Improvement of a Human/Robot Interface Through the Use of FORTH

Doug Thompson

International Robomation/Intelligence
Carlsbad, California

Abstract

International Robomation Intelligence (IRI) manufactures a computer controlled 5 axis robot arm for industrial uses such as machine loading unloading and parts transfer. IRI updated its computer operating system from the present assembly code version to FORTH. This paper describes the usage of FORTH for man machine interfaces. User programming is accomplished via IRI's simple and straight forward on-line Robot Command Language (RCL). Special emphasis will be placed on how FORTH has been adapted to the production environment where the end user may have little or no programming experience. The approach taken emphasizes at all times the relationship between human and robot rather than the physical control of the robot itself.

International Robomation Intelligence (IRI) is a California based company designing and manufacturing robot automation and artificial intelligence products. IRI has recently decided to use FORTH Inc.'s polyFORTH II in two of its products.

The IRI M-50 is a computer controlled 5 axis (with provision for six) robot arm for industrial uses such as machine loading unloading and parts transfer. Some of its unusual features are:

1. aerospace-derived construction techniques
2. air-servo motors
3. a compact computer control network onboard the robot

Typical uses:

Current major industry groups are:

1. Aircraft and suppliers
2. Auto Truck industry and suppliers
3. Electrical Electronic products and components
4. Farm Construction equipment and suppliers

Applications include:

1. Materials Handling Palletizing
2. Machine Loading
3. Die Casting
4. Investment Casting
5. Stamping Press Loading
6. Forge and Heat Treating
7. Injection molding
8. Cylindrical Grinding

All of the M-50's control electronics are on a single "monoboard" which is a 6-layer printed circuit board housing 8 computers. It also contains two RS232 ports, a manual move pendant connection and up to 16 digital input and 16 digital output lines.

The M-50 is controlled by a hierarchy of micro-computers with the Motorola 68000 16 32 bit processor as the manager. The 68000 has up to 128K bytes of RAM and 64K bytes of EPROM. Each axis has its own control computer and an additional CPU furnishes safety control. The robot may be

*Received July 1983.

programmed on-line in IRI's Robot Command Language (RCL) currently written in assembly language, but in the process of conversion to FORTH. Refer to Figure 1.

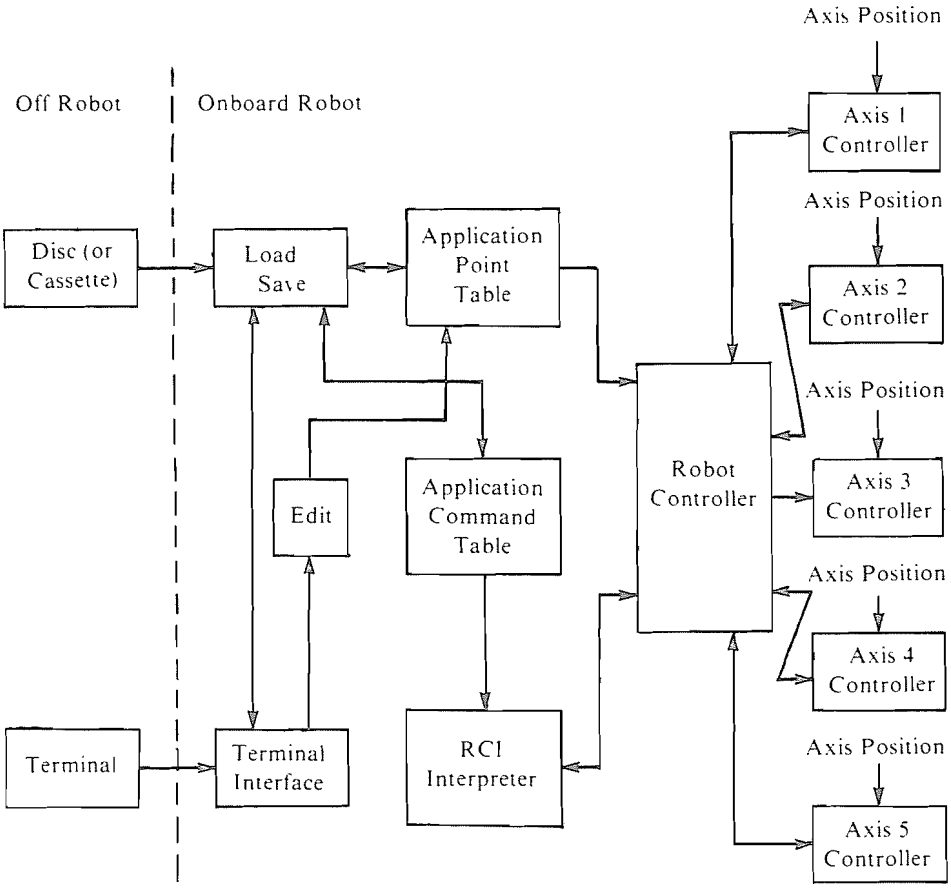

Figure 1. System Block Diagram

The users of the RCL will have diverse programming backgrounds. Based on current users, and our market in general, we expect that about 60% of future users will have no previous computer programming experience, 35% with some, and only 5% or less with broader experience.

The goal of a project currently underway at IRI is to improve the existing RCL interface. FORTH is an excellent language for the new RCL, and we believe that it will play a significant role in helping us to achieve our goal. This paper illustrates the rationale behind our choice to use FORTH.

Defining a Better RCL

The existing RCL is an interpretive programming language which allows simple command entry. Commands can be entered sequentially in program form or directly for immediate action. RCL and all other programming languages should have certain characteristics that make them desirable. T. W. Pratt [1] lists six characteristics that he feels are desirable. Pratt's list has been paraphrased below in order of relative importance from the viewpoint of the author.

1. Efficiency: The language should be designed to minimize the total time all users must spend to complete an application.
2. Clarity, simplicity, and the unity concept: The language should be clear, simple and follow a logical set of rules.

3. Debug and support facilities: The language should include built in debugging features and have a good supporting editor to help minimize development time.
4. Ease of extension: The language should be easily expandable to handle future needs.
5. Clarity of program strucure: The language should have structures which help create understandability in a program.
6. Naturalness for the application: The language should be designed to fit the level of the user.

In order to improve the existing RCL, we had two options. The first was to continue using the assembly language. The second was to use a higher level language. Of the higher level languages we considered two groups: Pascal-like languages, and threaded code languages. FORTH, because of its wide acceptance, was the only threaded code language considered. Languages used by other robot manufacturers range from assembly language for Unimation's VAL, FORTRAN for SRI International's RPL, to Pascal for General Elecric's HELP and PL1 for IBM's MAPLE [2]. The factors for our decision are compared in Figure 2.

	ASSEMBLER	PASCAL-like	FORTH
Natural to use? Clear?	Not natural.	Some English like constructs.	Reverse polish not natural.
Debugging facilities	Low level monitor. Changes made by patching code. Very error prone.	Modular testing and print statements. May have trace. Slow testing.	Word by word testing. Interpreter allows quick high level changes. Fast testing.
Support facilities	Off-line development.	Off-line development.	Resident compiler and interpreter.
Extensibility?	Very little to build on, hard to expand.	Good base, moderately difficult to expand.	Excellent base allows easy expansion.
Structured?	NO	YES	YES
Original application	General purpose.	Designed as good language to teach programming.	Designed for process control.
Size of memory usage	Small	Large if standard functions used.	Small
Ease of assembly linkage	Not applicable.	Not straightforward.	Quick and easy.

Figure 2. Comparison of Implementation Languages Considered

Our first option, to continue using assembly language was rejected on the basis of the above comparison. An additional factor was that the old RCL completely filled the available 64K of EPROM, mainly because of the large real-time operating system used. In order to add new functions one of two things had to take place: either change to 128K EPROMS, or restructure the code to use memory more efficiently. 128K EPROMS are still fairly expensive and adding new code to a system that was becoming less unified did not seem prudent. Because of the long development time and the lack of structure in assembly language, it was rejected.

Having ruled out assembly language, we were left with a choice between Pascal-like languages or FORTH. Pascal-like languages need large support systems such as the compiler, thus necessitating off-line programming. Off-line programming, in turn, causes increased development time. These languages were meant to be applied to a wide variety of uses. Because of this, much of its capability would not be used (such as the file system), and would be carried along as extra overhead. With memory a concern, these languages became inappropriate.

The selection of FORTH was not merely a process of elimination; it was chosen on its own merits. FORTH is simple, consistent, modular and thus well unified. Because of its interpreter, development time is shortened. A major advantage is that debug and development can be done directly on the robot, independent of any off-line support facilities. It is low in memory usage due to its modularity and reuse of code.

Now let's take a closer look at the first 6 characteristics of good programming languages and see where FORTH is beneficial, specifically for RCL.

1. Efficiency: Efficient use of resources is of prime concern to any profit oriented company. The major objective of the robot is to decrease production costs. The longer it takes to program the robot for a task, the less time the robot has to do productive work. This makes the user interface very important to the user. FORTH, because of its interpreter, gives the user almost instantaneous feedback. The ease of error detection implementation afforded by FORTH allows quick and efficient guidance of the user to program correctness. With the onboard compiler and interpreter, FORTH allows us to do our development on any robot. Resources are distributed rather than dependent upon one large off-line development system. Memory usage is also more efficient due to the high degree of modularity, reuse of code, and its threaded code design structure.

2. Clarity, simplicity and the unity concept: It was important for us to make the old RCL as simple for the user as possible, yet powerful enough to efficiently handle the application. The assembly code to implement this, unfortunately, was not so simple or clear. Most of the problems occured in the real-time operating system interface. The interconnections of tasks became less and less well defined as the system expanded. FORTH provides a very basic operating system that meets our needs. Task interaction is greatly simplified. Because FORTH streamlines the underlying structure of the RCL, the RCL itself becomes easier to implement. The emphasis can be placed on the clarity and use of the RCL commands rather than how they are to be implemented.

3. Debug and support facilities: The old RCL had good debugging features which allowed single stepping through program execution, breakpoints, and tracing during execution. These functions are being carried over into the FORTH version. To handle these functions, a new interpreter had to be developed. To trace each command we could have used Tom Asprey's [3] method of embedding a DEBUG command or by using a switch tested inside NEXT. This was ruled out because only the RCL commands were to be printed as found in the user's source code. This was resolved by feeding the FORTH interpreter one command at a time from the user's source program, keeping a pointer to the next source command.

This works very nicely and simply until structured control constructs are added. The FORTH interpreter will not accept the constructs such as BEGIN...UNTIL, so these constructs must be interpreted before handing down the proper command to "EXECUTE." As suggested by Peter Helmers [4] a user stack was set up to handle return positions and loop indexes just as the FORTH return stack does. These return positions point to the user's source code.

Error detection was easy to add by including a word into each RCL command which checks for the proper number of arguments on the stack and ensures that they are within the proper range. Undefined words entered by the user are rejected by the editor through the use of " - ' " . Control structures are checked during run time for proper matching order.

Because of FORTH's high degree of modularity, many of the FORTH interpreter's functions were available for use in the new interpreter, thus minimizing new code.

4. Ease of extension: The old RCL, as previously stated, became heavily enmeshed with the real-time operating system. It was designed for a multi-user disk based environment, instead of an on-line control system. Modularity in code was stressed, but exceptions occurred. The interdependent code made expansions more complex due to unanticipated side effects. I O was not general enough to allow easy communication to other devices. The RCL was also limited in that the user did not have an easy method of passing parameters to subroutines. FORTH's emphasis on modularity allows expansion to take place

at a small cost in memory usage. The new RCL allows parameter passing to subroutines via the stack. The simple and direct I O of FORTH will allow additions of any conceivable device pertinent to robotics.

5. Clarity of program structure: The old RCL allowed the GOTO, conditional GOTO, and subroutine call (without parameter passing). All labels were numeric instead of the more readable alphanumeric format. This led to code which was sometimes hard to follow. The GOTO statement has been the subject of much debate among even the structured language advocates during the last several years. For those experienced with sructured programming, the GOTO becomes unnecessary in all but a few exceptional cases. Here, the assumption that nearly 60% of our users have no programming experience has prompted us to retain a GOTO construct. For those users, the GOTO is easier to understand than BEGIN...WHILE...REPEAT. The GOTO can be appropriate for simple applications, and more cost effective for the user. Productivity and cost effectiveness are the primary goals.

Labels will be alphanumeric (like any other FORTH word) to enhance understandability. Structured constructs (once understood) will generally be far superior for productivity, because of their cohesiveness and better understandability. The new RCL will also include a higher level set of control structures similar to FORTH's.

FORTH's constructs have been criticized by many people because of the pre-test features that are unlike the English language. As Howard Goodell states, "(condition) IF (action) THEN is an insult to everyone's syntactic preconceptions, and one which takes a long time to overcome." [5] Again, we must focus on who the user is and what his needs are. Less than 10% of our users may have some experience with a structured programming language. With no prior experience, either design would have to be learned, therefore we believe FORTH's pre-test structure will not be significantly more difficult to grasp than post-test structures. The RCL control structures are listed in Figure 3.

< condition > IF < action > (ELSE < action >) THEN

< end > < start > DO < action > (LEAVE) < action > (LEAVE) < action > LOOP

BEGIN < action > (< condition > WHILE) < action >
(< condition > WHILE) < action > < condition > UNTIL

ENTRY name

SUB name

RETURN

Figure 3. RCL Control Structures

FORTH's control structures are good, but we will implement them with two changes we feel will benefit our users. The first deals with the classic debate on whether the DO loop should test the exit condition before entering the loop for the first time. We believe that it is inconsistent to force execution once, regardless of limit conditions. Our RCL DO loop will test the condition prior to loop entry simply by decrementing the starting index by 1 and jumping to the LOOP in the source code. +LOOP will not be implemented because at the level of our users, it would not be used often enough.

The second change is in the BEGIN...UNTIL loop. Here we will combine this with the BEGIN...WHILE...REPEAT loop to produce one structure. Its syntax is BEGIN...WHILE... WHILE...UNTIL. Any number of WHILEs are allowed so the loop can be exited at the beginning, middle, and or end. We believe the simplicity and consistency of having the one structure rather than two similar structures will benefit our user.

Subroutine capability will be handled through the word SUB followed by a name. The defining word SUB puts the name in the dictionary as a CONSTANT with the value of the source position directly following the name. When the name is used within the program, its run-time behavior is to save the current source position on the user stack and use the CONSTANT value as the new source position.

Execution continues at the new position. Any number of parameters can be passed on the parameter stack, just as with FORTH. The RETURN command simply takes the position off the stack and uses it as the new source position.

Entry points are used to designate where program execution should initially begin and to aid in debugging small sections of code at a time. The word ENTRY works identically to SUB except its run-time behavior first clears the user stack and then sets up the new source position without leaving a return position on the stack. A GOTO can be simulated simply by using the name of an entry point in the program.

Compilation of variables, constants, and entry points is done when the user exits the editor. This means that all entry points are compiled before execution. Therefore, forward referencing is allowed.

6. Naturalness for the application: We forsee three levels of users of our products. To respond to these needs we anticipate offering three levels of software capability.

Level 1 would be for those users who have no desire to expand the current commands of RCL other than through subroutines. This would pertain to almost all M-50 users. Disk usage is limited to LOAD and SAVE of the user program. Block structure is completely transparent to the user.

Level 2 would add the capability of the : definition. This would allow the user to add to the list of RCL commands. Editing of FORTH programs would be allowed on the disk as well as the SAVE and LOAD RCL usage.

Level 3 would allow the user complete access to the FORTH system. All source code would be available, hence the user could target compile a new system. Only the most sophisticated user would desire this level.

FORTH allows these divisions to be easily made through the use of sealed vocabularies and headerless definitions. By providing different levels of software, we anticipate our product will be useful in a wider range of applications.

RCL Example Usage

Lathe Part Loading Example With Dual Gripper

This describes a simple application of the IRI M50 using a double gripper. The double gripper is shown in Figure 4. The grippers are labeled A and B. This design is used so that a part may be removed with one gripper, move the other gripper into place with a new part and load it. This cuts the load unload time in approximately half. The program below uses this gripper to load and unload the chucker of a lathe. Seven points in space are defined. An Idle point (point #1) is described as away from the lathe and waiting for a signal from the lathe to move. Chuck-A position (point #2) is the point next to the chuck using gripper A. Chuck-B position (point #3) is the point next to the chuck using gripper B. Drop-off (point #4) is the point where finished parts are placed. Pick-up (point #5) is the point where new parts are to be picked up before machining. Away (point #6) is the point out from the chucker and far enough away to rotate the gripper so that the other gripper can move into the chucker. Sleep (point #7) is the position the robot assumes during the off shift times.

Figure 4. IRI D1 Gripper.

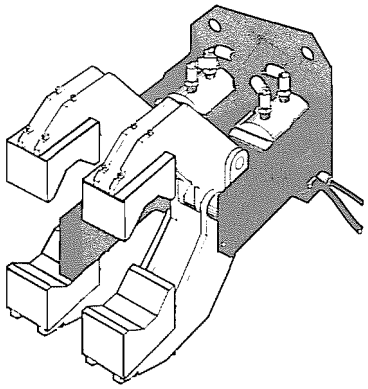

Figure 5 shows the points in relation to the robot and the lathe. Figure 6 shows the robot in action.

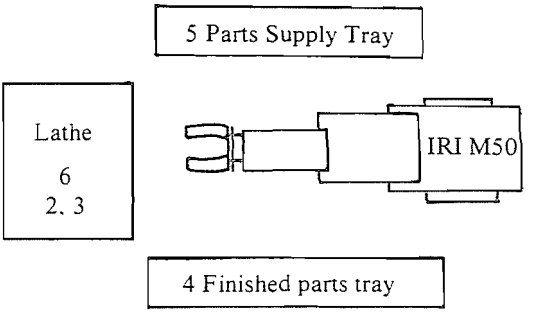

Figure 5. Station Layout

(Numbers in the drawings refer to point numbers in the program.)

For purposes of clarity and simplicity, the example program uses a small subset of the available RCL commands. This subset is defined below.

COMMAND EXPLANATIONS (n=integer; b= boolean)

CON	(--)	= constant definition (value and name follow)
VAR	(--)	= variable definition (name follows)
[	(--)	= comment (end with])
MOVER	(n--)	= move all robot axes to point # n
DELAY	(n--)	= delay for n milliseconds
OPEN	(--0)	= 0
CLOSE	(--1)	= 1
GRIP-A	(b--)	= send signal b to gripper A
GRIP-B	(b--)	= send signal b to gripper B
-OUT	(n--)	= reset digital output n
+OUT	(n--)	= set digital output n
IN	(n--b)	= read in digital input n
PDATE	(--)	= prints current date
PTIME	(--)	= prints current time
P"	(--)	= prints user message (end with ")
P#	(n--)	= prints value n
INCR	(--)	= increments by 1 the variable that follows
TO	(n--)	= indicates n to be stored in variable following

The following sample program consists of 2 main parts: part 1 contains constants and one variable with meaningful names to enhance program readability (lines 1 through 14); part 2 contains the program body (lines 16 through 23) and a subroutine (lines 24 through 45). Execution will begin at START and execute through line 23 once a day. Subroutine TASK does all the work of moving the robot during the work hours, and "parks" the robot after the day is over. Execution is started by typing "START RUN".

```
 1  CON 1 Idle                    [Idle wait position]
 2  CON 2 Chuck-A                 [Chuck position gripper A]
 3  CON 3 Chuck-B                 [Chuck position gripper B]
 4  CON 4 Drop-off                [Part drop-off point]
 5  CON 5 Pick-up                 [Part pick-up point]
 6  CON 6 Away                    [Approach point to chuck]
 7  CON 7 Sleep                   [Shift completed position]
 8  CON 15 #lbs                   [Weight of part to pick up]
 9  CON 1 Ready?                  [Ready switch for lathe]
10  CON 2 More?                   [Switch indicating more parts]
11  CON 3 Day-done?               [Switch indicating shift over]
12  CON 4 Lathe                   [Switch to turn on lathe]
13  CON 5 Open-chk                [Switch open and close lathe chuck]
14  VAR #parts                    [Number of parts machined]
15
16  ENTRY START                   [Main entry]
17    BEGIN Day-done? NOT UNTIL   [Wait for new day]
18    0 TO #parts                 [Initialize # of parts]
19    TASK                        [Execute TASK subroutine]
20    PDATE PTIME                 [Print date and time]
21    #parts P#                   [Print number of parts]
22    P" parts machined. " LINE   [ machined today]
23    GOTO START                  [Repeat]
24
25  SUB TASK
26    BEGIN
27      Idle MOVER                [Move to idle position]
28      BEGIN More? IN UNTIL      [Wait for next part]
29      Pick-up MOVER             [Move to pick-up point]
30      #lbs CLOSE GRIP-A         [Grab it]
31      BEGIN Ready? IN NOT       [Wait at idle position]
32        WHILE Idle MOVER 0 UNTIL  [ until lathe ready]
33      Chuck-B MOVER             [Move to chuck with grip B]
34      #lbs CLOSE GRIP-B         [Grab machined part]
35      Open-chk +OUT  500 DELAY  [Open chuck and wait]
36      Away MOVER  Chuck-A MOVER [Move away and back with A]
37      Open-chk -OUT  500 DELAY  [Close chuck and wait]
38      OPEN GRIP-A               [Release grasp]
39      Drop-off MOVER            [Move to drop off part]
40      Lathe +OUT                [Start lathe for new part]
41      OPEN GRIP-B               [Drop off machined part]
42      INCR #parts               [Add 1 to # of parts done]
43      Day-done? UNTIL           [Continue until shift done]
44  Sleep MOVER                   [Move to rest position]
45  RETURN                        [Return to caller]
```

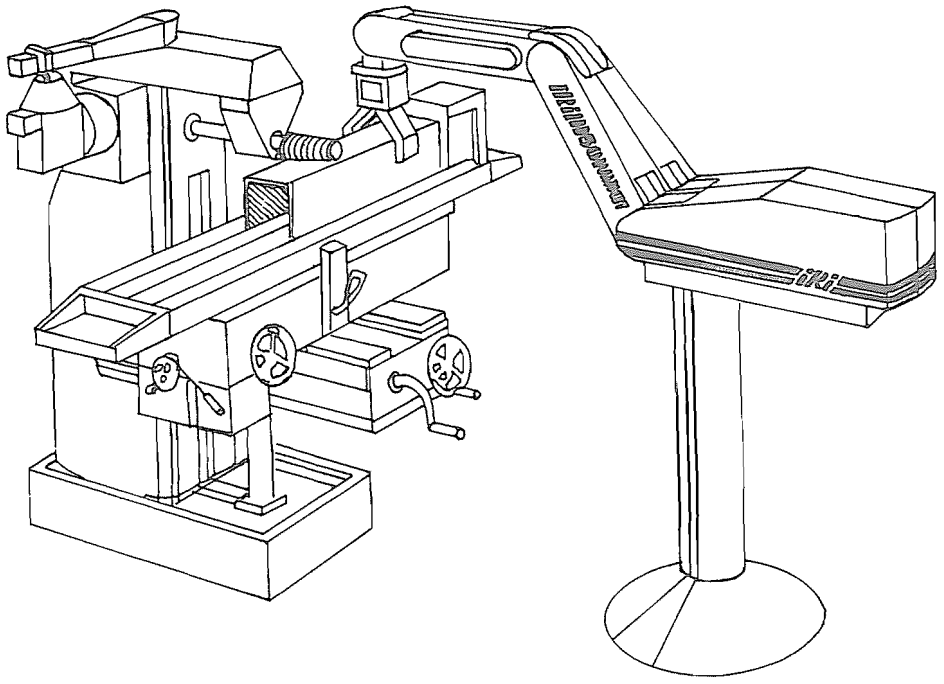

Figure 6. The IRI M50 Robot in action.

Conclusion

The goal of this project was to improve the existing RCL. Particular needs were ease of extensibility, program structure, unity, and debug enhancement. These are best attainable through the use of FORTH because of FORTH's high degree of modularity, small size, interactive capability, and the ease of program structure implementation. Problems such as reverse polish notation will have a minimal impact on our user because of his lack of traditional programming experience. Though the project is not yet complete, it is far enough along to appreciate the power of FORTH's simplicity.

References

1) T. W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, N.J., 1975.

2) Susan Bonner and Kang G. Shin, "A Comparative Study of Robot Languages", *COMPUTER*, Dec. 1982, pp. 82-96.

3) Tom Asprey, "A FORTH Execution Simulator for Debugging", *Proceedings: 1980 FORML Conference*, Asilomar, CA., Nov. 26-28, 1980, pp. 181-186.

4) Peter H. Helmers, "Userstack", *FORTH Dimensions*, Vol. III, No. 1, pp. 20-22.

5) Howard Goodell, "Comprehensive Control Structures", *Proceedings: 1981 FORML Conference*, Asilomar, CA., Nov. 25-27, 1981, Vol. I, pp. 259.

Mr. Thompson received a BA in computer science from the University of California, San Diego in 1979 and did graduate work in artificial intelligence at the University of California, Berkley in 1982. He is interested in robotics and artificial intelligence, and aspects of work and energy conservation. Currently he is a software engineer developing a robotics operating system and user interfaces.

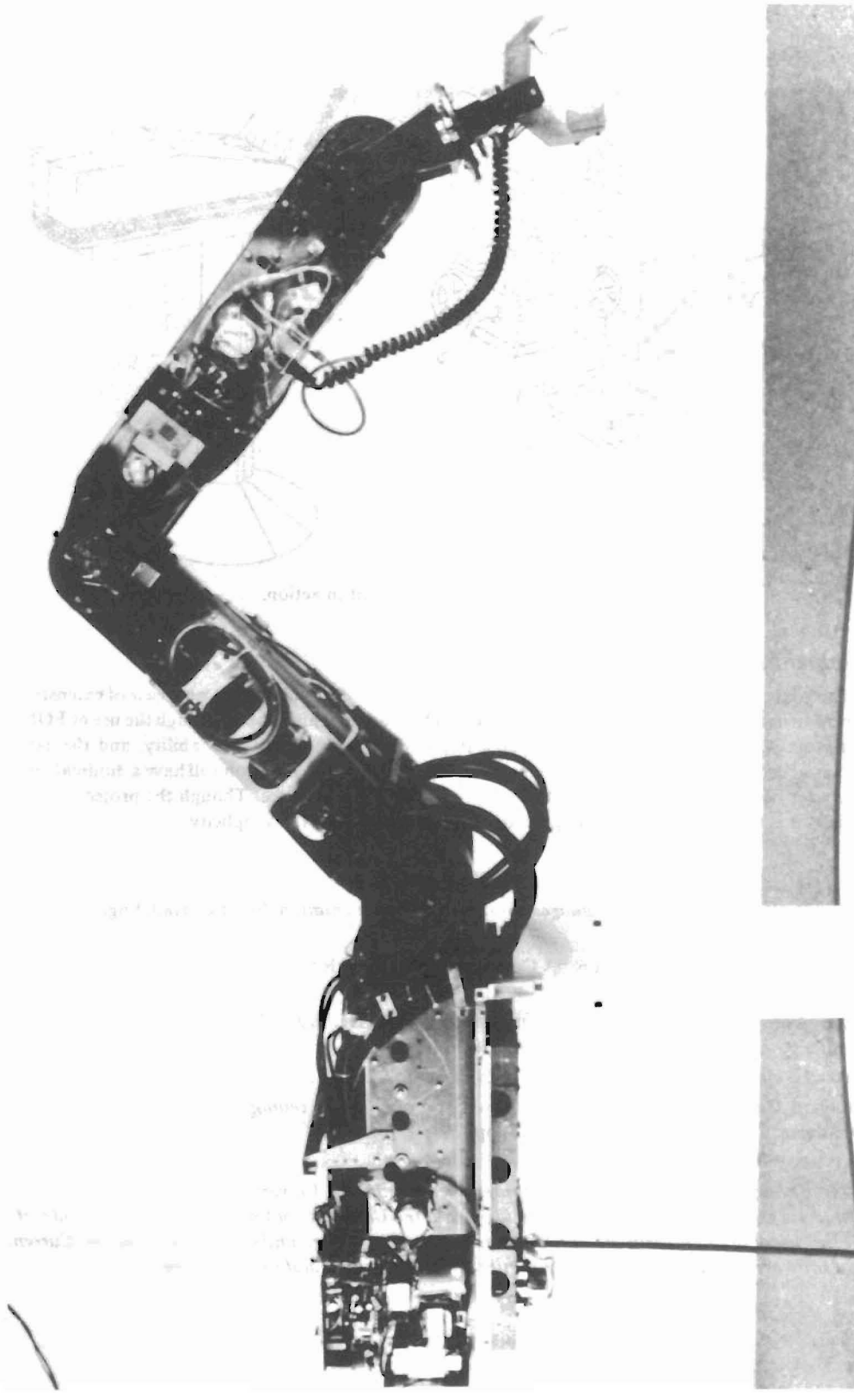

Figure 7 The M50 Robot Arm.