based upon information that is not easily stated or even consciously recalled. For example, a physician may be able to diagnose an illness in a very short time, perhaps less than a minute. However, if asked to describe the basis of the diagnosis, the physician may require hours to describe the information from the books, medical school courses, and case histories that formed the basis for the judgment. One approach to designing an expert machine is to couple an expert, such as the physician, with a computer scientist, and let them work together to develop a computer program that integrates the expert's knowledge into a computer system. Some examples of difficult problems that have been solved in this manner include medical diagnosis, oil exploration, geologic fault isolation, genetic experimentation, and visual recognition. One implementation of this approach involves writing a computer program that asks questions of the expert to develop a knowledge data base, then continues to

form a relational model from the data base, and finally provides an expert program that can make judgements similar to those of the human expert. An example is a program that can write other computer programs. A set of questions that defines the problem and solution is asked by the machine and answered by the human. If sufficient information is given, a computer program to solve the problem can be developed. Other examples may be found in the area of computer-aided design (CAD). Here, a collection of design and analysis programs is made so that the designer simply specifies the parameters of the design and the computer analyzes the performance of the specified system. The results of the analysis are presented to the designer so that they may be compared to the original design goals. The design may then be analyzed and altered repeatedly to provide a final design. As an example of the need for this iterative process, consider the drawing of an "impossible" figure shown in Figure 4–1a. On first glance the drawing seems reasonable. However, upon checking the consistency of the drawing, it can be clearly determined that the figure cannot be built although the attempts shown in Figure 4–1b are as close as possible.

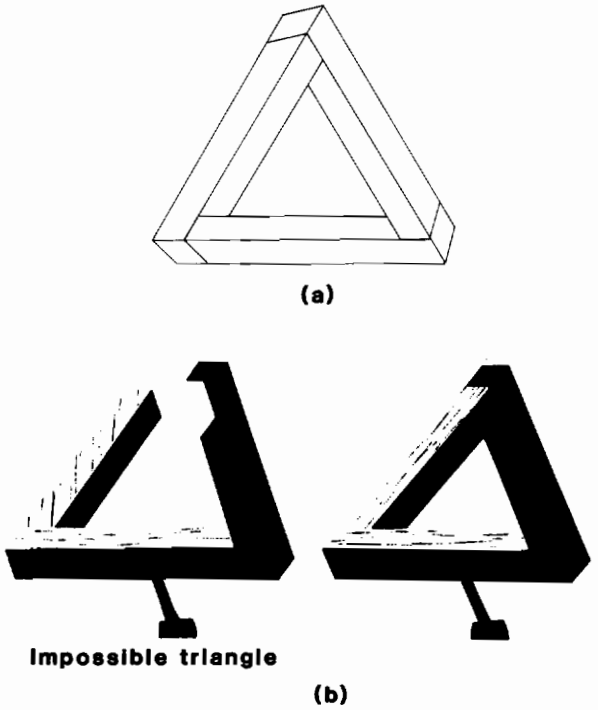


Figure 4–1. An impossible figure. (a) Drawing that appears reasonable. (b) A realization that closely resembles the design. (Adapted with permission from *Engineering Intelligent Systems: Concepts, Theory, and Applications*, by Robert M. Glorioso and Fernando C. Colon Osorio, Jr. Copyright Digital Press/Digital Equipment Corp, Bedford, Massachusetts, 1980.)

Learning from experience is another characteristic of human intelligence. One approach to machine intelligence to learn in this manner provides the machine with prototypes or examples of the desired pattern to be recognized. The machine extracts features from the prototypes and develops a decision rule based upon these features. Learning with a teacher describes the method in which a human provides the prototypes in a labeled manner so that the features of a particular class may be grouped together and then compared to the features from another pattern class. This approach has been successfully used in many applications but may require a large effort in the collection of prototypes from each pattern class. Learning without a teacher may also be accomplished by simply presenting the machine with an assortment of patterns. The machine must determine the number of classes and characteristic patterns that are used to make decisions on new test samples. This process may lead to new discoveries of pattern classes; however, long training periods may be required. Interestingly, it is common to require as many as 20 years of formal education with teachers for humans to learn from the collective experience of humankind and develop the ability to work independently. Only a few intelligent machines or programs have been developed over a 10-year period. Perhaps a longer view of machine learning is required of machines just as it is for humans.

Understanding natural language is one area that has been studied for about 20 years, with some considerable progress. Word recognition and synthesis can now be demonstrated on microprocessors in a rather limited but effective manner. Since this is a very effective form of human communication, more research can be expected in this area.

Performing appropriate actions in unusual circumstances is another characteristic of high-level intelligence. For limited action situations, such as games, the universe of all possible solutions can be represented by a tree structure to permit a machine to make a response to any possible situation by searching the tree branches in a clever manner. A clever search is required if the number of branches is too large to be searched exhaustively. We are still far from achieving this level of intelligence in the general situation. Although one can conceptualize searching all possible paths and deleting those that do not lead to a desired goal, it is possible that the computing time required to do this could run into hundreds of years, such as would be required for a game like chess.

The concepts of generalization and specialization are important to mention in this discussion of intelligence. We are constantly striving to generalize our knowledge about ourselves and the world. This process is endless. The knowledge that has been accumulated permits us to specify and engineer solutions that may be useful although limited in scope. In only a few instances have we developed universal solutions. The computer, which performs a finite number of operations to permit an infinite number of calculations, and the robot, which permits a finite number of degrees of manipulative freedom to provide an infinite number of possible motions, are noteworthy examples. The modern robot, which combines these two capabilities, offers unlimited application. Even when we do not have a general solution, a specialization of the solution may be useful. Examples are machine recognition of printed characters and limited speed recognition.

Perceiving the world around us and responding to the changes is a relatively modest

form of intelligence. For example, using an umbrella on a rainy day is a rather mundane, or commonsense reaction. This second sense of the definition of intelligence is very closely related to the form of intelligence we are trying to build into today's "intelligent robot." Interestingly, this is not such a simple task. In addition to the computer and manipulator, the intelligent robot requires sensors to permit it to understand the environment and provide a basis for reacting to changes. Sensors require design, processing, and simple implementation to be useful. Robots without sensors cannot possibly be intelligent. Robots with sensors may or may not appear to be intelligent, depending upon their program, but they do have the capability to be intelligent.

Artificial intelligence has now been studied seriously for the past 25 years. For example, Professor Marvin Minsky used robots, vision systems, and computers in his work at MIT, as shown in Figure 4–2. In the early days, direct analogies between the human brain and the elements of a computer were made. The following are examples of such analogies:

Brain.....Computer
Knowledge.....Data

Figure 4–2. Studies in intelligent robotics conducted by Professor Marvin Minsky at the Massachusetts Institute of Technology. Note the vision system, the robot, and an artificial intelligence program manipulating objects from the blocks world. (Courtesy of the Massachusetts Institute of Technology Museum, Cambridge, Massachusetts.)

Memory	Storage
Judgments	Decision making
Stream of consciousness	Program execution
Reasoning	Heuristic search
Learning	Pattern recognition
Psychology	Artificial intelligence
Vision	Image processing
Hearing	Language understanding
Speech	Voice synthesis
Movements	Robots
Common sense	Knowledge reasoning

Many consider such analogies naive, and we would have to concede. The differences between a brain and a computer are perhaps greater than the similarities. However, a great deal has been learned about human intelligence by attempting to describe intelligent activities and phenomena in sufficient detail to permit one to write a program to simulate it on a computer. However, several computer programs have been written that can demonstrate rather conclusively that the computer can outperform the human in many cases, such as at playing games. How do human and computer storage capacities compare in terms of understanding the world we live in? As Carl Sagan (1979) states, "we do not have the storage capacity either in the human brain or in our largest computers to understand a barely visible grain of salt." Let's explore this statement.

Salt is sodium chloride, or NaCl. The atomic weight of salt (Na) is 22.99 and that of chlorine (Cl) is 35.45. Therefore, the molecular weight of NaCl is 58.44. One molecular weight of NaCl weighs 58.44 grams and contains 6.02×10^{23} molecules. A barely visible grain of salt weighs approximately 1 microgram. The number of molecules in a microgram of NaCl is (6.02×10^{23} molecules $\times$ 10^{-6} grams)/58.44 grams, which is approximately 10^{+14}. Therefore, to describe the three-dimensional position of each molecule requires 3×10^{14} numbers. If each number is represented by a 7-bit binary number, then the storage capacity required to understand this barely visible grain of salt is about 2×10^{15} bits.

The human brain is known to have about 10^{11} neurons. If we assume that storage capacity is represented by a dendrite connection between neurons and that each neuron is connected to about 1000 others, then the total storage capacity of the human brain is about 10^{14} bits of information. Thus, we see that the storage capacity of the brain is less than 5 percent of that required to specify the position of the molecules in a microgram of salt.

Rather than feel limited by this comparison, however, let us compare the storage capacity of the human brain to some other forms of information storage. A book of 500 pages with 500 words per page and 5 characters per word contains about 1,250,000 characters. If each character is represented by a 7-bit code, such as the American Standard Code for Information Interchange (ASCII), then the book would contain 8,750,000 bits of information. The human brain could store the contents of more than 10 million such books.

The on-line storage memory of a supercomputer, such as the CRAY, is about 8

million characters, or 56 million bits. The storage capacity of a large magnetic disk is about 300 million characters, or 2.1 billion bits, which is still much less than the human brain. The *Encyclopedia Brittanica* has about 12,500 million characters, or 87.5 billion bits. All the written information in the National Archives has been estimated at 12,500,000 million characters, or 8.75×10^{13} bits. Even this large amount of information would not fill the 10^{14} capacity of the human brain.

The storage capacity of a robot controller may only be 64,000 characters, or about 448,000 bits. This is certainly a modest amount when compared with a human's capacity of 100 trillion bits.

4.2 Machine Intelligence

The concept of a universal machine, which has not only the intelligence capacities of the general-purpose computer, but also the manipulative capacity of a general-purpose robot, has such far-reaching applications, consequences, and perhaps pitfalls that it is difficult to describe them. The number of actual implementations of intelligent robots is still rather limited in both quantity and quality. However, the need for such systems in industrial situations is clear.

Sensors for sight, touch, hearing, sound, taste, and smell are very important to humans because they permit us to accommodate to changes in our environment. Similar sensors are useful to the intelligent robot. However, extensions of the common human senses developed over thousands of years of science can also be added to the robot.

Tools of great variety would also be needed by the universal machine. Grippers with a variety of fingers, ranging from a simple hook that could pick up a bucket to a two-fingered gripper that could pick up an object, to a three-fingered gripper that could roll a pencil in its fingers, to a multifingered manipulator that could throw a perfect football pass with the appropriate spin, could be used. A dexterous hand similar to the human hand being studied by NASA for future space applications is shown in Figure 4–3. Such tools as screwdrivers, impact wrenches, welding devices, spray painters, and even calligraphic painting brushes have already been developed for robots. The list of tools or end effectors for the universal manipulator is also quite long. Is there a universal set of tools?

Robots capable of storing and performing a fixed number of operations can be made to perform a great variety of motions. Using a control device called a "teach pendant," which permits the robot to be positioned in space and programmed, a human trainer can instruct the computer-controlled robot, which will be able to perform the sequence taught with a far higher degree of repeatability than could its human trainer. Furthermore, by recalling the sequence from memory, the robot can perform the task repetitively and consistently. However, if something in the robot's work environment changes—say, the part the robot is to pick up is missing—then the robot will appear to be "dumb," since it must still follow those same, programmed motions just as if the part were present. To appear intelligent, the robot must be able to respond to changes in its environment: the robot needs intelligence. This ability to adapt has been crucial in the evolution and survival of humans and will be essential in the development of versatile

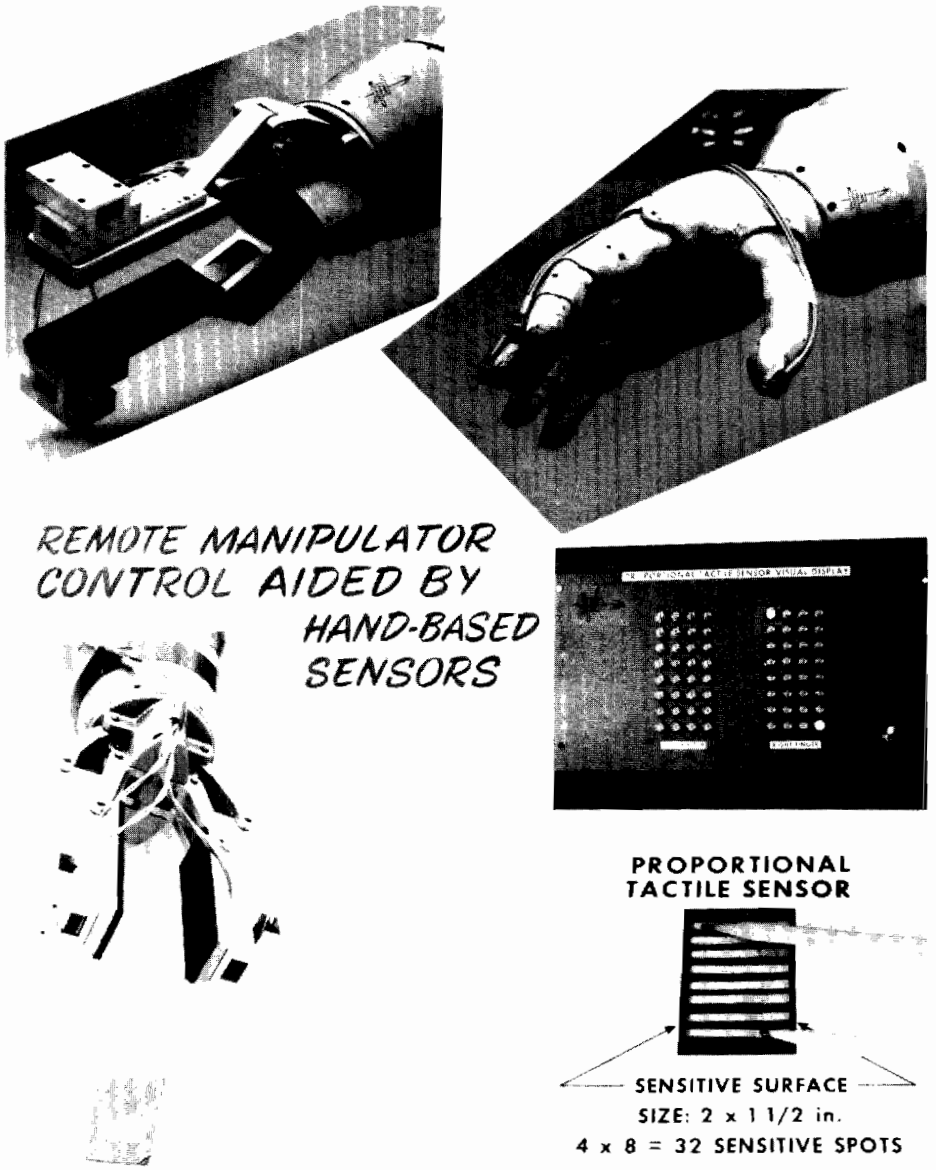


Figure 4–3 Dexterous hand being developed for versatile object manipulation. (Courtesy of JPL/NASA.)

robots. If the robot can sense changes in its environment—by using sensory perception—then it will have a better chance to solve problems for itself.

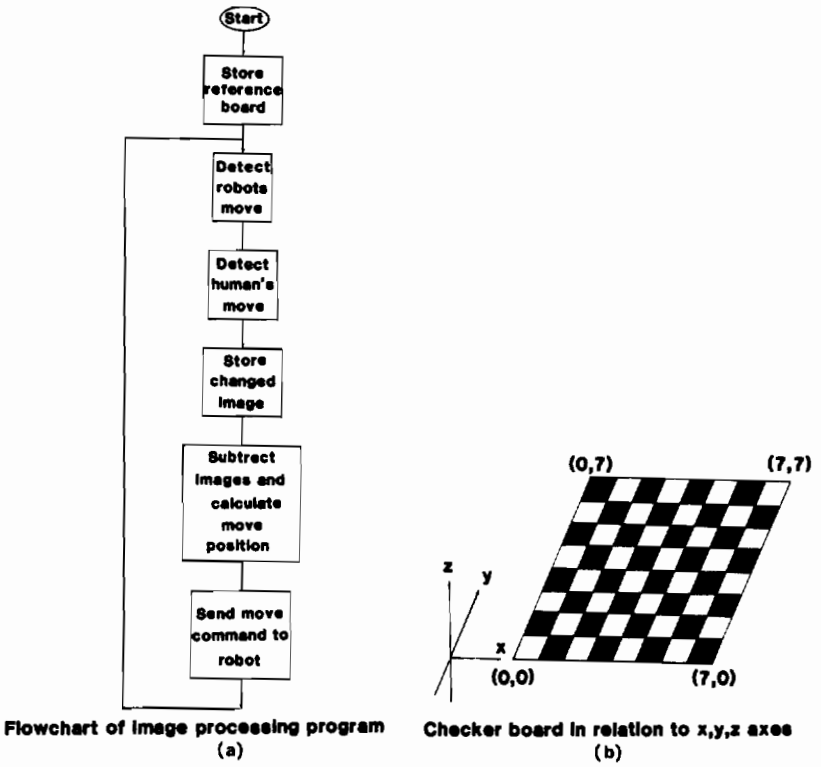
Robot Checkers Player

Equipping the robot with this adaptability involves the use of sensors and machine intelligence. Sensors are used to determine how the environment has changed, and machine intelligence is used to determine how the robot should respond to the changes.

To illustrate the steps involved in an intelligent robot, let's consider the cyclopean checkers player developed at the University of Tennessee (Hall et al., 1982). Since checkers is a well-known game, we won't go over the rules, but just the sequence of steps in the game shown in Figure 4–4. Let the robot move first. The robot arm moves out, and the gripper closes on one of its front pieces. It moves the piece forward. It is now the human's turn to move. Since the human has four possible checkers to move, the intelligent robot must sense the changes in the environment to determine where and when the human moves his checker. In the cyclopean system, this sensing is accomplished with a small, solid-state camera that monitors the board 30 times each second. It first stores an image of the board after the robot arm has made a move and has positioned itself out of the camera's view. When the human reaches to move his checker, the image in the camera changes, since the human's hand is in the field of view. This change can be detected and used to signal when the human starts and ends his move. After the human has removed his hand, a second image is taken and compared with the first. The first reference image is subtracted from the second image, which eliminates all information that did not change between the two images. Where the piece was and where it ends up are thus easily detected. At this point, the image-processing and pattern-recognition processes have extracted the relevant information from the scene. The robot now knows when and where the human has moved, and will next determine its response. The robot must now use machine intelligence to analyze the possible valid moves and to select the "best" move. The number of possible responses and the subsequent number of games are so large that an exhaustive search of all possible combinations is not feasible. Instead, a method or algorithm must be used that will search only a few of the possibilities and select a "good" move. Once the good move is determined, the robot simply reaches out and moves its piece to the desired position. The cyclopean robot plays a very natural-appearing game of checkers, even though it is implemented with microprocessors to control the camera and robot. It can accommodate multiple jumps, crown pieces, and remove jumped pieces from the board. It can even speak; for example, it may ask its human partner to wait until it finishes a move or to inform him that an illegal move has been made. It illustrates the key ingredients of any intelligent robot system—sensors, computers, and robot manipulators.

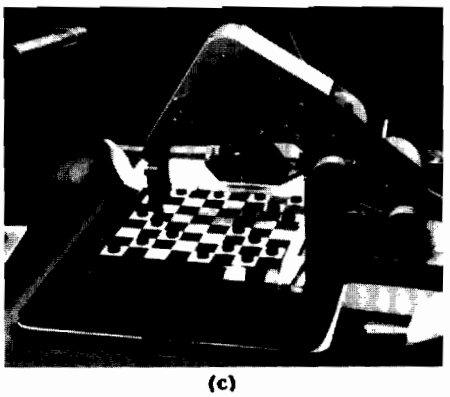
A Robot Solution to Rubik's Cube

If checkers is not challenging enough a game to convince you that intelligent robots can be "smart," then let's consider another game, solving Rubik's cube. A compact, self-contained robot called Cubot has been built for solving the three-dimensional



Flowchart of image processing program
(a)

Checker board in relation to x,y,z axes
(b)



(c)

Figure 4–4. Intelligent robot vision system designed to play the game of checkers. (a) A flowchart of the image-processing program. A reference image is first stored. The robot's move is detected from the image. The human's move is then detected. A new image of the altered positions is recorded, then subtracted from the reference image to determine the human's move. The detected positions are then sent to the robot, which uses a checkers program to determine the appropriate move for the robot. (b) The checkerboard in relation to the global coordinate axes. (c) The robot manipulator moving a game piece. (d) The overall system with camera, robot, and checkerboard and Mr. John Lesac, one of the designers.

(d)

Figure 4–4 (continued)

Rubik's cube puzzle. The system, shown in Figure 4–5, was built by researchers at Battelle Pacific Northwestern Laboratories to demonstrate the capabilities of intelligent robots (Reich et al., 1983).

Erno Rubik's cube has six surfaces, each divided into nine facets that can be manipulated to produce a solid color for each face. Rubik originally intended his invention to be used by his students as an exercise in spatial thinking at the School of Commercial Artists in Budapest, Hungary. The mathematical implications of his invention fall into the branch of rather advanced mathematics called group theory. The number of positions that can be achieved by turning faces of the cube is about 4.3×10^{19}. It would take a computer over 1 million years just to go through all the possible combinations of all the possible positions of the cube. In light of this, it is a tribute to human ingenuity that even children have solved the cube in less than 1 minute.

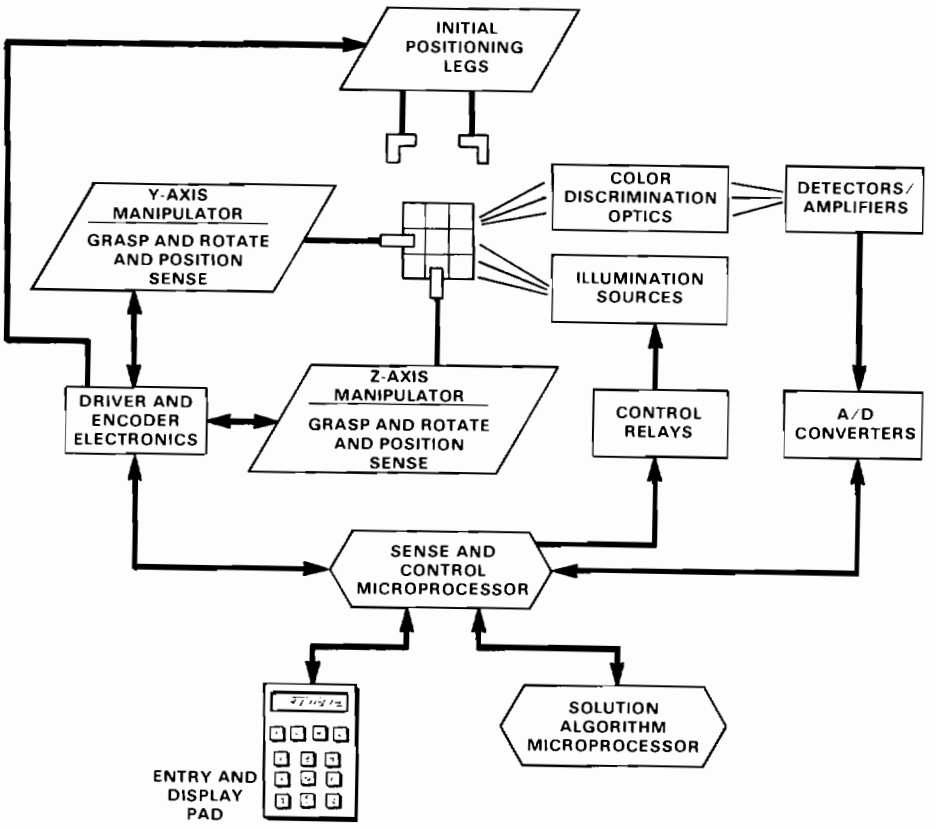
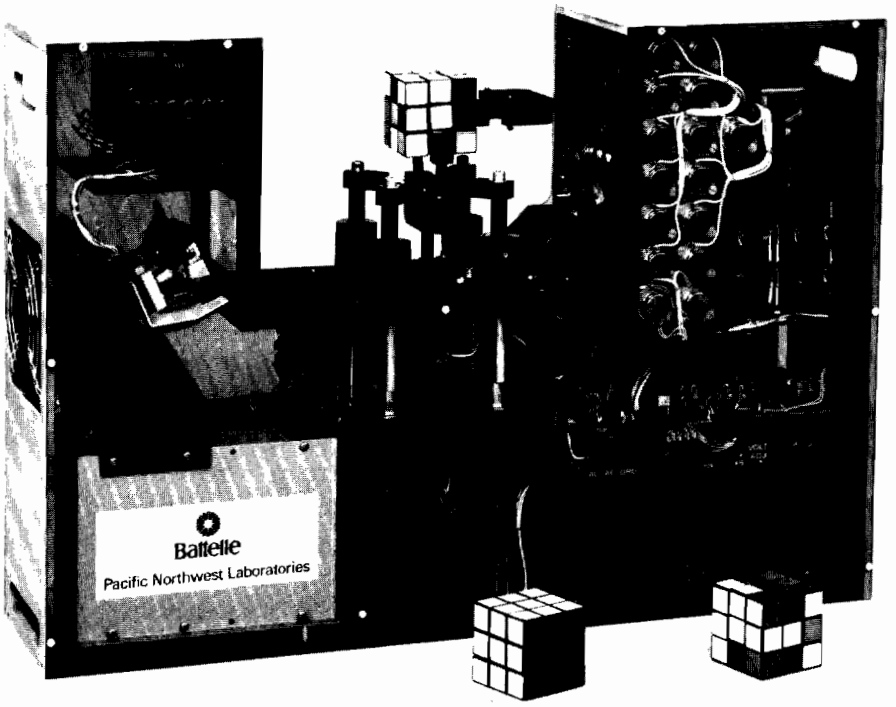


Figure 4–5. Intelligent robot that solves Rubik's cube. (a) Cubot system block diagram showing the components of the mechanical, optical, and electronic systems. (b) Actual system of Cubot. (Courtesy of Battelle Memorial Institute, Pacific Northwest Laboratories.)

Computer algorithms can determine a solution sequence in less than 1 second using AI techniques. However, the robotic solution also consists of determining the initial position and physically manipulating the cube to its solution. Students at the University of Illinois recently completed the first robot solution to the cube. The robot solution described here is a bit more sophisticated than the first solution and solves the cube without human intervention by using sensors, a computer, and a special robot.

The Cubot has three major subsystems. The electro-optical system illuminates the cube and provides a color readout of the cube faces. The microprocessor interprets these sensed data, computes a solution, and formulates an instruction sequence for the robot manipulator. The manipulators and grippers then rotate the cube faces based upon the solution sequence. Tests of the system indicate that it can solve the most scrambled cube in less than 3 minutes. Although it is not as fast as humans, the Cubot is a demonstration of the type of intelligent robot technology that can surely be called "smart."



(b)

Figure 4–5 (continued)

As we have seen, cameras, visual point sensors, and tactile touch sensors may be used to sense the environment. In general, we may divide sensors into two categories, those that require contact with the surface and noncontact sensors. The camera is popular because it is a noncontact method of measurement and because it gives a set of measurements over a spatial region and not just at a point. It is natural to use a camera to guide a robot to a position, but it may be necessary to use a contact sensor to perhaps pick up a part at the position. In the cyclopean checkers player, a force sensor in the hand is used to check that the robot actually picked up a piece. In other situations, such as fitting a chess piece into a hole, a contact sensor is used to determine the location of the piece.

4.3 Voice Control of Robots

The use of speech recognition for the control of a robot provides a desirable method for human-machine interaction. Speech-recognition equipment is used in many applications, such as extending the capabilities of the physically handicapped. There are now a number of cases of people running complete offices from their homes, including typing, telephoning, and environmental control, such as lights and windows, using speech

control. The use of speech control of a robot manipulator should provide a valuable extension in the ease of use of robots. In 1983, at the Automan exhibition in Birmingham, England, a Cincinnati Milacron T3 726 robot was demonstrated, that was controlled by a voice. Visitors would tell their initials to the robot, and the robot would engrave those initials on a glass paperweight submerged in water.

There are now about 250 different speech-recognition systems available worldwide. Recognition capabilities are limited from about 32 to 256 words. A training mode is used to permit the system to adjust to one or several speakers. Recognition rates of 99 percent have been reported. Representative companies include Verbex of Bedford, Massachusetts, a Division of Exxon Enterprises, Votan of Fremont, California, and Scott Instruments of Denton, Texas. This area appears quite rich for further developments.

4.4 Programming a Robot

The bridge between the robot, its sensors, and its application is the human programmer. A human must write the program, or at least write the program that writes the program to be used. The easier it is to develop or modify the program in a robot, the easier it is to solve problems for a given application. The need for ease of programming is great in giving the intelligent robot versatility. Of course, in many applications, an industrial robot will be installed and can run the same program for years. However, if a robot is to be in a work cell making small batches of products or if product changes are made periodically, then new or modified programs will be required almost continuously. Since the robot is designed to be versatile, the software should also be versatile.

Certain elements of the robot controller must be written in very efficient languages for speed. The basic servo control loops and drivers for sensors fall into this category. However, perhaps just as important as speed of operation is ease of use. This is why most manufacturers also supply a robot programming language and operating system support for the robot controllers for the software changes required to make the robot truly versatile.

Even if each manufacturer supplied a language to make its robots easy to use, we encounter a dilemma: there are almost as many robot languages as there are robots. One consequence of this is, if a manufacturer attempts to use more than one brand of robot in an application, the technical staff must learn more than one language. This is not an impossible task for the trained computer engineer or scientist, but it can be a burden to operators and technicians. One practical solution to this problem is for the company to specify a given type of controller and interface when purchasing a variety of robots. However, this is not the best solution.

If we look at the general computer world, the same phenomena occurred and are still going on. Each computer manufacturer must develop an assembly language unique to the architecture of its computer. However, groups of users quickly discovered that they were expending a good deal of redundant effort in developing assembly language subroutines to multiply, divide, compute trigonometric functions, and solve linear

systems of equations. The invention of such high-level languages as FORTRAN for scientific users, COBOL for business users, BASIC for beginning users, ALGOL for mathematics, LISP for symbol manipulation, PROLOG for logic programming, and more recent structured languages, like PASCAL, provided a base for the development of machine-independent and somewhat transportable programs. A compiler, which translates the high-level language into a specific machine language, is required for each machine. However, rather than this becoming a complete Tower of Babel, with each machine requiring its own language, we have arrived at a sort of international state, with only half a dozen or so languages required of the user. Each language has its proponents and adversaries. The difficulties involved with writing workable software and writing readable software have not yet been overcome. In the near future, we cannot expect that a single acceptable robot language will come about. However, we can expect the number of languages to narrow down to just a few to facilitate robotic operations by the use of translator programs used for off-line programming, that is, programming not done directly on the robot controller.

Because of the different requirements in speed and complexity for the versatile robot, there are some natural divisions in the requirements for programming languages. As with hierarchical control, we may also think of a hierarchy of programming languages. At the top of the hierarchy, there is a need for a production control language (PCL), such as the one being developed at the National Bureau of Standards, which provides a common interface between the functional levels and applications areas of the automated factory (McLean et al., 1983). There is also a clear need for an off-line programming language, such as a manufacturing language (AML) developed by IBM, or the Stanford artificial intelligence language (AL), which permits the off-line development and simulation of robot control programs that can then be down-loaded to the robot controller for final testing and implementation. Off-line programming permits many programmers to develop programs for a single robot. There is also a need for an on-line programming language for each robot that permits teach control and real-time programming and testing. Furthermore, if we are to use much of the previously developed software and expertise from the artificial intelligence community, then perhaps some attention to symbol-manipulating languages, such as LISP, might be prudent. Rather than answer the unanswerable question about which programming language is superior to all others, we will simply observe that some language is necessary and review some of the languages we might encounter.

First, any computer has a set of instructions, called machine language, that specifies the operation to be performed by the machine as it cycles through its program. For example, on a 68000 microprocessor, the operation code that instructs the machine to add two numbers is the binary number 1101, which is the decimal number 13, or hexadecimal number D. In the binary number system, strings of the binary digits 0 and 1 are used to represent a number. The digits in the string are multiplied by the powers of 2, such as 1, 2, 4, 8, and 16. In the octal number system, strings of octal digits, 0, 1, 2, 3, 4, 5, 6, 7, are used to represent a number. The digits in the string are multiplied by powers of 8, such as 1, 8, 64, and 4096. Octal digits exactly represent groups of three binary digits. In the decimal number system, strings of decimal digits, 0, 1, 2, . . ., 9, are used

to represent a number. The digits in the string are multiplied by powers of 10, such as 0, 10, 100, and 1000. In the hexadecimal number system, a number is represented by a string of hexadecimal digits, 0, 1, 2, . . . , 9, A, B, C, . . . , D. Hexadecimal digits exactly represent groups of four binary digits. The digits in the string are multiplied by powers of 16, such as 1, 16, 256, and 65,536, to evaluate the number. A total program can be developed in machine code, as is required on the Heath Company Heathkit HERO 1 robot. However, it is much easier to use a mnemonic, such as ADD, which can be directly translated into the binary number 1101 but is much easier to remember. An assembly language consists of a set of mnemonics and operations that can be directly translated into machine code. An assembly language program is translated into machine code by a program called an assembler. Assembly language is the most difficult language in which to write a program, because it involves thinking of operations in terms of the basic computer operations, such as register transfers. However, it generally results in the fastest and most powerful way to use a computer.

An interpreter language is a method for implementing a higher level language. Languages of this form have instructions that can be directly interpreted into machine code, often in a one-to-many translation, then executed. BASIC (beginners' all-purpose symbolic instruction code) is one of the most popular interpreters. For example, the instruction X = 1 + 2 may generate a machine code that instructs the computer to add the two numbers and store the result in the memory location labeled X. Interpreters are the easiest languages in which programs may be written, since they can be executed at any time and provide direct feedback about errors. However, it is also the slowest in terms of execution, since the commands are retranslated each time the program is executed.

A compiler program is used with yet higher level languages that interpret commands into many machine language commands and can also optimize the code for such functions as high-speed execution or minimum storage. A compiler requires that a complete program be translated at the same time, so that a single typing error can make it necessary to repeat the compilation process. Another program, called a linker, is often required so that subroutines and other program segments may be compiled separately, then combined to form an executable module.

Each of these approaches uses another program, called an editor, to permit one to enter commands. The editor, assembler, interpreter, compiler, and applications programs are managed by an operating system program, which supervises the use of memory, storage, and machine control. Various operating systems are available for each brand of computer.

We might also want to distinguish between a general-purpose computer and a programmable controller. The main distinction is that a programmable controller is a general-purpose computer used in a limited application without such peripherals as a line printer or a mass-storage device, but with excellent interface signal support. Also, a special logical notation, called ladder logic, is often used in industry to develop programs for programmable controllers. Ladder logic is especially useful for systems with many switches and timing controls.

With the great variety of other languages available, why do we need a robot language? The main reason is that, in controlling a robot, certain primitive operations,

such as moving the robot to a position in space and opening or closing a gripper, are used repeatedly, and these operations are specific to robots.

Various levels of robot programming languages have been described by Bonner (Bonner and Shin, 1982). Level 1 is at the microcomputer hardware level and is used for direct robot actuator control. A special machine language called microcode or normal machine language is used at this level. Level 2 consists of a robot path controller high-level language. Teach-by-example programming is indicative of that in which the programmer moves the robot to a specified point in space and pushes a program button to record the actuator locations. A high-level language in the robot controller translates these commands into level 1 commands. Level 3 is also programmed in the robot controller at the primitive motion level, but has basic operations that can be specified by such mnemonics as MOVE. The higher level languages are usually used for off-line programming. Level 4 is a structured programming level in which complex data structures, the use of predefined state variables, and sensor commands are built into the language. Level 5 is a task-oriented level in which English language-type commands may be given. This type of language would depend upon a world model being stored in the computer so that such commands as PICK UP THE TOOL could be interpreted unambiguously. We might even envision a level 6 language in which strategies, motion paths, and a knowledge base are built into the overall system to such an extent that such commands as BUILD TRUCKS might be interpretable.

Some of the characteristics of a good robot language include clarity, simplicity, naturalness, debug and support capabilities, ease of extension, decision-making capabilities, ease of interaction with external devices and sensors, concurrent or parallel operations, and interaction with data base systems.

Currently, the main division of interest in industrial robot applications is between on-line and off-line programming. From the short-term industrial point of view, on-line programming using teach by example offers a simple and low-cost solution that can be quickly mastered by an operator. Also, the microprocessor for on-line programming is simpler and costs less than more complex systems. From the long-term point of view, as CAD and computer-aided manufacturing (CAM) merge together in factories, off-line programming offers advantages of multiuser capabilities, distributed processing, and system integration. Both methods are useful.

Level 1 Robot Languages

Let us now consider some specific robot-programming examples. At the first level, the program would be written in a specific assembly language for high-speed, real-time response. As an example, let us consider a machine language portion of the higher level language called ARMBASIC, which was developed in Z80 assembly language for the Microbot, Inc., MiniMover 5 robot (Microbot, 1980).

The MiniMover 5 uses stepper motors to actuate its 5 joint degrees of freedom and gripper. These motors have four coils, each driven by a power transistor. The command signals are digital. That is, the motors may be driven clockwise or counterclockwise by the execution of the following binary commands on the four drive lines going to each motor.

Motor drive signals				
d_1	d_2	d_3	d_4	Hexadecimal value
0	0	0	1	1
0	1	0	1	5
0	1	0	0	4
0	1	1	0	6
0	0	1	0	2
1	0	1	0	A
1	0	0	0	8
1	0	0	1	9

A sequence of commands that goes down the table drives the motor clockwise; a sequence that goes up the table drives the motor counterclockwise. When either end of the table is reached, it is necessary to go to the other end of the table and continue sequentially. The program that performs this control of the motors is called STEP. Only the segment of the program that steps all six motors will be considered. A flowchart of the motor drive program is shown in Figure 4–6. To introduce the level 1 assembly language program, let's first consider the following high-level language description of the program segment.

```
THE MAIN LOOP TO STEP ALL MOTORS
FOR N ::= 1 TO MAX
     FOR I ::= 1 TO 6
          SUM(I) ::= SUM(I) -DELT(I)
          IF SUM(I) < 0 THEN
          SUM(I) ::= SUM(I) + MAX
          PHASE(I) ::= (PHASE(I) + DIRC(I))MOD 8
          SEND MOTOR NO AND PHASE CODE TO OUTPUT
          ELSE DELAY FOR SAME AMOUNT OF TIME
          END IF
     END FOR
DELAY LOOP;COUNT DOWN "CONT"
END FOR
```

The execution of this program is as follows. A form of proportional path motion is implemented in which all motor rotations will end at the same time. Therefore, the value of MAX is the largest number of steps required of any of the six motors. Thus, the first FOR loop will index this largest number of steps. The statements that start and end this loop are

```
FOR N ::= 1 TO MAX
END FOR
```

Since there are six motors, the inner FOR loop will be executed for each motor. The statements that start and end this loop are

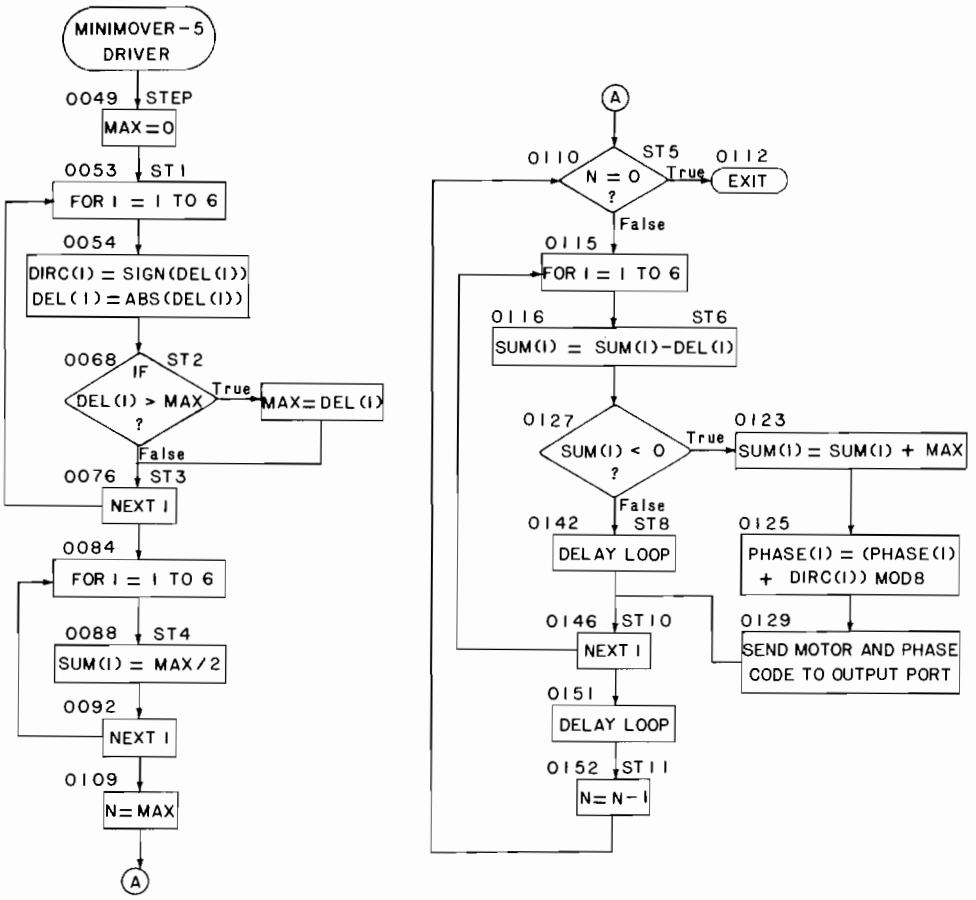


Figure 4–6. Flowchart of the Microbot MiniMover 5 robot, stepper motor drive program. (Courtesy of Microbot, Inc., Mountain View, California.)

```
FOR I ::= 1 TO 6
END FOR
```

All elements of the array SUM have been initialized to the value of MAX/2. The array DELT contains the number of steps required for each motor. The first loop will be performed the maximum number of times. However, only the motor making the maximum number of steps is moved each time. The others are delayed some of the time. If DEL = MAX, then the motor should step each time through the loop. If DEL = MAX/2, then the motor should step each second time through the loop. If DEL = MAX/3, then the motor should step each third time through the loop. Finally, if DEL = MAX/MAX = 1, the motor should only step once. The logic is implemented with the IF-THEN-ELSE statement:

```
IF SUM(I) < 0 THEN
  SUM(I) ::= SUM(I) + MAX
  PHASE(I) ::=(PHASE(I) + DIRC(I))MOD 8
ELSE DELAY FOR SAME AMOUNT OF TIME
```

The algorithm used to determine the number of increments is based on a clever digital differential analyzer used in machine tool control (Seim, 1980). Let's consider the following simplified version for a single motor.

```
DEL = 7
MAX = 10
SUM = MAX/2
   FOR N = 1 TO MAX
    SUM = SUM-DEL
    PRINT "SUM"; SUM
    IF SUM < 0 THEN SUM = SUM + MAX
   NEXT N
```

When this simple BASIC version is executed, the following sequence of values of SUM are generated: –2, 1, –6, –3, 0, –7, –4, –1, 2, –5. Note that exactly seven negative numbers are generated. For each of these, the motor is stepped. This example may be executed with various values of DEL and MAX to verify that the SUM will be less than 0, only DEL times through the loop. Therefore, when this condition is satisfied, the motor should be stepped.

Now, let's consider the assembly language version of this motor drive. For clarity, we will place the high-level command as a comment for the assembly code.

```
LABEL    OP CODE  OPERANDS        ;COMMENTS

              DATA
CONT     DEFS 2
ARRAY    EQU  $
DEL      EQU  $-ARRAY
         DEFS 2 * 6
SUM      EQU  $-ARRAY
         DEFS 2 * 6
DIRC     EQU  $-ARRAY
PHASE    EQU  $-ARRAY + 1
         DEFS 2 * 6
MAX      DEFS 2 * 1
         LD   BC, (MAX)          ;LOAD THE VALUE OF MAX
ST5      LD   A,B                ;FOR N ::= 1 TO MAX
         OR C
         RET  Z                  ;THE EXIT CONDITION
         PUSH BC
         LD   B,6                ;B IS COUNTER FOR MOTOR
                                 ;LOOP
```

```
        LD    IX,ARRAY          ;REFERENCE POINTER FOR
                                ;DEL
ST6     LD    H, (IX+SUM+1)     ;FOR I ::= 0 TO 5
        LD    L, (IX+SUM)       ;HL ::= SUM(I) - DEL(I)
        LD    D, (IX+DEL+1)
        LD    E, (IX+DEL)
        XOR   A
        SBC   HL,DE
        JR    NC,ST8            ;IF HL < 0 THEN
        LD    DE, (MAX)         ;HL ::= HL + MAX
        ADD   HL,DE
        LD    A, (IX+PHASE)     ;(PHASE(I) ::=
        ADD   A, (IX+DIRC)      ; (PHASE(I)+DIRC(I) )
        AND   7                 ;MOD 8
        LD    (IX+PHASE),A
        PUSH  HL                ;SAVE CONTENTS OF HL
        LD    HL, TABP
        ADD   A,L
        LD    L,A
        JR    NC,ST7
        INC   H
ST7     LD    D,(HL)            ;TABP(PHASE(I) )
        POP   HL                ;RESTORE HL VALUE
        LD    A,PORT+6          ;PORT FOR MOTOR I
        SUB   B
        LD    C,A
        OUT   (C),A             ;SEND COMMAND TO MOTOR
        JR    ST10              ;ELSE
ST8     PUSH  BC
        LD    B,1               ;DELAY FOR EQUAL TIME
ST9     DJNZ  ST9
        POP   BC                ;END IF
ST10    LD    (IX+SUM+1),H      ;SUM(I) ::= HL
        LD    (IX+SUM),L
        INC   IX
        INC   IX
        DJNZ  ST6               ;END FOR
        LD    BC,(CONT)         ;DELAY LOOP
ST11    DEC   BC
        LD    A,B
        OR    C
        JR    NZ,ST11
        POP   BC
        DEC   BC
        JR    ST5               ;END FOR
;
;       TABLE OF PHASE CODE
;
TABP    DEFB  1,5,4,6,2,0AH,8,9
```

The assembly language version requires one to operate at the basic machine level. This requires a knowledge of the machine architecture as well as the instruction set. The architecture of the Z80 includes two sets of registers; however, only one will be used in this example. The registers are called A, F, B, C, D, E, and H, L. Their contents can be operated in 8-bit byte segments or as 16-bit register pairs for the BC, DE, and HL pairs. The machine also contains other 16-bit registers, such as the index registers IX and IY, the program counter PC, and the stack pointer SP. The A register or main accumulator holds one operand for adds, subtracts, and some other operations; the other operand may come from the other registers or memory. This permits two operand instructions. The F register holds the conditional bits or flags, such as the negative, zero, and carry bits, which may be set as the result of an operation. Let's go through the assembly language program statement by statement, just as it is executed.

The first statement simply loads the contents of the memory location labeled MAX into the BC register pair. Note that the variable names, such as MAX in the previous BASIC programs are implemented by assigning a memory location to the variable name and storing the value of the contents in the memory location. For example, MAX=7 may correspond to assigning memory location 1000 as MAX and storing 7 in the memory location. The instruction LD BC,(MAX) puts the contents of (MAX) into the B register and the contents of (MAX+1) into the C register. Note that the syntax first gives the operation LD, then the operands BC and (MAX). Since BC is a register, its address is known to the machine and thus its contents are simply referred to as BC; however, for memory location MAX, the address of this location, for example, 1000, would be obtained by using the syntax MAX. To obtain the contents of the location, we must use the syntax (MAX).

The next statement starts the loop for the number of steps:

```
LD A,B
```

Suppose the value of (MAX) is 7. In the previous instruction, this value was transferred to the B register. The value 7 is now transferred to the A register for ease of further operations. If the value (MAX) were less than 255 and (MAX+1) had previously been initialized to a value 0, the C register should contain a zero.

The next instruction simply sets the flags to determine if the contents of the A register equal 0. This is accomplished by

```
OR C
```

This instruction logically ORs the contents of the A and C registers and sets condition flags in the F register. Since C contains 0, the result of the OR instruction is simply equal to the contents of A. Only when all the bits of A are zero will the zero flag in the F register be set.

The next instruction

```
RET Z
```

returns to the instruction immediately following the one that called this program if the zero flag Z is set. The loop from 1 to MAX is therefore implemented in an A register, then later decrementing A and returning when its value is zero. Note that this comparison at the beginning of the operation counts the correct number of iterations. For example, if MAX=7, the sequence would be 7, test, 6, test, 5, test, 4, test, 3, test, 2, test, 1, test, 0, exit.

The next few instructions set up the loop counter for the six motors by first saving the contents of BC and then putting the value 6 into the B register, that is,

```
LD B,6
```

Next, the memory address of the array ARRAY is loaded into the index register IX. This technique permits IX to be used as a pointer to values of the array, using the indexed addressing mode. Two arrays are used. One is used to store the six values of SUM(I) and the other to store the values of DEL(I). These values are stored at locations IX+SUM and IX+DEL, with two memory words used for each parameter. For example, SUM could equal 0 and DEL could equal 12.

The next four statements put particular values of SUM(I) into the HL register and DELT(I) into the DE register. Note that the 16-bit integers are stored with the most significant byte in the low-order memory location and the least significant byte in the high-order memory. Thus, the individual bytes are transferred to the four individual registers in the following manner.

```
LD    H, (IX + SUM+1)
LD    L, (IX + SUM)
LD    D, (IX + DEL+1)
LD    E, (IX + DEL)
```

Note the difference between loading the HL register by LD HL, (IX + SUM).

The next instruction,

```
XOR A
```

performs a logical, exclusive OR of the contents of the A register with itself and sets flags. The exclusive OR of two 1 bits is a zero, so this instruction essentially forms a zero result and resets the carry and sign flags and sets the zero flag.

The next instruction,

```
SBC HL,DE
```

subtracts the contents of DE from HL and puts the result in HL. The next instruction,

```
JR NC, ST8
```

jumps or branches to location ST8 if the results of the subtraction SUM(I) did not set the carry flag, that is, were positive.

If the result SUM(I) was negative, then the next two instructions add back the value of (MAX), that is,

```
LD DE, (MAX)
ADD HL, DE
```

The next instruction loads the PHASE(I) value into the A register, that is,

```
LD , (IX+PHASE)
```

The phase value is stored following the DEL(I) values.

The next instruction adds the direction value DIRC(I) and saves the result in the A register, that is,

```
ADD A , (IX+DIRC)
```

To compute the sum modulo 8, we simply throw away any bits past the third bit or numbers greater than 7. This is accomplished by the logical AND of the contents of the A register and the number 7, that is,

```
AND 7
```

This result is now stored as the new value of PHASE:

```
LD (IX+PHASE) , A
```

At this point, the HL register contains the updated value of SUM(I). It is now saved for the output to the motor and the next iteration computation by pushing it on the stack, that is,

```
PUSH HL
```

The next sequence of instructions sets the phase codes to control the direction and step code for the motor. The address of the stepper motor codes is first loaded into the HL register, that is,

```
LD HL , TABP
```

Next, the value of the phase offset is added to the address:

```
ADD A,L
```

This value is now placed back into the L register so that the HL pair now points to the appropriate phase value in the table:

```
LD L,A
```

A jump to ST7 is now made if the ADD instruction did not result in a carry. If a carry did result, the H register is incremented, that is,

```
INC H
```

At ST7, the command values are output to the motor. First, the motor code is loaded into the D register by

```
ST7  LD D, (HL)
```

The old value of SUM(I) is put back into the HL register by

```
POP HL
```

The port number is now loaded into the A register by

```
LD A , PORT+6
```

The loop counter value is subtracted from the A register

```
SUB B
```

The resulting port value is now placed in the C register by

```
LD C,A
```

Finally, the command is sent to the motor by

```
OUT  (C),A
```

The program now jumps to ST10. At this location, the value of SUM(I) is updated in the memory array by

```
ST10                              LD (IX+SUM+1), H
                                  LD (IX+SUM), L
```

Also, the array counter is incremented by 2 to skip past the previous value by

```
INC  IX
INC  IX
```

The B register contains the loop counter. The next instruction decrements this loop counter and branches to ST6 to start the next iteration if the result is not zero by

```
DJNZ ST6
```

If the result is zero, the loop is completed and the next step sequence can be started. To control the motor speed a delay loop between the steps is started. This permits a speed control of the movements. A counter value is loaded into the BC register by

```
LD   BC,(CONT)
```

The BC value is then decremented by

```
ST11 DEC BC
```

The most significant byte is then loaded into the A register by

```
LD   A,B
```

The logical OR of this and the C register is then computed by

```
OR   C
```

When both bytes of BC contain zero, this operation will set the zero flag. The next instruction tests this condition:

```
JR   NZ, ST11
```

After the delay, the saved value of BC, which is (MAX), is started again by

```
POP  BC
DEC  DC
JR   ST5
```

One might ask why it is necessary to write the motor drive program in assembly language, since it requires a great deal of effort to read and write such a machine-dependent language. The main answer is speed. Some of the motor drive commands might operate at rates of a million per second. High-level languages have not yet reached this operating speed. Assembly language and microprogramming, which we have not discussed, are both much easier than building an electronic circuit to perform the operation. However, from the user's viewpoint, higher level languages are definitely desirable. Let's now look at some of the higher level robot languages.

The second level of programming uses a language designed for the primitive operations of the robot and is operable on the robot controller. For the MiniMover, a language called ARMBASIC is used, which has the following primitives.

@STEP Moves the manipulator with the delay between pulses to the motors and the number of steps for each motor as a parameter.
@CLOSE Closes the gripper with the delay for the gripper motor as a parameter.
@SET Allows the manipulator to move under manual control with delay as a parameter.
@RESET Zeroes the arm position and motor currents and is used for arm position initialization.
@READ Returns the current position of the manipulator by returning the values of the six position registers for the six drive motors.
@ARM Selects the port number ARMBASIC uses. This permits the control of the robot arms.

A simple example of the type of program that can be easily written in ARMBASIC is a pick-and-place program. Suppose an object is to be picked up at location A and placed at location B on a flat surface. If the object is to be picked up from the top, we may need to insert intermediate positions, such as A' directly above A and B' directly above B, so that slower approach and depart trajectories from A to A' and B' to B may be used. Assuming that the robot starts at A', the desired sequence of operations is as follows.

1. Move down to A.
2. Close gripper slowly.
3. Move back to A' at slow speed.
4. Move to B' at high speed.
5. Move to B at slow speed.
6. Open gripper slowly.
7. Move to B' at slow speed.
8. Move to A' at high speed.

For performance of these movements, we must also specify the locations of the STEP primitive and the speeds for the delays. The position vectors are needed in terms of the number of steps required for each motor. This number of steps is directly related to the angle of motion required and may be called joint-step coordinates. Suppose the number of joint steps required to move from A' to A, A' to B', and B' to B are P, Q, and R, respectively, where

P = (P1,P2,P3,P4,P5,P6)
Q = (Q1,Q2,Q3,Q4,Q5,Q6)
R = (R1,R2,R3,R4,R5,R6)

Let the slow speed be specified by a delay S, the high speed by a delay H, and the gripping speed by a delay G. Finally, let the number of steps required to open the gripper be CG. The following ARMBASIC program would implement this pick-and-place task repeatedly.

```
10    STEP        S, -P1, -P2, -P3, -P4, -P5, -P6
20    CLOSE       G
30    STEP        S, P1, P2, P3, P4, P5, P6
40    STEP        H, Q1, Q2, Q3, Q4, Q5, Q6
50    STEP        S, R1, R2, R3, R4, R5, R6
60    STEP        G, 0, 0, 0, 0, 0, CG
70    STEP        S, -R1, -R2, -R3, -R4, -R5, -R6
80    STEP        H, -Q1, -Q2, -Q3, -Q4, -Q5, -Q6
90    GOTO        10
```

Upon executing this program, you must be ready to keep placing objects at position A. The manufacturer also provides numerous example programs to illustrate the use of the MiniMover robot. Other examples are described by Hemenway (1983).

This example has one basic problem from the user's viewpoint. For each location, the number of motor steps in the joint-step coordinates must be known. This requires some rather tedious calculations, especially for a large number of positions. Two solutions are offered for this problem.

The first is called teach-by-example programming. With this method, a human programmer interactively moves the robot to a desired position and pushes a button to record the joint angles or position. This procedure can be repeated for each desired position. The human programmer must also specify velocities and tool operations; however, the difficult calculation of joint angles or joint-step values can be done by the computer.

The second approach is to develop the transformations from a more natural coordinate system, such as Cartesian coordinates to the joint angle coordinates. Note that both the position (x,y,z) as well as the orientation angles of the gripper or tool are required to determine the joint angles. For the Microbot MiniMover, there are five joint angles. In general, six are required. Given the joint angles of the robot, it is relatively easy to determine the forward, kinematic problem solution. However, given the Cartesian coordinates, it is relatively difficult to determine the corresponding joint angles. This is called the inverse kinematic problem and solution. The inverse problem is of course solved for each robot in use.

The difference between teach-by-example programming and the coordinate transformation method is more fundamentally in the operation than in the mathematics. Let's now explore teach-by-example programming to illustrate the power of this approach.

Teach-by-Example Programming

Let's use the Cincinnati Milacron heavy-duty T3 (HT3) hydraulic-powered robot language for an example of a higher level language (Operating Manual for the Cincinnati Milacron T3 Industrial Robot). This language provides for teaching by example (level 2), as well as transformations, primitive motions, and external control, which are level 3 features (Holt, 1977). Certain conditions must be met to actually operate the robot. The power must be turned on for the controller and the hydraulics. Then, the operator may enter either a teach mode or an automatic mode. Safety is always the primary consideration. Although there are manual stops that restrict the robot's motions somewhat, the

robot, like any moving machinery, requires careful usage. In the teach mode, a teach pendant, such as that shown in Figure 4–7, permits the operator to position the robot, program a point, operate the grippers, and single step through a sequence of programmed points. The teach pendant has a large button for emergency stop. Depressing this button disables the hydraulic power, which stops the powered motion; however, the arm can still drift downward. The position buttons permit the operator to position the arm in the coordinate systems selected. Rectangular, cylindrical, or hand coordinates, as shown in Figure 4–8, may be selected. Note that the robot controller assists the operator by permitting the selection of different coordinate systems. The teach pendant also has three sets of buttons to control orientation in terms of yaw, pitch, and roll. Tool functions may also be selected for two possible tools. When these buttons are depressed, the tool status changes. If the tool is open, it closes, and vice versa. When the robot is positioned at the desired location, the program button is pressed, which stores the actuator locations in memory. Such functions as velocity may also be selected by depressing the function

Figure 4–7. Teach pendant for the Cincinnati Milacron T3 robot. (Courtesy of Cincinnati Milacron.)

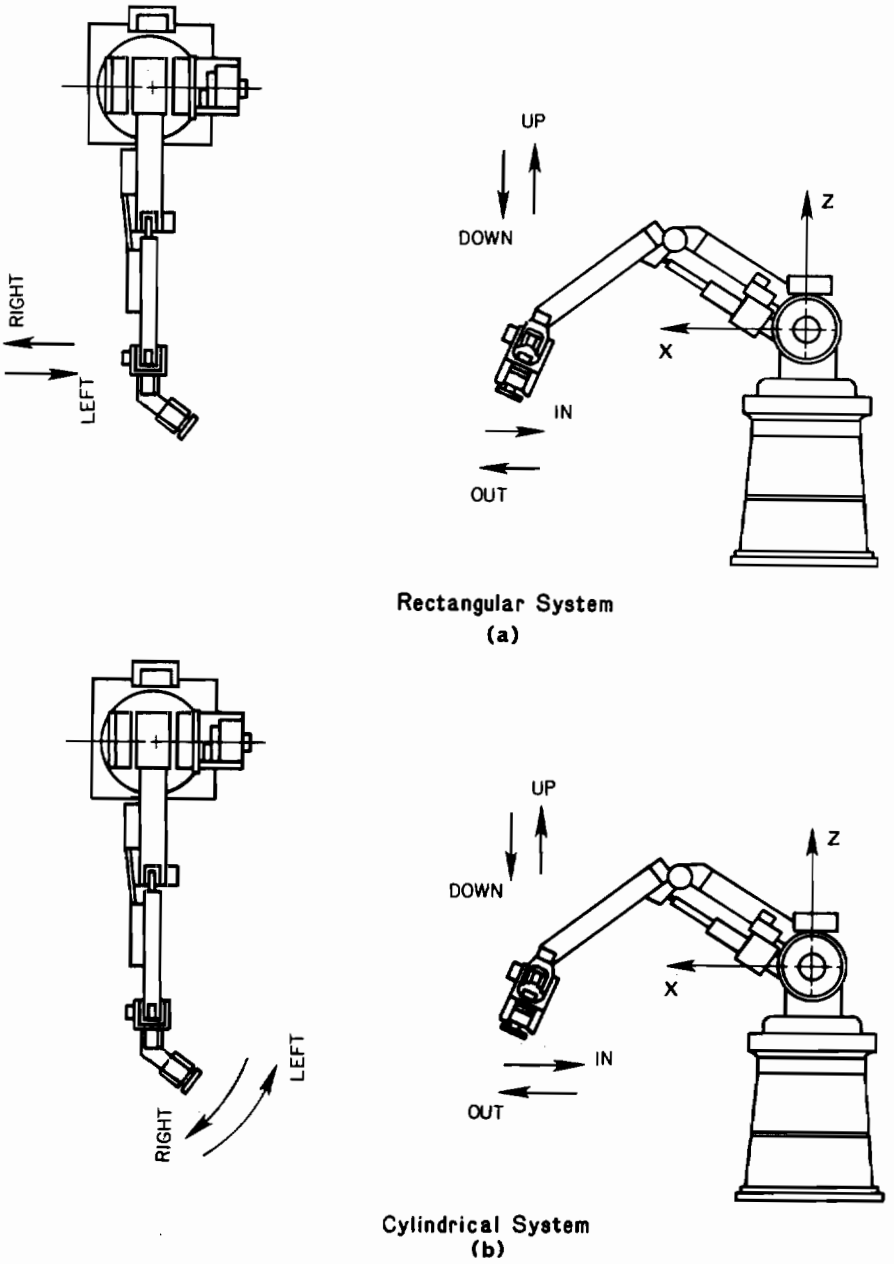


Figure 4–8. Teach coordinate systems available for the Cincinnati Milacron T3 robots. (a) Rectangular coordinates. (b) Cylindrical coordinates. (c) Hand coordinates. (Courtesy of Cincinnati Milacron.)

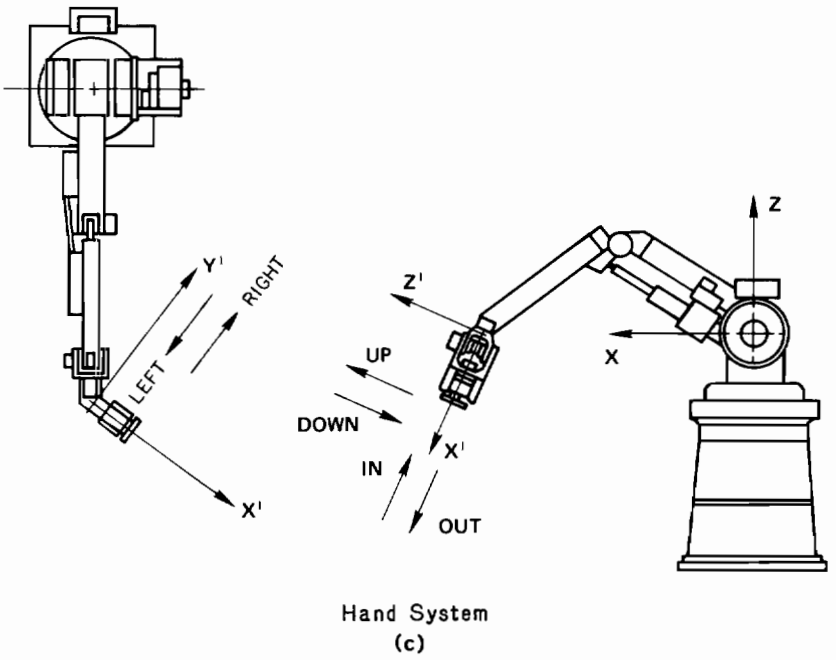


Figure 4–8 (continued)

button and entering the desired value. A cathode-ray tube (CRT) and keyboard are also available to monitor the operation and permit program entry or modification or activate automatic mode.

A program must start with the robot in the home position. The home position provides a common reference position on all the manipulator axes. A program is developed by using the teach pendant to move the robot to the desired position. Then a function, velocity, and tool dimension are selected. If no particular function is desired, a no operation NOP may be selected. A program button is then depressed to store the information in the robot's memory. A velocity, which may vary from 0 to the maximum velocity, may be selected. The tool dimension specifies a point a given distance in front center of the wrist faceplate, which provides an orientation reference. A program consists of a main line sequence, and up to 63 non-main line sequences. A maximum of 999 points is permitted in a sequence. A main line sequence is shown in Figure 4–9. Note that the home position is numbered as position 00.001. The first two digits refer to the sequence number, which is 00 for the main line sequence. The other three digits refer to the point in the sequence. After a sequence has been programmed, a close path operation is performed to a point in the main line sequence that has a NOP function assigned to it. This point will become the cycle start point. Also, the cycle start position is numbered 00.002. The cycle start must contain a NOP function.

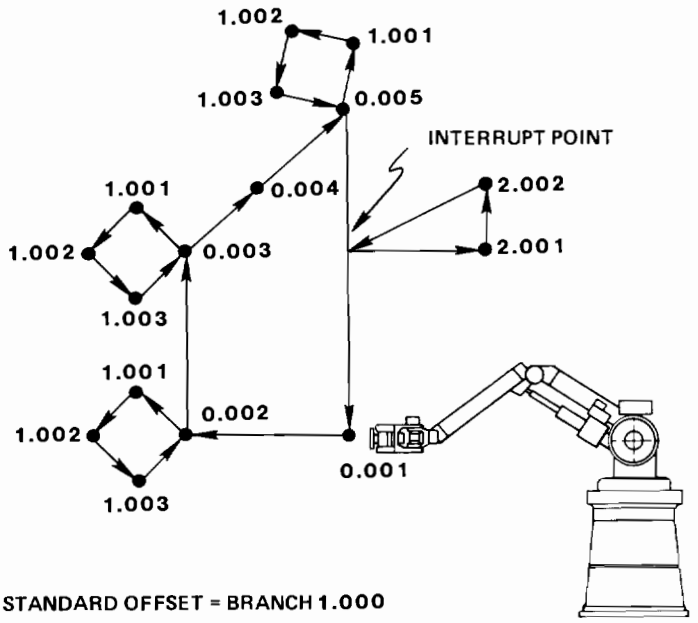


Figure 4–9. Diagram of the Cincinnati Milacron T3 program, showing main line and subroutine sequences. (Courtesy of Cincinnati Milacron.)

Let's now consider a more realistic example that uses signals. We will look at a diagram of the process shown in Figure 4–9. A program corresponding to this diagram is shown below.

```
SEQ. POINT   LOCATION              FUNCTION
0.001        1 or Home            NOP
0.002        2                    PERFORM,1,C ALL
0.003        3                    PERFORM,1,C ALL
0.004        4                    NOP
(Cycle start)
0.005        5                    PERFORM,1,C ALL
(Close path, 0.001)

(PERFORM 1)
1.001                             NOP
1.002                             NOP
1.003                             NOP
(PERFORM 2 on interrupt, close path subroutine, relocatable)
2.001                             NOP
2.002                             NOP
(Abort routine for main line program)
```

As illustrated, the main line program starts at statement 0.001, which is the home position. It then moves to position 2 and performs the subroutine sequence 1. It then moves to position 3 and again performs subroutine 1. It then moves to position 4 and does nothing. It then moves to position 5 and again performs subroutine 1. The interrupt subroutine 2 could be an abort routine, which is performed if something goes wrong during the program execution.

This simplified routine shows the main line and subroutine structure but is not directly executable since the velocity, tool dimension, and tool status are not given.

Let's now consider a more realistic example that has all the parameters specified and also uses input and output signals. The robot must pick up a part from the incoming conveyor, load it into the machine, and then unload it and place it on the outgoing conveyor. Signal 1 will indicate to the controller that a part is ready to be picked up. Signal 2 will indicate that the machine is ready. These are both input signals to the robot. The robot will send out signal 3 when the part is loaded into the machine so that it can start its operation. The program is shown below.

```
SEQ. POINT   LOCATION FUNCTION            VEL. DIM. STATUS
00.001       HOME     HOME                  1   1
00.002       1        NOP                  20   1
00.003       1        PERFORM,1,+S01        0   1
                      (CLOSE PATH,00.002)
01.001       1        TOOL,1                0   1      +N
01.002       2        TOOL,1               10   1      -N
01.003       4        NOP,CONTINUE         20   1
01.004       3        OUTPUT,+01            5   1
01.005       3        DELAY,5.0             0   1
01.006       3        TOOL,1                0   1      +N
01.007       4        WAIT,U,+S02          20   1
01.008       3        TOOL,1                5   1      -N
01.009       3        OUTPUT,+03            0   1
01.010       4        NOP,CONTINUE         20   1
01.011       5        TOOL,1               10   1      +N
                      (CLOSE PATH)SUBROUTINE
```

The program has a main sequence consisting of the first three statements and a subsequence consisting of the next eleven statements. At each statement, a function, velocity, and tool function may be specified. Let's examine the program in detail.

The first statement, 00.001, corresponds to the HOME position of the robot. The next statement sets the velocity at 1 inch/sec and the tool dimension at 1 inch. The tool dimension also determines a position with respect to the wrist faceplate in which straight-line controlled path motion takes place. That is, the points, which are the control points for interpolated motion, are specified by the tool dimension. The next statement instructs the robot to wait for signal 1 from the input conveyor before moving to sequence 1. Upon receiving signal 1, the robot performs sequence 1, which starts at

statement 01.001. At the start of the sequence the robot opens tool 1 but stays at the same position. At step 2, the robot moves to position 2 and closes the gripper on the part. At step 3, the robot moves in front of the machine. At step 4, the robot slows down and presents the part to the machine. Next, at step 5, the robot waits for the machine to grasp the part. At step 6, the gripper opens. At step 7, the robot waits, untimed, for the machine to finish. Next, the robot opens the gripper, enters the machine, and regrasps the part. At step 9, the robot signals the machine to release the part. Next, at step 10, the robot backs out of the machine. Finally, at step 11, the robot moves to the output conveyor and releases the part. The close path to position 2 returns control to statement 2 in the main line sequence, where the robot again waits for a part to be present. The sequence is then repeated continually.

This simple example illustrates the unique requirements of a robot programming language for a machine that can interact with other machines through input and output signals, open and close tools, and move to different points at different speeds. The T3 language also has provisions for external control from other computers, which permits off-line programming.

As another example, consider the following VAL program for a Unimate/Westinghouse robot (User's Manual to VAL). This sample program performs the following tasks. The robot first waits for a part to be in place in a feeder. It then picks up the part and carries it to an inspection station. At this position, it sends a signal to the inspection station that a part is in place. The station then determines whether the part is type A or type B. If the part is type A, one subroutine is performed. If the part is type B, another subroutine is performed. If the part is neither type A or B, a process reject is performed. The cycle repeats indefinitely.

The program is also set up to branch to an emergency subroutine if this condition is indicated by input signal 7, at any time from the start of the program until the IGNORE statement.

```
      REMARK Start of program
      REMARK
      REMARK Initialize signal line
      SIGNAL -2
      REMARK Make sure hand is initially open
      OPENI 100.00
      REMARK
10    REMARK Start of loop to process parts
      REMARK
      REMARK Start looking at emergency signal on input
             channel 7
      REACTI 7,EMERGENCY ALWAYS
      REMARK Wait for "part in place" signal on input
             channel 1
      WAIT 1
      REMARK Pick up part from feeder
      SPEED 200.00
```

```
      APPRO PART, 50.00
      MOVES PART
      CLOSEI 0.00
      DEPARTS 50.00
      REMARK Move to inspection station
      APPRO TEST, 75.00
      MOVES TEST
      REMARK Stop checking for emergency signal
      IGNORE 7 ALWAYS
      REMARK Signal that part is in place
      SIGNAL 2
      REMARK Wait for "inspection done" signal on input
             channel 6
      WAIT 6
      REMARK Withdraw from inspection station
      DEPART 100
      REMARK Reset "part in place" signal
      SIGNAL -2
      REMARK Test results of inspection; first for part
             "A"
      IFSIG -3,4,-5, THEN 20
      REMARK Then for part B
      IFSIG 3,-4,-5, THEN 30
      REMARK Part is neither "A" nor "B"--process
             reject
      GOSUB REJECT
      GOTO 40
20    REMARK Process part "A"
      GOSUB PART,A
      GOTO 40
30    REMARK Process part "B"
      GOSUB PART,B
40    REMARK Part processing completed, get another part
      GOTO 10
      REMARK End of program
```

This language appears similar to BASIC because of the REMARK, GOTO, and GOSUB statements. However, the robot primitives are embedded in the language. Let's also consider the operation of this program.

The first executable statement, SIGNAL, transmits a negative signal on channel 2. The next statement, OPENI, immediately opens the gripper 100 millimeters. The REACTI statement initiates continuous monitoring of signal 7. The next statement, WAIT, puts the program in a wait state until a signal is received on channel 1. The next statement, SPEED, requests that the next arm motion be performed at a speed twice as great as normal, which is speed 100 millimeters/second. The APPRO statement moves the end effector to the position specified by array PART, offset by a distance of 50 millimeters in the z direction. The CLOSEI positions the tool near the part. Next, a

MOVES command moves the tool to the position and orientation specified by the array PART. The CLOSEI closes the gripper. The DEPARTS statement moves the hand a distance of 50 millimeters along the current axis of rotation of the last joint. Since the distance is positive, the hand retracts. Next, the APPRO statement moves the hand to the position specified in array TEST, offset by a distance of 75 millimeters. The MOVES statement then moves the hand to the position and orientation specified in array TEST. The IGNORE statement stops testing for the emergency signal. A positive signal is then sent out on channel 2 to inform the inspection unit that a part is in place. A WAIT of 6 seconds is then initiated. The DEPART statement moves the hand back a distance of 100 millimeters. A negative signal is then sent out by the SIGNAL statement. A test is now made. The IFSIG command tests for a negative signal on channels 3 and 5 and a positive signal on channel 4. If these exact conditions are met, program control branches to statement 20. A similar test with different conditions is made and can send control to statement 30. Otherwise, the GOSUB statement branches to REJECT. At statements 20 and 30, various processing on parts A and B can be accomplished by the GOSUB statements. Also, even though the positions, such as PART and TEST, are arrays, the values in these arrays must be specified by teach programming or some other method.

Now let's look, as an example of a level 4 language, at an off-line program example using the AML language developed by IBM (Taylor, et al., 1982). Since AML is an off-line programing language, all positions and orientations must be specified in the program. Upon execution, a calibration procedure must be accomplished to ensure that program positions and physical positions correspond. The sample program moves to a position above a part, checks to see if the part is present with a sensor, grasps the part if it is present, and moves back. Since AML is structured, all variables are specified at the beginning of the program.

```
;GLOBAL DATA DEFINITION
PICKUP
  POINT=(10.000,15.000,10.000,90.00,180.00,90.00)
DISTANCE = 7.0
FUNCTION GETPART
GLOBAL PICKUP POINT, DISTANCE
BEGIN
       SPEED = FAST
;APPROACH SOME DISTANCE ABOVE THE POINT TO START
APPROACH DISTANCE FROM PICKUP POINT
;MOVE TO THE POINT--STOP AS SOON AS WE CAN SEE IT
  WITH SENSOR
MOVE PICKUP POINT UNLESS PART PRESENT = = ON
;CHECK IF IT IS PRESENT
IF PART PRESENT = = ON THEN
       BEGIN
;             PART WAS PRESENT, MOVE TO THE POINT
                WITH A SLOW SPEED
MOVE PICKUP POINT WITH SPEEDSCHED(1)
```

```
;GRASP THE PART
CLOSE
        END
ELSE
        BEGIN
;       PART WAS NOT PRESENT, PRINT ERROR MESSAGE
        WRITE(ERROR - PART WAS NOT PRESENT IN
          FIXTURE)
        END
;MOVE UP A SAFE DISTANCE FROM CURRENT POSITION
DEPART DISTANCE
END
```

The English-like nature of AML makes the program easy to read; therefore, it will not be described in detail.

Finally, let's consider another level 4, off-line programming example using the language AL developed at the Stanford Artificial Intelligence Laboratory (Mujtaba and Goldman, 1979). The following program segment was written to grasp the handle of a pencil sharpener and place it into an assembly fixture. The AL program statements are separated by ";" and surrounded by the reserved words BEGIN and END.

```
BEGIN
DEFINE ARM = "BARM"
DEFINE STOP ARM = "BARM"
DEFINE FING = "BHAND"
DEFINE HAND = "BHAND"
DEFINE PARK = "BPARK"
DEFINE INCH = "2.54*CM"
DEFINE INCHES = "INCH"
DEFINE OZ = "28*GM"
DEFINE / = "COMMENT"
DEFINE $ = "COMMENT"
DEFINE DIRECTLY = "WITH ARRIVAL=NILDEPROACH WITH DEPARTURE=NILDEPROACH";
DEFINE NO ARRIVAL = "WITH ARRIVAL = NILDEPROACH ";
DEFINE NO DEPARTURE="WITH DEPARTURE=NILDEPROACH ";
DISTANCE FRAME ORIGIN;
        ORIGIN =FRAME(NILROTN,VECTOR(0.0,40,.13) );
FRAME JIG BOTTOM;
        JIG BOTTOM = ORIGIN*TRANS(NILROTN,
                                  VECTOR(14.8,3.06,1.06));
DISTANCE VECTOR UP3;
        UP3 = 3*ZHAT;
(NOTE "BEGINNING");
/ ASSEMBLE HANDLE;
BEGIN
/ INITIALIZE POSITION OF ARM;
BARM == BPARK
MOVE BARM TO BPARK;
DISTANCE FRAME HANDLE REF, HANDLE GRASP;
        HANDLE REF = ORIGIN*TRANS(NILROTN,
          VECTOR(2.5,8,1.06) );
HANDLE GRASP = HANDLE REF*TRANS(YHAT
                  ROT-180,VECTOR(0.8,0,0));
AFFIX HANDLE GRASP TO HANDLE REF RIGIDLY;
```

```
OPEN HAND TO 0.6;
MOVE ARM TO HANDLE GRASP;
CENTER ARM;
WHILE HAND < 0.2
       DO BEGIN
              OPEN HAND TO 0.6;
              MOVE ARM TO .+ ZHAT*3;
              ABORT("I THINK HANDLE IS NOT IN THE
                RIGHT POSITION: PLEASE RECTIFY");
              MOVE ARM TO HANDLE GRASP DIRECTLY
              CENTER ARM;
       END;                  /HALLELUJAH, WE GOT THE
                               HANDLE;);
AFFIX HANDLE TO GRASP TO ARM RIGIDLY;
MOVE HANDLE REF TO JIG BOTOM*TRANS(ZHAT ROT
  -86,ZHAT*0.5);
/NOW DROP THE HANDLE;
/ OPEN HAND TO 0.5;
UNFIX HANDLE GRASP FROM ARM;
MOVE ARM TO .+ UP3;
CLOSE HAND TO 0.0;
/NOW SIT ON THE HANDLE;
MOVE ARM TO .-UP3
       ON FORCE(ZHAT) > 12 DO STOP ARM;
ARM == HANDLE GRASP+0.3*ZHAT;
MOVE ARM TO .+0.2*YHAT
/ARM HAS PUT HANDLE FLUSH ON THE FIXTURE
$ WE CAN NOW DESTROY ALL AFFIXMENTS TO HANDLE
  SINCE WE DON'T NEED IT ANY MORE;
END;
```

These examples illustrate the various types and levels of programs used in robot programming. The need for an industry standard is clear, but one has not been achieved. Fortunately, the basis of a new language may be mastered in a relatively short time, especially if one already has a solid background in programming.

Off-Line Programming

Off-line programming is the process of programming the robot on an off-line computer. An illustration of this process using a McDonnell Douglas Automation Company, McAuto robotics simulation system is shown in Figure 4–10. This concept is important, since it ties together the concepts of computer-aided design with the concept of computer-aided manufacturing. The design process and the manufacturing processes can be integrated, simulated, tested, and verified before actually setting up the manufacturing process. To accomplish this integration will require advances in all the areas of intelligent robot programming discussed in this chapter.

Questions

1. Compare the features of the on-line programming languages, T3 and VAL, with the off-line languages, AML and AL. Describe the features present in each class and the advantages and disadvantages of each.

Figure 4–10. An off-line programming example. (Courtesy of Cincinnati Milacron.)

2. Compare the features of interpretative and compiled robot programming languages. List the advantages and disadvantages of each.
3. Describe how learning by experience may be implemented in an intelligent robot system (*hint:* pattern recognition and tree searches).
4. Discuss the direct application of optimal linear control theory for robot control.
5. Discuss the problems in robot control for the four situations:

   a. Robot path not specified and no obstacles
   b. Robot path specified and no obstacles
   c. Robot path specified but obstacles in path
   d. Robot path not specified but obstacles in path

Which problem is most commonly encountered by humans; by industrial robots? Which problem is most difficult?