

SCELBAL - A HIGHER LEVEL LANGUAGE FOR 8008/8080 SYSTEMS

TABLE OF CONTENTS

	Introduction
Chapter ONE	Interpreter Versus Compiler
Chapter TWO	The Fundamental Capabilities of SCELBAL
Chapter THREE	Fundamental Operation of SCELBAL
Chapter FOUR	The Executive
Chapter FIVE	The Main Syntax Routine
Chapter SIX	Statement Interpretation
Chapter SEVEN	Evaluating Mathematical Expressions
Chapter EIGHT	The Parser Routine
Chapter NINE	Function and Optional Array Handling Routines
Chapter TEN	Mathematical Routines
Chapter ELEVEN	I/O Routines
Chapter TWELVE	SCELBAL Assembled for Operation on an 8008 System
Chapter THIRTEEN	SCELBAL Assembled for Operation on an 8080 System
Chapter FOURTEEN	Operating SCELBAL
Chapter FIFTEEN	Suggestions for Program Tinkerers
Appendix I	SCELBAL Labels Reference List

SCELBAL - A HIGHER LEVEL LANGUAGE FOR 8008/8080 SYSTEMS

BY

Mark Arnold
and
Nat Wadsworth

© Copyright 1976
SCELBI COMPUTER CONSULTING, INC.
1322 Rear - Boston Post Road
Milford, CT. 06460
203-874 1573
- ALL RIGHTS RESERVED -

IMPORTANT NOTICE

Other than using the information detailed herein on the purchaser's individual computer system, no part of this publication may be reproduced, transmitted, stored in a retrieval system, or otherwise duplicated in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior express written consent of the copyright owner.

The information in this publication has been carefully reviewed and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies or for the success or failure of various applications to which the information herein might be applied.

The authors wish to thank the following members of the staff
at Scelbi Computer Consulting, Inc., for their dedicated assis-
tance in the preparation of this publication:

Robert Findley
Raymond Edwards
Ms. Gabrielle Tingley

INTRODUCTION

In the early 1970's technology produced the integrated circuit microprocessor. The advent of this device offered the promise of making low cost computing elements available to the general public at large and raised the hopes of many citizens that the power of the computer could finally be accessed by individuals of limited means. This promise was most exciting for in the past the use of computers had been fairly limited, for economic reasons, to institutions that could afford the use of their incredible power.

For the first several years after their introduction, microprocessors remained primarily in the domain of highly educated scientist and engineers who were backed by organizations equipped to exploit the device's capabilities in a variety of fields. Gradually, however, as knowledge spread, their capabilities became known to the general public. People, many of them electronic enthusiasts and hobbyists, wanting to harness the power of these devices for personal use began to clamor for low cost computing systems. The old laws of supply and demand came into effect. Within a short time span, a number of small corporations began to offer the hardware for small personalized systems. Initially, only individuals with appropriate technical backgrounds were able to capitalize on the availability of these low cost systems and put them to effective use. Some people, enthralled by the exciting potential of such systems, had some rude awakenings. For, while the microprocessor is touted as being able to do any and everything, it turns out that these little devices are virtually worthless without SOFTWARE or PROGRAMS that can direct their activities. The development of useful software using early machine language techniques is no trivial task. It takes a considerable amount of individual effort to get to the point where one can program a computer using the most fundamental programming method, which is machine or assembly language programming. These programming methods require an intimate knowledge of

the detailed operation of a computer on a step-by-step basis. The development of even seemingly simple tasks using these programming methods can take an inordinate amount of time. This is particularly so if one is not skilled in the art and practice.

The limitations of machine language programming have been known for many years since the beginnings of computer technology some 30 years ago. Over the years a number of HIGHER LEVEL LANGUAGES have been developed so that people other than computer experts could work effectively with computers. Higher level language programs are actually programs written in machine or assembler language by skilled personnel that will in turn allow other people to communicate with the computer using simple commands and statements. The degree of programming efficiency that may be achieved using a higher level language is many orders of magnitude over that required to perform the same tasks using the fundamental machine language programming methods. For instance, a simple directive such as:

LET X = (Y + 145*Z) ↑ (2*N - M)

might require several THOUSAND individual machine language instructions to achieve a general solution capability. A person who had many such equations to solve would soon opt to forget the use of a computer if such a task had to be performed for each variation of similar problems. It may be apparent, however, that such equations, while individually different in detail, consist of similar operations (such as multiply, add, raise to a power and so forth). A higher level language is designed to take advantage of such similarities in a generalized fashion.

On the other hand, while a higher level language yields such tremendous increases in programming efficiency, this increase is not achieved without sacrifice! It takes many

thousands of man hours to develop such a generalized higher level language, and this investment in labor must be made each time such a language is created. It is not always easy to get a group of people together and make the type of investment necessary to initially develop such a language. Additionally, the individual user who desires to install such a language on a computer, must pay for the increased programming efficiency by budgeting a significant amount of the available memory in the computer for the exclusive use of the operating portion of the higher level language program. What is left over may then be used to hold the user's program (in the higher level language form) along with any data that is to be manipulated or processed. For the small system user, the "significant" amount of memory set aside for the operating portion of the higher level language, for the program described herein, will be some six to seven thousand bytes of memory. This is indeed a good chunk of memory for the system owner who has but 8 K of that precious commodity!

The individual user must also sacrifice certain aspects of a computer's capability when utilizing a higher level language. For instance, it is virtually impossible to program real-time routines whose precise execution times can be controlled when using the higher level syntax. This is because the higher level syntax does not give the programmer access to individual machine language instructions. Additionally, many types of instructions available in machine language (for instance, rotating a register to the right or to the left) have no direct counter-part in the higher level language. (However, the student of this publication will be in a position to incorporate subroutines that can be accessed by higher level language programs and can thus enjoy the benefits of both types of programming!)

Despite the relatively large memory requirements of a high level language, and the other types of limitations mentioned, it is felt that the time has arrived when such a

language would be welcomed by small systems owners when presented in the detailed manner of this publication.

The higher level language to be presented in this publication has been given the acronym SCELBAL. This stands for SCientific ELe-mentary BAsic Language. It has been patterned after a commonly used higher level language referred to as BASIC.

SCELBAL was specifically developed to be able to run on systems using the ubiquitous 8008 CPU. This CPU is generally acknowledged as being the first true 8-bit CPU to be manufactured on an integrated circuit. It was first developed by a California based firm, Intel Corporation. SCELBAL is believed to be the first such higher level language to be specifically developed to run on the 8008 CPU and be made generally available to the public. The program described herein can also be run on systems using the more powerful 8080 CPU though it is not as memory efficient as it could have been if the program had forsaken 8008 capability.

While this publication was specifically prepared to demonstrate the details of the language as developed for 8008/8080 machines, the publication should be of considerable interest to users of other types of similar computing devices. Indeed, the experienced programmer, armed with the knowledge presented in this book, should be in a pretty good position to implement a similar language on just about any other microprocessor by simply translating the machine code instructions to those of the machine of particular interest to the user. (While such a project might seem monumental to some, the information in this book would make the task considerably less difficult than approaching such a task without the practical, detailed information which is presented herein!)

The major objectives of this publication are to:

- 1.) Present a higher level language that can

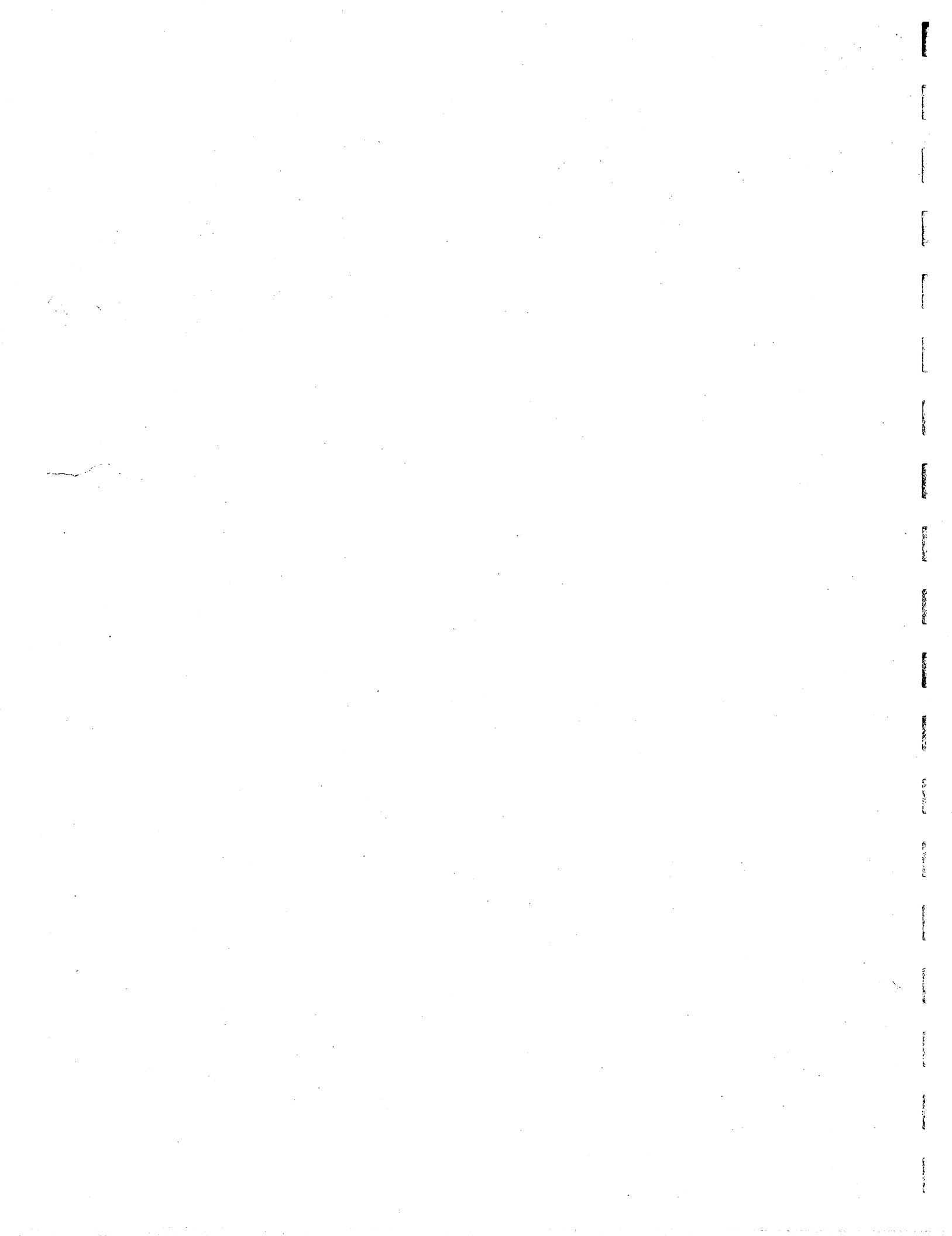
be implemented on 8008/8080 microprocessor systems with the user having the freedom to adapt the package to various individual I/O configurations.

2.) Present the intimate details of its operation so that it may be readily modified and adapted to individual user's applications and requirements.

3.) Serve as an educational and stimulative tool for the future development of similar languages, possibly of a more advanced nature.

Much thought in the preparation of the overall program went into just what capabilities to provide given the various technical trade-offs that one must consider. It

was known at the start that the program could not be developed to satisfy every potential user. Nobody has a system with that much memory available! Care was taken to provide a good fundamental selection of syntax statements and functions in the language. From that point, backed by the descriptions of the program's organization, general flow charts, and highly commented listings provided in this publication, it is felt that the user will be equipped to add extended capabilities depending on memory available, or willingness to sacrifice described functions. For many users, it is felt that the program as presented, will be entirely satisfactory. The extra measure of providing the information so that the user may go further if desired, is the fundamental premise behind this publication.



INTERPRETER VERSUS COMPILER

SCELBAL was developed as an INTERPRETIVE language, not a compiler. Some readers might be asking, "What's the difference?"

There is a lot of difference. An interpretive language is one that essentially processes each line or statement in the source code of the higher level syntax and then executes the directive before going on to the next line or statement. It does this by calling on machine language routines that perform the various functions as soon as it has been determined which job is to be accomplished. A compiler operates quite differently. Each time it processes a statement in the higher level language syntax it PRODUCES some machine language coding that can later be executed to perform the desired task.

From this brief introduction it may be apparent that there are some major organizational differences between the two types of higher level language processors. The key ingredient is that the INTERPRETER immediately interprets and executes. The compiler COMPILES, that is it produces machine code, and the machine code it produces is executed at a later stage.

What does this mean from an organizational and systems view point? Perhaps the best way to obtain the overall view is to present the typical practical operation of both types of systems.

COMPILER OPERATION

The general sequence of operations to get a program written in a higher level language into actual operation using a compiler oriented language is as follows.

First, a program written in the higher level language syntax is prepared. This might be done using an Editor program

on the computer. Note that if such is the case, that first an Editor program must be loaded into the computer's memory and the computer system used for editing purposes. When the high level language source listing has been prepared, it must usually be saved or stored on some external medium such as punched paper tape or magnetic tape.

Next the COMPILE portion of the higher level compiler program would be loaded into the system's memory and the original source listing of the high level language program processed. Generally this procedure requires several passes or readings of the source listing. The final result of this operation is the production of machine language code, which once again would usually have to be stored on some sort of external medium.

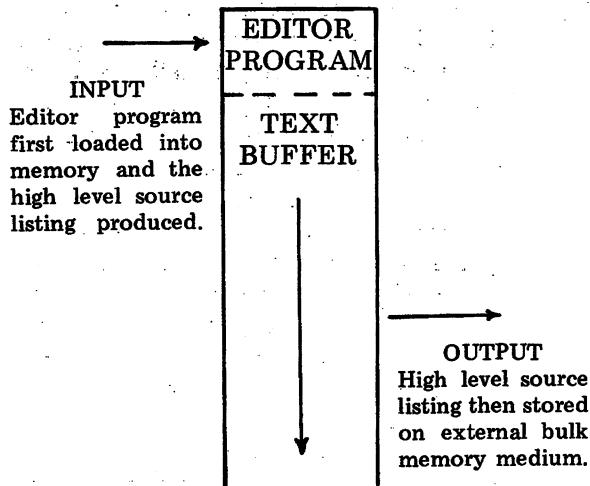
Finally, the RUN or EXECUTE portion of the compiler program would be loaded into the system's memory along with the machine language code that was produced previously by the COMPILE portion of the compiler. At this point, the user's program, originally written in the higher level syntax, would be ready to operate, having been converted to machine code.

The first two stages of a compiler oriented language can be considered as analogous to the sequence of operations necessary to create a program using an Editor and Assembler. The only difference being that the source listing when using an assembler would consist of the machine language mnemonics, while when using a compiler it would consist of the higher level language syntax.

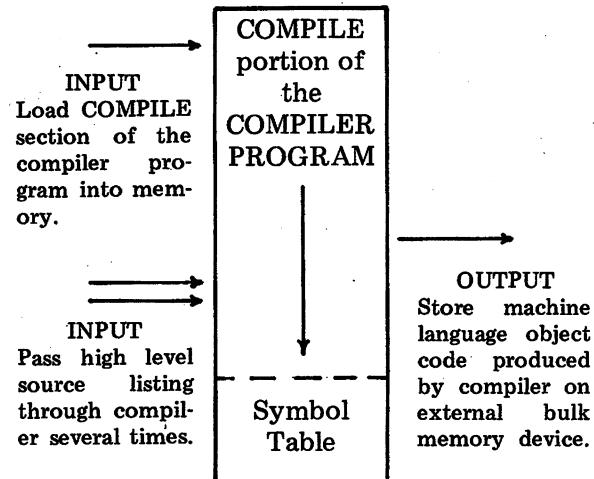
The final stage of a compiler oriented language is generally not quite the same as would be the case if machine code was produced by an assembler. This is because

the run or execute portion of the compiler typically provides some control over the compiled program by the operator. Additionally, this portion of the compiler program has a number of routines that the program that has been compiled is able to utilize, such as, a floating point arithmetic package. At this point, when the RUN portion of the compiler along with the machine code produced by the COMPILE portion are both residing in memory, the user is finally able to execute the original program that was written using the higher level syntax.

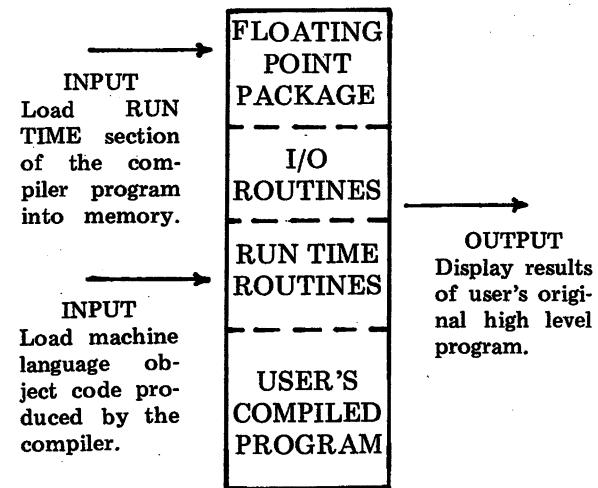
It may now be apparent that a compiler oriented language is highly dependent on the host system having fast and reliable I/O capability with an external bulk memory device. This is because of the constant need to input the various sections of the compiler program and output the intermediate information during the program development process. This requirement for the constant use of an external memory medium may be observed more clearly by reviewing the development process for a higher level language, going from the creation of the high level source listing to final execution of the high level program, as illustrated in the following diagrams.



COMPILER OPERATION - STEP 1



COMPILER OPERATION - STEP 2



COMPILER OPERATION - STEP 3

The fact that a compiler oriented version of a high level language is so dependent on I/O operations with an external bulk device is the primary reason that SCELBAL was not developed as a compiler. Most small system owners must be satisfied with either paper tape or audio cassette magnetic tape bulk storage devices. Both of these types of

peripherals are relatively slow in operation and not as reliable as commercially oriented magnetic tape systems. For convenient compiler operations a system really needs a disc peripheral unit that will allow the rapid loading of programs and storage of intermediate data (such as the object code produced during the second step of compiler operations discussed above). It could take as much time as an hour or more to attempt to compile a higher level language program on a small system equipped with slow peripherals. The task of operating a compiler would quickly become quite frustrating if the programmer was a novice and frequently made programming errors in the source syntax. Remember, for the system just described, that if a program error was not detected until compiler RUN TIME, the user would have to go all the way back to the first step of loading an Editor program back into the computer and correcting the source listing of the high level language program!

As a matter of interest, if a compiler is so much trouble to use, what good is a higher level language that utilizes the method? Well, first of all, a compiler is not so difficult to use if one has a computer system equipped with a disc or other high speed memory peripherals. With such equipment it takes just a few seconds to load in a program or save the results of intermediate operations. Remember, the choice was made to not use the compiler method for SCELBAL based on the consideration that most small system owners could not afford the luxury of such speedy peripherals. There are, of course, institutions and organizations that do have such capabilities. For them, a compiler oriented system can have advantages.

A few advantages of using a compiler are as follows.

As a general rule of thumb, a compiler program can be created to operate in less actual read and write memory in the computer than an interpretive version. This is almost self-evident from the presentation of the information that a compiler is generally

split into several portions, the COMPILE part, and the RUN or EXECUTE portion. Thus, had SCELBAL been developed as a compiler it might have been possible to provide the same capabilities (from the final results view point of having a program executed that was originally written in a higher level syntax) with a program that only required, say, 4 K of RAM memory in the computer at any one time.

Second, the final operating version of the higher level program will generally function at a considerably faster speed than the same program executed in an interpretive fashion. This too is easy to see since one now knows that the interpreter must examine and interpret each statement as it goes along, whereas the compiled version had already accomplished that task when it produced the machine code that will result in the desired functions being performed at program execution time. This final speed of the program may be important when massive amounts of calculations are being performed, or in real-time situations. It is not likely to be that critical when a small system (that is probably severely restricted by I/O timing considerations) is being utilized.

Third, in line with what has already been mentioned about a compiler oriented program requiring less actual memory in the computer, the final machine code version of the program that has been compiled will generally be much more efficient memory usage-wise. This again is pretty much self-evident when one considers that the compiled program will only have machine language routines that perform the specific functions asked for in the actual program that was compiled. The interpretive package, on the other hand, must have all the possible functions for the language available in memory, since it is not known which functions may be utilized by a particular program.

In summary, it might be stated that a compiler becomes much more attractive when viewed in the context of larger computing systems with high speed peripherals available.

From the microprocessor view point, compiler oriented higher level languages, implemented on larger machines, are quite valuable if one is interested in developing a relatively large number of programs that will operate in microprocessor systems when they are part of a product. For instance, a manufacturer that desired to produce a line of test instruments, each of which would utilize a microprocessor, but with a special software package for each type of instrument, would be well off to use a compiler to create the programs. Compilers operating on microprocessor systems themselves, however, for the reasons indicated, are simply not practical for most small system users.

INTERPRETER OPERATION

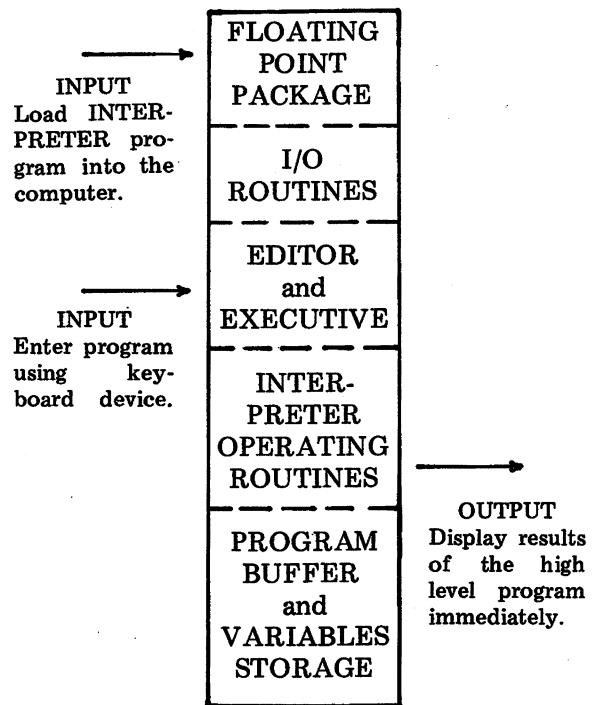
An interpretive version of a higher level language, while not as memory efficient as a compiler, is much convenient for the small systems user. In the context of being able to prepare and execute many different kinds of programs in a short time span, it is much more efficient in terms of overall program development to execution time. This is particularly true for inexperienced programmers as they can almost instantaneously be notified of syntax errors and immediately make corrections to the program being created on an on-line, real-time basis.

An interpreter differs from a compiler, as mentioned previously, in the fact that each line of the source syntax is interpreted and then executed before going on to the next line. The execution is performed by calling on various routines provided as part of the interpreter package. There is no production of intermediate machine code as in the sense of the compiler (though there may be the production of intermediate data, symbols, etc.).

An interpreter such as SCELBAL has everything required to create and execute a program residing in memory at one time. Thus, once the interpreter program itself has been loaded into memory, there is no need to use external bulk memory devices (unless one

wants to save a higher level program, or restore one previously saved on such an external memory storage device). This eliminates all the critical bulk memory operations necessary for the successful development of such programs when using a compiler.

The following diagram illustrates a memory map view of a typical interpreter program.



INTERPRETER OPERATION

The diagram above illustrates that the interpretive oriented program really consists of an Editor program (to enter and edit the high level syntax into a program buffer), an Executive (to direct the operation of the various portions of the package as directed by the user), and an Interpretive/Operating section that is able to analyze the contents of the program buffer and call on the desired routines as indicated by the statements it interprets.

With this type of arrangement one can typi-

cally create and execute higher level language programs in seconds or minutes versus an hour or two.

Thus, SCELBAL was developed to operate

as an INTERPRETER. The details of its operation will be presented in this manual. To find out the fundamental capabilites of SCELBAL just continue reading into the next chapter.

THE FUNDAMENTAL CAPABILITIES OF SCELBAL

As explained in Chapter One, SCELBAL was developed to operate in an INTERPRETIVE mode. This means that the entire program resides in memory at one time along with the program written in the higher level language that is to be executed. When the INTERPRETER is given the RUN command it immediately proceeds to INTERPRET each line of the higher level language program and perform the necessary calculations and functions.

SCELBAL has actually been designed so that it may operate in a "calculator" mode or operate in a stored program mode. In the calculator mode, each statement is executed immediately after it is entered on the input device. In this mode, the program is ideal for solving simple formulas when the user only needs to obtain a few values.

For instance, if one typed in the statement:

```
PRINT 2*2 + 3*3 +4*4
```

the value:

29

would be displayed as soon as the end of line code (carriage-return) was issued at the end of the PRINT statement.

One may use the calculator mode to solve more complex problems. For instance, if one entered a series of statements such as:

```
LET A = 2  
LET B = 3  
LET C = 4
```

and then entered:

```
PRINT A*A + B*B + C*C
```

the answer:

29

would immediately be displayed. This is because, in the calculator mode, the values assigned to A, B and C would be immediately assigned and available for use in solving the formula given in the PRINT statement above.

When it is not desired to operate in the calculator mode, but rather in a stored program mode, the user simply inserts a line number in front of each statement. A whole series of statements may then be arranged to form a program. When it is desired to execute the steps in the program, a special executive RUN command is issued. This command will cause the INTERPRETER to proceed to execute the program one statement at a time.

SCELBAL is able to handle actual numeric values using a floating point package which is an integral part of the interpreter. While a floating point package is used to perform all calculations, inputs and outputs to the program may be in fixed format within certain ranges.

When inputting information or specifying values within a program, the user may use fixed point notation for numbers in the range plus or minus 0.999999 to 999999. Numbers smaller or greater than this must be stated in floating point format, such as:

+0.123456E-10

or

-654321E+12

The minimum and maximum powers that

the floating point package used in SCELBAL can handle is ten to the plus or minus thirty-eighth.

SCELBAL automatically outputs numbers in the range plus or minus 1.0 to approximately 9999999 in fixed point format. Outside this range, output automatically switches to floating point notation.

The floating point package itself provides SCELBAL with the four most fundamental arithmetic capabilities. They are addition, subtraction, multiplication and division. All calculations in the floating point package are maintained to twenty-three significant binary bits in the mantissa, with the multiplication and division routines providing binary rounding when calculations yield numbers that exceed twenty-three binary bits.

While the floating point package provides the essential capability to handle the operators: +, -, * (multiply) and / (divide), the language, using supplementary routines, can also recognize the operators \uparrow (raise to a power), and parenthesis "(" and ")" which may be used to group or nest mathematical statements.

Up to twenty user defined variables are permitted at one time when using the language. However, in order to conserve memory space, variables must be limited to a maximum of two characters. Variables must begin with a letter of the alphabet.

The Executive portion of SCELBAL allows the user to control the overall operation of the program from an I/O device such as a keyboard and teleprinter. The user can create a program in the higher level language and have it executed using the features of the Executive portion of the program. A portion of the Executive is actually a small Editor program that allows the user to "edit" the information (program) in the program buffer at any time. Lines may be deleted and new lines entered. Clerical errors on a line may be corrected. Furthermore, a portion of the

Executive checks for various types of syntax errors as each line is entered. If an error is detected, an error code message is presented to the operator. This feature is extremely valuable for novice programmers, (and though some of them might not admit it, is quite comforting to the old professionals as well).

The Executive portion of SCELBAL has five major commands available to the operator which are defined and explained briefly below.

SCR is used to indicate the SCRATCH command. This command effectively clears out any previous program stored in the program buffer along with any previous user defined variables. It is used in preparation for entering a new high level program into the program storage area.

The LIST command does just that! It causes the contents of the program buffer to be displayed or "listed" on the system's output device so that it may be reviewed by the operator.

RUN directs the interpreter to begin operations and execute the program stored in the program buffer.

SAVE. This command may be used to direct the program to save a copy of the program stored in the program buffer on the system's external bulk storage device. A program saved using this command can later be restored for further use by using the command presented next.

LOAD. This command directs the program to read in a copy of a program from an external bulk storage device (previously written thereon using the above SAVE command) into the program buffer so that it may be executed by the interpreter.

The higher level language SCELBAL consists of STATEMENTS that are interpreted by the program resulting in selected operations being performed. SCELBAL recognizes the

following types of statements.

The REM for REMarks statement indicates a comment which is to be ignored as far as the interpreter is concerned. Information on a line prefaced by a REM statement is intended only for the use of programmers and may be used to document a program.

The LET statement is used to set a variable equal to a numerical value, another variable, or an expression. For instance, the statement:

LET X = (Y*Y + 2*Y - 5)*(Z + 3)

would mean that the variable X was to be given the value of the expression on the right hand side of the equal sign.

Since the LET statement is such a frequently used directive, SCELBAL also recognizes an implied LET statement. Thus, the simple statement:

X = (Y*Y + 2*Y - 5)*(Z + 3)

would be interpreted as though the LET directive had been stated.

The IF combined with the THEN statement allows the higher level program to make decisions. SCELBAL will allow one or two conditions to be expressed in an IF....THEN statement. Thus, the statement:

IF X = Y THEN LL

would be interpreted to mean that if, and only if, X is equal to Y, then the program would branch to line number LL in the program.

While the directive:

IF X <= Y THEN LL

would mean that if X was less than OR equal to Y (two conditions), that the program was to go to line number LL.

Similarly, the statement:

IF X <> Y THEN LL

would mean that if X was less than OR greater than Y that the program was to branch (again two conditions).

If the condition(s) in an IF...THEN statement are not met, then the program continues by going directly to the next sequential statement in the program and does not execute the branch directive.

The GOTO statement directs the program to effectively JUMP to a specified line number in a program. The GOTO statement may be used to skip over a block of instructions in a multiple segment or subroutines program.

The FOR, NEXT and STEP statements provide capability for the programmer to form program loops. For example, the series of statements:

FOR X = 1 TO 10
LET Z = X*X + 2*X + 5
NEXT X

would result in Z being calculated for all the integer values of X from 1 to 10. While SCELBAL does not require the insertion of a STEP directive in a FOR - NEXT loop, a STEP value may be defined if desired. The implied STEP value if not specifically stated is always 1. However, it may be set to a value other than 1 by following the FOR range statement by a STEP directive that dictates the desired STEP size. Thus, the statement line:

FOR X = 1 TO 10 STEP 2

would result in X assuming values of 1, 3, 5, 7 and 9 as the FOR - NEXT loop was traversed.

GOSUB is a statement that is used to direct the program to perform another statement or group of statements as a subroutine. The statement is used in conjunction with a line number which design-

nates where subroutine execution is to begin.

A RETURN statement is used to indicate the end of a subroutine. When a RETURN statement is encountered, the program will return to the next statement immediately following the GOSUB directive which was used to call the subroutine.

SCELBAL permits multiple nesting of subroutines (up to eight levels) within a program.

INPUT is used to direct the interpreter to wait for an operator to INPUT information to the program. After the information has been received operation of the program automatically continues.

The PRINT statement is used to output information from a program. By using the PRINT statement the user may direct the program to display the values of variables, expressions, or other types of information such as messages. The PRINT statement in SCELBAL permits mixed types of output on the same line (numerical values and alphanumeric messages), and the option of providing a carriage-return and line-feed after outputting information or the suppression of that function. For instance, the statement:

PRINT 'X IS EQUAL TO: ' ; X

would result in the program first printing the text message "X IS EQUAL TO:" and then the value of the variable X on the same line. After the value of the variable X had been displayed a carriage-return and line-feed combination would be issued. To suppress the issuing of the CR & LF function in the above example, the programmer would only need to include another semicolon at the end of the statement!

The PRINT statement is augmented by several functions and features. For instance, a comma sign in a PRINT statement may be used to cause the display device to space over to the next TAB position before continuing

to output more data. A special TAB function that will be discussed later may also be used with the PRINT statement to format the outputting of data. And, another special function which will be presented shortly will provide capability for SCELBAL to convert decimal numbers (representing ASCII codes) into alphanumeric characters for display.

The END statement is used to designate the conclusion of a higher level program in the program buffer. When this statement is interpreted control will return to the Executive portion of SCELBAL.

There is an optional statement available in SCELBAL that may be added to the package if the user desires to utilize the capability and has sufficient memory to adequately support the statement. This is the DIM for DIMension statement. It is used to specify the formation of a one dimensional array in a program. Up to four such arrays having a total of up to 64 entries are permitted in a program when the optional feature is included in the user's version of SCELBAL. Thus, when a user elects to provide the capability, the statement:

DIM K(20)

would set up space for an array containing 20 entries. (The array size must be specified using a numerical value, not a variable.)

The power of SCELBAL is further enhanced by the addition of seven functions that may be used within statements. These functions are discussed below.

INT returns the INTeger value of the expression, variable or number requested as the argument. The integer value is defined as the greatest integer number less than or equal to the argument. Thus, a statement which contained:

INT(X)

would result in the value, for instance, 5.0 being returned if X at the time the func-

tion was encountered was greater than or equal to 5.0 but less than 6.0 (such as 5.0001, 5.54321, 5.99999).

SGN returns the SiGN of the variable, number, or expression. If the value is greater than zero, the value +1.0 is returned. If the value is less than zero the value -1.0 is returned. The value 0 is returned when the expression or variable is zero.

ABS returns the ABSolute value (magnitude without regard to sign) of the variable or expression identified as the argument of the function.

SQR returns the SQuare Root of the expression, variable, or number.

RND produces a semi-psuedo-RaNDom number in the range of 0 to 0.99. This function is particularly useful to have available for games programs or when it is desired to have random values when doing statistical analysis problems. The random number generated may be operated on to produce random numbers within a desired range. For instance, the statement:

```
LET X = RND(0)*10
```

would result in X being assigned values in the range of 0 to 9.99.

CHR is a special CHaRacter function. It may be used in a PRINT statement and will cause the ASCII character corresponding to the decimal value of the argument to be displayed. Thus, if:

```
CHR(193)
```

was contained in a PRINT statement, the letter A would be displayed. The argument portion of the CHR function may be a user defined variable so that different characters would be displayed depending on the value of the variable at the time the PRINT statement was executed.

A reverse function is available for use in

an INPUT statement. This function is specified by placing a dollar sign (\$) immediately after a variable in an INPUT statement. This function will cause the decimal value for the ASCII code of the letter that is inputted to be returned to the program. Thus, if an INPUT statement contained the directive:

```
INPUT A$
```

and the operator entered the letter Y as an input to the program, the value 217 would be returned as the value for the variable A. This function is valuable in a number of applications. For instance, if the programmer desired to have a user answer a question in a program with a yes or no response, the function enables the higher level program to ascertain which response was entered by testing the decimal value received.

A TAB function is available for use in a PRINT statement. This function allows the programmer to direct the display device to space over to the column number specified as the argument of the function. This function thus allows the programmer to format the output into neat columns. Thus, the statement:

```
PRINT X;TAB(10);Y;TAB(20);Z
```

would result in the value for X being displayed starting at column 1, the value Y starting at column 10, and the value of Z starting at column 20.

SCELBAL is designed to run in a system having a minimum of 8 K of read and write memory. In an 8 K system, the program, leaving out the optional DIMension (single dimension array) capability, provides about 1,250 bytes of memory for storage of the users higher level language program. While it is possible to include the DIMension capability in an 8 K system, doing so would reduce the program storage area in about half. One nice feature about SCELBAL is that the user with more than 8 K of memory can use the additional memory for higher level pro-

gram storage. A user with, for instance, a 12 K system, may configure the package so that there are about 5,000 bytes of memory available for storage of a program. It is recommended that those desiring to include the DIMension capability of SCELBAL have 9 or 10 K of memory in the system so that the program storage area will not be prohibitively small. The package has been arranged so that those that desire the DIMension option can install this section in the upper portion of available memory. Those that do not desire this feature, may leave it out to provide additional program storage room.

Even with just an 8 K system, surprisingly complex programs can be executed. A game such as Lunar Landing is easily accommodated if one reduces the number and lengths of the messages issued to the player. An 8 K system will be adequate for many users who are primarily interested in using the package as a sophisticated programmable calculator.

A 12 K system will support quite sophisticated programs with plenty of alphanumeric messages. With approximately 5 K bytes of memory available for program storage in such a system, the user would have the capability to execute programs that contained several hundred statements.

While most 8008 based systems are limited to a maximum of 16 K of memory, those utilizing the 8080 version of SCELBAL could conceivably have a program storage area (in a 64 K system) in excess of 56 thousand bytes. The kinds of programs one could run in that amount of memory could fill books alone!

The execution speed of SCELBAL, while slow compared to higher level languages that are designed to run on large computers, is

surprisingly good. The 8008 version is, of course, about an order of magnitude slower than the 8080 version due to the relative speeds of the two types of CPUs. The execution speed of an 8008 version can be almost doubled if one installs an 8008-1 CPU in their system. Some users may want to consider that option. However, even on an 8008 based unit, the execution speed of SCELBAL is quite tolerable. For instance, the typical response time between the displaying of a new set of parameters when running a Lunar Landing game is in the order of six to seven seconds. A program that calculates the mortgage payments on a house on a monthly basis and displays such data as the payment number and current balance after each payment requires but a few seconds between the displaying of each new line of data. A dice playing game responds with new throws of the dice in the order of a second or so when using a formula that includes the use of the random number generating function. These times are by no means fast but they are in the general range that one might obtain when solving formulas of similar complexity on commonly used programmable hand held calculators. Remember, these times are for the slowest 8008 version. They are lowered by an order of magnitude on an 8080 based system.

The information presented in this chapter is merely to whet the reader's appetite and present an overall picture of the fundamental capabilities of SCELBAL. The detailed use of the language will be presented in a later chapter along with numerous actual programming examples. It is now time to start learning how SCELBAL is organized as an overall package and then proceed to discuss the various portions of the program in detail. This coverage starts with the next chapter.

FUNDAMENTAL OPERATION OF SCELBAL

The following brief description provides a summary of the manner in which SCELBAL proceeds to process a higher level program. It should help the reader who needs some confidence building before digging into a software package that may initially seem complex due to the large number of individual machine language instructions that make up the overall package. The reader will hopefully soon see that all the individual machine language instructions are organized into relatively small routines and these in turn are carefully organized into a surprisingly simple scheme. The essential concepts of this simple scheme are presented in this section.

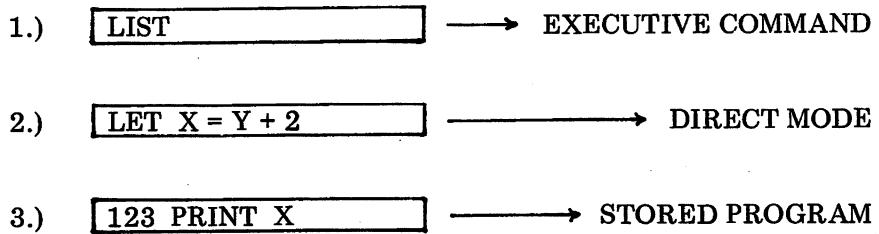
SCELBAL, as discussed in the opening chapter, is an interpretive language. The program simply operates by analyzing each line of source coding which the operator inputs in the defined higher level language format using the defined syntax. As the program analyzes each portion of a line, it performs the operations indicated.

Virtually all of the analysis of a line of source coding is accomplished when the information is residing in a temporary storage buffer in memory called the LINE INPUT

BUFFER. This LINE INPUT BUFFER is used to initially store data as it is inputted to the program from the operator's console, which would typically be an input device such as an ASCII encoded electronic keyboard. As will be illustrated shortly, information stored in the LINE INPUT BUFFER can be transferred to a USER PROGRAM BUFFER. Or, information in the LINE INPUT BUFFER can be analyzed and interpreted. Finally, a line of information in the USER PROGRAM BUFFER can be transferred back to the LINE INPUT BUFFER.

A LINE of information is simply a string of allowable ASCII encoded characters which may consist of COMMANDS, NUMBERS, STATEMENTS, FUNCTIONS, user defined VARIABLES and mathematical OPERATORS. A LINE is always terminated (during operator input) when a line ending terminator, the ASCII code for a carriage-return (CR) is detected.

The pictorial below illustrates three general formats for lines of information. These three general formats essentially provide a means of controlling the overall operation of SCELBAL.



The first line format illustrated above has an EXECUTIVE COMMAND as the first word in the line. Each time a line of information is entered into the LINE INPUT BUFFER from the system's input device, the EXECUTIVE portion of SCELBAL checks to see if the

first word in the line represents any one of the valid SCELBAL commands such as LIST, RUN, SCRATCH, SAVE or LOAD. If so, appropriate action is taken such as LISTing the contents of the USER PROGRAM BUFFER or SCRatching (clearing out the USER PRO-

GRAM BUFFER).

If the first word in a line is not an EXECUTIVE COMMAND, SCELBAL checks to see if the first string of characters represents a LINE NUMBER such as shown in example number three on the previous page. If such is the case it means that the line of information is to be stored in the USER PROGRAM BUFFER as part of a high level stored program being created by the user. Appropriate steps are then taken by the program to append, insert, change or delete information in the USER PROGRAM BUFFER.

If a LINE NUMBER is not detected at the start of a line, the program assumes that the information in the line represents a higher level program STATEMENT which is to be DIRECTly interpreted. This would be the situation when the user desired to use SCELBAL in the "calculator" mode.

In this case, the program would proceed to EVALuate the information by SCANning the information in the LINE INPUT BUFFER. This is done by examining the SYNTAX of the line and initially testing to see if the first word in the line represents a statement KEYWORD such as LET, FOR, IF, GOSUB etc. Upon ascertaining the type of STATEMENT that is to be processed, the program is directed to an appropriate routine that will further evaluate and process the information on the line. This is accomplished by calling on routines that SCAN the line and decode the information, then performing the indicated operations. To do this, other routines such as a PARSER (routine to detect and decode mathematical operators), FUNCTION subroutines (such as SQR, TAB, INT), and FLOATING POINT mathematical routines may be called on to perform the operations specified by the higher level syntax. This process is accomplished on a step-by-step basis following logical rules that establish a HEIRarchy for performing the various types of operations that will be explained in detail in the appropriate sections of this publication.

O.K. The reader now knows how three basic line formats direct SCELBAL to perform an executive function, or place a line of information into the USER PROGRAM BUFFER, or DIRECTly execute a line of information being held in the INPUT LINE BUFFER. What happens when it is desired to execute a higher level program that has been stored in the USER PROGRAM BUFFER?

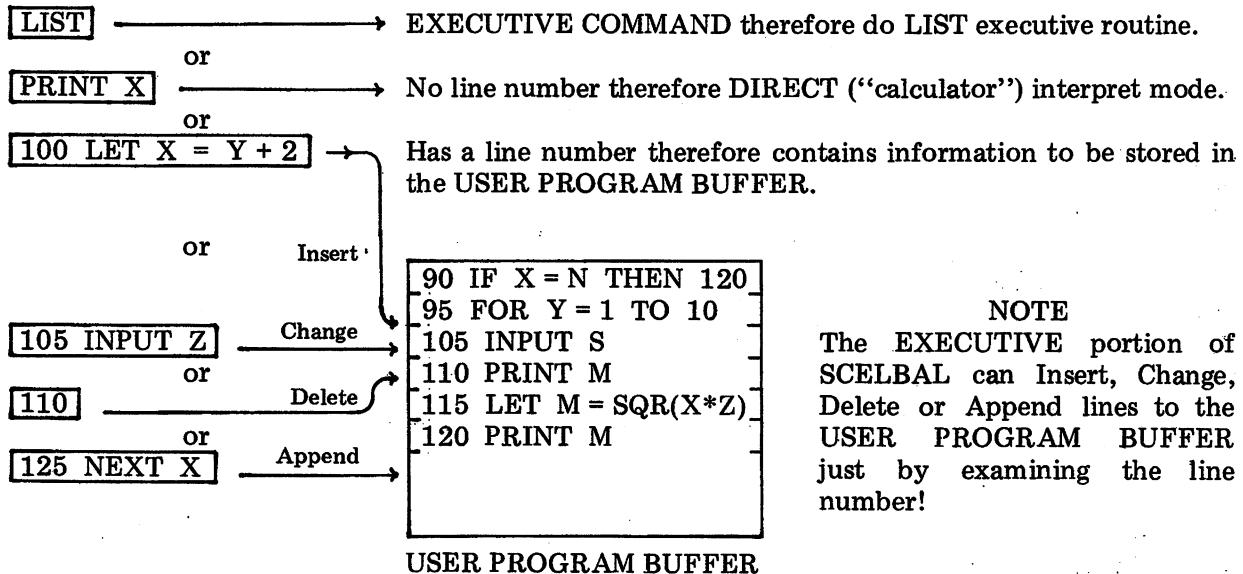
The scheme is still very simple. When the executive portion of SCELBAL detects a line containing the executive RUN command the program simply does the following. It goes to the start of the USER PROGRAM BUFFER and pulls a copy of the first line of information from that storage area back into the INPUT LINE BUFFER. As it does this it strips off the LINE NUMBER. The information in the LINE INPUT BUFFER is then simply processed in the same manner in which a DIRECT type of line would be handled. When the directives contained in that line have been performed, the program proceeds to get the next line in the USER PROGRAM BUFFER (unless directed otherwise by such statements as IF, GOSUB and so forth), strip off the line number, and DIRECTly execute that statement. This process continues until the end of the USER PROGRAM BUFFER has been reached, or an END statement is encountered!

These operational concepts, the reader may now agree, are indeed quite straightforward. True, it does take thousands of machine language instructions to accomplish the tasks, the concepts of which are so easily conveyed in just a few paragraphs. However, the essential point being made is that the overall plan is quite simple. The reader should keep this simple picture in mind as the various sections are discussed in detail. A similar pattern of simplicity will hopefully emerge as the various levels of detail are presented in the following chapters. Readers should refer to this section whenever they feel they are becoming too immersed in the details of individual routines to review where the particular

process being discussed fits in to the basically simple scheme of SCELBAL. The pictorials provided below serve as a summary of what

has just been presented as a quick and easy review when desired.

Representative lines in
LINE INPUT BUFFER



SUMMARY OF FUNDAMENTAL OPERATION OF SCELBAL AS CONTROLLED
BY THE THREE DIFFERENT TYPES OF LINES IN THE LINE INPUT BUFFER

LINE INPUT BUFFER

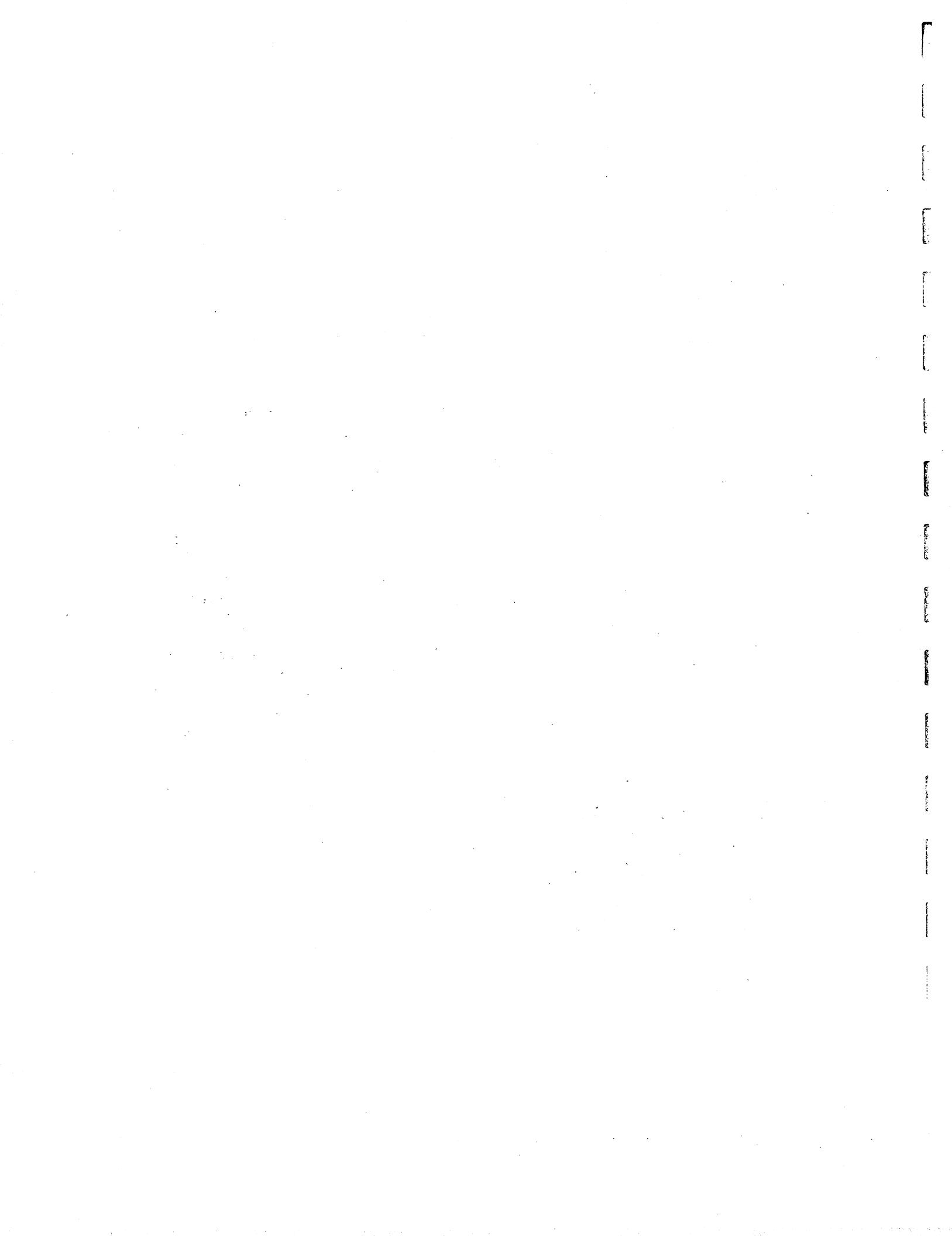
IF X = N THEN 120

When SCELBAL is in the RUN mode each line is pulled from the USER PROGRAM BUFFER. The line number is stripped off and the information in the line is interpreted and executed.

```
90 IF X = N THEN 120
95 FOR Y = 1 TO 10
100 LET X = Y + 2
105 INPUT Z
115 LET M = SQR(X*Z)
120 PRINT M
125 NEXT X
```

USER PROGRAM BUFFER

OPERATION OF SCELBAL WHEN IN THE PROGRAM RUN MODE



THE EXECUTIVE

The EXECUTIVE portion of SCELBAL is the part that essentially enables the operator to control the primary operations of the program from a keyboard device. This part of the program actually performs two types of operations. It can decode and direct the program to execute any of the defined executive COMMANDS which are SCRatch, LIST, RUN, SAVE and LOAD. It also serves as an Editor to enable information to be arranged in the USER PROGRAM BUFFER. This buffer is an area in memory used to hold a user created program in the high level syntax of SCELBAL. The executive RUN command causes a program stored in this area to be executed as a stored program.

Before beginning to present the routines that make up SCELBAL it will be beneficial to explain some aspects of the presentation techniques to be used in this publication.

As each section of the program is discussed the actual source listings for that section of the program will be presented with highly detailed comments. These source listings will refer to the assembled version of the program for an 8008 machine that will be presented later in this publication. (An assembled version for an 8080 machine will also be presented.) That is, the values of pointers, counters, temporary storage locations, and buffers used in the source listings will be those values used in the actual assembled example listing.

SCELBAL uses three PAGES of memory for the storage of pointers, counters, temporary data areas and look up tables. In the assembled program presented in this publication these areas were assigned to pages 01, 26 and 27 in memory. A considerable number of machine language instructions in the program are devoted to establishing pointers to these areas through the use of LLI XXX and LHI YYY instructions. It is likely that some users may desire to assemble the package to reside in areas of memory

other than those used by the version provided. In such an event, if the storage locations assigned to pages 01, 26 and 27 were altered, the user would have to alter the values used when setting up pointers to those areas. As an aid to those that might undertake this task, those LHI YYY instructions that point to those areas in memory have been "flagged" with a double asterisk (**) at the beginning of the associated comments lines. (It is assumed that the locations of storage areas within a page would not be altered.) Thus, a person desiring to create a new assembly of the program would be able to easily spot those instructions to which particular attention would have to be paid.

While discussing the subject of pointers, counters, temporary storage locations, etc., it will be pointed out that the actual locations of all these storage locations will be presented in the final assembled listing of SCELBAL. During the discussion and presentation of the various routines that make up the program during the next several chapters, the reader does not have to be concerned with where each and every such storage location resides. Indeed, there are too many of them for a person to even attempt to keep close tabs on. The actual locations of such storage areas is not important during the description process as it is only necessary for the reader to realize that such locations do exist and to understand the functions that they perform when required.

During the course of the following chapters, virtually each and every routine used in SCELBAL will be presented in its source listing format. However, due to the general complexity of the program (in the microscopic view point of individual instructions, remember, the fundamental concepts are quite simple), some routines may not be explained or presented in detail the first time they are utilized in the source listing. In these cases the user need only understand that there is a routine or subroutine that

will perform a particular function, the details of which will eventually be presented. This is particularly true in the next several chapters as the beginning sections of the program are discussed.

LINE FORMAT

In the preceding chapter, the general format of a line of information as it came

021 261 260 260 240 314 305 324 240 330 240 275 240 331 240 253 240 262
cc 1 0 0 sp L E T sp X sp . = sp Y sp + sp 2

The first line in the above illustration shows the actual machine code that would be stored in successive locations in the INPUT LINE BUFFER. The line beneath it gives the data the code represents in the example. The reader should note that the first entry in the string represents a CHARACTER COUNT. That is a binary count of the number of bytes that the character string consumes. This CHARACTER COUNT (cc) will always be the first byte of data in a character string that is processed by the program. The remaining bytes in a character string are occupied by the ASCII code for the characters being represented shown in eight-bit octal format with the parity bit always being defined in this program as being in a marking (logic one) state. The CHARACTER COUNT for a line of information is calculated by simply reserving the first loca-

from the system's input device was presented. The precise format will now be shown.

Whenever the operator enters information on the system's input device an input routine (labeled STRIN) will arrange a line of information in an INPUT BUFFER in the following format which is illustrated for the example input:

100 LET X = Y + 2

tion in a character string buffer for that information, counting the number of characters inputted until a line terminating character (carriage-return) is received, and then storing the value of that count in the first byte of the character string buffer. The character count for a line of information is an important piece of data that is utilized by many parts of the program package. The reader will soon see how this information is utilized when manipulating lines of data in the Executive/Editor portion of SCELBAL.

With the precise manner in which character strings are stored now explained, one can proceed to present the first major section of SCELBAL. The section to be presented is illustrated by the flow chart shown on the next two pages. The commented source listing begins below.

SCELBAL and EXECUTIVE start here. This first part sets a pointer to a buffer containing the message READY and calls on a subroutine to display this to the operator indicating program is in the EXECUTIVE COMMAND mode.

EXEC, LLI 352
LHI 001
CAL TEXTC

Load L with address of READY message
** Load H with page of READY message
Call subroutine to display the READY message

1400 →
1401 → 39.52

TO Change PRINT to :

27 020 - 001 (cc) for PRINT

021 - 272 :

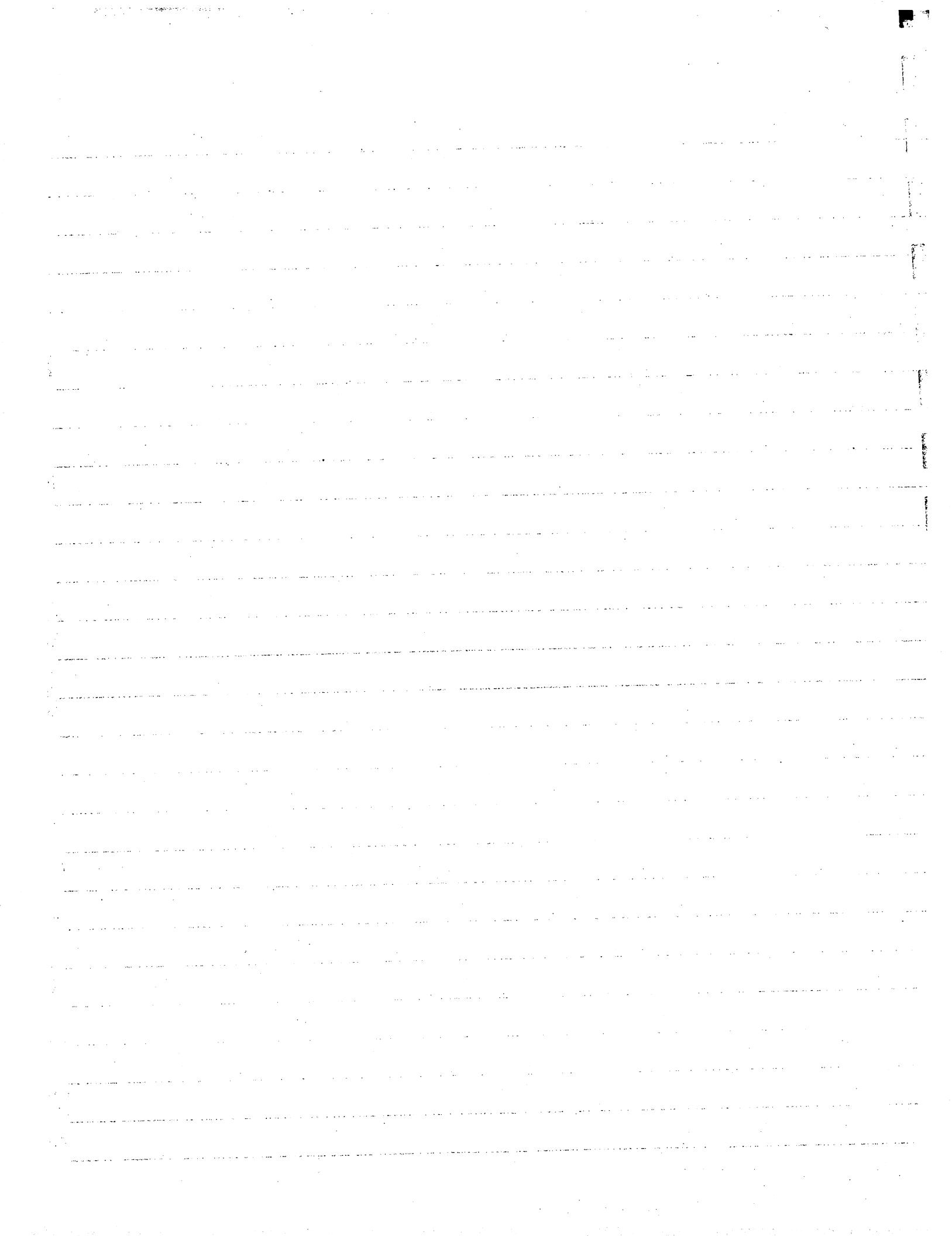
line up all the below

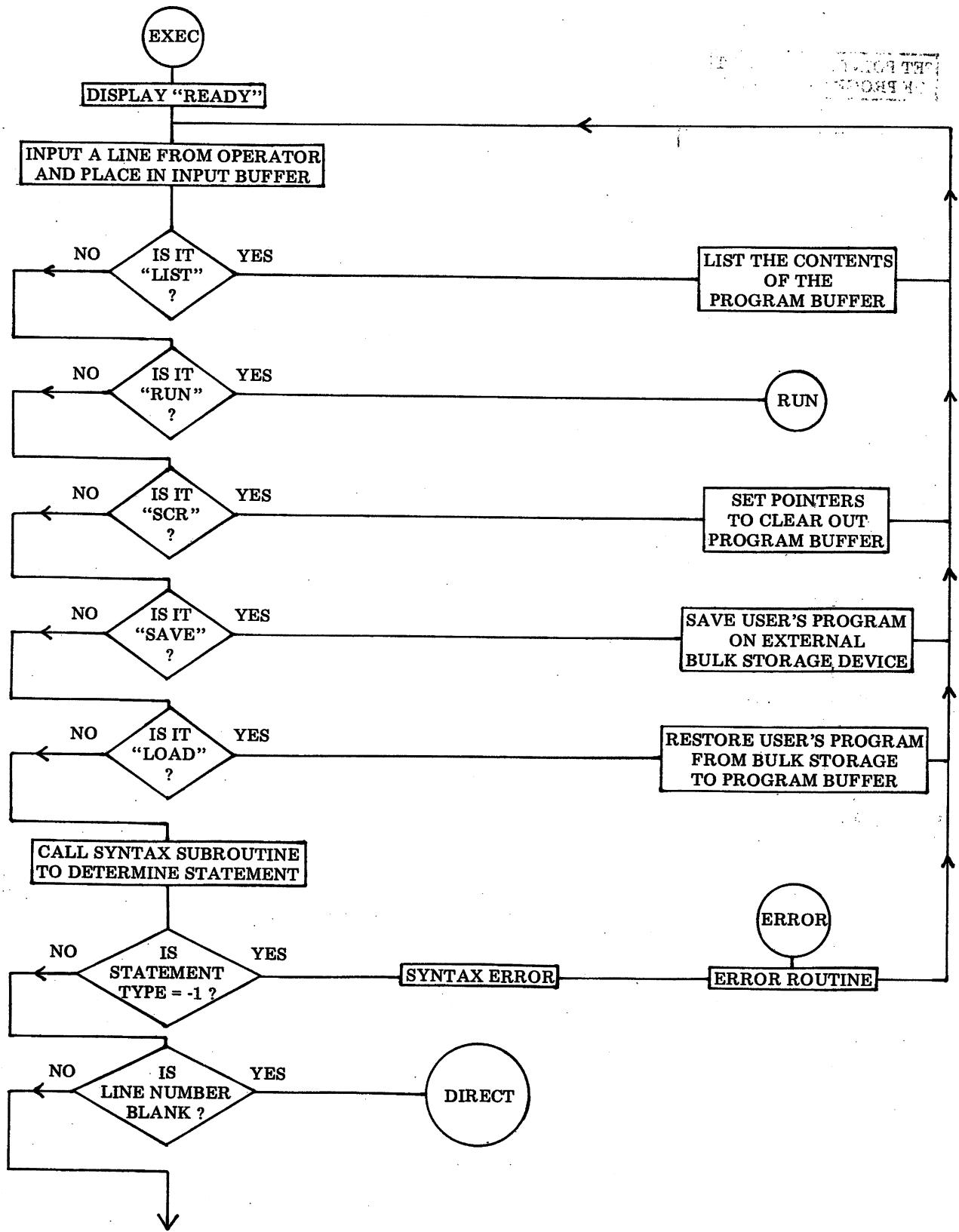
TO CHANGE > to S-

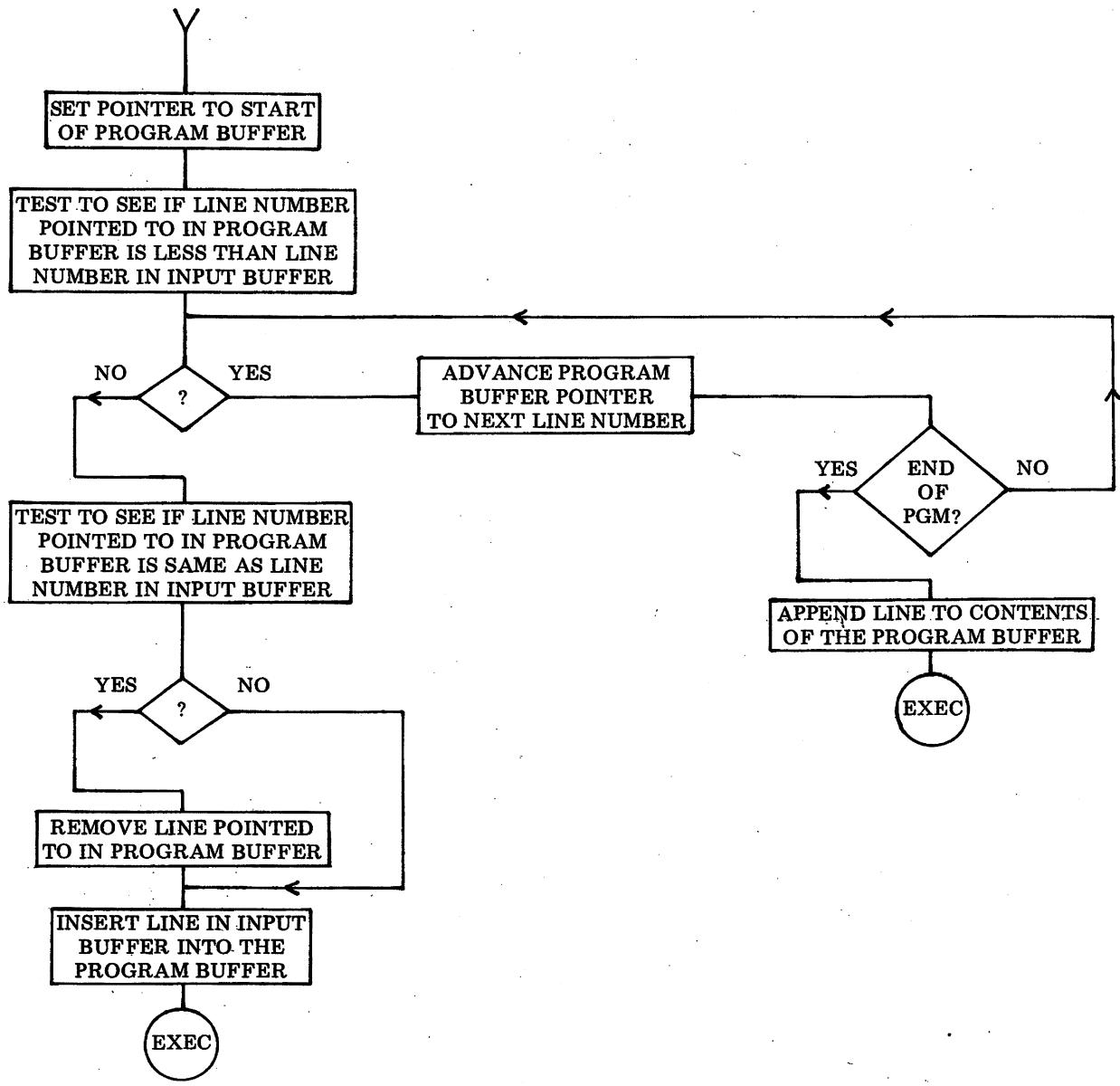
< to

SCAN12 04101

SCAN9 04206







This next section fetches a line from the operator's input device into the INPUT LINE BUFFER. After making sure that the line contains data it tests to see if the first word in the line is the command LIST. If so, it sets up to perform the LIST directive.

10215
 EXEC1, LLI 000
 LHI 026
 CAL STRIN
 LAM
 NDA
 JTZ EXEC1

Load L with starting address of INPUT LINE BUFFER
 ** Load H with page of INPUT LINE BUFFER
 Call subroutine to input a line into the buffer
 The STRIN subroutine will exit with pointer set to the CHARACTER COUNT for the line inputted. Fetch the Value of the counter, if it is zero then line was blank.

LLI 335	Load L with address of LIST in look up table
LHI 001	** Load H with address of LIST in look up table
LDI 026	** Load D with page of line input buffer
LEI 000	Load E with start of line input buffer
CAL STRCP	Call string compare subroutine to see if first word in
JFZ NOLIST	Input buffer is LIST. Jump ahead if not LIST.
LLI 000	If LIST, set up pointers to start of USER PROGRAM
LHI 033	†† BUFFER. (Note user could alter this starting addr)

Next portion of program will LIST the contents of the USER PROGRAM BUFFER until an end of buffer (zero byte) indicator is detected.

LIST, LAM
1033
 NDA
 JTZ EXEC
 CAL TEXTC
 CAL ADV
 CAL CRLF
 JMP LIST

Fetch the first byte of a line in the USER PROGRAM BUFFER and see if it is zero. If so, have finished LIST So go back to start of Executive and display READY. Else call subroutine to display a line of information Now call subroutine to advance buffer pointer to Character count in next line. Also display a CR & LF. Continue LISTing process

If line inputted by operator did not contain a LIST command, continue program to see if RUN or SCRatch command.

NOLIST, LLI 342
1035X
 LHI 001
 LEI 000
 LDI 026
 LEI 000
 CAL STRCP
 JTZ RUN
 LDI 026
 LEI 000
 LLI 346
 LHI 001
 CAL STRCP
 JFZ NOSCR
 LHI 026
 LLI 364
 LMI 033
 INL
 LMI 000
 LLI 077
 LHI 027
 LMI 001
 LLI 075
 LMI 000
 LLI 120
 LMI 000
 LLI 210

Load L with address of RUN in look up table
 ** Load H with address of RUN in look up table
 Load E with start of line input buffer
 ** Load D with page of line input buffer
 (Reserve 2 locs in case of patching by duplicating above)
 Call string compare subroutine to see if first word in Input buffer is RUN. Go to RUN routine if match.
 ** If not RUN command, reset address pointers back To the start of the line input buffer
 Load L with address of SCR in look up table
 ** Load H with page of SCR in look up table
 Call string compare subroutine to see if first word in Input buffer is SCR. If not then jump ahead.
 ** If found SCR command then load memory pointer With address of a pointer storage location. Set that †† Storage location to page of start of USER PROGRAM BUFFER. (Buffer start loc may be altered). Then adv pntr and do same for low addr portion of pntr Now set pointer to address of VARIABLES counter
 ** Storage location. Initialize this counter by placing The count of one into it. Now change the memory pntr To storage location for number of dimensioned arrays @@ And initialize to zero. (@@ = Substitute NOPs if @@ DIMension capability not used in package.) Also @@ Initialize 1'st byte of array name table to zero. Set pointer to storage location for the first byte of the

	LMI 000 INL LMI 000 LHI 033 LLI 000 LMI 000 LHI 057	VARIABLES symbol table. Initialize it to zero too. Advance the pointer and zero the second location In the Variables table also. †† Load H with page of start of USER PROGRAM BUFFER. (Buffer start location could be altered.) Clear first location to indicate end of user program. @@ Load H with page of ARRAYS storage
SCRLOP,	LMI 000 INL JFZ SCRLOP JMP EXEC	@@ And form a loop to clear out all the locations @@ On the ARRAYS storage page. (@@ These become @@ NOPs if DIMension capability deleted fm package.) SCRatch operations completed, go back to EXEC.
		If line inputted did not contain RUN or SCRatch com- mand, program continues by testing for SAVE or LOAD commands. If it does not find either of these com- mands, then operator did not input an executive com- mand. Program then sets up to see if the first entry in the line inputted is a LINE NUMBER.
NOSCR,	LEI 272 LDI 001 LHI 026 LLI 000 CAL STRCP JTZ SAVE LLI 277 LHI 001 LDI 026 LEI 000 CAL STRCP JTZ LOAD LLI 360 LHI 026 LMI 033 INL LMI 000 CAL SYNTAX LLI 203 LHI 026 LAM NDA JFS SYNTOK	Load E with address of SAVE in look up table ** Load D with page of look up table ** Load H with page of input line buffer Set L to start of input line buffer Call string compare subroutine to see if first word in †† Input buffer is SAVE. If so, go to user's SAVE rtn If not SAVE then load L with address of LOAD in look ** Up table and load H with page of look up table ** Load D with page of input line buffer And L to start of input line buffer Call string compare subroutine to see if first word in †† Input buffer is LOAD. If so, go to user's LOAD rtn If not LOAD then set pointer to address of storage loc ** For USER PROGRAM BUFFER pointer. Initialize this †† Pointer to the starting address of the program buffer. Advance memory pntr. Since pointer storage requires Two locations, initialize the low addr portion also. Call the SYNTAX subroutine to obtain a TOKEN indi- Cator which will be stored in this location. Upon return ** From SYNTAX subroutine set memory pointer to The TOKEN indicator storage location and fetch the Value of the TOKEN. If the value of the syntax TOKEN Is positive then have a valid entry.
SYNERR,	LAI 323 LCI 381 JMP ERROR	However, if SYNTAX returns a negative value TOKEN Then have an error condition. Set up the letters SY in ASCII code and go to display error message to operator.
SYNTOK,	LLI 340 LAM NDA	Set pointer to start of LINE NUMBER storage area First byte there will contain the length of the line Number character string. Fetch that value (cc).

	JTZ DIRECT LLI 360 LMI 033 INL LMI 000	If line number blank, have a DIRECT statement! If have a line number must get line in input buffer into †† User program buffer. Initialize pointer to user buffer. This is a two byte pointer so after initializing page addr Advance pointer and initialize location on page address
		If the line in the LINE INPUT BUFFER has a line number then the line is to be placed in the USER PROGRAM BUFFER. It is now necessary to determine where the new line is to be placed in the USER PROGRAM BUFFER. This is dictated by the value of the new line number in relation to the line numbers currently in the program buffer. The next portion of the program goes through the contents of the USER PROGRAM BUFFER comparing the values of the line numbers already stored against the value of the line number currently being held in the LINE INPUT BUFFER. Appropriate action is then taken to Insert or Append, Change, or Delete a line in the program buffer.
GETAUX, 11177	LLI 201 LHI 026 LMI 001 LLI 350 LMI 000	Set memory pointer to line character pointer storage ** Location and then initialize that storage location To point to the 1'st character in a line
GETAU0, 11211	LLI 201 CAL GETCHP JTZ GETAU1 CPI 260 JTS GETAU2 CPI 272 JFS GETAU2 LLI 350 LHI 026 CAL CONCT1	Set memory pointer to addr of start of auxiliary line Number storage area and initialize first byte to zero Set memory pointer to line character pointer storage loc Fetch a char in line pointed to by line pointer If character is a space, skip it by going to advance pntrs If not a space check to see if character represents a Valid decimal digit in the range 0 to 9 by testing the ASCII code value obtained. If not a decimal digit then Assume have obtained the line number. Go process. If valid decimal digit want to append the digit to the ** Current string being built up in the auxiliary line Number storage area so call sub to concat a character.
GETAU1, 11242	LLI 201 LHI 026 LBM INB LMB LLI 360 LHI 026 LCM INL LLM LHC LAM DCB CPB JFZ GETAU0	Reset memory pointer to line character pntr storage loc ** On the appropriate page. Fetch the pointer, increment it, and restore new value Set memory pointer to pgm buff line pntr storage loc ** Bring the high order byte of this double byte pointer Into CPU register C. Then advance the memory pntr And bring the low order byte into register L. Now transfer the higher order portion into memory pointer H. Obtain the char cntr (cc) which indicates the length of The line being pointed to by the user program line pntr Compare this with the value of the chars processed so Far in current line. If not equal, continue getting line nr.

GETAU2,	LLI 360 LHI 026 LDM INL LLM LHD LAM NDA JFZ NOTEND JMP NOSAME	Reset mem pntr to pgm buffer line pntr storage ** On this page and place the high order byte Of this pointer into CPU register D Advance the memory pointer, fetch the second Byte of the pgm buffer line pointer into register L Now make the memory pointer equal to this value Fetch the first byte of a line in the program buffer Test to see if end of contents of pgm buff (zero byte) If not zero continue processing. If zero have reached End of buffer contents so go APPEND line to buffer.
NOTEND,	LLI 350 LHI 026 LDI 026 LEI 340 CAL STRCP JTS CONTIN JFZ NOSAME LLI 360 LHI 026 LCM INL LLM LHC LBM INB CAL REMOVE LLI 203 LHI 026 LAM NDA JTZ EXEC	Load L with addr of auxiliary line number storage loc ** Load H with addr of aux line number storage loc ** Load D with addr of line number buffer location Load E with address of line number buffer location Compare line nr in input buffer with line number in User program buffer. If lesser in value keep looking. If greater in value then go to Insert line in pgm buffer If same values then must remove the line with the same ** Line number from the user program buffer. Set up The CPU memory pointer to point to the current Position in the user program buffer by retrieving that Pointer from its storage location. Then obtain the first Byte of data pointed to which will be the character Count for that line (cc). Add one to the cc value to take Account of the (cc) byte itself and then remove that Many bytes to effectively delete the line fm the user Program buffer. Now see if line in input buffer consists ** Only of a line number by checking SYNTAX TOKEN value. Fetch the TOKEN value from its Storage location. If it is zero then input buffer only Contains a line number. Action is a pure Delete.
NOSAME,,	LLI 360 LHI 026 LDM INL LEM LLI 000 LHI 026 LBM INB CAL INSERT LLI 360 LHI 026 LDM INL LEM LLI 000 LHI 026 CAL MOVEC JMP EXEC1	Reset memory pointer to program buffer ** Line pointer storage location Load high order byte into CPU register D Advance memory pointer Load low order byte into CPU register E Load L with address of start of line input buffer ** Do same for CPU register H Get length of line input buffer Advance length by one to include (cc) byte Go make room to insert line into user program buffer Reset memory pointer to program buffer ** Line pointer storage location Load higher byte into CPU register D Advance memory pointer Load low order byte into CPU register E Load L with address of start of line input buffer ** Do same for CPU register H Call subroutine to Insert line in input buffer into the User program buffer then go back to start of EXEC.

<i>1204K</i>	MOVEC, MOVEPG,	LBM INB LAM CAL ADV CAL SWITCH LMA CAL ADV CAL SWITCH DCB JFZ MOVEPG RET	Fetch length of string in line input buffer Increment that value to provide for (cc) Fetch character from line input buffer Advance pointer for line input buffer Switch memory pointer to point to user pgm buffer Deposit character fm input buff into user pgm buff Advance pointer for user program buffer Switch memory pntr back to point to input buffer Decrement character counter stored in CPU register B If counter does not go to zero continue transfer ops When counter equals zero return to calling routine
<i>1205O</i>	CONTIN,	LLI 360 LHI 026 LDM INL LEM LHD LLE LBM INB CAL ADBDE LLI 360 LHI 026 LMD INL LME JMP GETAUX	Reset memory pointer to program buffer ** Line pointer storage location Load high order byte into CPU register D Advance memory pointer Load low order byte into CPU register E Now set CPU register H to high part of address And set CPU register L to low part of address Fetch the character counter (cc) byte fm line in Program buffer and add one to compensate for (cc) Add length of line value to old value to get new pointer Reset memory pointer to program buffer ** Line pointer storage location Restore new high portion Advance memory pointer And restore new low portion Continue til find point at which to enter new line
<i>1207J</i>	GETCHP,	LHI 026 LBM LLI 360 LDM INL LEM CAL ADBDE LHD LLE LAM CPI 240 RET	** Load H with pointer page (low portion set upon Entry). Now fetch pointer into CPU register B. Reset pntr to pgm buffer line pointer storage location Load high order byte into CPU register D Advance memory pointer Load low order byte into CPU register E Add pointer to pgm buffer pointer to obtain address of Desired character. Place high part of new addr in H. And low part of new address in E. Fetch character from position in line in user pgm buffer See if it is the ASCII code for space Return to caller with flags set to indicate result
<i>1212J</i>	REMOVE,	CAL INDEXB LCM CAL SUBHL LMC LAC NDA JTZ REMOV1 CAL ADV JMP REMOVE	Add (cc) plus one to addr of start of line Obtain byte from indexed location and Subtract character count to obtain old location Put new byte in old location As well as in the Accumulator Test to see if zero byte to indicate end of user pgm buff If it is end of user pgm buffer, go complete process Otherwise add one to the present pointer value And continue removing characters from the user pgm bf

REMOV1, LLI 364
¹²¹⁶
 LHI 026
 LDM
 INL
 LAM
 SUB
 LMA
 RFC
 DCL
 DCD
 LMD
 RET

Load L with end of user pgm buffer pointer storage loc
 ** Load H with page of that pointer storage location
 Get page portion of end of pgm buffer address
 Advance memory pointer
 And get low portion of end of pgm buffer address into
 Accumulator then subtract displacement value in B
 Restore new low portion of end of pgm buffer address
 If subtract did not cause carry can return now
 Otherwise decrement memory pointer back to page
 Storage location, decrement page value to give new page
 And store new page value back in buffer pntr storage loc
 Then return to calling routine

INSERT, LLI 364
¹²¹⁷
 LHI 026
 LAM
 INL
¹²¹⁸
 LLM
 LHA
 CAL INDEXB
 LAH
 CPI 054
 JFS BIGERR
 CAL SUBHL

Load L with end of user pgm buffer pointer storage loc
 ** Load H with page of that pointer storage location
 Get page portion of end of program buffer address
 Advance memory pointer
 Load low portion of end of program buffer address
 Into L and finish setting up memory pointer
 Add (cc) of line in input buffer to form new end of
 Program buffer address. Fetch new end of buffer page
 ↑↑ Address and see if this value would exceed user's
 System capability. Go display error message if so!
 Else restore original value of end of buffer address

INSER1, LCM
¹²¹⁹
 CAL INDEXB
 LMC
 CAL SUBHL
 CAL CPHLDE
 JTZ INSER3
 CAL DEC
 JMP INSER1

Bring byte pointed to by H & L into CPU register C
 Add displacement value to current memory pointer
 Store the byte in the new location
 Now subtract displacement value from H & L
 Compare this with the address stored in D & E
 If same then go finish up Insert operation
 Else set pointer to the byte before the byte just
 Processed and continue the Insert operation

INSER3, INCLIN, LLI 000
¹²²⁰
 LHI 026
 LBM
 INB
 LLI 364
 LDM
 INL
 LEM
 CAL ADBDE
 LME
 DCL
 LMD
 RET

Load L with start of line input buffer
 ** Load H with page of start of line input buffer
 Fetch length of the line in line input buffer
 Increment value by one to include (cc) byte
 Set memory pointer to end of user pgm buffer pointer
 Storage location on same page and fetch page address
 Of this pointer into D. Then advance memory pointer
 And get low part of this pointer into CPU register E.
 Now add displacement (cc) of line in input buffer to
 The end of program buffer pointer. Replace the updated
 Low portion of the new pointer value back in storage
 And restore the new page value back into storage
 Then return to calling routine

The following are small subroutines used by the EXECutive and other parts of SCELBAL.

CPHLDE, 12277	LAH CPD RFZ LAL CPE RET	Subroutine to compare if the contents of CPU registers H & L are equal to registers D & E. First compare Register H to D. Return with flags set if not equal. If Equal continue by comparing register L to E. IF L equals E then H & L equal to D & E so return to Calling routines with flags set to equality status
ADBDE, 12305	LAE ADB LEA RFC IND RET	Subroutine to add the contents of CPU register B (single Byte value) to the double byte value in registers D & E. First add B to E to form new least significant byte Restore new value to E and exit if no carry resulted. If had a carry then must increment most significant byte In register D before returning to calling routine
CTRLC, 12317	LAI 336 LCI 303 JMP ERROR	Set up ASCII code for ↑ (up arrow) in Accumulator. Set up ASCII code for letter 'C' in CPU register C. Go display the 'Control C' condition message.
FINERR, 12322	LLI 340 LHI 026 LAM NDA JTZ FINER1 LLI 366 LHI 001 CAL TEXTC LLI 340 LHI 026 CAL TEXTC FINER1, 12321	Load L with starting address of line number storage area ** Load H with page of line number storage area Get (cc) for line number string. If length is zero meaning There is no line number stored in the buffer then jump Ahead to avoid displaying "AT LINE" message Else load L with address of start of "AT LINE" message ** Stored on this page Call subroutine to display the "AT LINE" message Now reset L to starting address of line number storage ** Area and do same for CPU register H Call subroutine to display the line number Call subroutine to provide a carriage-return and line-feed To the display device then return to EXEC UTIVE.
DVERR, 12323	LAI 304 LCI 332 JMP ERROR	Set up ASCII code for letter 'D' in Accumulator Set up ASCII code for letter 'Z' in CPU register C Go display the 'DZ' (divide by zero) error message
FIXERR, 12326	LAI 306 LCI 330 JMP ERROR	Set up ASCII code for letter 'F' in Accumulator Set up ASCII code for letter 'X' in CPU register C Go display the 'FX' (FiX) error message
NUMERR, 12324	LAI 311 LCI 316 LLI 220 LHI 001 LMI 000 JMP ERROR	Set up ASCII code for letter 'I' in Accumulator Set up ASCII code for letter 'N' in CPU register C Load L with address of pointer used by DINPUT ** Routine. Do same for register H. Clear the location Go display the 'IN' (Illegal Number) error message

The following subroutine, used by various sections of SCELBAL, will search the LINE INPUT BUFFER for a character string which is contained in a buffer starting at the address pointed to by CPU registers H & L when the subroutine is entered.

INSTR,	LDI 026 LEI 000	** Set D to starting page of LINE INPUT BUFFER Load E with starting location of LINE INPUT BUFFER
INSTR1,	CAL ADVDE CAL SAVEHL LBM CAL ADV CAL STRCPC JTZ RESTHL CAL RESTHL LLI 000 LHI 026 LAM CPE JTZ INSTR2 CAL RESTHL JMP INSTR1 HLT	Advance D & E pointer to the next location (input Buffer). Now save contents of D, E, H & L before the Compare operations. Get length of TEST buffer in B. Advance H & L pointer to first char in TEST buffer Compare contents of TEST buffer against input buffer For length B. If match, restore pntrs and exit to caller. If no match, restore pointers for loop test. Load L with start of input buffer (to get the char cntr) ** Load H with page of input buffer. Get length of buffer (cc) into the accumulator. Compare with current input buffer pointer value. If at end of input buffer, jump ahead. Else restore test string address (H&L) and input buffer Address (D&E). Look for occurrence of test string in ln. Safety halt. If program reaches here have system failure.
INSTR2,	LEI 000 RET	If reach end of input buffer without finding a match Load E with 000 as an indicator and return to caller.
ADVDE,	INE RFZ IND RET	Subroutine to advance the pointer stored in the register Pair D & E. Advance contents of E. Return if not zero. If register E goes to zero when advanced, then advance Register D too. Exit to calling routine.

THE MAIN SYNTAX ROUTINE

In order to avoid confusing the reader with the title of this chapter, it will be pointed out that the word SYNTAX generally refers to the complete set of rules or grammar associated with a language such as SCELBAL. The above title implies more than this single chapter will cover. The preceding chapter actually began explaining the complete syntax of SCELBAL by showing how Executive commands were processed and defining the use of line numbers. Other rules of the syntax defined for SCELBAL will become apparent as other chapters are presented. The section of SCELBAL to be discussed in this chapter is limited to the first major subset of the language which consists of the statement classifications. Statements are the major types of higher level directives which the language can interpret and execute such as LET, GOTO, IF, FOR etc. When SCELBAL finds one of these statements in a line of higher level coding, it will know what major type of operation it is to perform. The portion of the program that makes this initial syntax determination has been labeled SYNTAX, hence the title name of this chapter.

The SYNTAX subroutine to be presented in this chapter is not difficult to understand once the reader gets an overall view of the process. Referring to the flow chart for the routine illustrated on the next several pages will help the reader get the essential concepts involved.

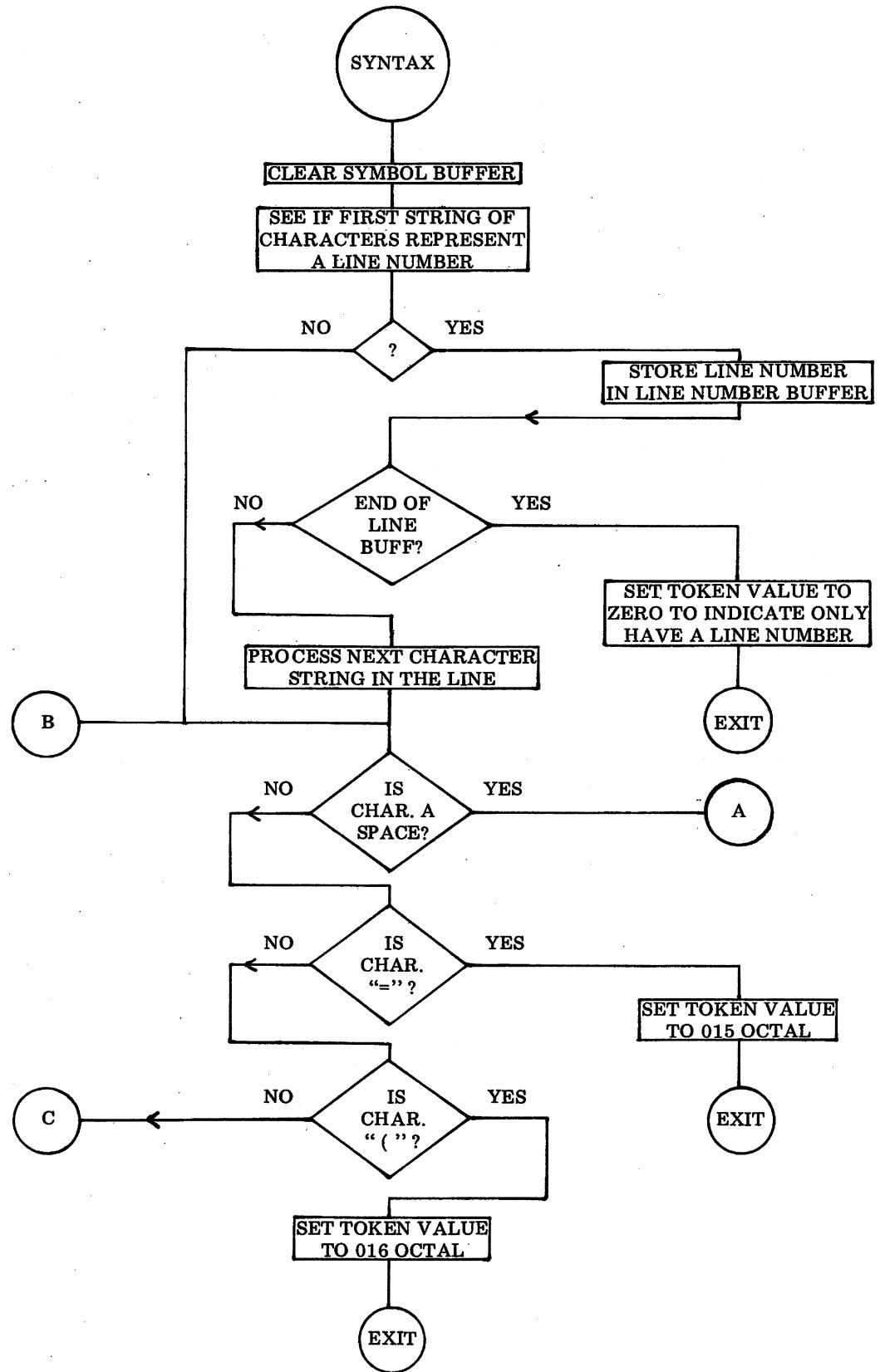
The purpose of the routine is simply to determine whether a group of characters (taken from the contents of the LINE INPUT BUFFER) represent a program line number, and a valid statement KEYWORD. A KEYWORD in this context is simply a group of characters that form the name of a valid statement such as LET, GOSUB, FOR, NEXT and so forth. If a line number is found, and/or a valid KEYWORD is found, the routine will place a TOKEN value in a special TOKEN BUFFER to indicate what the SYNTAX subroutine processed. A TOKEN value in this

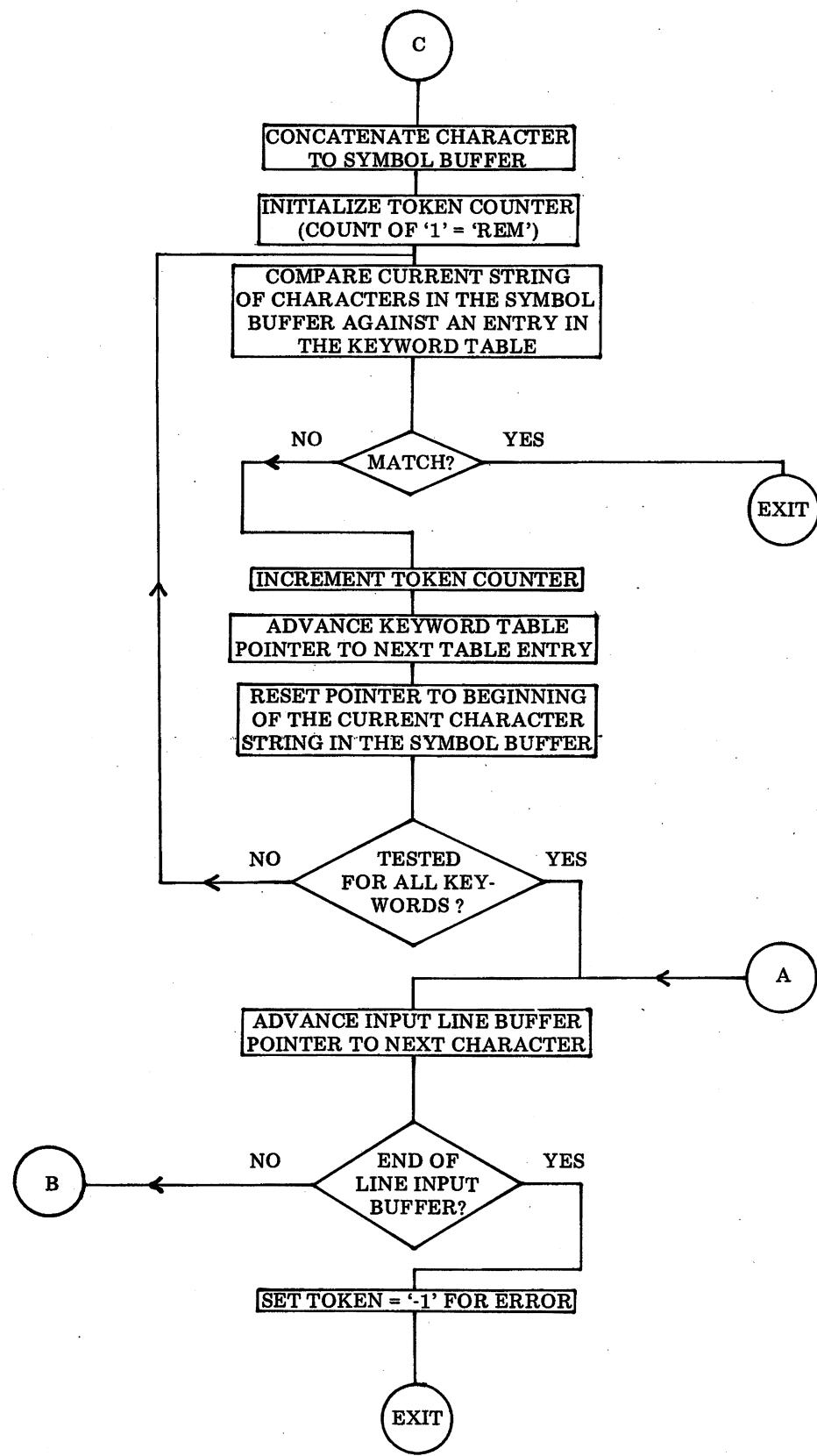
context is simply a numerical value used to symbolize the finding of a particular type of character string. It is a sort of shorthand notation that serves to reduce the amount of data that must be processed by the computer in the future.

Thus, for instance, if during the operation of the SYNTAX routine, the keyword REM is detected, a token value of 001 (octal) will be established. The finding of the keyword GOTO would result in a token value of 004 being set up. Each valid keyword has a token value associated with it. The token value established is then used later by other portions of SCELBAL to signify a particular type of operation using much less storage space than would be required if one had to refer to an entire string of ASCII characters that make up a keyword. The technique of establishing a token value to represent a particular string of characters is thus a powerful method in the process of converting higher level English language directives which are convenient for human programmers, down to the simple numerical directives that the computer needs for sustenance!

The process by which keywords are converted to token values is shown quite clearly in the flow chart provided. Essentially the routine seeks to find a match between a group of characters (taken from the line input buffer and examined while in a working register) to determine if they match any entry in a keyword look-up table. The keyword look-up table utilized by this routine is formatted as follows:

CCC	Number of characters in keyword.
AAA	ASCII code for 1'st letter of keyword
BBB	ASCII code for 2'nd letter of keyword
NNN	ASCII code for N'th letter of keyword
CCC	Number of characters in next keyword
AAA	ASCII code for 1'st letter of the next entry in the keyword table, etc.





The table contains all the valid keywords defined for statement types used in the high level language SCELBAL. These are: REM, IF, LET, GOTO, PRINT, INPUT, FOR, NEXT, GOSUB, RETURN, DIM and END. They appear in the table in the order just presented.

Since the number of characters making up a keyword can vary, the technique used to look for a match between a group of characters in the line input buffer and the look-up table is as follows.

Characters are taken one at a time from the line input buffer and placed in a special buffer (referred to as the SYMBOL buffer). Each time a character is added to the symbol buffer, a search is made through the keyword look-up table. At the start of the search a TOKEN value of 001 (octal) is set in the TOKEN VALUE storage register. Now, as each entry in the look-up table is compared against the character string currently in the symbol buffer and fails to match, the token value is incremented. This technique results, if a match IS found, in the token value already being set to the proper token value. For instance, if a match was found for the keyword PRINT, the token value would be at 005. (Print is the fifth entry in the look-up table.) If a match is not found during the search of the table, the routine goes back and appends another character from the input buffer onto the symbol buffer. It then reinitializes the token value back to 001 and tries searching the table again. This process continues until either a match is found or an end of character string terminator is detected. Notice that if a keyword is not found, once the table look-up process is started, that an error condition (SYntax error) is assumed to exist. For such an error condition, a negative value (377 octal) is placed in the token value register so that the routine calling SYNTAX will be able to detect the error condition.

The reader should note that the flow chart illustrates two special syntax conditions. One is when an equal (=) sign is detected. Finding an equal sign before a keyword has been es-

tablished can occur for a special situation called the IMPLIED LET. The IMPLIED LET statement enables SCELBAL to interpret a statement such as:

X = Y

without having to put in the actual LET keyword. An IMPLIED LET statement signified by an equal sign at the point in a line where the SYNTAX routine would be processing the information is handled as a special keyword and given the token value of 015.

A second special case is defined for handling array (subscripted) variables in an IMPLIED LET situation. The use of a left hand parenthesis "(" at this point in a line is assigned a token value of 016.

One of the principal functions of the SYNTAX subroutine, which is shown at the beginning of the flow chart, is to see if the line being processed contains a line number and to store the line number in a special line number buffer. This is because the SYNTAX routine is the first routine to be called when SCELBAL is in the RUN mode each time a new line is processed. Lines stored in the program buffer start with a line number, and then the keyword statement. Naturally, the SYNTAX subroutine must get beyond the line number before it can look for the keyword in the line. However, there are certain cases, such as when SYNTAX is called by the EXECutive routine (described in the previous chapter) where a line in the input buffer may contain just a line number and no keyword. (This is the situation when an operator wishes to delete a specific line number from the user's program buffer.) For this special case, the SYNTAX subroutine assigns a token value of 000.

The converse case can occur when a DIRECT (calculator mode) statement is being processed. In that case there would be no line number. The flow chart illustrates that if the first group of characters in a line is not num-

erical the routine proceeds to just look for a keyword.

The reader should now be prepared to fol-

low the detailed source listing for this section of the program as presented next starting at the instruction labeled SYNTAX. The reader may review from the flow chart as desired.

SYNTAX,	CAL CLESYM LLI 340 LHI 026 LMI 000 LLI 201 LMI 001	Clear the SYMBOL BUFFER area Set L to start of LINE NUMBER BUFFER ** Set H to page of LINE NUMBER BUFFER Initialize line number buff by placing zero as (cc) Change pointer to syntax counter/pntr storage loc. Set pointer to first character (after cc) in line buffer
SYNTX1,	LLI 201 CAL GETCHR JTZ SYNTX2 CPI 260 JTS SYNTX3 CPI 272 JFS SYNTX3 LLI 340 CAL CONCT1	Set pointer to syntax cntr/pntr storage location Fetch the character pointed to by contents of syntax Cntr/pntr from the line input buffer. If character was A space, ignore. Else, test to see if character was ASCII Code for a decimal digit. If not a decimal digit, consider Line number to have been processed by jumping Over the remainder of this SYNTX1 section. If have decimal digit, set pointer to start of LINE NUMBER BUFFER and append incoming digit there.
SYNTX2,	LLI 201 CAL LOOP JFZ SYNTX1 LLI 203 LMI 000 RET	Reset L to syntax cntr/pntr storage location. Call sub-Routine to advance pntr and test for end of input buffer If not end of input buffer, go back for next digit If end of buffer, only had a line number in the line. Set pntr to TOKEN storage location. Set TOKEN = 000. Return to caller.
SYNTX3,	LLI 201 LBM LLI 202 LMB	Reset pointer to syntax cntr/pntr and fetch Position of next character after the line number Change pntr to SCAN pntr storage location Store address when SCAN takes up after line number
SYNTX4,	LLI 202 CAL GETCHR JTZ SYNTX6 CPI 275 JTZ SYNTX7 CPI 250 JTZ SYNTX8 CAL CONCTS LLI 203 LMI 001 LHI 027 LLI 000	Set pntr to SCAN pntr storage location Fetch the character pointed to by contents of the SCAN Pointer storage location. If character was ASCII code For space, ignore. Else, compare character with "=" sign If is an equal sign, go set TOKEN for IMPLIED LET. Else, compare character with left parenthesis "(" If left parenthesis, go set TOKEN for implied array LET Otherwise, concatenate the character onto the string Being constructed in the SYMBOL BUFFER. Now set Up TOKEN storage location to an initial value of 001. ** Set H to point to start of KEYWORD TABLE. Set L to point to start of KEYWORD TABLE.

SYNTX5,	LDI 026 LEI 120 CAL STRCP RTZ CAL SWITCH	** Set D to page of SYMBOL BUFFER Set E to start of SYMBOL BUFFER Compare char string presently in SYMBOL BUFFER With entry in KEYWORD TABLE. Exit if match. TOKEN will be set to keyword found. Else, switch
SYNTXL,	INL LAM NDI 300 JFZ SYNTXL CAL SWITCH LLI 203 LHI 026 LBM INB LMB CAL SWITCH LAB CPI 015 JFZ SYNTX5	Pointers to get table address back and advance pntr to KEYWORD TABLE. Now look for start of next entry In KEYWORD TABLE by looking for (cc) byte which Will NOT have a one in the two most sig. bits. Advance Pntr til next entry found. Then switch pointers again so Table pointer is in D&E. Put addr of TOKEN in L. ** And page of TOKEN in H. Fetch the value currently In TOKEN and advance it to account for going on to The next entry in the KEYWORD TABLE. Restore the updated TOKEN value back to storage. Restore the keyword table pointer back to H&L. Put TOKEN count in ACC. See if have tested all entries in the keyword table. If not, continue checking the keyword table.
SYNTX6,	LLI 202 LHI 026 CAL LOOP JFZ SYNTX4 LLI 203 LMI 377 RET	Set L to SCAN pointer storage location ** Set H to page of SCAN pointer storage location Call routine to advance pntr & test for end of ln buffer Go back and add another character to SYMBOL BUFF And search table for KEYWORD again. Unless reach End of line input buffer. In which case set TOKEN=377 As an error indicator and exit to calling routine.
SYNTX7,	LLI 203 LMI 015 RET	Set pointer to TOKEN storage register. Set TOKEN Equal to 015 when “=” sign found for IMPLIED LET. Exit to calling routine.
SYNTX8,	LLI 203 LMI 016 RET	Set pointer to TOKEN storage register. Set TOKEN Equal to 016 when “(“ found for IMPLIED array LET. Exit to calling routine.
		The following are subroutines used by SYNTAX and other routines in SCELBAL.
BIGERR,	LAI 302 LCI 307	Load ASCII code for letters B and G to indicate BIG ERROR (For when buffer, stack, etc., overflows.)
ERROR,	CAL ECHO LAC CAL ECHO JMP FINERR	Call user provided display routine to print ASCII code In accumulator. Transfer ASCII code from C to ACC And repeat to display error codes. Go complete error message (AT LINE) as required.
GETCHR,	LAM CPI 120 JFS BIGERR	Get pointer from memory location pointed to by H&L See if within range of line input buffer If not then have an overflow condition = error.

	LLA LHI 026 LAM CPI 240 RET	Else can use it as addr of character to fetch from the ** LINE INPUT BUFFER by setting up H too. Fetch the character from the line input buffer. See if it is ASCII code for space. Return to caller with flags set according to comparison.
CLESYM,	LLI 120 LHI 026 LMI 000 RET	Set L to start of SYMBOL BUFFER. ** Set H to page of SYMBOL BUFFER. Place a zero byte at start of SYMBOL BUFFER. To effectively clear the buffer. Then exit to caller.
		Subroutine to concatenate (append) a character to the SYMBOL BUFFER. Character must be alphanumeric.
CONCTA,	CPI 301 JTS CONCTN CPI 333 JTS CONCTS	See if character code less than that for letter A. If so, go see if it is numeric. See if character code greater than that for letter Z. If not, have valid alphabetical character.
CONCTN,	CPI 260 JTS CONCTE CPI 272 JFS CONCTE	Else, see if character in valid numeric range. If not, have an error condition. Continue to check for valid number. If not, have an error condition.
CONCTS,	LLI 120 LHI 026	If character alphanumeric, can concatenate. Set pointer ** To starting address of SYMBOL BUFFER.
CONCT1,	LCM INC LMC LBA CAL INDEXC LMB LAI 000 RET	Fetch old character count in SYMBOL BUFFER. Increment the value to account for adding new Character to the buffer. Restore updated (cc). Save character to be appended in register B. Add (cc) to address in H & L to get new end of buffer Address and append the new character to buffer Clear the accumulator Exit to caller
CONCTE,	JMP SYNERR	If character to be appended not alphanumeric, ERROR!
		Subroutine to compare character strings pointed to by register pairs D & E and H & L.
STRCP,	LAM CAL SWITCH LBM CPB RFZ CAL SWITCH	Fetch (cc) of first string. Switch pointers and fetch length of second string (cc) Into register B. Compare the lengths of the two strings. If they are not the same Return to caller with flags set to non-zero condition Else, exchange the pointers back to first string.
STRCPL,	CAL ADV LAM CAL SWITCH	Advance the pointer to string number 1 and fetch a Character from that string into the accumulator. Now switch the pointers to string number 2.

	CAL ADV	Advance the pointer in line number 2.
STRCPE,	CPM RFZ CAL SWITCH DCB JFZ STRCPL RET	Compare char in string 1 (ACC) to string 2 (memory) If not equal, return to caller with flags set to non-zero Else, exchange pointers to restore pptr to string 1 Decrement the string length counter in register B If not finished, continue testing entire string If complete match, return with flag in zero condition
STRCPC,	LAM CAL SWITCH JMP STRCPE	Fetch character pointed to by pointer to string 1 Exchange pointer to examine string 2 Continue the string comparison loop
		Subroutine to advance the two byte value in CPU registers H and L.
ADV,	INL RFZ INH RET	Advance value in register L. If new value not zero, return to caller. Else must increment value in H Before returning to caller
		Subroutine to advance a buffer pointer and test to see if the end of the buffer has been reached.
LOOP,	LBM INB LMB LLI 000 LAM DCB CPB RET	Fetch memory location pointed to by H & L into B. Increment the value. Restore it back to memory. Change pointer to start of INPUT LINE BUFFER Fetch buffer length (cc) value into the accumulator Make value in B original value See if buffer length same as that in B Return with flags yielding results of the comparison
		The following subroutine is used to input characters from the system's input device (such as a keyboard) into the LINE INPUT BUFFER. Routine has limited editing capability included. (Rubout = delete previous character(s) entered.)
STRIN,	LCI 000	Initialize register C to zero.
STRIN1,	CAL CINPUT CPI 377 JFZ NOTDEL LAI 334 CAL ECHO DCC JTS STRIN CAL DEC JMP STRIN1	Call user provided device input subroutine to fetch one Character from the input device. Is it ASCII code for Rubout? Skip to next section if not rubout. Else, load ASCII code for backslash into ACC. Call user display driver to present backslash as a delete Indicator. Now decrement the input character counter. If at beginning of line do NOT decrement H and L. Else, decrement H & L line pointer to erase previous Entry, then go back for a new input.

NOTDEL,	CPI 203 JTZ CTRLC CPI 215 JTZ STRINF CPI 212 JTZ STRIN1 CAL ADV INC LMA LAC CPI 120 JFS BIGERR JMP STRIN1	See if character inputted was 'CONTROL C' If so, stop inputting and go back to the EXECutive If not, see if character was carriage-return If so, have end of line of input If not, see if character was line-feed If so, ignore the input, get another character If none of the above, advance contents of H & L Increment the character counter Store the new character in the line input buffer Put new character count in the accumulator Make sure maximum buffer size not exceeded If buffer size exceeded, go display BG error message Else can go back to look for next input
STRINF,	LBC CAL SUBHL LMC CAL CRLF RET	Transfer character count from C to B Subtract B from H & L to get starting address of The string and place the character count (cc) there Provide a line ending CR & LF combination on the Display device. Then exit to caller.
		Subroutine to subtract contents of CPU register B from the two byte value in CPU registers H & L.
SUBHL,	LAL SUB LLA RFC DCH RET	Load contents of register L into the accumulator Subtract the contents of register B Restore the new value back to L If no carry, then no underflow. Exit to caller. Else must also decrement contents of H. Before returning to caller.
		Subroutine to display a character string on the system's display device.
TEXTC,	LCM LAM NDA RTZ	Fetch (cc) from the first location in the buffer (H & L Pointing there upon entry) into register B and ACC. Test the character count value. No display if (cc) is zero.
TEXTCL,	CAL ADV LAM CAL ECHO DCC JFZ TEXTCL RET	Advance pointer to next location in buffer Fetch a character from the buffer into ACC Call the user's display driver subroutine Decrement the (cc) If character counter not zero, continue display Exit to caller when (cc) is zero.
		Subroutine to provide carriage-return and line-feed combination to system's display device. Routine also initializes a column counter to zero. Column counter is used by selected output routines to count the num- ber of characters that have been displayed on a line.

CRLF,	LAI 215 CAL ECHO LAI 212 CAL ECHO LLI 043 LHI 001 LMI 001 LHD LLE RET	Load ASCII code for carriage-return into ACC Call user provided display driver subroutine Load ASCII code for line-feed into ACC Call user provided display driver subroutine Set L to point to COLUMN COUNTER storage location ** Set H to page of COLUMN COUNTER Initialize COLUMN COUNTER to a value of one Restore H from D (saved by ECHO subroutine) Restore L from E (saved by ECHO subroutine) Then exit to calling routine
		Subroutine to decrement double-byte value in CPU registers H and L.
DEC,	DCL INL JFZ DECNO DCH	Decrement contents of L Now increment to exercise CPU flags If L not presently zero, skip decrementing H Else decrement H
DECNO,	DCL RET	Do the actual decrement of L Return to caller
INDEXB,	LAL ADB LLA RFC INH RET	Subroutine to index the value in CPU registers H and L by the contents of CPU register B.
ECHO,	LDH LEL LLI 043 LHI 001 LBM INB LMB CAL ††† ††† LHD LLE RET	Load L into the accumulator Add B to that value Restore the new value to L If no carry, return to caller Else, increment value in H Before returning to caller
CINPUT,	JMP ††† †††	The following subroutine is used to display the ASCII encoded character in the ACC on the system's display device. This routine calls a routine labeled CINPUT which must be provided by the user to actually drive the system's output device. The subroutine below also increments an output column counter each time it is used. Save entry value of H in register D And save entry value of L in register E Set L to point to COLUMN COUNTER storage location ** Set H to page of COLUMN COUNTER Fetch the value in the COLUMN COUNTER And increment it for each character displayed Restore the updated count in memory †† Call the user's device driver subroutine Restore entry value of H from D Restore entry value of L from E Return to calling routine

STATEMENT INTERPRETATION

The reader has now been presented with the knowledge of how SCELBAL utilizes an Executive routine to store a user created high level language program in memory. Additionally, the reader has been shown how the SYNTAX routine is used to analyze the first portion of a line in order to obtain the line number and to set up a token value representing the finding of a particular type of statement in the beginning portion of a line. (A line referring to a line of the source coding in the higher level language.) The reader should now be prepared to learn how a program stored in the user program buffer (or a single line "calculator mode" directive residing in the line input buffer) is further processed.

The flow chart on the next page will once again illustrate how the program continues to operate in a straightforward, conceptually simple manner. It illustrates that when the Executive interprets a RUN command, the program proceeds to perform operations in the following fashion.

The first line stored in the user program buffer is pulled into the line input buffer. Then the SYNTAX subroutine is used to find out what type of statement is contained in the line. A TOKEN value representing the type of statement found is returned by the SYNTAX subroutine. This token value is then used to direct the program to go to a particular routine that will perform the type of operation dictated by the statement type. It is as simple as that!

There is then a whole series of routines, one for each type of statement used in the language, that processes the remaining data on a line after the statement keyword. This chapter will present the details for each of these routines.

When the execution of a statement routine has been completed, the program con-

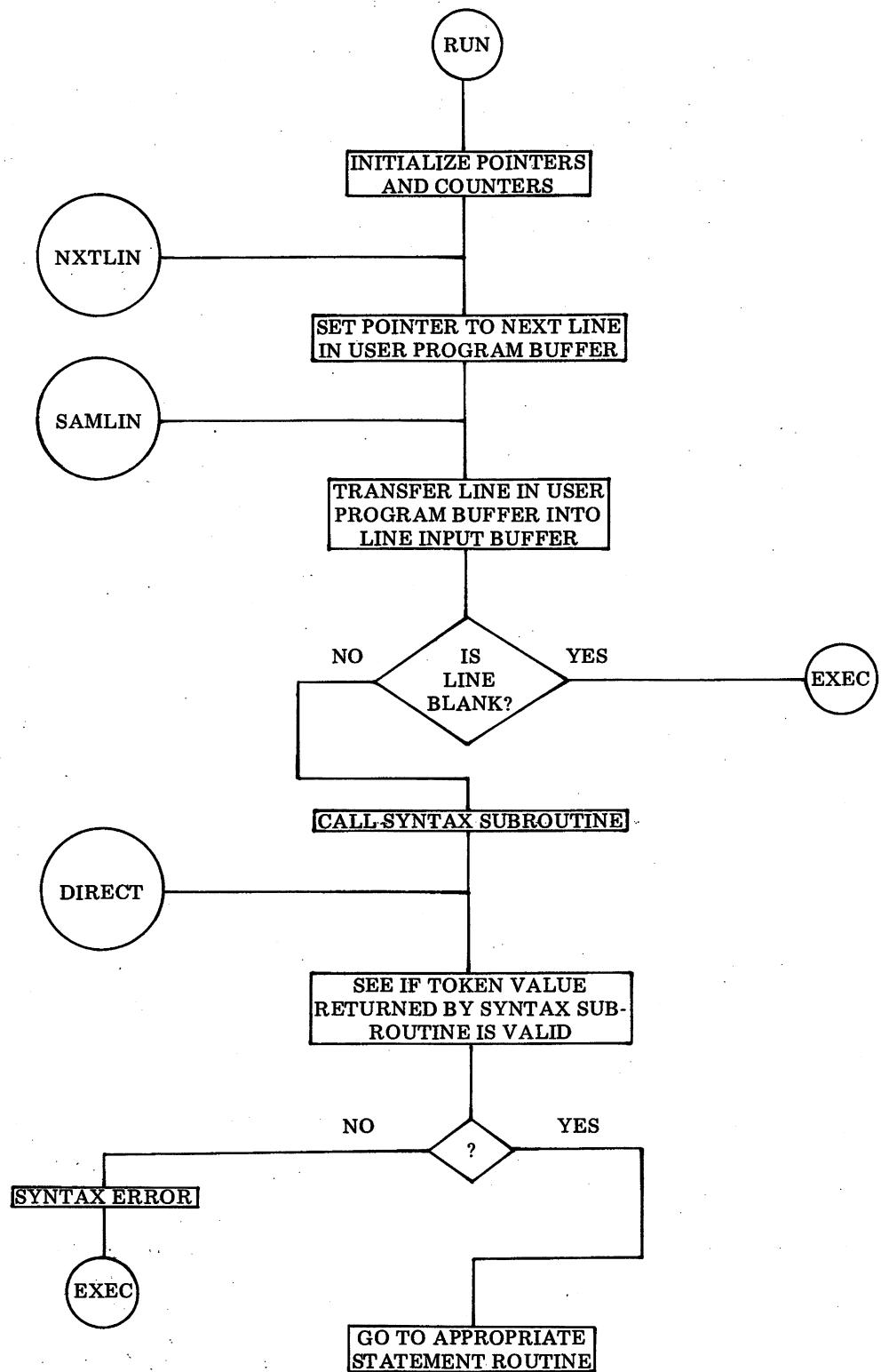
tinues by simply extracting the next line of information stored in the user program buffer and repeating the process.

In the DIRECT, or "calculator" mode, the program simply restricts its operation to processing the line of information stored in the input line buffer, instead of extracting lines from the user program buffer. The reader may observe that the RUN flow chart shows several entry points to various subsections of the program. The reader can see that there is a DIRECT entry to the routine which is used when the program is interpreting a single line statement in the "calculator" mode.

The reader might also note that there are two special entry points in the RUN routine named NXTLIN and SAMLIN. The first entry point is used when the program has finished the execution of a statement and is to proceed to interpret the next line of information in the user program buffer. The second entry point is used in special situations which will be explained more fully later in this chapter. One such case is when the program has executed a GOTO statement. This is because, the routine that processes a GOTO statement will search for the line number in the user program buffer that was specified in the GOTO directive. When it finds that line number, the program will already have the user program buffer pointer set up to point to the line that should be processed next!

The various statement routines presented in this chapter will call on subroutines whose functions will be described in detail in following chapters. However, the reader should be able to discern the essential operations of each type of statement as they are presented. The supplementary subroutines will fall into logical order once the information in this chapter has been digested and is understood.

The source listing for the RUN routine and associated subsections of that routine are presented immediately following the flow chart.



RUN,	LLI 073 LHI 027 LMI 000 LLI 205 LMI 000 LLI 360 LHI 026 LMI 033 INL LMI 000 JMP SAMLIN	Load L with addr of GOSUB/RETURN stack pointer ** Load H with page of same pointer Initialize the GOSUB/RETURN stack pointer to zero Load L with addr of FOR/NEXT stack pointer Initialize the FOR/NEXT stack pointer to zero Load L with addr of user pgm buffer line pointer ** Load H with page of user pgm buffer line pointer †† Initialize pointer (may be altered by user) Advance memory pointer to low portion of user pgm Buffer pointer and initialize to start of buffer Start executing user program with first line in buffer
NXTLIN,	LLI 360 LHI 026 LDM INL LEM LHD LLE LBM INB CAL ADBDE LLI 360 LHI 026 LMD INL LME LLI 340 LHI 026 LAM NDA JTZ EXEC LAA LAA	Load L with addr of user program buffer line pointer ** Load H with page of user pgm buffer line pointer Place page addr of pgm buffer line pointer in D Advance the memory pointer Place low addr of pgm buffer line pointer in E Also put page addr of pgm buffer line pointer in H And low addr of pgm buffer line pointer in L Now fetch the (cc) of current line into register B Add one to account for (cc) byte itself Add value in B to D&E to point to next line in User program buffer. Reset L to addr of user pgm ** Buffer pointer storage location. Store the new Updated user pgm line pointer in pointer storage Location. Store both the high portion And low portion. (Now points to next line to be Processed from user program buffer.) Change pointer ** To address of line number buffer. Fetch the last Line number (length) processed. Test to see if it was Blank. If it was blank Then stop processing and return to the Executive Insert two effective NOPs here In case of patching
SAMLIN,	LLI 360 LHI 026 LCM INL LLM LHC LDI 026 LEI 000 CAL MOVEC LLI 000 LHI 026 LAM NDA JTZ EXEC CAL SYNTAX	Load L with addr of user program buffer line pointer ** Load H with page of same pointer Fetch the high portion of the pointer into register C Advance the memory pointer Fetch the low portion of the pointer into register L Now move the high portion into register H ** Set D to page of line input buffer Set E to address of start of line input buffer Move the line from the user program buffer into the Line input buffer. Now reset the pointer to the start ** Of the line input buffer. Fetch the first byte of the line input buffer (cc) Test (cc) value to see if fetched a blank line If fetched a blank line, return to the Executive Else call subrtn to strip off line nr & set statement token

DIRECT,	LLI 203	Load L with address of syntax TOKEN storage location
	LHI 026	** Load H with page of syntax TOKEN location
	LAM	Fetch the TOKEN value into the accumulator
	CPI 001	Is it token value for REM statement? If so, ignore the
	JTZ NXTLIN	Current line and go on to the next line in pgm buffer.
	CPI 002	Is it token value for IF statement?
	JTZ IF	If yes, then go to the IF statement routine.
	CPI 003	Is it token value for LET statement? (Using keyword)
	JTZ LET	If yes, then go to the LET statement routine.
	CPI 004	Is it token value for GOTO statement?
	JTZ GOTO	If yes, then go to the GOTO statement routine.
	CPI 005	Is it token value for PRINT statement?
	JTZ PRINT	If yes, then go to the PRINT statement routine.
	CPI 006	Is it token value for INPUT statement?
	JTZ INPUT	If yes, then go to the INPUT statement routine.
	CPI 007	Is it token value for FOR statement?
	JTZ FOR	If yes, then go to the FOR statement routine.
	CPI 010	Is it token value for NEXT statement?
	JTZ NEXT	If yes, then go to the NEXT statement routine.
	CPI 011	Is it token value for GOSUB statement?
	JTZ GOSUB	If yes, then go to the GOSUB statement routine.
	CPI 012	Is it token value for RETURN statement?
	JTZ RETURN	If yes, then go to the RETURN statement routine.
	CPI 013	Is it token value for DIM statement?
	JTZ DIM	If yes, then go to the DIM statement routine.
	CPI 014	Is it token value for END statement?
	JTZ EXEC	If yes, then go back to the Executive, user pgm finished!
	CPI 015	Is it token value for IMPLIED LET statement?
	JTZ LET0	If yes, then go to special LET entry point.
	CPI 016	@@ Is it token value for ARRAY IMPLIED LET?
	JFZ SYNERR	If not, then assume a syntax error condition.
	CAL ARRAY1	@@ Else, perform array storage set up subroutine.
	LLI 206	@@ Set L to array pointer storage location.
	LHI 026	@@ ** Set H to array pointer storage location.
	LBM	@@ Fetch array pointer to register B.
	LLI 202	@@ Change memory pointer to syntax pntr storage loc.
	LMB	@@ Save array pointer value there.
	CAL SAVSYM	@@ Save array name in auxiliary symbol buffer
	JMP LET1	@@ Go to special array implied LET entry point.

THE PRINT STATEMENT ROUTINE

The PRINT statement routine is used to output data as directed by the creator of a SCELBAL program. There are several types of information that the PRINT statement can display. It can display text messages that have been enclosed by single ('.....') or

double (".....") quotation marks on the line containing the PRINT statement. It is also used to display the numerical values of variables or expressions referred to in the line containing the PRINT directive. Finally, the PRINT statement may be used to

TAB (space over) to a TABBING POSITION (every sixteenth column) and control the occurrence of a line-feed and carriage-return combination after the displaying of information. (The PRINT statement may also be used to perform two special functions that will be explained in a later chapter. These relate to the capability to TAB to a specific column position specified by the user, and the capability to display a certain range of numbers as an alphanumeric character through the use of the CHR function.)

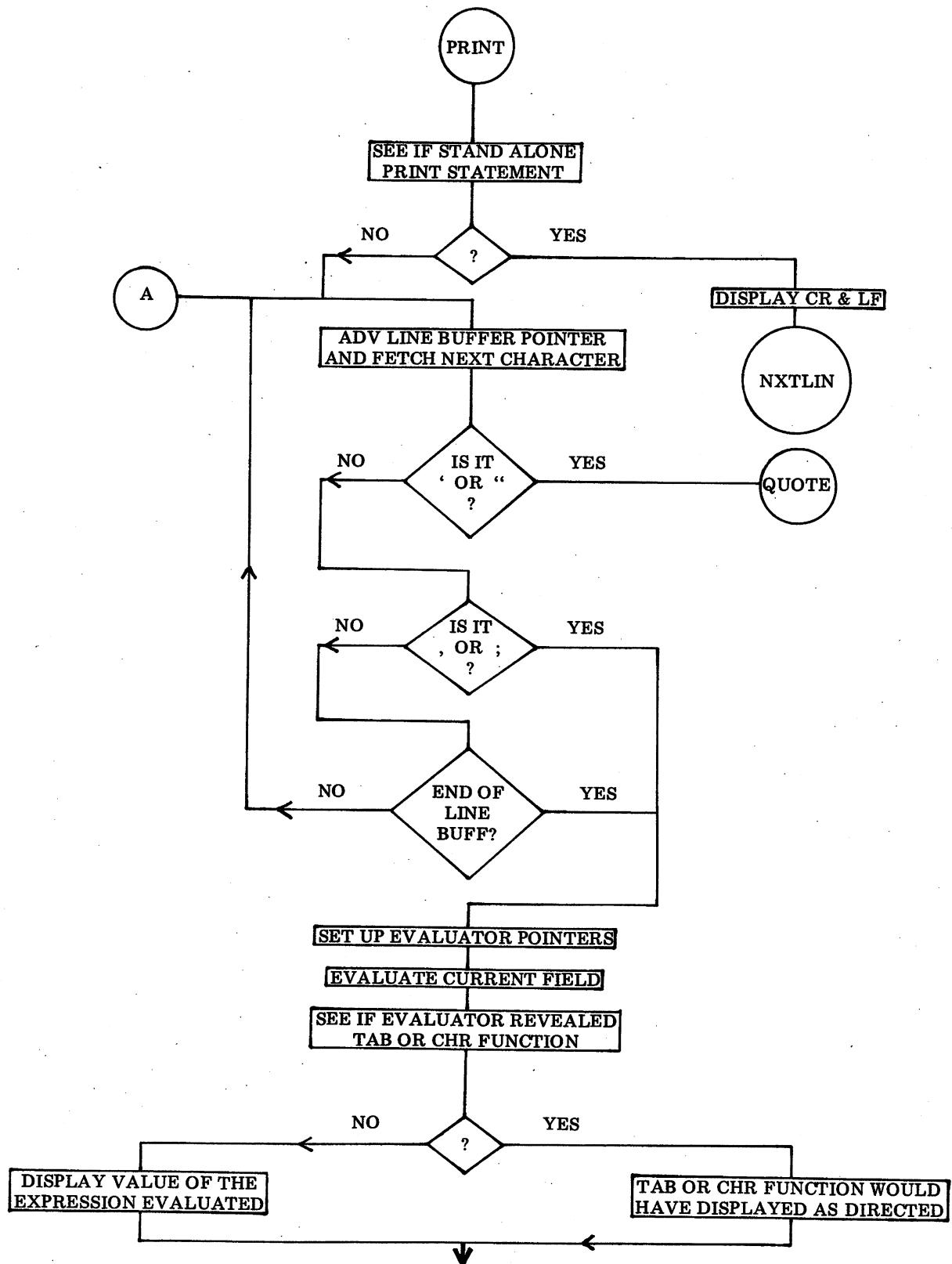
The PRINT routine is split into two major sections. The first section is primarily concerned with determining whether the statement line requires the outputting of text information (enclosed in single or double quotation marks) or the displaying of the value of an expression. If the value of an expression is to be displayed, the program calls on relevant portions of SCELBAL to obtain the value to be outputted and then displays the value. The second section of the PRINT routine starts with the label QUOTE. It is used to display text information enclosed by quotation marks in the PRINT statement line.

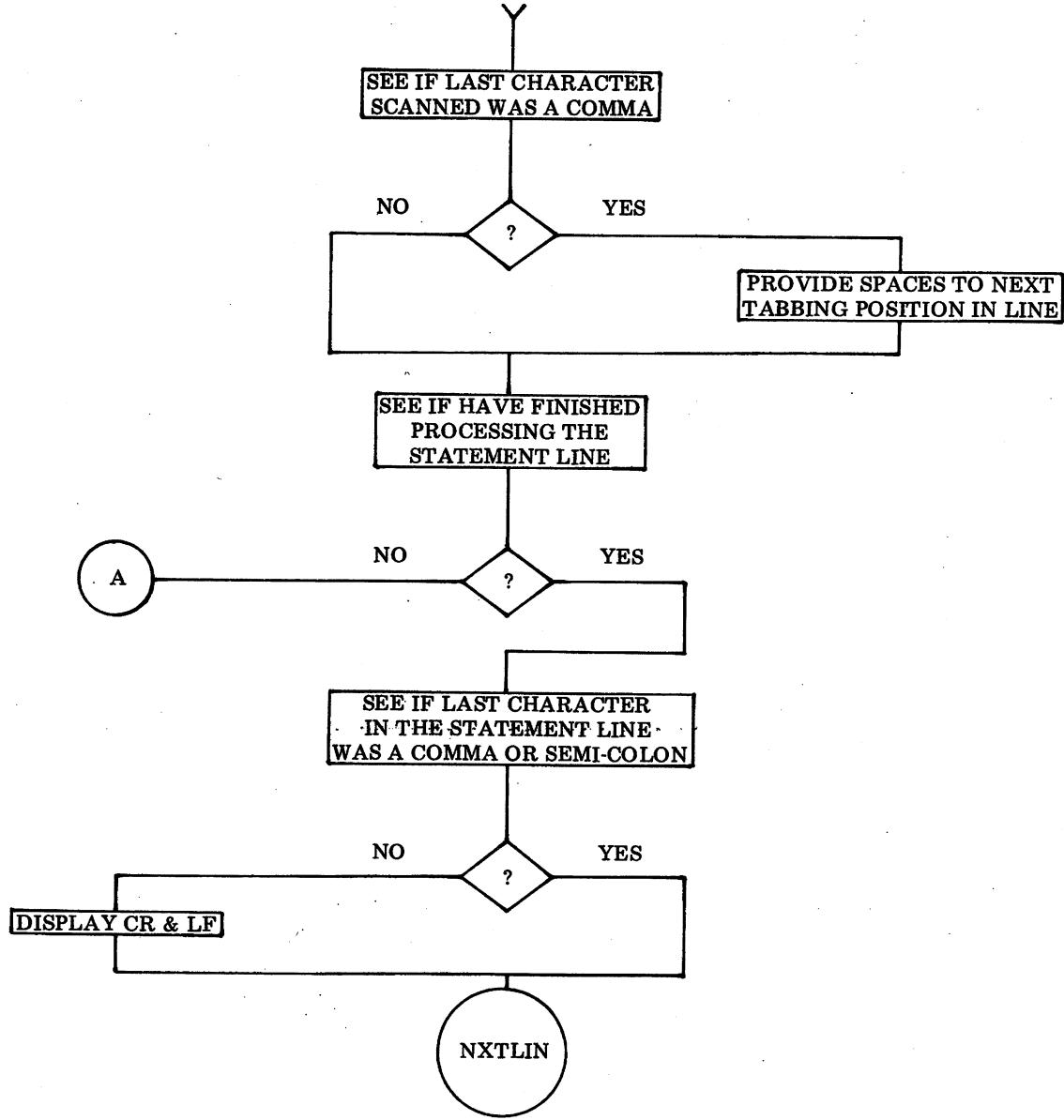
Since a PRINT statement line can contain both expressions and text strings, the routine essentially operates by splitting the line into appropriate fields and processing each field independently, either outputting the value of an expression, or a text string as required.

The flow chart on the next two pages illustrates the key portions of the first section of the PRINT routine. The source listing for this section starts below. The QUOTE portion of the routine is then presented along with its flow chart. The reader may note that the QUOTE portion of the routine may direct program operation back to the first section when it is finished processing a text field. This is indicated in the QUOTE flow chart by the exit point marked A which refers to the A entry point in the PRINT flow chart.

The PRINT routine may at first appear somewhat complicated because a good deal of pointer manipulation is required by the routine as it analyzes fields within a line. Reference to the flow charts will show, though, that its operation is really quite straightforward in concept.

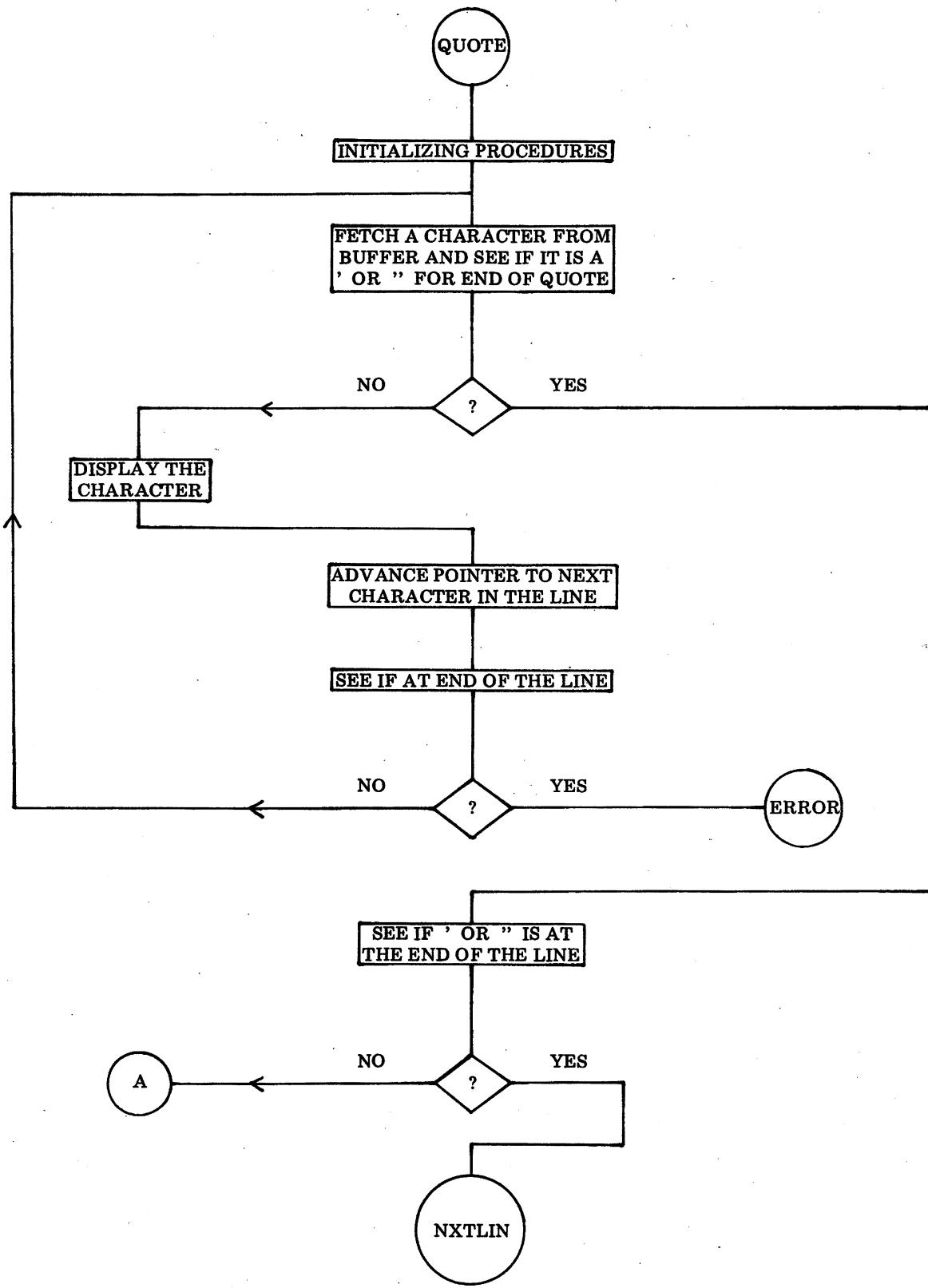
PRINT,	LLI 202 LHI 026 LAM LLI 000 CPM JTS PRINT1 CAL CRLF JMP NXTLIN	Load L with address of SCAN pointer storage location ** Load H with page of SCAN pointer Fetch the pointer value (last character scanned by the SYNTAX routine). Change pointer to line buffer (cc). Compare pointer value to buffer length. If not equal Then line contains more than stand alone PRINT statement. However, if just have PRINT statement then issue A carriage-return & line-feed combination, then exit.
PRINT1,	CAL CLESYM LLI 202 LHI 026 LBM INB LLI 203 LMB	Initialize the SYMBOL buffer for new entry. Load L with address of SCAN buffer pointer ** Load H with page of SCAN pointer Pointer points to last char scanned by SYNTAX. Need To increment it to point to next char in statement line. Load L with address of former TOKEN value. Use it as Storage location for a PRINT statement pointer.
PRINT2,	LLI 203 CAL GETCHR CPI 247	Set memory pointer to PRINT pointer storage location Fetch character in input buffer pointed to by PRINT Pointer. See if it is ASCII code for single quote mark.





JTZ QUOTE	If so, go to QUOTE section to process text string.
CPI 242	If not, see if it is ASCII code for double quote mark.
JTZ QUOTE	If so, go to QUOTE section to process text string.
CPI 254	If not, see if it is ASCII code for comma sign.
JTZ PRINT3	If so, go evaluate expression.
CPI 273	If not, see if it is ASCII code for semi-colon sign.
JTZ PRINT3	If so, go evaluate expression.
LLI 203	Load L with address of PRINT pointer storage location.
CAL LOOP	Increment pointer and test for end of line.
JFZ PRINT2	If not end of line, fetch the next character.

PRINT3,	LLI 202 LBM INB LLI 276 LMB LLI 203 LBM DCB LLI 277 LMB LLI 367 LAM NDA JTZ PRINT4 LMI 000 JMP PRINT6	Load L with address of SCAN pointer storage location Fetch value of the pointer (last letter of KEYWORD) Add one to point to first character of expression Load L with addr of EVAL pointer storage location Store addr at which EVAL should start scanning Load L with address of PRINT pointer Which points to field terminator Decrement pointer value to last character of expression Load L with address of EVAL FINISH pntr storage loc. Place address value of last char in PRINT field there Load L with address of QUOTE flag Fetch the value of the QUOTE flag into the ACC Test the QUOTE flag status If field not quoted, proceed to evaluate expression If field quoted, then clear the QUOTE flag for next field And skip the evaluation procedure
PRINT4,	CAL EVAL LLI 177 LHI 026 LAM NDA LLI 110 LHI 001 LMI 377	Evaluate the current PRINT field Then load L with address of the TAB flag ** Load H with the page of the TAB flag Fetch the value of the TAB flag into the accumulator Test the TAB flag Change L to the FIXED/FLOAT flag location ** Change H to the FIXED/FLOAT flag page Set FIXED/FLOAT flag to fixed point
PRINT5,	CTZ PFPOUT LLI 177 LHI 026 LMI 000	If TAB flag not set, display value of expression Load L with address of TAB flag ** Load H with page of TAB flag Reset TAB flag for next PRINT field
PRINT6,	LLI 203 CAL GETCHR CPI 254 CTZ PCOMMA LLI 203 LHI 026 LBM LLI 202 LMB LLI 000 LAB CPM JTS PRINT1 LLI 000 CAL GETCHR CPI 254 JTZ NXTLIN CPI 273 JTZ NXTLIN CAL CRLF JMP NXTLIN	Load L with address of PRINT pointer storage location Fetch the character pointed to by the PRINT pointer See if the last character scanned was a comma sign If so, then display spaces to next TAB location Reset L to address of PRINT pointer storage location ** Reset H to page of PRINT pointer storage location Fetch the value of the pointer into register B Change L to SCAN pointer storage location Place end of last field processed into SCAN pointer Change pointer to start of line input buffer Place pntr to last char scanned into the accumulator Compare this value to the (cc) for the line buffer If not end of line, continue to process next field If end of line, fetch the last character in the line And check to see if it Was a comma. If it was, go on to the next line in the User program buffer without displaying a CR & LF. If not a comma, check to see if it was a semi-colon. If so, do not provide a CR & LF combination. If not comma or semi-colon, provide CR & LF at end Of a PRINT statement. Go process next line of pgm.



QUOTE,	LLI 367 LMA CAL CLESYM LLI 203 LBM INB LLI 204 LMB	Load L with address of QUOTE flag Store type of quote in flag storage location Initialize the SYMBOL buffer for new entry Load L with address of PRINT pointer Fetch the PRINT pointer into register B Add one to advance over quote character Load L with address of QUOTE pointer Store the beginning of the QUOTE field pointer
QUOTE1,	LLI 204 CAL GETCHR LLI 367 CPM JTZ QUOTE2 CAL ECHO LLI 204 CAL LOOP JFZ QUOTE1	Load L with address of QUOTE pointer Fetch the next character in the TEXT field Load L with the QUOTE flag (type of quote) Compare to see if latest character this quote mark If so, finish up this quote field If not, display the character as part of TEXT Reset L to QUOTE pointer storage location Increment QUOTE pointer and test for end of line If not end of line, continue processing TEXT field
QUOTER,	LAI 311 LCI 321 LLI 367 LHI 026 LMI 000 JMP ERROR	If end of line before closing quote mark have an error So load ACC with I and register C with Q Load L with the address of the QUOTE flag ** Load H with the page of the QUOTE flag Clear the QUOTE flag for future use Go display the IQ (Illegal Quote) error message
QUOTE2,	LLI 204 LBM LLI 202 LMB LAB LLI 000 CPM JFZ PRINT1 CAL CRLF LLI 367 LHI 026 LMI 000 JMP NXTLIN	Load L with address of QUOTE pointer Fetch the QUOTE pointer into register B Load L with address of SCAN pointer storage location Store former QUOTE pointer as start of next field Place QUOTE pointer into the accumulator Change L to point to start of the input line buffer Compare QUOTE pointer value with (cc) value If not end of line, process next PRINT field Else display a CR & LF combination at end of line Load L with the address of the TAB flag ** Load H with the page of the TAB flag Clear the TAB flag for future use Go process next line of the program.
PFPOUT,	LLI 126 LHI 001 LAM NDA JTZ ZERO INL	The following subroutines are utilized by the PRINT routine. Load L with the address of the FPACC MSW (Floating ** Point ACC). Load H with page of the FPACC MSW. Fetch the FPACC MSW into the accumulator. Test to See if the FPACC MSW is zero. If so, then simply go and Display the value "0" Else advance the pointer to the FPACC Exponent

	LAM NDA JTZ FRAC JMP FPOUT	Fetch the FPACC Exponent into the accumulator See if any exponent value. If not, mantissa is in range 0.5 to 1.0. Treat number as a fraction. Else perform regular numerical output routine.
ZERO,	LAI 240 CAL ECHO LAI 260 JMP ECHO	Load ASCII code for space into the ACC Display the space Load ASCII code for 0 into the ACC Display 0 and exit to calling routine
FRAC,	LLI 110 LMI 000 JMP FPOUT	Load L with address of FIXED/FLOAT flag Reset it to indicate floating point mode Display floating point number and return to caller
PCOMMA,	LLI 000 LAM LLI 203 SUM RTS LLI 043 LHI 001 LAM NDI 360 ADI 020 SUM LCA LAI 240	Load L with address of (cc) in line input buffer Fetch the (cc) for the line into the ACC Change pointer to PRINT pointer storage location Subtract value of PRINT pointer from line (cc) If at end of buffer, do not TAB If not end, load L with address of COLUMN COUNTER ** Set H to page of COLUMN COUNTER Fetch COLUMN COUNTER into the accumulator Find the last TAB position (multiple of 16 decimal) Add 16 (decimal) to get new TAB position Subtract current position from next TAB position Store this value in register C as a counter Load the ACC with the ASCII code for space
PCOM1,	CAL ECHO DCC JFZ PCOM1 RET	Display the space Decrement the loop counter Continue displaying spaces until loop counter is zero Then return to calling routine

THE LET STATEMENT ROUTINE

The LET statement is used to set a variable equal to the value of another variable, an expression, or a specific number. This is illustrated by the following examples.

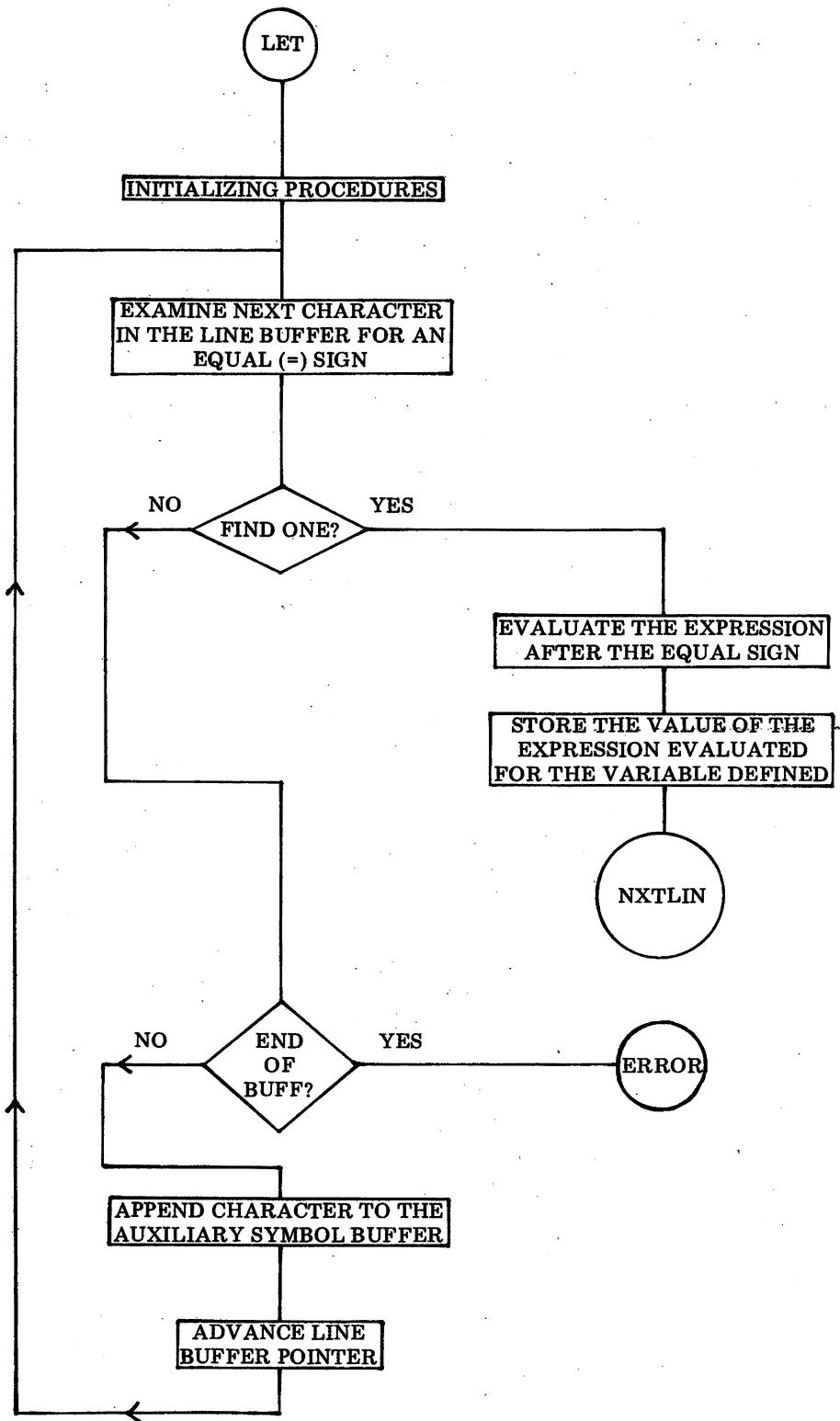
```

LET X = Y
      or
LET X = (Y*2 + 3*Y + 4)*(N - M)
      or
LET X = 3.14159
  
```

The operation of the LET routine simply consists of defining the variable on the left

hand side of the equal sign in a statement line (by defining, it is meant determining what character(s) are being used to represent the variable) and then calculating the value of the expression contained on the right hand side of the equal sign. This value is then stored along with the variable in a variables symbol table.

The operation of the LET statement routine is summarized in the flow chart shown on the next page. The source listing for the routine is then presented.



LET0,	CAL SAVSYM LLI 202 LHI 026 LBM LLI 203 LMB JMP LET5	Entry point for IMPLIED LET statement. Save the Variable (to left of the equal sign). Set L to the SCAN ** Pointer. Set H to the page of the SCAN pointer. Fetch value of SCAN pointer. (Points to = sign in ln bf) Change pointer to LET pointer (was TOKEN value) Place the SCAN pointer value into the LET pointer Continue processing the LET statement line
LET,	CAL CLESYM LLI 144 LHI 026 LMI 000	Initialize the SYMBOL BUFFER for new entry Load L with address of start of AUX SYMBOL BUFF ** Load H with page of AUX SYMBOL BUFFER Initialize AUX SYMBOL BUFFER
LET1,	LLI 202 LHI 026 LBM INB LLI 203 LMB	Entry point for ARRAY IMPLIED LET statement. ** Set pointer to SCAN pointer storage location Fetch the SCAN pointer value (last letter scanned by SYNTAX subroutine) and add one to next character Change L to LET pointer storage location Store former SCAN value (updated) in LET pointer
LET2,	LLI 203 CAL GETCHR JTZ LET4 CPI 275 JTZ LET5 CPI 250 JFZ LET3 CAL ARRAY LLI 206 LHI 026 LBM LLI 203 LMB JMP LET4	Set L to storage location of LET pointer Fetch the character pointed to by the LET pointer If character is a space, ignore it See if character is the equal (=) sign If so, go process other side of the statement (after =) @@ If not, see if character is a right parenthesis "(" If not, continue looking for equal sign @@ If so, have subscript. Call array set up subroutine. @@ Load L with address of ARRAY pointer @@ ** Load H with page of ARRAY pointer @@ Fetch value (points to ")" character of subscript) @@ Load L with address of LET pointer @@ Place ARRAY pointer value as new LET pointer @@ Continue to look for = sign in statement line
LET3,	LLI 144 LHI 026 CAL CONCT1	Reset L to start of AUX SYMBOL BUFFER ** Load H with page of AUX SYMBOL BUFFER Concatenate character to the AUX SYMBOL BUFFER
LET4,	LLI 203 CAL LOOP JFZ LET2	Load L with address of LET pointer storage location Add one to pointer and test for end of line input buffer If not end of line, continue looking for the equal sign
LETERR,	LAI 314 LCI 305 JMP ERROR	If do not find an equal sign in the LET statement line Then have a LE (Let Error). Load the code for L and E Into registers ACC and C and go display the error msg.
LET5,	LLI 203 LHI 026 LBM INB	When find the equal sign, reset L to point to the LET ** Pointer and H to the proper page. Fetch the pointer Value into register B and add one to advance pointer Over the equal sign to first char in the expression.

LLI 276	Set L to point to the address of the EVAL pointer
LMB	Set EVAL pointer to start evaluating right after the
LLI 000	Equal sign. Now change L to start of line input buffer.
LBM	Fetch the (cc) value into register B. (Length of line.)
LLI 277	Load L with EVAL FINISH pointer storage location.
LMB	Set it to stop evaluating at end of the line.
CAL EVAL	Call the subroutine to evaluate the expression.
CAL RESTSY	Restore the name of the variable to receive new value.
CAL STOSYM	Store the new value for the variable in variables table.
JMP NXTLIN	Go process next line of the program.

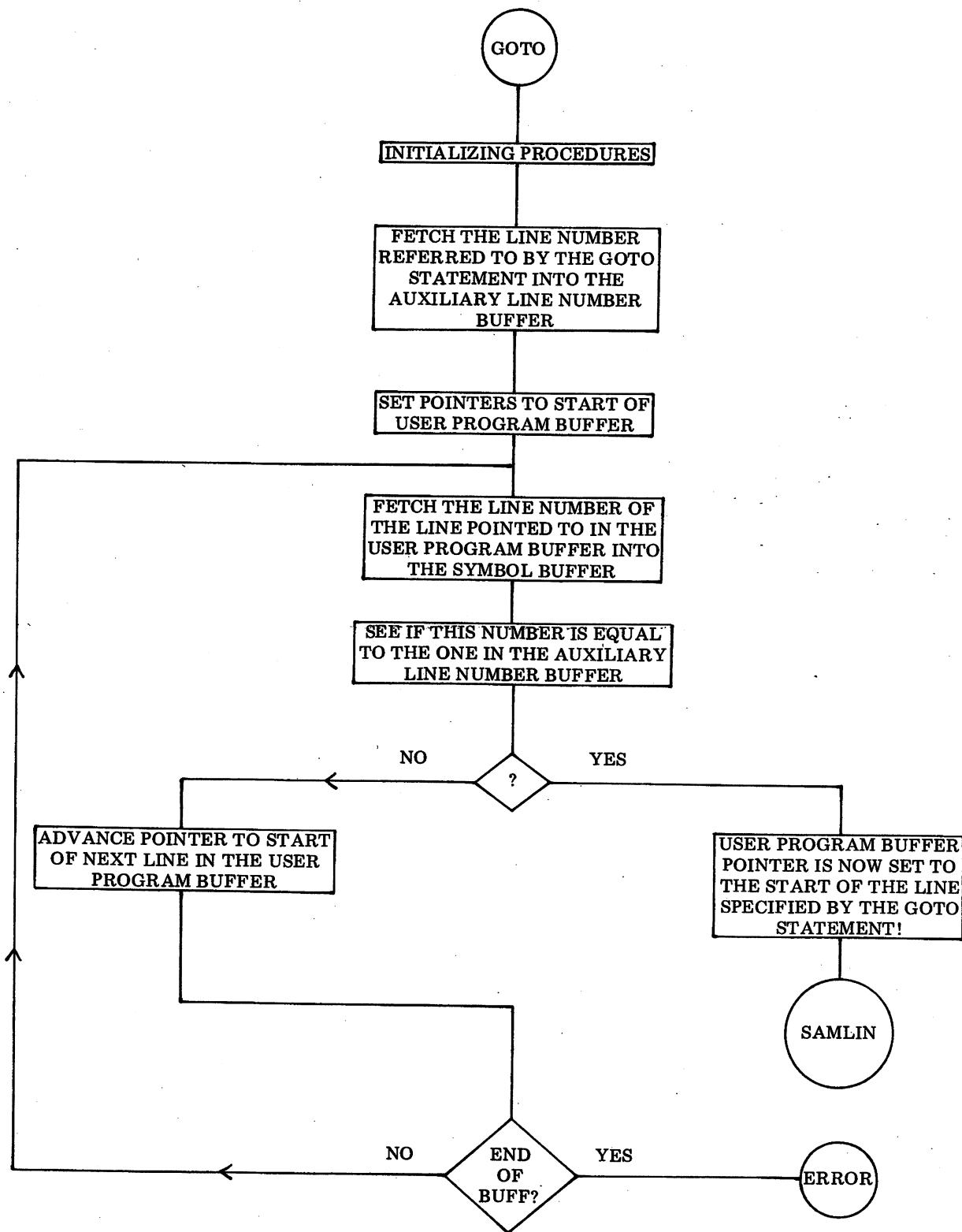
THE GOTO STATEMENT ROUTINE

The GOTO statement is one of the easiest statements to process even though the source listing is somewhat longer than the LET routine just described. The reason for the relatively lengthy source listing is because a lot of pointer manipulation is required. Conceptually, the process involves nothing more than searching the user program buffer for

the line containing the line number specified as part of the GOTO statement. Once it is located, the program simply continues executing the high level program with that line!

The source listing for the GOTO statement is presented below. The reader may correlate it with the flow chart on the next page.

GOTO,	LLI 350	Load L with start of AUX LINE NR BUFFER
	LHI 026	** Load H with page of AUX LINE NR BUFFER
	LMI 000	Initialize the AUX LINE NR BUFFER to zero
	LLI 202	Load L with address of SCAN pointer storage location
	LBM	Fetch pointer value (last char scanned by SYNTAX)
	INB	Add one to skip over the last O in GOTO keyword
	LLI 203	Change pointer to GOTO pointer (formerly TOKEN)
	LMB	Store the updated SCAN pointer as the GOTO pointer
GOTO1,	LLI 203	Load L with address of GOTO pointer
	CAL GETCHR	Fetch the character pointed to by the GOTO pointer
	JTZ GOTO2	If character was a space, ignore it
	CPI 260	See if character is in the range of a decimal digit
	JTS GOTO3	If not, must have end of the line number digit string
	CPI 272	Continue to test for decimal digit
	JFS GOTO3	If not, must have end of the line number digit string
	LLI 350	If valid decimal digit, load L with addr of AUX LINE
	CAL CONCT1	NR BUFFER and concatenate digit to the buffer.
GOTO2,	LLI 203	Reset pointer to GOTO pointer storage location
	CAL LOOP	Advance the pointer value and test for end of line
	JFZ GOTO1	If not end of line, fetch next digit in GOTO line number



GOTO3,	LLI 360 LHI 026 LMI 033 INL LMI 000	Set L to user program buffer pointer storage location ** Set H to page of program buffer pointer †† Initialize high part of pointer to start of pgm buffer Advance the memory pointer Initialize the low part of pointer to start of pgm buffer
GOTO4,	CAL CLESYM LLI 204 LMI 001	Clear the SYMBOL BUFFER Load L with address of GOTO SEARCH pointer Initialize to one for first char of line
GOTO5,	LLI 204 CAL GETCHP JTZ GOTO6 CPI 260 JTS GOTO7 CPI 272 JFS GOTO7 CAL CONCTS	Load L with address of GOTO SEARCH pointer Fetch character pointed to by GOTO SEARCH pointer From line pointed to in user program buffer. Ignore Spaces. Check to see if character is a decimal digit. If not, then have processed line number at the start of The current line. Continue the check for a valid decimal Digit. If have a decimal digit then concatenate the digit Onto the current string in the SYMBOL BUFFER.
GOTO6,	LLI 204 LHI 026 LBM INB LMB LLI 360 LCM INL LLM LHC LAM DCB CPB JFZ GOTO5	Change L to the address of the GOTO SEARCH pointer ** And H to the proper page of the pointer Fetch the GOTO SEARCH pointer value Increment the GOTO SEARCH pointer And restore it back to memory Change L to address of user program buffer pointer Save the high part of this pointer value in register C Advance L to the low part of the pgm buffer pointer Now load it into L And transfer C into H to point to start of the line Fetch the (cc) of the current line being pointed to in the User pgm buff. Decrement B to previous value. Compare GOTO SEARCH pointer value to length of current line. If not end of line then continue getting current line nr.
GOTO7,	LLI 120 LHI 026 LDI 026 LEI 350 CAL STRCP JTZ SAMLIN LLI 360 LHI 026 LDM INL LEM LHD LLE LBM INB CAL ADBDE LLI 360	Load L with address of start of the SYMBOL BUFFER ** Set H to the page of the SYMBOL BUFFER ** Set D to the page of the AUX LINE NR BUFFER Set E to the start of the AUX LINE NR BUFFER Compare GOTO line number against current line nr. If they match, found GOTO line. Pick up ops there! Else, set L to user program buffer pntr storage location ** Set H to page of user program buffer pointer Fetch the high part of this pointer into register D Advance the memory pointer Fetch the low part into register E Transfer the pointer to H And L. Fetch the (cc) of the current line into register B and then add one to account for the (cc) byte to get Total length of the current line in the user pgm buffer Add the total length to the pointer value in D & E To get the starting address of the next line in the user

LHI 026	** User program buffer. Place the new value for the user Program buffer pointer back into the user program Buffer pointer storage locations so that it points to the Next line to be processed in the user program buffer.
LMD	Load L with address of end of user pgm buffer storage Location (page address) and fetch end of buffer page.
INL	Compare this with next line pointer (updated).
LME	If not end of buffer, keep looking for the specified line
LLI 364	If have same page addresses, check the low address
LAD	Portions to see if
CPM	Have reached end of user program buffer
JFZ GOTO4	If not, continue looking. If end of buffer without
INL	
LAE	
CPM	
JFZ GOTO4	
GOTOER,	Finding specified line, then have an error condition.
LAI 325	Load ACC and register C with code for "UN" and go
LCI 316	Display "Undefined Line" error message.
JMP ERROR	

THE IF STATEMENT ROUTINE

The IF statement routine is a little more complicated than most statement routines presented so far. This is because the statement line may take several forms. The typical forms the IF statement may appear in are illustrated here:

IF X = Y + 2 GOTO 120

or

IF X = Y + 2 THEN 120

or

IF X = Y + 2 THEN Z = 3.14159

The first two examples of the IF statement format are relatively straightforward. If the specified condition is not met, the user program simply continues with the next high level statement in the program. If the condition is satisfied, the program simply proceeds directly to the line number specified after the GOTO or THEN directive.

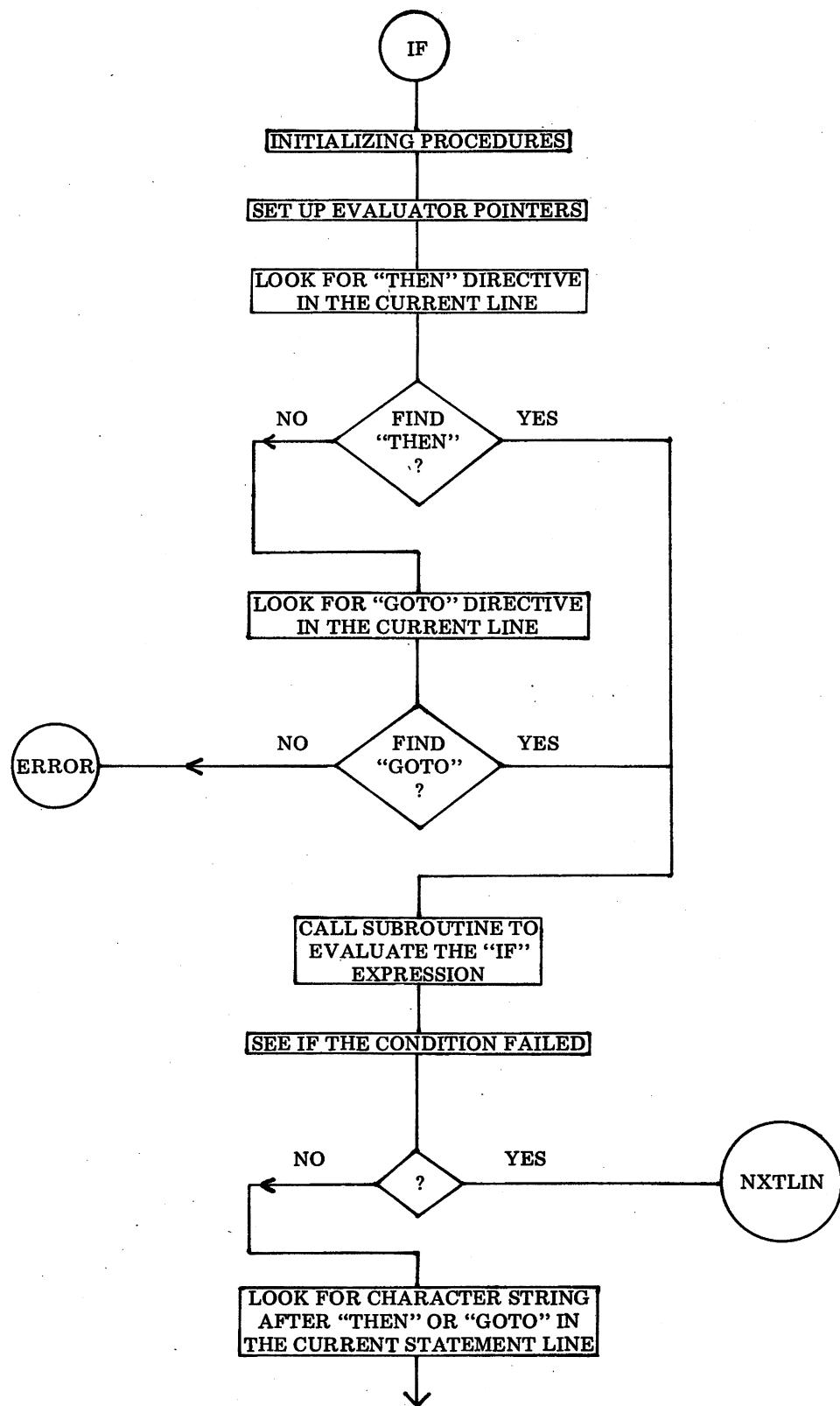
The third example effectively results in a line of the user's high level program contain-

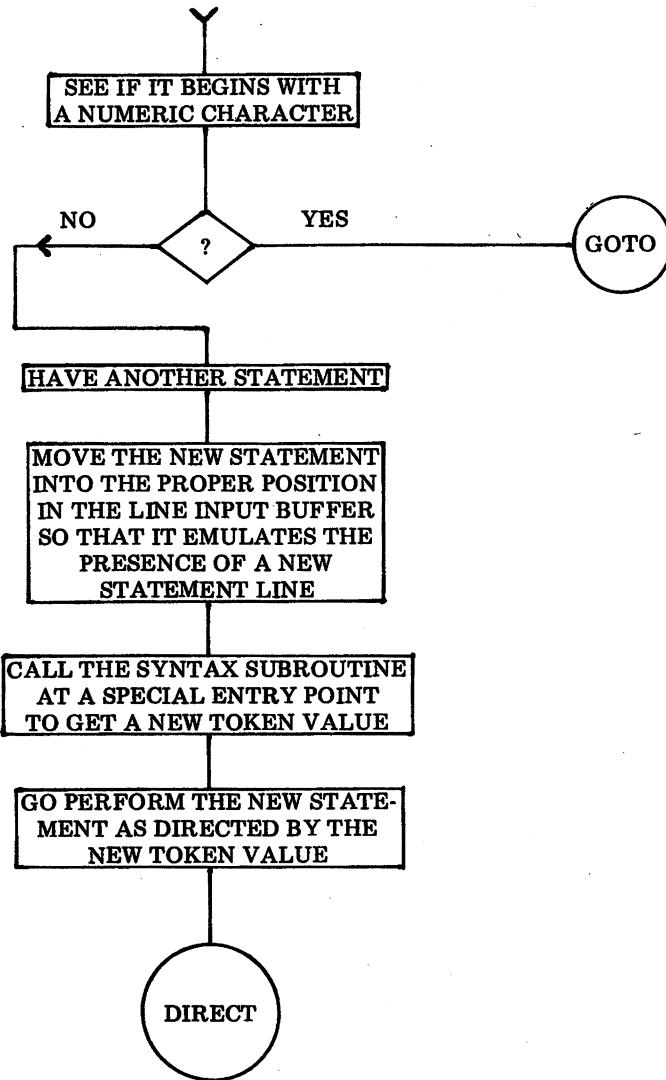
ing two statements. The first statement in the example is the IF directive, the second is an IMPLIED LET provided that the IF condition is satisfied.

It should be noted that the IMPLIED LET part of the line in the example could be replaced by other types of SCELBAL statements.

The processing of an IF statement is outlined in the flow chart shown on the next several pages. The case where a line number follows the THEN or GOTO directive in the statement is handled effectively as a JUMP to the designated line number in the user program buffer. The case where another statement follows the THEN directive is handled as if the program actually was processing a new line of the higher level program except that the line number remains the same as that used for the originating IF statement!

The reader may refer to the flow chart when necessary to understand the operation of this portion of SCELBAL while studying the source listing of the IF statement routine.





IF,	LLI 202	Set L to SCAN pointer storage location.
	LHI 026	** Load H to page of SCAN pointer storage location.
	LBM	Fetch the SCAN pointer value to register B.
	INB	Add one to advance pointer over last char scanned.
	LLI 276	Change L to address of EVAL pointer. Set up EVAL
	LMB	Pointer to begin evaluation with next char in the line.
	CAL CLESYM	Clear the SYMBOL BUFFER.
	LLI 320	Set L to starting address of THEN in look-up table.
	LHI 001	** Set H to page of the look-up table.
	CAL INSTR	Search for occurrence of THEN in the line input buffer.
	LAE	Transfer register E to ACC. If THEN not found
	NDA	The value in E will be zero.
	JFZ IF1	If THEN found, can evaluate the IF expression.
	LLI 013	If THEN not found, set L to starting address of GOTO
	LHI 027	** In the KEYWORD look-up table. Set H to table
	CAL INSTR	Search for occurrence of GOTO in the line input buffer.

	LAE NDA JFZ IF1	Transfer E to ACC. If GOTO not found The value in E will be zero. If GOTO found, can evaluate the IF expression.
IFERR,	LAI 311 LCI 306 JMP ERROR	Set ASCII code for letter I in ACC And code for letter F in register C Go display the IF error message
IF1,	LLI 277 LHI 026 DCE LME CAL EVAL LLI 126 LHI 001 LAM NDA JTZ NXTLIN LLI 277 LHI 026 LAM ADI 005 LLI 202 LMA LBA INB LLI 204 LMB	Load L with addr of EVAL FINISH pointer storage loc ** Load H with page of storage location Subtract one from pointer in E and set the EVAL FINISH pointer so that it will evaluate up to the THEN Or GOTO directive. Evaluate the expression. Load L with address of FPACC Most Significant Word ** Load H with page of FPACC MSW Fetch the FPACC MSW into the accumulator Test the value of the FPACC MSW If it is zero, IF condition failed, ignore rest of line. If not, load L with addr of EVAL FINISH pointer ** Set H to the appropriate page Fetch the value in the EVAL FINISH pointer Add five to skip over THEN or GOTO directive Change L to SCAN pointer storage location Set up the SCAN pointer to location after THEN or GOTO directive. Also put this value in register B. Add one to the value in B to point to next character After THEN or GOTO. Change L to addr of THEN pnter Storage location and store the pointer value.
IF2,	LLI 204 CAL GETCHR JFZ IF3 LLI 204 CAL LOOP JFZ IF2 JMP IFERR	Load L with the address of the THEN pointer Fetch the character pointed to by the THEN pointer If character is not a space, exit this loop If fetch a space, ignore. Reset L to the THEN pointer Add one to the THEN pointer and test for end of line If not end of line, keep looking for a character other Than a space. If reach end of line first, then error
IF3,	CPI 260 JTS IF4 CPI 272 JTS GOTO	When find a character see if it is numeric. If not numeric, then should have a new type of Statement. If numeric, then should have a line number. So process as though have a GOTO statement!
IF4,	LLI 000 LAM LLI 204 SUM LBA INB LCM LLI 000 LMB	Load L with addr of start of line input buffer. Fetch the (cc) byte to get length of line value. Change L to current value of THEN pointer (where first Non-space char. found after THEN or GOTO). Subtract This value from length of line to get remainder. Now Have length of second statement portion. Add one for (cc) count. Save THEN pointer value in register C. Reset L to start of line input buffer. Now put length of Second statement into (cc) position of input buffer.

LLC	Set L to where second statement starts.
LDI 026	** Set D to page of line input buffer.
LEI 001	Set E to first character position of line input buffer.
CAL MOVEIT	Move the second statement up in line to become first!
LLI 202	Load L with address of new SCAN pointer. Load
LMI 001	It with starting position for SYNTAX scan.
CAL SYNTX4	Use special entry to SYNTAX to get new TOKEN value.
JMP DIRECT	Process the second statement in the original line.

THE GOSUB STATEMENT ROUTINE

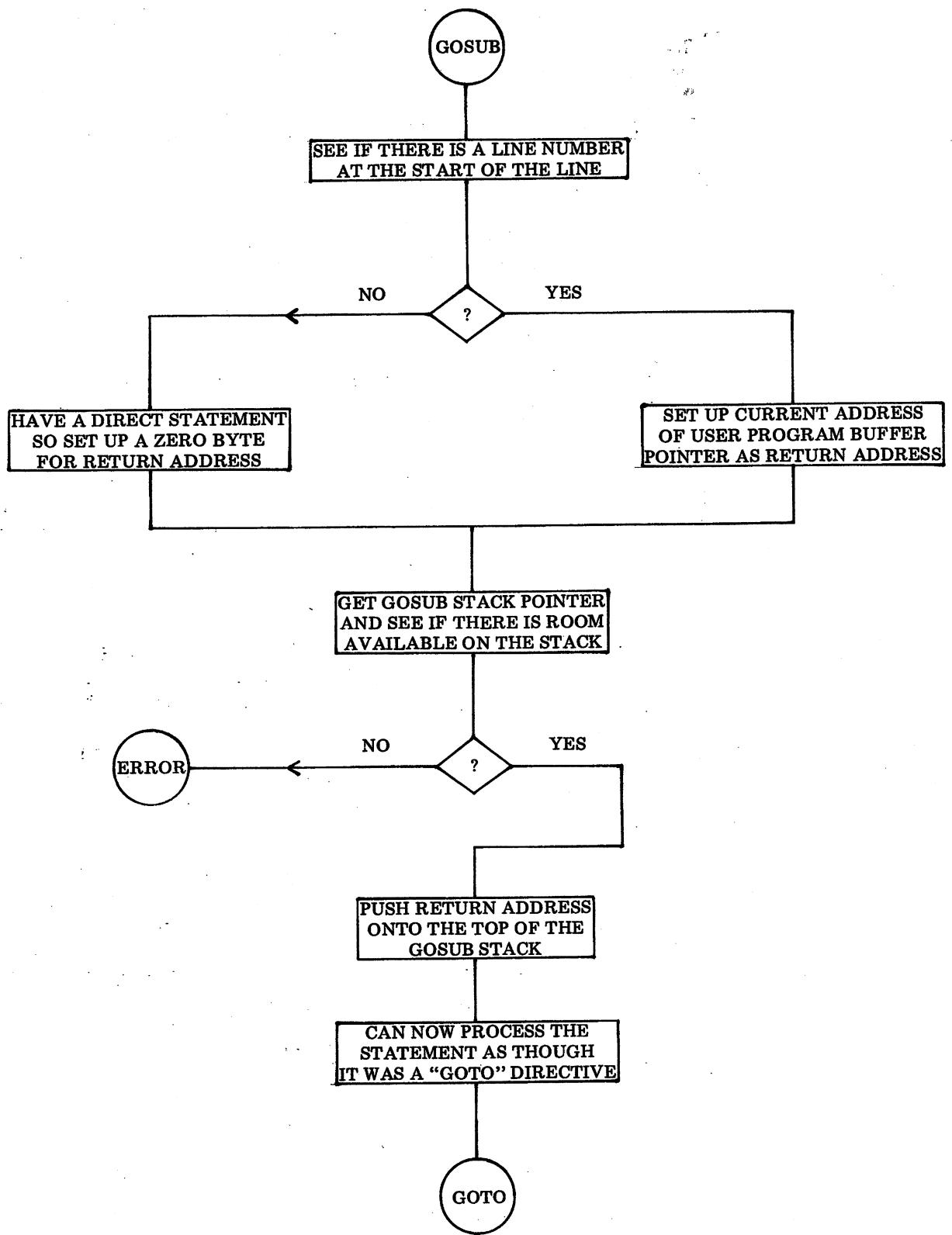
The GOSUB statement routine creates a software STACK so that the high level program can return, after executing the subroutine, to the next line in the user program buffer following the GOSUB statement. The software stack created is merely a group of locations in memory where addresses are stored and a stack pointer system that indicates what position in the stack is in use. The software stack utilized for GOSUB statements has enough room reserved in it to nest GOSUB statements up to eight levels.

The GOSUB software stack operates in a push-down manner. Each time a GOSUB statement is encountered, the current address of the user program buffer line pointer is placed on the top of the stack, with any

previous addresses on the stack being pushed down. The RETURN statement, to be discussed shortly, causes the reverse to occur. The address on the top of the stack is removed (as the returning address) and any remaining addresses on the stack are popped up.

The GOSUB flow chart on the following page illustrates the procedure followed when a GOSUB statement is encountered. Once the current user program buffer line pointer has been placed on the GOSUB stack, the GOSUB directive is handled as an effective GOTO statement. This use of the GOTO routine already presented, to complete the GOSUB process, makes the source listing for the GOSUB routine quite short as illustrated below.

GOSUB,	LLI 340 LHI 026 LDM IND DCD JTZ GOSUB1 LLI 360 LDM INL LEM	Load L with start of LINE NUMBER BUFFER ** Load H with page of LINE NUMBER BUFFER Fetch (cc) of current line number into register D Test contents of register by first incrementing And then decrementing the value in the register If no line number, then processing a DIRECT statement Else, load L with address of user pgm buff line pointer Fetch high value (page) of pgm line pointer to D Advance the memory pointer Fetch the low part of pgm line pointer to E
GOSUB1,	LLI 073 LHI 027 LAM ADI 002 CPI 021	Set L to address of GOSUB STACK POINTER ** Set H to page of GOSUB STACK POINTER Fetch value in GOSUB stack pointer to ACC Add two to current stack pointer for new data to be Placed on the stack and see if stack overflows



JFS GOSERR	If stack filled, have an error condition
LMA	Else, store updated stack pointer
LLI 076	Load L with address of start of stack less offset (2)
ADL	Add GOSUB stack pointer to base address
LLA	To get pointer to top of stack (page byte)
LMD	Store page part of pgm buffer line pointer in stack
INL	Advance pointer to next byte in stack
LME	Store low part of pgm buffer line pointer in stack
JMP GOTO	Proceed from here as though processing a GOTO

THE RETURN STATEMENT ROUTINE

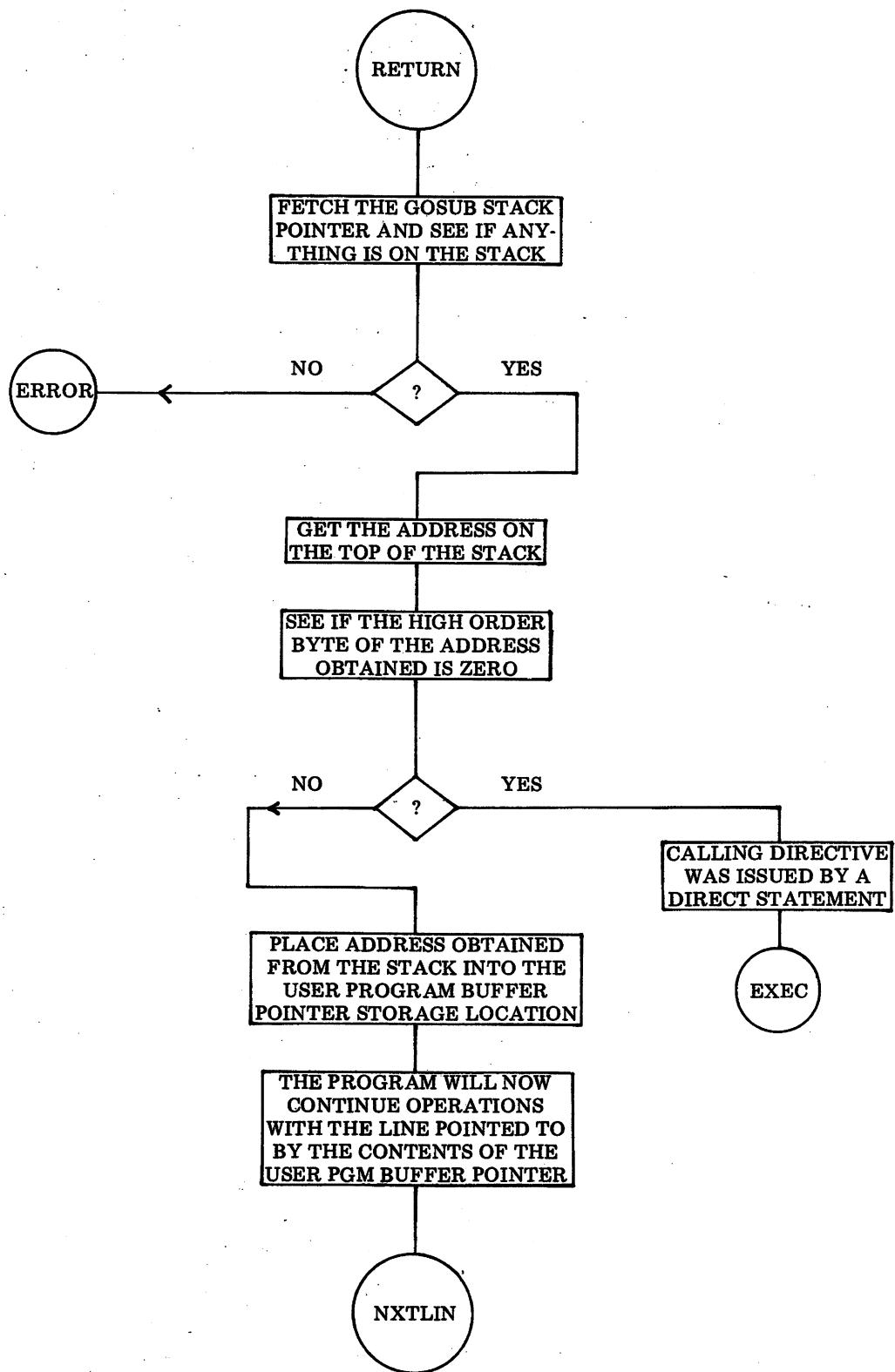
The RETURN statement routine takes the address residing on the top of the GOSUB stack just discussed and places it in the user program buffer line pointer. This operation will cause the high level program to continue with the next statement following the original GOSUB directive. Any remaining addresses on the GOSUB stack are popped up, as

mentioned in the discussion of the GOSUB statement, so that nested subroutines may be properly handled.

The flow chart provided on the next page illustrates the RETURN statement execution process. The source listing for this short routine is presented below.

RETURN, LLI 073	Set L to address of GOSUB STACK POINTER
LHI 027	** Set H to page of GOSUB STACK POINTER
LAM	Fetch the value of GOSUB stack pointer to ACC
SUI 002	Subtract two for data to be removed from stack
JTS RETERR	If stack underflow, then have an error condition
LMA	Restore new stack pointer to memory
ADI 002	Add two to point to previous top of stack
LLI 076	Load L with address of start of GOSUB stack less two
ADL	Add address of previous top of stack to base value
LLA	Set pointer to high address value in the stack
LDM	Fetch the high address value from stack to register D
IND	Exercise the register contents to see if high address
DCD	Obtained is zero. If so, original GOSUB statement was
JTZ EXEC	A DIRECT statement. Must return to Executive!
INL	Else, advance pointer to get low address value from the
LEM	Stack into CPU register E.
LLI 360	Load L with address of user pgm line pointer storage
LHI 026	** Location. Load H with page of user pgm line pnt.
LMD	Put high address from stack into pgm line pointer.
INL	Advance the memory pointer
LME	Put low address from stack into pgm line pointer.
JMP NXTLIN	Execute the next line after originating GOSUB line!

(Two short error routines used by the GOSUB and RETURN routines are shown following the flow chart.)



GO\$ERR,	LAI 307 LCI 323 JMP ERROR	Load ASCII code for letter G into accumulator Load ASCII code for letter S into register C Go display GoSub (GS) error message.
RE\$ERR,	LAI 322 LCI 324 JMP ERROR	Load ASCII code for letter R into accumulator Load ASCII code for letter T into register C Go display ReTurn (RT) error message.

THE INPUT STATEMENT ROUTINE

The INPUT statement routine is used to input the values for user defined variables during the operation of a high level program from the system's input device such as a keyboard. The values that are inputted from the operator are then stored in the variables symbol table.

The flow chart on the following page illustrates the essential operation of the statement routine. However, not illustrated in the flow chart is the fact that the INPUT statement routine has a special capability that is essentially the reverse of the CHR function. The CHR function was mentioned in the discussion of the PRINT statement and will be detailed in a later chapter.

The reverse of the CHR function is the capability to accept a character from an input device and convert the character to a numerical value corresponding to its ASCII code (in decimal for SCELBAL).

When a programmer using SCELBAL wants to have the operator enter a character as an input for a variable value, a dollar sign (\$) must be placed immediately after the variable in the statement directive. Thus:

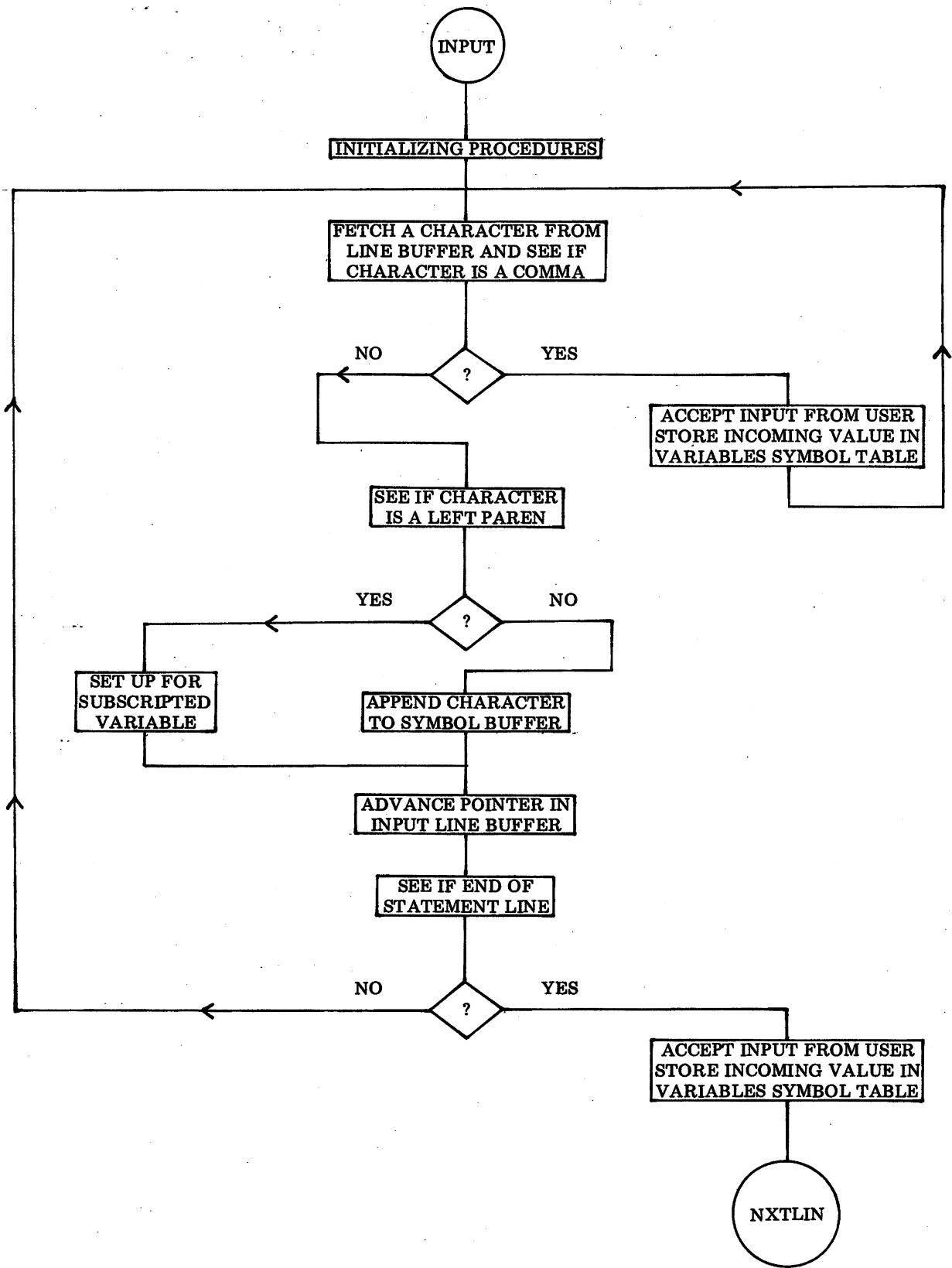
INPUT A\$,B,C,D\$

as an INPUT statement would mean that the variables B and C were to be entered as numerical values, while variables A and D were to be entered as alphanumeric characters (which will then be converted to numerical values according to their ASCII code equivalents).

When the INPUT statement routine is processing the statement line, it checks to see if the last character of each variable is a dollar sign. If so, the routine converts the character inputted by the operator for the variable value to its decimal ASCII code numerical value. That numerical value thus becomes the value assigned to the variable. If the dollar sign is not present as the last character of a variable, then the operator input is assumed to represent the actual numerical value entered.

This special capability is provided in the portion of the INPUT statement routine labeled INPUTX. The source listing which follows illustrates that the capability is a small subset of the fundamental INPUT statement routine. Hence, it is not highlighted in the flow chart.

INPUT,	CAL CLESYM LLI 202 LBM INB LLI 203 LMB	Clear the SYMBOL BUFFER Load L with address of SCAN pointer storage location Fetch value of SCAN pointer to register B Increment value to point to next character Change L to point to INPUT pointer (formerly TOKEN) Updated SCAN pointer becomes INPUT pointer
--------	---	---



INPUT1,	LLI 203 CAL GETCHR JTZ INPUT3 CPI 254 JTZ INPUT4 CPI 250 JFZ INPUT2 CAL ARRAY2 LLI 206 LHI 026 LBM LLI 203 LMB JMP INPUT3	Load L with address of INPUT pointer Fetch a character from the line input buffer If character is a space, ignore it. Else, See if character is a comma. If so, process the Variable that precedes the comma. If not, see if character is a left parenthesis. If not, continue processing to build up symbolic variable @@ If so, call array subscripting subroutine @@ Load L with address of array set up pointer @@ ** Load H with page of array set up pointer @@ Fetch pointer value (point to "(" of subscript) @@ Change pointer to address of INPUT pointer @@ Update INPUT pointer @@ Jump over concatenate instruction below
INPUT2,	CAL CONCTS	Concatenate character to SYMBOL BUFFER
INPUT3,	LLI 203 CAL LOOP JFZ INPUT1 CAL INPUTX CAL STOSYM JMP NXTLIN	Load L with address of INPUT pointer Increment INPUT pointer and test for end of line If not end of line, go get next character If end of buffer, get input for variable in the SYMBOL BUFFER and store the value in the VARIABLES table Then continue to interpret next statement line
INPUT4,	CAL INPUTX CAL STOSYM LHI 026 LLI 203 LBM LLI 202 LMB JMP INPUT	Get input from user for variable in SYMBOL BUFFER Store the inputted value in the VARIABLES table ** Set H to page of INPUT pointer Set L to location of INPUT pointer Fetch pointer value for last character examined Change L to point to SCAN pointer storage location Update the SCAN pointer Continue processing statement line for next variable
INPUTX,	LLI 120 LAM ADL LLA LAM CPI 244 JFZ INPUTN LLI 120 LBM DCB LMB CAL FP0 CAL CINPUT LLI 124 LMA JMP FPFLT	Load L with start of SYMBOL BUFFER (contains cc) Fetch the (cc) (length of symbol in the buffer) to ACC Add (cc) to base address to set up Pointer to last character in the SYMBOL BUFFER Fetch the last character in the SYMBOL BUFFER See if the last character was a \$ sign If not a \$ sign, get variable value as a numerical entry If \$ sign, reset L to start of the SYMBOL BUFFER Fetch the (cc) for the variable in the SYMBOL BUFF Subtract one from (cc) to chop off the \$ sign Restore the new (cc) for the SYMBOL BUFFER Call subroutine to zero the floating point accumulator Input one character from system input device Load L with address of the LSW of the FPACC Place the ASCII code for the character inputted there Convert value to floating point format in FPACC

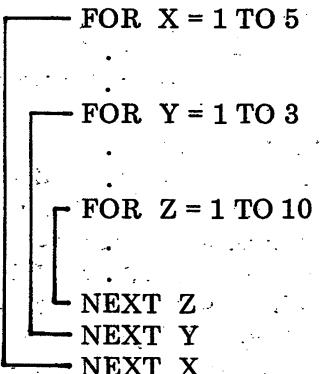
INPUTN,	LLI 144 LHI 026 LAI 277 CAL ECHO CAL STRIN JMP DINPUT	Load L with address of start of AUX SYMBOL BUFF ** Load H with page of AUX SYMBOL BUFFER Load accumulator with ASCII code for ? mark Call output subroutine to display the ? mark Input string of characters (number) fm input device Convert decimal string into binary floating point nr.
FPO,	LHI 001 JMP CFALSE	** Load H with floating point working registers page Zero the floating point accumulator & exit to caller

THE FOR STATEMENT ROUTINE

The FOR statement routine actually only performs part of the tasks related to the statement. The NEXT statement routine, which will be described in the following section, performs the major portion of the operations using the data entered on the FOR statement line.

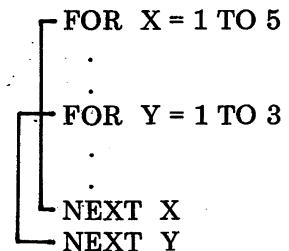
The use of the combination of the FOR and NEXT statements permits the high level language programmer to form iterative programming loops. These statements must always be used in pairs. The FOR statement initiates an iterative loop. The NEXT statement ends the loop. Statements in between a FOR and a NEXT statement may be used to perform desired operations.

FOR/NEXT loops may be nested one inside another provided that the nesting occurs in the following fashion.



In other words, the deepest loop must

be closed out by a NEXT statement first! Attempting to nest loops in the following manner:



will result in an error condition.

In order to allow for the nesting of FOR/NEXT loops, a FOR/NEXT STACK implemented by software is maintained similar in operation (push-down, pop-up) to the software stack established for GOSUB/RETURN statements. However, the FOR/NEXT stack requires four bytes for each nested loop. Two bytes are used to store the address of the user program buffer line pointer when a FOR statement is encountered, and two are used to store the symbolic name of the variable which is iterated. (Remember, the GOSUB/RETURN stack just required two bytes per statement. These were used to store the address of the GOSUB statement that initiated the subroutine call operation.)

Room has been provided in one of the special pointer/counters/look-up table pages used in SCELBAL for a FOR/NEXT stack area that will allow nesting of FOR/NEXT

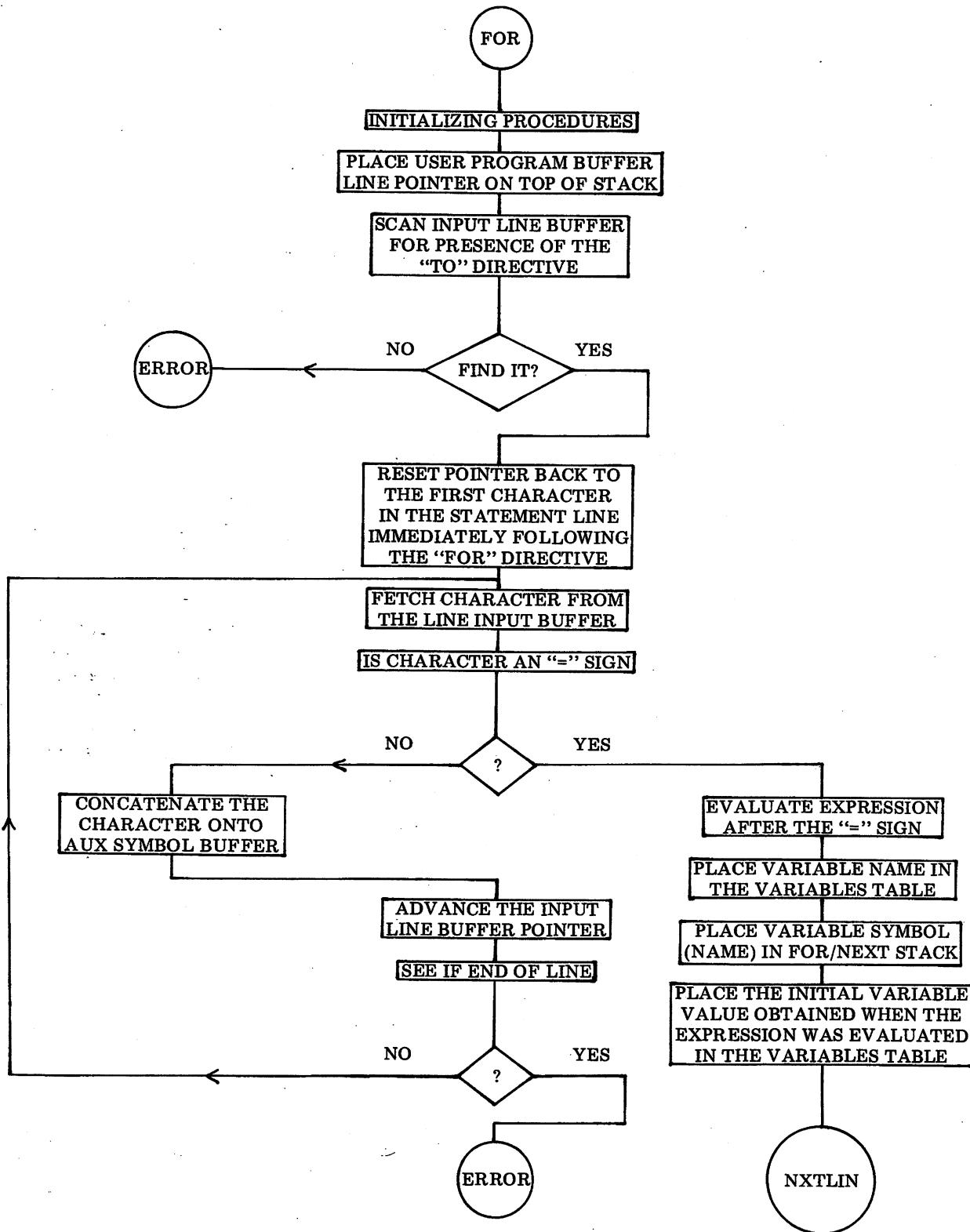
loops up to eight levels. A stack pointer is used to point to the proper locations in the stack area as a function of the nesting level at any given time.

The flow chart presented on the following

page illustrates that the major function of the FOR statement routine is to place the appropriate information on the FOR/NEXT stack.

The source listing for the routine starts below.

FOR,	LLI 144 LHI 026 LMI 000 LLI 146 LMI 000 LLI 205 LHI 027 LBM INB LMB LLI 360 LHI 026 LDM INL LEM LAB RLC RLC ADI 134 LLA LHI 027 LMD INL LME LLI 325 LHI 001 CAL INSTR LAE NDA JFZ FOR1	Load L with address of AUX SYMBOL BUFFER ** Load H with page of AUX SYMBOL BUFFER Initialize buffer by clearing first byte Load L with location of second character in buffer Clear that location in case of single character variable Load L with address of FOR/NEXT STACK pointer ** Load H with page of FOR/NEXT STACK pointer Fetch the FOR/NEXT STACK pointer Increment it in preparation for pushing operation Restore it back to its storage location Load L with address of user pgm buffer line pointer ** Set H to page of line pointer Fetch page address of pgm buffer line pntr into D Advance the memory pointer to pick up low part Fetch low address of pgm buffer line pntr into E Restore updated FOR/NEXT STACK pointer to ACC Rotate it left to multiply by two, then rotate it again to Multiply by four. Add this value to the base address of The FOR/NEXT STACK to point to the new top of The FOR/NEXT STACK and set up to point to stack ** Set H for page of the FOR/NEXT STACK Store the page portion of the user pgm buffer line pntr In the FOR/NEXT STACK, advance register L, then Store the low portion of the pgm line pntr on the stack Change L to point to start of TO string which is stored ** In a text strings storage area on this page Search the statement line for the occurrence of TO Register E will be zero if TO not found. Move E to ACC To make a test. If TO found then proceed with FOR statement
FORERR,	LAI 306 LCI 305 JMP ERROR	Else have a For Error. Load ACC with ASCII code for Letter F and register C with code for letter E. Then go display the FE message.
FOR1,	LLI 202 LHI 026 LBM INB LLI 204 LMB LLI 203 LME	Load L with address of SCAN pointer storage location ** Set H to page of the SCAN pointer Fetch pointer value to ACC (points to letter R in the For directive). Increment it to point to next character In the line. Change register L and set this value up As an updated FOR pointer. Set L to address of TO pointer (formerly TOKEN) Save pointer to TO in the TO pointer!



FOR2,	LLI 204 CAL GETCHR JTZ FOR3 CPI 275 JTZ FOR4 LLI 144 CAL CONCT1	Load L with address of the FOR pointer Fetch a character from the statement line If it is a space, ignore it Test to see if character is the “=” sign If so, variable name is in the AUX SYMBOL BUFFER If not, then set L to point to start of the AUX SYMBOL BUFFER and concatenate the character onto the buffer
FOR3,	LLI 204 CAL LOOP JFZ FOR2 JMP FORERR	Reset L to address of the FOR pointer Increment the pointer and see if end of line If not end of line, continue looking for the “=” sign If reach end of line before “=” sign, then have error
FOR4,	LLI 204 LBM INB LLI 276 LMB LLI 203 LBM DCB LLI 277 LMB CAL EVAL CAL RESTSY LLI 144 LHI 026 LAM CPI 001 JFZ FOR5 LLI 146 LMI 000 JMP FOR5	Set L with address of the FOR pointer Fetch pointer value to ACC (pointing to “=” sign) Increment it to skip over the “=” sign Set L to address of the EVAL pointer Restore the updated pointer to storage Set L to the address of the TO pointer Fetch pointer value to ACC (pointing to letter T in TO) Decrement it to point to character before the T in TO Set L to EVAL FINISH pointer storage location Store the EVAL FINISH pointer value Evaluate the expression between the “=” sign and TO Directive. Place the variable name in the variables table. Load L with starting address of the AUX SYMBOL BF ** Load H with the page of the AUX SYMBOL BUFF Fetch the (cc) for the name in the buffer See if the symbol (name) length is just one character If not, go directly to place name in FOR/NEXT STACK If so, set L to point to second character location in the AUX SYMBOL BUFFER and set it equal to zero. This jump directs program over pntrs/cntrs/table area
FOR5,	LLI 205 LHI 027 LAM RLC RLC ADI 136 LEA LDH LLI 145 LHI 026 LBI 002 CAL MOVEIT CAL STOSYM JMP NXTLIN	Load L with address of the FOR/NEXT STACK pointer ** Load H with page of the FOR/NEXT STACK pntr Fetch the stack pointer to the ACC. Rotate it left to multiply by two, then rotate it again to Multiply by four. Add this value to the base address Plus two of the base address to point to the next part of The FOR/NEXT STACK. Place this value in register E. Set D to the FOR/NEXT STACK area page. Load L with the address of the first character in the ** AUX SYMBOL BUFFER and set up H to this page. Set up register B as a number of bytes to move counter. Move the variable name into the FOR/NEXT STACK. Store initial variable value in the VARIABLES TABLE. Continue with next line in user program buffer.

THE NEXT STATEMENT ROUTINE

The NEXT statement routine is the work horse portion of the FOR/NEXT combination. As indicated in the preceding section, the statement types must always appear in pairs in a high level program. When a NEXT statement is used it must be followed (in the statement line) by the identifying variable that associates it with an originating FOR statement.

The flow chart on the next several pages illustrates the essential operations of the NEXT statement. This flow chart is amplified by the following discussion.

The first thing the NEXT statement routine accomplishes is to go to the FOR/NEXT stack to obtain the starting address of the associated FOR statement line in the user program buffer. As a check for proper FOR/NEXT nesting, a test is made to see if the variable in the FOR statement line pointed to by the entry in the FOR/NEXT stack is the same as that specified in the NEXT statement being processed. If not, improper FOR/NEXT nesting has been attempted.

The NEXT statement routine then proceeds to process the information on the originating FOR statement line. Remember, the originating FOR statement line contains the the variable range (limit) and step size for the FOR/NEXT loop being processed.

A FOR statement may be formatted in one of two possible ways. The statement:

FOR X = 1 TO 5

represents an IMPLIED STEP SIZE. That is, since no STEP size is specified, the statement is to be interpreted as having an implied value of 1.0.

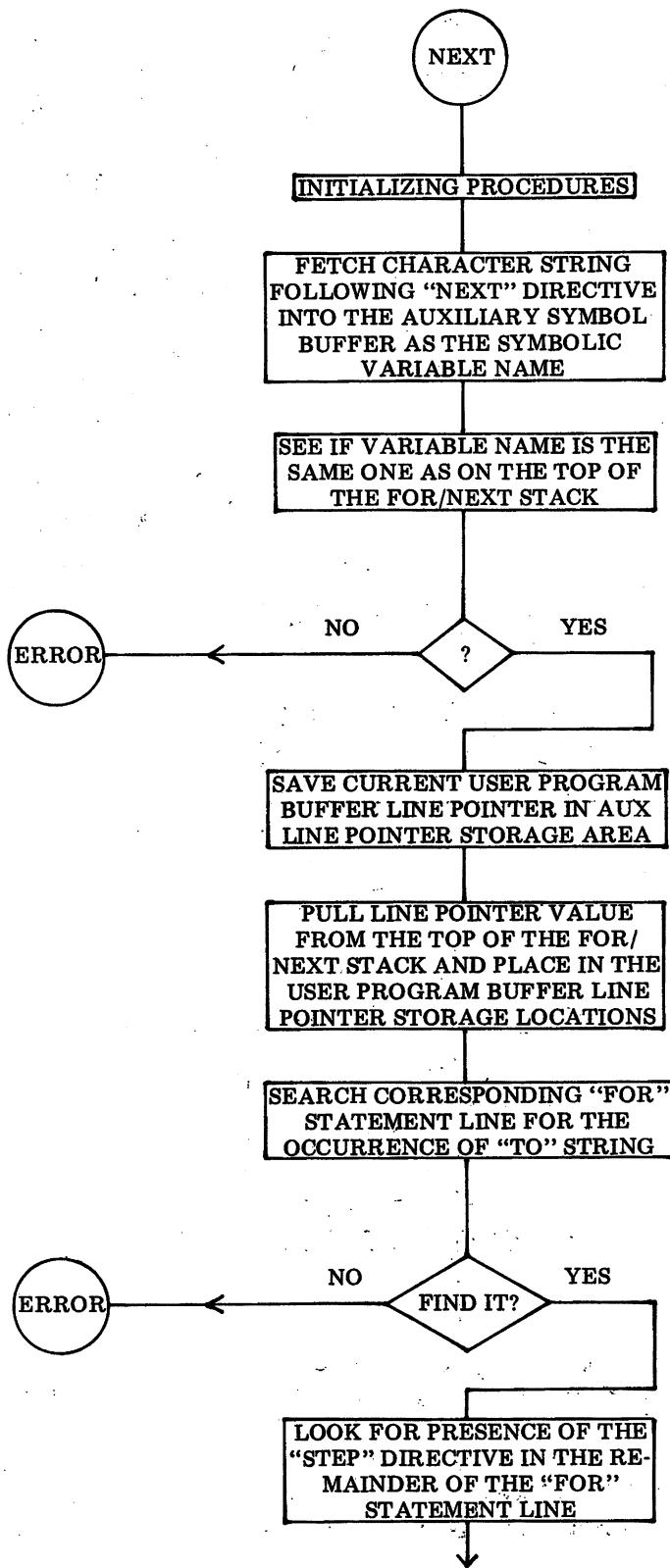
If desired, the high level language programmer may specify a STEP size in a FOR statement such as in the example:

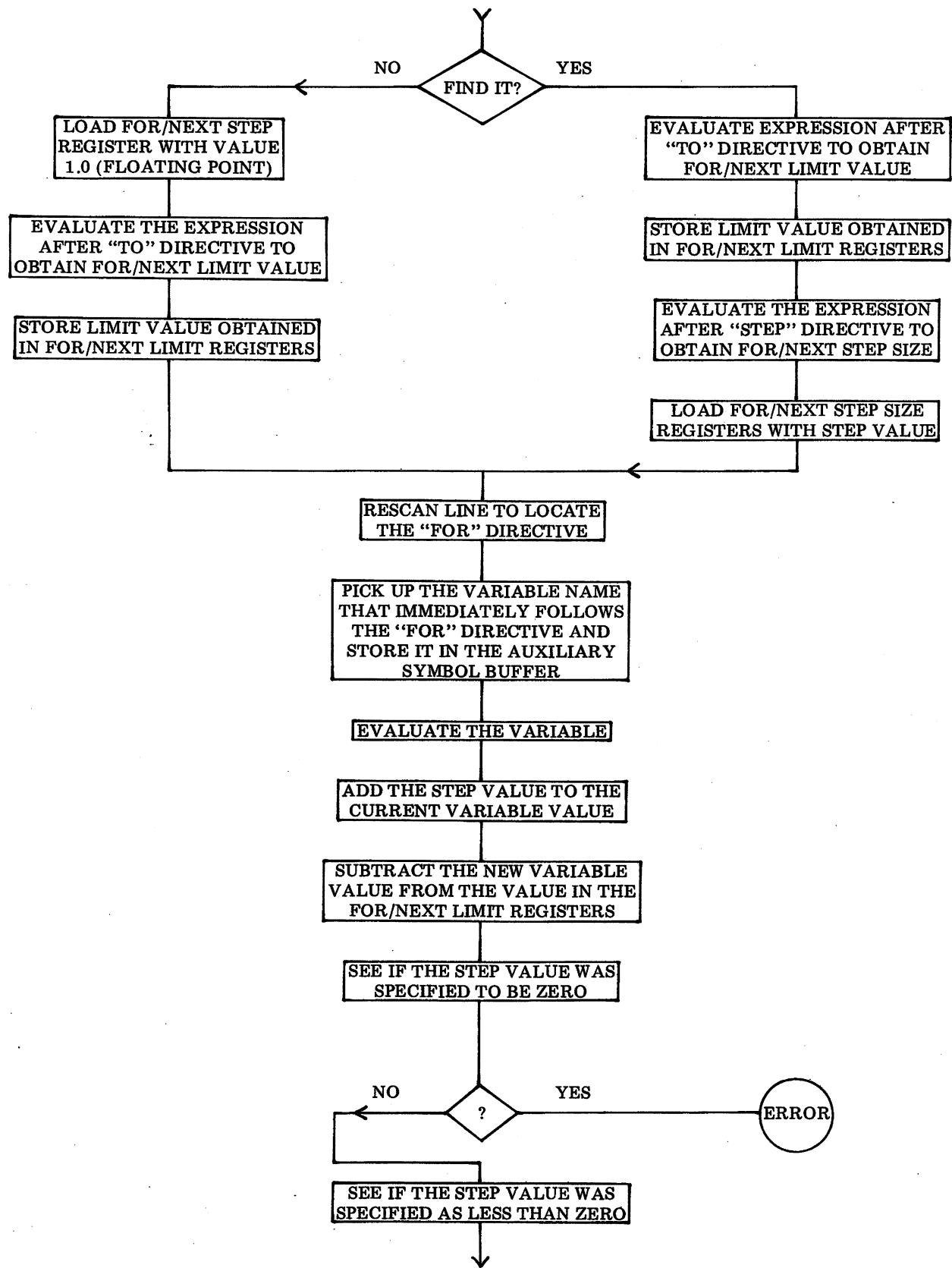
FOR X = 1 TO 5 STEP (2)

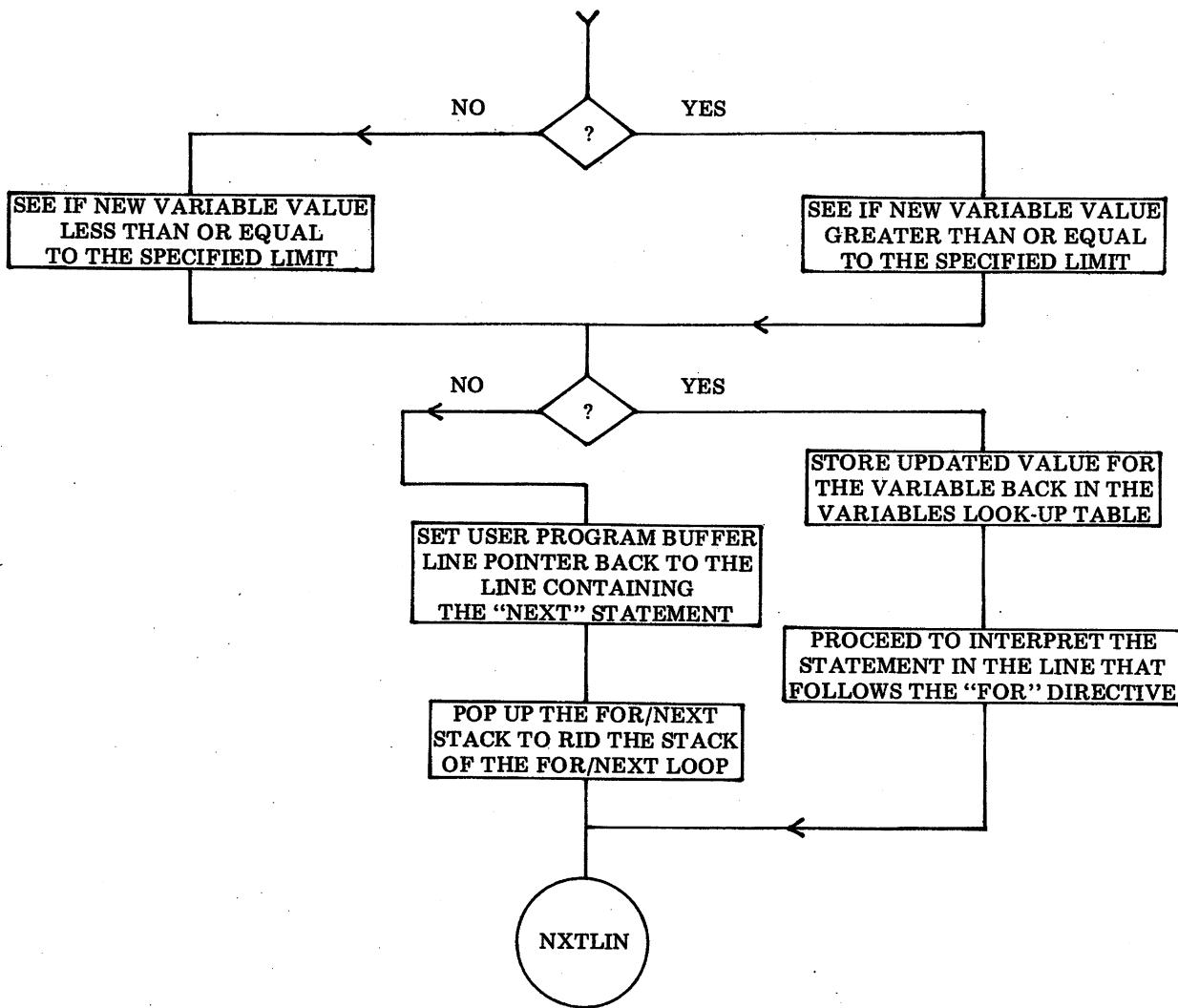
In this case, the STEP size will be whatever value is dictated by the programmer in the term that follows the STEP directive.

Thus, the NEXT statement routine must determine whether an implied or specific STEP size is involved. When this has been accomplished, the STEP size is added to the current value of the associated variable specified in the FOR/NEXT loop. A test is made to see if the new variable value thus obtained is within the range limit specified in the FOR statement line. If the new value causes the variable to exceed the limit value, then the FOR/NEXT loop must be terminated. This is accomplished by removing the associated data from the top of the FOR/NEXT stack and then directing program operation to continue with the statement that follows the NEXT statement. (And NOT the statement following the FOR statement line!) If, on the other hand, the new variable value is still within the specified limit range, then the FOR/NEXT loop must be executed again. In this case, the updated variable value is stored for future use and the statement following the FOR statement will be the next program line executed by the interpreter.

This flow of operations is apparent in the accompanying flow chart. The details of the routine's execution are presented in the source listing which follows the flow chart.







NEXT,	LLI 144 LHI 026 LMI 000 LLI 202 LBM INB LLI 201 LMB	Load L with start of AUX SYMBOL BUFFER ** Set H to page of AUX SYMBOL BUFFER Initialize AUX SYMBOL BUFFER by clearing first byte Change L to address of SCAN pointer Fetch pointer value to CPU register B Add one to the current pointer value Load L with address of NEXT pointer storage location Place the updated SCAN pointer as the NEXT pointer
NEXT1,	LLI 201 CAL GETCHR JTZ NEXT2 LLI 144 CAL CONCT1	Reset L to address of NEXT pointer storage location Fetch the character pointed to by the NEXT pointer If the character is a space, ignore it Else, load L with start of AUX SYMBOL BUFFER Concatenate the character onto the AUX SYMBOL BF

NEXT2,	LLI 201 CAL LOOP JFZ NEXT1 LLI 144 LAM CPI 001 JFZ NEXT3 LLI 146 LMI 000	Reset L to address of NEXT pointer storage location Advance the NEXT pointer and see if end of line Fetch next character in line if not end of line When reach end of line, should have variable name In the AUX SYMBOL BUFFER. Fetch the (cc) for The buffer and see if variable name is just one letter If more than one proceed directly to look for name In FOR/NEXT STACK. If have just a one letter name Then set second character in buffer to zero
NEXT3,	LLI 205 LHI 027 LAM RLC RLC ADI 136 LHI 027 LLA LDI 026 LEI 145 LBI 002 CAL STRCPC JTZ NEXT4	Load L with address of FOR/NEXT STACK pointer ** Set H to page of FOR/NEXT STACK pointer Fetch the FOR/NEXT STACK pointer value to ACC Rotate value left to multiply by two. Then rotate it Left again to multiply by four. Add base address plus Two to form pointer to variable name in top of stack ** Set H to page of FOR/NEXT STACK Move pointer value from ACC to CPU register L ** Set register D to page of AUX SYMBOL BUFFER Set register E to first character in the buffer Set B to serve as a character counter See if variable name in the NEXT statement same as That stored in the top of the FOR/NEXT STACK
FORNXT,	LAI 306 LCI 316 JMP ERROR	Load ACC with ASCII code for letter F Load register C with ASCII code for letter N Display For/Next (FN) error message if required
NEXT4,	LLI 360 LHI 026 LDM INL LEM INL LMD INL LME LLI 205 LHI 027 LAM RLC RLC ADI 134 LLA LDM INL LEM LLI 360 LHI 026 LMD INL	Load L with address of user program line pointer ** Load H with page of user pgm line pntr storage loc. Fetch the page portion of the line pointer into D Advance the memory pointer Fetch the low portion of the line pointer into E Advance pntr to AUXILIARY LINE POINTER storage Location and store value of line pointer there too (page) Advance pointer to second byte of AUXILIARY line Pointer and store value of line pointer (low portion) Load L with address of FOR/NEXT STACK pointer ** Set H to page of FOR/NEXT STACK pointer Fetch the FOR/NEXT STACK pointer value to ACC Rotate value left to multiply by two. Then rotate it Left again to multiply by four. Add base address to Form pointer to top of FOR/NEXT STACK and place The pointer value into CPU register L. Fetch the page Address of the associated FOR statement line pointer Into register D. Advance the pointer and fetch the low Address value into register E. Prepare to change user Program line pointer to the FOR statement line by ** Setting H & L to the user pgm line pntr storage loc. Place the page value in the pointer storage location Advance the memory pointer

	LME	Place the low value in the pointer storage location
	LHD	Now set up H and L to point to the start of the
	LLE	Associated FOR statement line in the user pgm buffer
	LDI 026	** Change D to point to the line input buffer
	LEI 000	And set L to the start of the line input buffer
	CAL MOVEC	Move the associated FOR statement line into the input
	LLI 325	Line buffer. Set L to point to start of TO string which is
	LHI 001	** Stored in a text strings storage area on this page
	CAL INSTR	Search the statement line for the occurrence of TO
	LAE	Register E will be zero if TO not found. Move E to ACC
	NDA	To make a test. If TO found then proceed to set up for
	JTZ FORNXT	Evaluation. If TO not found, then have error condition.
	ADI 002	Advance the pointer over the characters in TO string
	LLI 276	Change L to point to EVAL pointer storage location
	LHI 026	** Set H to page of EVAL pointer. Set up the starting
	LMA	Position for the EVAL subroutine (after TO string)
	LLI 330	Set L to point to start of STEP string which is stored
	LHI 001	** In text strings storage area on this page. Search the
	CAL INSTR	Statement line for the occurrence of STEP
	LAE	Register E will be zero if STEP not found. Move E to
	NDA	The accumulator to make a test. If STEP found must
	JFZ NEXT5	Evaluate expression after STEP to get STEP SIZE.
	LLI 004	Else, have an IMPLIED STEP SIZE of 1.0. Set pointer
	LHI 001	** To start of storage area for 1.0 in floating point
	CAL FLOAD	Format and call subroutine to load FPACC with 1.0
	LLI 304	Set L to start of FOR/NEXT STEP SIZE storage loc.
	CAL FSTORE	Store the value 1.0 in the F/N STEP SIZE registers
	LLI 000	Change L to the start of the input line buffer
	LHI 026	** Set H to the page of the input line buffer
	LBM	Fetch the (cc) into CPU register B (length of FOR line)
	LLI 277	Change L to EVAL FINISH pointer storage location
	LMB	Set the EVAL FINISH pointer to the end of the line
	CAL EVAL	Evaluate the LIMIT expression to obtain FOR LIMIT
	LLI 310	Load L with address of start of F/N LIMIT registers
	LHI 001	** Load H with page of FOR/NEXT LIMIT registers
	CAL FSTORE	Store the FOR/NEXT LIMIT value
	JMP NEXT6	Since have IMPLIED STEP jump ahead
NEXT5,	DCE	When have STEP directive, subtract one from pointer
	LLI 277	To get to character before S in STEP. Save this value in
	LHI 026	** The EVAL FINISH pointer storage location to serve
	LME	As evaluation end location when obtaining TO limit
	CAL EVAL	Evaluate the LIMIT expression to obtain FOR LIMIT
	LLI 310	Load L with address of start of F/N LIMIT registers
	LHI 001	** Load H with page of FOR/NEXT LIMIT registers
	CAL FSTORE	Store the FOR/NEXT LIMIT value
	LLI 277	Reset L to EVAL FINISH pointer storage location
	LHI 026	** Set H to page of EVAL FINISH pointer storage loc.
	LAM	Fetch the pointer value (character before S in STEP)
	ADI 005	Add five to change pointer to character after P in STEP
	DCL	Decrement L to point to EVAL (start) pointer

	LMA LLI 000 LBM LLI 277 LMB CAL EVAL LLI 304 LHI 001 CAL FSTORE	Set up the starting position for the EVAL subroutine Load L with starting address of the line input buffer Fetch the (cc) for the line input buffer (line length) Change L to the EVAL FINISH storage location Set the EVAL FINISH pointer Evaluate the STEP SIZE expression Load L with address of start of F/N STEP registers ** Set H to page of F/N STEP registers Store the FOR/NEXT STEP SIZE value
NEXT6,	LLI 144 LHI 026 LMI 000 LLI 034 LHI 027 CAL INSTR LAE NDA LLI 202 LHI 026 LMA JTZ FORNXT ADI 003 LLI 203 LMA	Load L with address of AUX SYMBOL BUFFER ** Set H to page of the AUX SYMBOL BUFFER Initialize AUX SYMBOL BUFFER with a zero byte Set L to start of FOR string which is stored in the ** KEYWORD look-up table on this page Search the statement line for the FOR directive Register E will be zero if FOR not found. Move E to ACC and make test to see if FOR directive located Load L with address of SCAN pointer ** Load H with page of SCAN pointer Set up pointer to occurrence of FOR directive in line If FOR not found, have an error condition If have FOR, add three to advance pointer over FOR Set L to point to F/N pointer storage location Set F/N pointer to character after FOR directive
NEXT7,	LLI 203 CAL GETCHR JTZ NEXT8 CPI 275 JTZ NEXT9 LLI 144 CAL CONCT1	Set L to point to F/N pointer storage location Fetch a character from position pointed to by F/N pnter If character is a space, ignore it Else, test to see if character is “=” sign If yes, have picked up variable name, jump ahead If not, set L to the start of the AUX SYMBOL BUFFER And store the character in the AUX SYMBOL BUFFER
NEXT8,	LLI 203 CAL LOOP JFZ NEXT7 JMP FORNXT	Load L with address of the F/N pointer Increment the pointer and see if end of the line If not, continue fetching characters If end of line before “=” sign then have error condx
NEXT9,	LLI 202 LHI 026 LAM ADI 003 LLI 276 LMA LLI 203 LBM DCB LLI 277 LMB CAL EVAL	Load L with address of SCAN pointer ** Load H with page of SCAN pointer Fetch pointer value to ACC (points to start of FOR Directive) and add three to move pointer over FOR Directive. Change L to EVAL pointer storage location Set EVAL pointer to character after FOR in line Load L with address of F/N pointer storage location Fetch pointer to register B (points to “=” sign) and Decrement the pointer (to character before “=” sign) Load L with address of EVAL FINISH pointer Set EVAL FINISH pointer Call subroutine to obtain current value of the variable

	LLI 304 LHI 001 CAL FAXOP CAL FPADD LLI 314 LHI 001 CAL FSTORE LLI 310 CAL FAXOP CAL FPSUB LLI 306 LAM NDA LLI 126 LAM JTZ FORNXT JTS NEXT11 NDA JTS NEXT12 JTZ NEXT12	Load L with address of start of F/N STEP registers ** Set H to page of F/N STEP registers Call subroutine to set up FP registers for addition Add F/N STEP size to current VARIABLE value Load L with address of F/N TEMP storage registers **Set H to page of F/N TEMP storage registers Save the result of the addition in F/N TEMP registers Load L with starting address of F/N LIMIT registers Call subroutine to set up FP registers for subtraction Subtract F/N LIMIT value from VARIABLE value Set pointer to MSW of F/N STEP registers Fetch this value into the ACC Test to see if STEP value might be zero Load L with address of MSW of FPACC Fetch this value into the ACC If STEP size was zero, then endless loop, an error condx If STEP size less than zero make alternate test on limit Test the contents of the MSW of the FPACC Continue FOR/NEXT loop if current variable value is Less than or equal to the F/N LIMIT value
NEXT10,	LLI 363 LHI 026 LEM DCL LDM DCL LME DCL LMD LLI 205 LHI 027 LBM DCB LMB JMP NXTLIN	If out of LIMIT range, load L with address of the AUX ** PGM LINE pointer. (Contains pointer to the NEXT Statement line that initiated this routine.) Fetch the Low part of the address into E, decrement the memory And get the page part of the address into CPU register Decrement memory pointer to the low portion of the User pgm buffer line pointer (regular pointer) and set it With the value from the AUX line pntr, decrement the Pointer and do the same for the page portion Set L to address of FOR/NEXT STACK pointer ** Set H to page of FOR/NEXT STACK pointer Fetch and decrement the FOR/NEXT STACK pointer value To perform effective popping operation Statement line after NEXT statement is done next
NEXT11,	NDA JFS NEXT12 JMP NEXT10	When F/N STEP is negative, reverse test so that if the Variable value is greater than or equal to the F/N LIMIT The FOR/NEXT loop continues. Else it is finished.
NEXT12,	LLI 314 LHI 001 CAL FLOAD CAL RESTSY CAL STOSYM JMP NXTLIN	Load L with address of F/N TEMP storage registers ** Set H to F/N TEMP storage registers page Transfer the updated variable value to the FPACC Restore the variable name and value In the VARIABLES table. Exit routine so that Statement line after FOR statement is done next

THE OPTIONAL DIM STATEMENT ROUTINE

The DIM statement routine is an optional statement routine that may be included in SCELBAL depending on whether the user desires to utilize its capabilities and sacrifice the memory space that it and routines associated with it utilize.

The purpose of the DIM statement routine is to allow the defining of single character ARRAY VARIABLES and to reserve space in an ARRAY VALUES TABLE for the specified number of entries that the array will occupy.

The DIM statement capability in SCELBAL is restricted to single dimension arrays. To conserve memory space, the DIM routine to be presented restricts the total amount of memory used to store the values at points in an array to 256 bytes (one page). The storage of floating point numbers in the format used in SCELBAL requires four bytes of memory to store a value. Thus, the total number of array points that may be set aside in one program is 256 divided by 4 or 64 (decimal).

To keep the DIM capability in line with the storage space allotted for array values, the number of arrays that may be created in a program is restricted to four. However, regardless of whether one, two, three or four array variables are defined, the total number of array subscripts for all the variables must not exceed 64 because of the limitation discussed in the previous paragraph.

Thus, one could DIMension a single array to have 64 locations. One could specify two arrays, each using 32 entries. One could create four array variables and DIMension 16 locations for each. Or, any other combination may be specified as long as the total number of array variable names does not exceed four, and the total number of subscripted array points does not exceed 64!

The reader must remember that an array

variable name may only consist of one letter followed by a subscript. Thus, a four element array having the symbolic variable name A would consist of the elements:

A(1)
A(2)
A(3)
A(4)

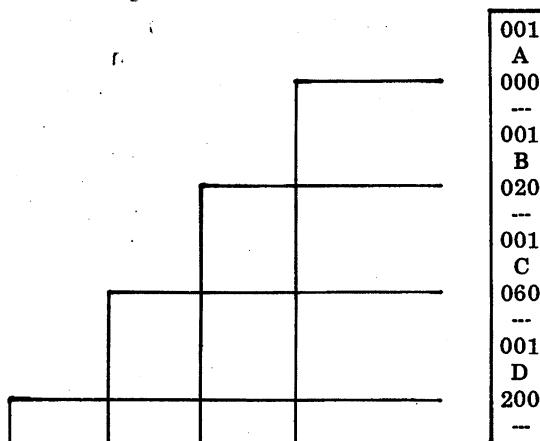
Since the above array would need to have four locations reserved for it in the ARRAY VALUES TABLE, the DIMension statement for it would appear as:

DIM A(4)

The reader must note too, that the array size in a DIMension statement must always be given in the form of an integer value (less than or equal to 64) and may not be another variable.

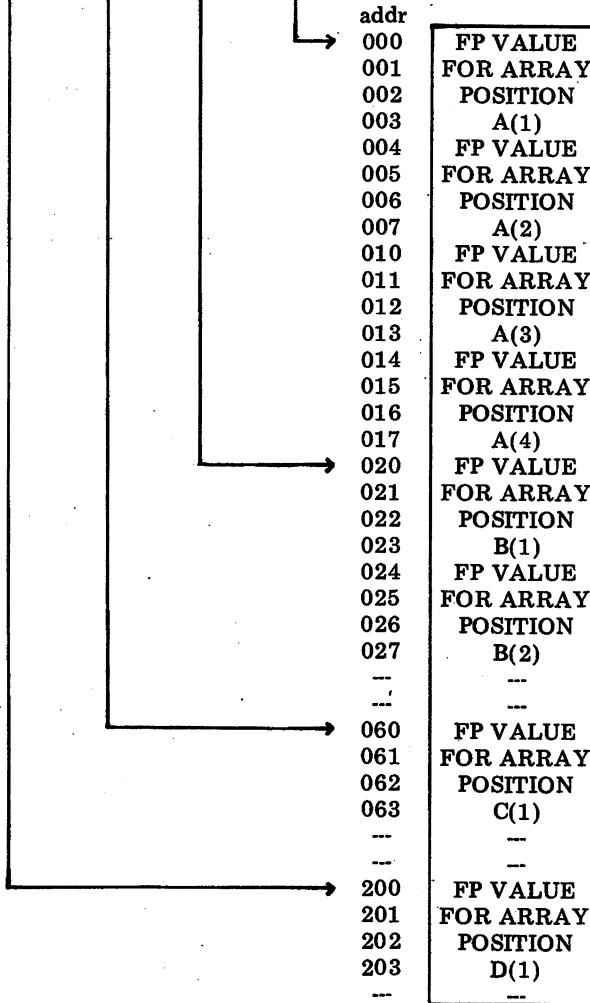
Associated with the ARRAY VALUES TABLE is another table called the ARRAY VARIABLES TABLE. This short table, having room for a maximum of four entries, contains the array name(s) and the starting location(s) in the ARRAY VALUES TABLE for the first array value associated with an array name. The ARRAY VARIABLES TABLE reserves four bytes for each array specified in a program. Two are used to store the array name. (This is done using string format, thus the first byte will always be 001 to indicate a one byte character string and the second byte will be the alphabetical character designated as the name of the array.) The third byte in an ARRAY VARIABLES TABLE entry is used to store the starting location for the first element in the associated ARRAY VALUES TABLE. The fourth byte is reserved for possible use by the user who might desire to modify and expand the array capability of SCELBAL. It could be used to store the page address value in the ARRAY VALUES TABLE if that table crossed page boundaries.

ARRAY VARIABLES TABLE



The ARRAY VARIABLES TABLE holds the array variable names and points to the starting location for each series of subscripted array entries associated with an array name. In this example the array named A has had room for four entries reserved for it. The array named B has had eight value locations reserved. C has 16 and D has 32.

ARRAY VALUES TABLE



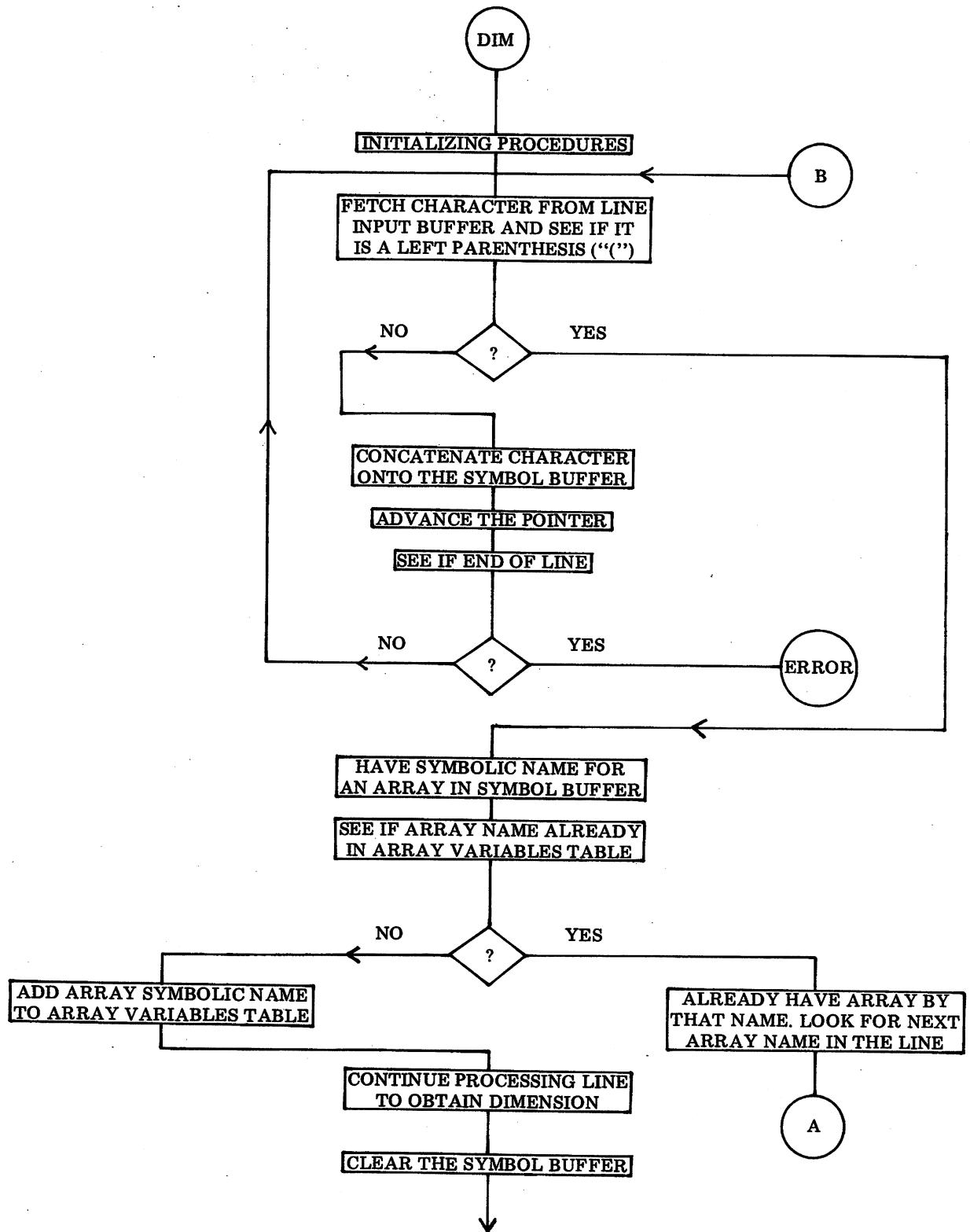
The ARRAY VALUES TABLE is used to hold the numerical value for each position in the array. Numerical values are stored in floating point format and require four bytes each. Note that the starting address for each series of values associated with an array name is that address pointed to in the ARRAY VARIABLES TABLE. The address for a particular point in an array is calculated as a function of the subscript specified.

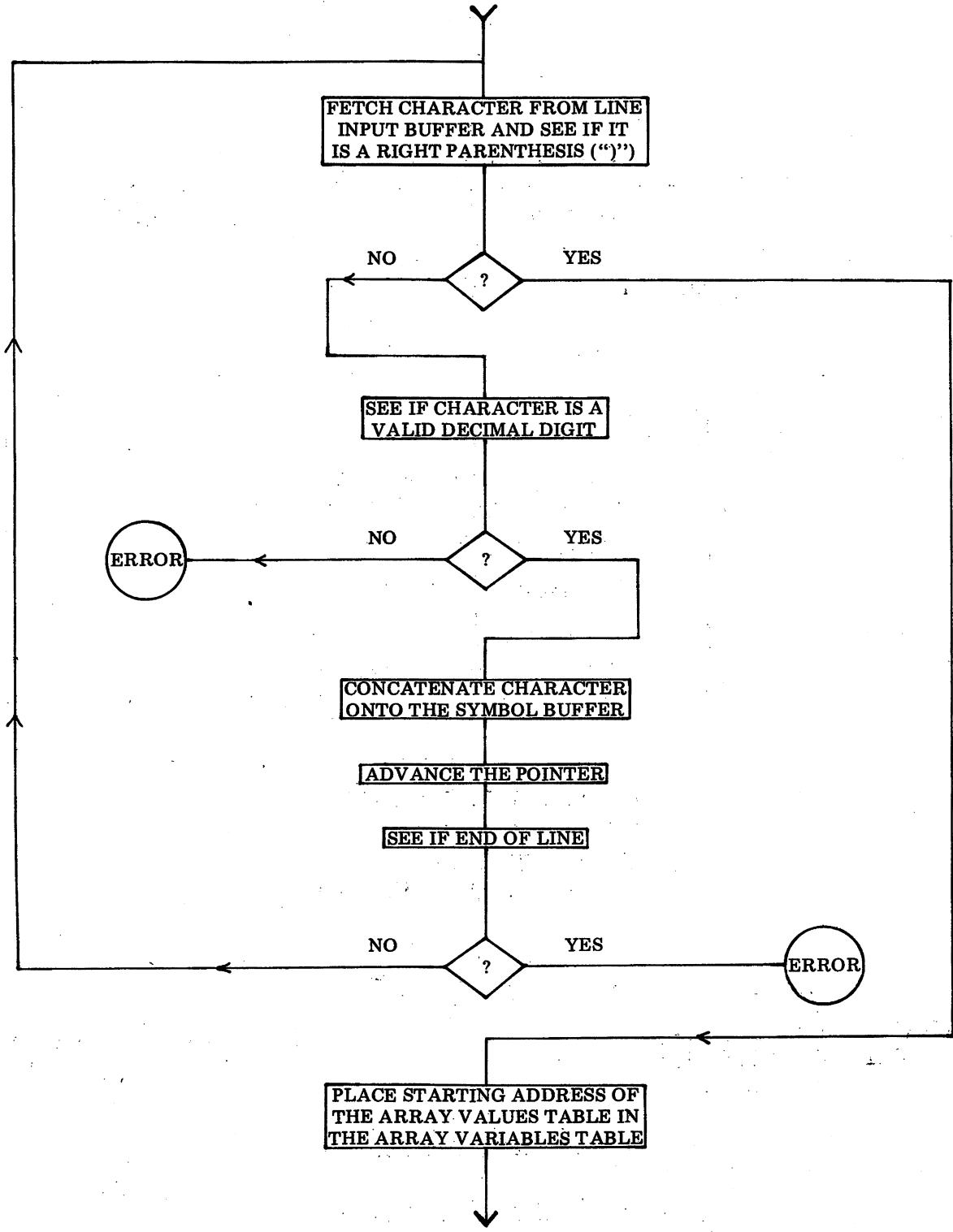
The relationship between the ARRAY VARIABLES TABLE and the ARRAY VALUES TABLE may be seen a little more clearly by examining the pictorial illustration presented on the preceding page.

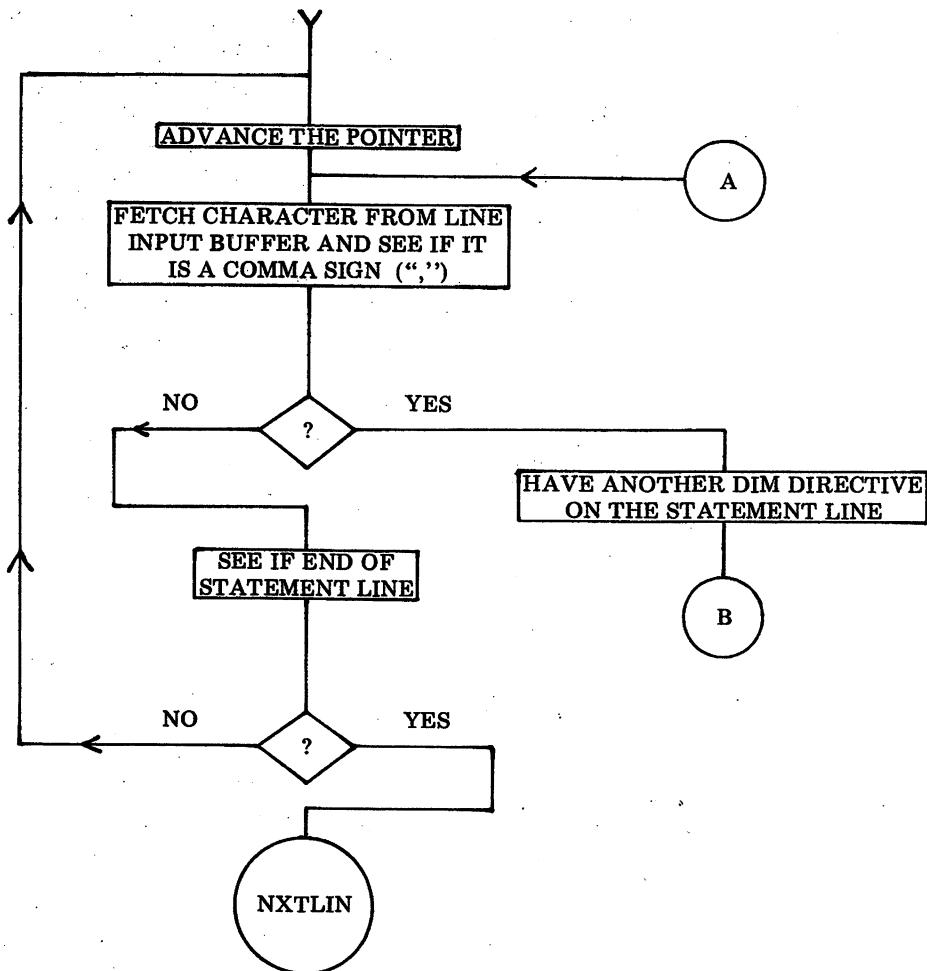
The flow chart on the next several pages summarizes the operation of the DIM routine as just discussed. The commented source listing for the routine starts below.

Remember, this routine is an optional routine. If array capability is not desired this routine may be left out of SCELBAL (along with related routines which will be presented later). If the routine is not incorporated in the reader's individual version of SCELBAL the various locations throughout the program identified by an @@ mark should be changed to effective no-operation instructions (such as LAA) as previously explained.

DIM,	CAL CLESYM LLI 202 LBM INB LLI 203 LMB	Initialize the SYMBOL BUFFER to cleared condition Load L with address of SCAN pointer Fetch SCAN pointer value into register B Add one to the SCAN pointer value Change L to DIM pointer (formerly TOKEN) storage Store the updated SCAN pointer as the DIM pointer
DIM1,	LLI 203 CAL GETCHR JTZ DIM2 CPI 250 JTZ DIM3 CAL CONCTS	Load L with the address of DIM pointer storage location Fetch a character from the line input buffer If character fetched is a space, ignore it Else see if character is "(" left parenthesis If so, should have ARRAY VARIABLE name in buffer If not, append the character to the SYMBOL BUFFER
DIM2,	LLI 203 CAL LOOP JFZ DIM1 JMP DIMERR	Load L with the address of DIM pointer storage location Increment the pointer and see if end of line If not end of line, fetch next character Else have a DIMension error condition
DIM3,	LLI 206 LMI 000	Load L with address of ARRAY pointer storage loc Initialize ARRAY pointer to starting value of zero
DIM4,	LLI 206 LHI 026 LAM RLC RLC ADI 114 LHI 027 LLA LEI 120 LDI 026 CAL STRCP JTZ DIM9 LLI 206 LHI 026 LBM	Load L with address of ARRAY pointer storage loc ** Set H to page of ARRAY pointer storage location Fetch value in ARRAY pointer to ACC (effectively Represents number of arrays defined in pgm). Rotate Left twice to multiply by four (number of bytes per entry in ARRAY VARIABLES table). Add to base ** Address to form pointer to ARRAY VARIABLES Table and set up H & L as the memory pointer. Load E with starting address of the SYMBOL BUFFER ** Load D with the page address of the SYMBOL BUFF Compare contents of SYMBOL BF to entry in ARRAY VARIABLES table. If same, have duplicate array name. Else, load L with address of ARRAY pointer storage ** Load H with page of ARRAY pointer storage Fetch the ARRAY pointer value to register B







INB	Increment the value
LMB	Restore it to ARRAY pointer storage location
LLI 075	Change L to number of arrays storage location
LHI 027	** Set H to page of the number of arrays storage loc.
LAM	Fetch the number of arrays value to the ACC
DCB	Restore B to previous count
CPB	Compare number of arrays tested against nr defined
JFZ DIM4	If not equal, continue searching ARRAY VARIABLES
LLI 075	Table: When table searched with no match, then must
LHI 027	** Append name to table. First set pointer to number
LBM	Of arrays storage location. Fetch that value and
INB	Add one to account for new name being added.
LMB	Restore the updated value back to memory.
LLI 076	Change pointer to ARRAY TEMP pointer storage
LMB	Store pointer to current array in ARRAY TEMP too.
LLI 206	Load L with address of ARRAY pointer storage loc.
LHI 026	** Set H to page of ARRAY pointer storage location
LMB	And update it also for new array being added.

	LAM	Fetch the current ARRAY pointer value to the ACC
	RLC	Multiply it times four by performing two rotate left Operations and add it to base value to form address in The ARRAY VARIABLES table. Place the low part Of this calculated address value into register E.
	RLC	** Set register D to the page of the table.
	ADI 114	Load L with the start of the SYMBOL BUFFER
	LEA	** Load H with the page of the SYMBOL BUFFER
	LDI 027	Move the array name from the SYMBOL BUFFER to The ARRAY VARIABLES table. Then clear the SYMBOL BUFFER. Reset L to the DIM pointer storage
	LLI 120	** Location. Set H to the DIM pointer page.
	LHI 026	Fetch the pointer value (points to "(" part of DIM Statement). Increment the pointer to next character in The line input buffer. Change L to DIMEN pointer.
	CAL MOVEC	Store the updated DIM pointer in DIMEN storage loc.
	CAL CLESYM	
	LLI 203	
	LHI 026	
	LBM	
	INB	
	LLI 204	
	LMB	
DIM5,	LLI 204	Set L to DIMEN pointer storage location
	CAL GETCHR	Fetch character in line input buffer
	JTZ DIM6	Ignore character for space
	CPI 251	If not space, see if character is right parenthesis ("").
	JTZ DIM7	If yes, process DIMension size (array length)
	CPI 260	If not, see if character is a valid decimal number
	JTS DIMERR	If not valid number, have DIMension error condition
	CPI 272	Continue testing for valid decimal number
	JFS DIMERR	If not valid number, then DIMension error condition
	CAL CONCTS	If valid decimal number, append digit to SYMBOL BF
DIM6,	LLI 204	Set L to DIMEN pointer storage location
	CAL LOOP	Advance the pointer value and check for end of the line
	JFZ DIM5	If not end of line, continue fetching DIMension size
	JMP DIMERR	If end of line before right parenthesis, have error condx.
DIM7,	LLI 120	Load L with address of start of SYMBOL BUFFER
	LHI 026	** Load H with page of SYMBOL BUFFER. (Now Contains DIMension size.) Convert buffer to floating
	CAL DINPUT	Point number and then reformat to fixed point.
	CAL FPFIX	Load L with address of LSW of fixed point number
	LLI 124	And fetch the low order byte of the nr into the ACC
	LAM	Rotate it left two times to multiply it by four (the Number of bytes required to store a floating point nr).
	RLC	Store this value in CPU register C temporarily.
	RLC	Set L to ARRAY TEMP storage location.
	LCA	** Set H to ARRAY TEMP pointer page.
	LLI 076	Fetch the value in ARRAY TEMP (points to ARRAY VARIABLES table). Subtract one from the pointer
	LHI 027	Value and multiply the result by four using rotate left Instructions. Add this value to a base address
	LAM	(Augmented by two) to point to ARRAY VALUES
	SUI 001	Pointer storage location in the ARRAY VARIABLES
	RLC	** Table and set the pointer up in registers H & L.
	RLC	
	ADI 122	
	LLA	
	LHI 027	

	LBM ADI 004 LLA LAB ADC LMA	Fetch the starting address in the ARRAY VALUES Table for the previous array into register B. Now add Four to the ARRAY VARIABLES table pointer to Point to current ARRAY VALUES starting address. Add the previous array starting address plus number of Bytes required and store as starting loc for next array
DIM8,	LLI 204 LHI 026 LBM LLI 203 LMB	Set L to address of DIMEN pointer storage location ** Set H to page of DIMEN pointer Fetch pointer value (points to ")" in line) Change L to DIM pointer storage location Store former DIMEN value back in DIM pointer
DIM9,	LLI 203 CAL GETCHR CPI 254 JTZ DIM10 LLI 203 CAL LOOP JFZ DIM9 JMP NXTLIN	Load L with address of DIM pointer storage location Fetch a character from the line input buffer See if character is a comma (,) sign If yes, have another array being defined on the line If not, reset L to the DIM pointer Increment the pointer and see if end of the line If not end of the line, keep looking for a comma Else exit the DIM statement routine to continue pgm
DIM10,	LLI 203 LBM LLI 202 LMB JMP DIM	Set L to DIM pointer storage location Fetch pointer value (points to comma sign just found) Load L with address of SCAN pointer storage location Place DIM pointer into the SCAN pointer Continue processing DIM statement line for next array
DIMERR,	LAI 304 LCI 305 JMP ERROR	On error condition, load ASCII code for letter D in ACC And ASCII code for letter E in CPU register C Go display the Dimension Error (DE) message.

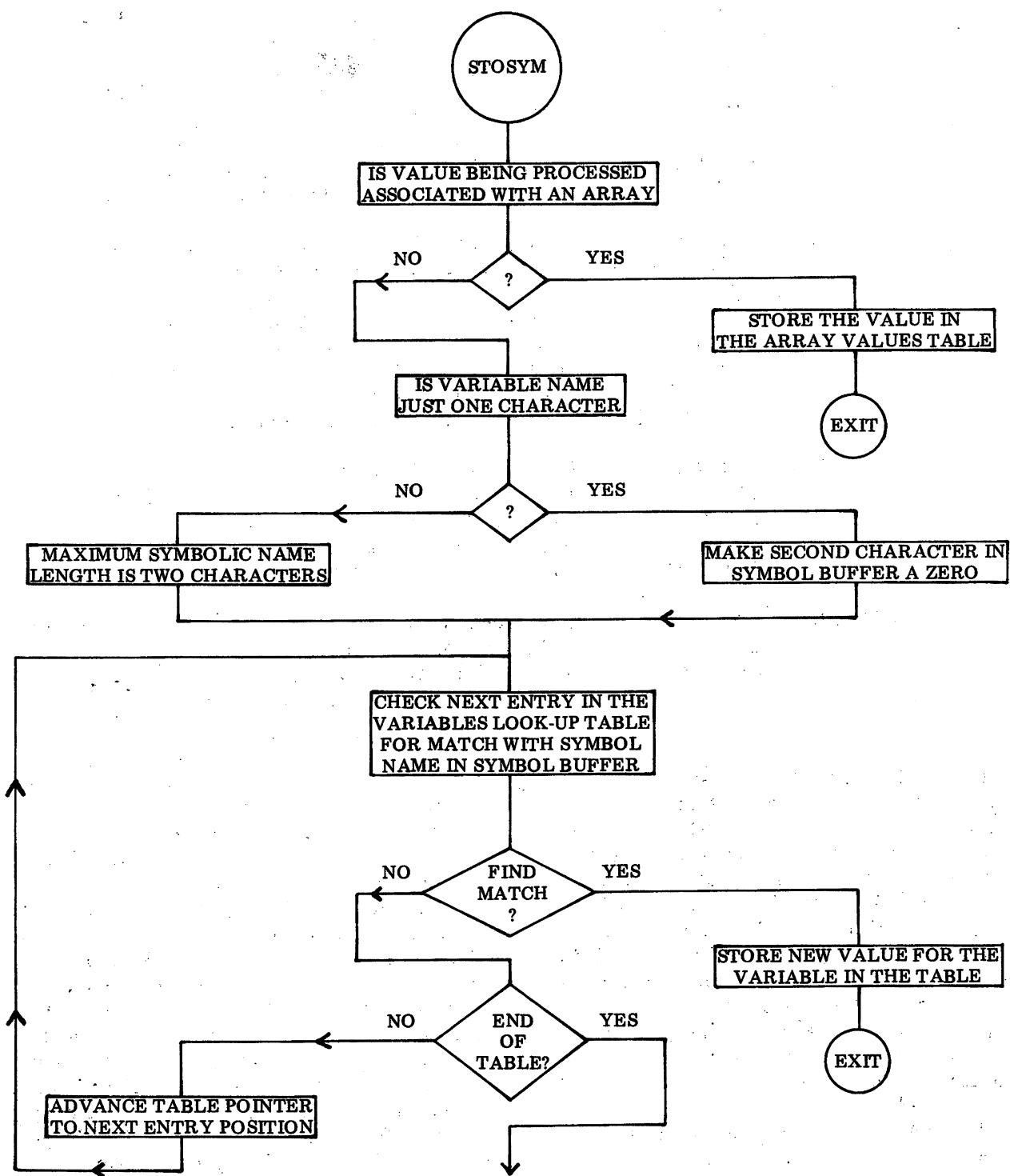
The final routine to be discussed in this chapter is used by several of the statement routines such as the LET and INPUT routines. This routine is used to store the name of a variable and its numerical value in a table called the VARIABLES LOOK-UP TABLE. (A portion of the routine is also used to handle the storing of values assigned to array variables (which are stored in a separate table) if the user elects to utilize the single DIMension array handling capability of SCELBAL. The array handling routines themselves are discussed in a later chapter.)

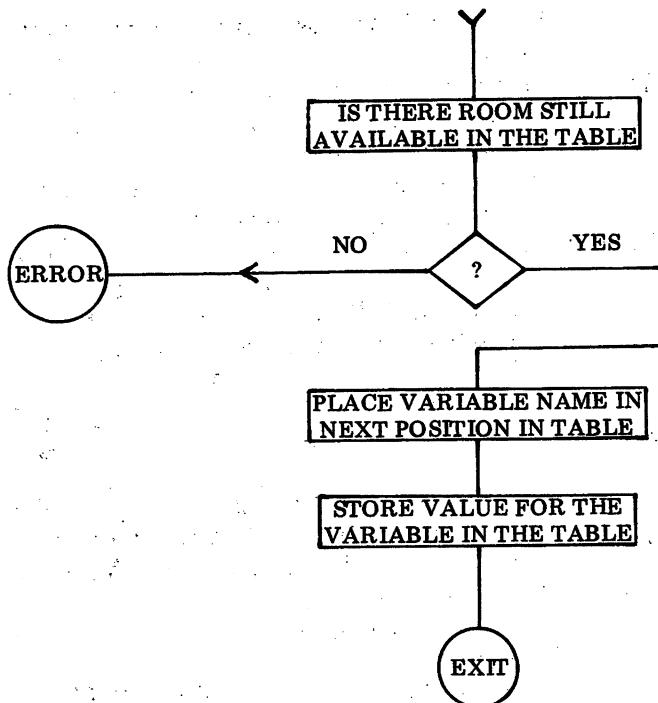
The VARIABLES LOOK-UP TABLE is organized to hold the variable names and

the current values assigned to those names in the following format. The first two bytes of an entry are used to hold the one or two letter NAME for the variable. (If the variable name only consist of one letter, the second byte of the entry will be zero.) The next four bytes in an entry are used to store the current value of the variable in floating point format. (This format for storing mathematical values will be presented in a later chapter.) Thus, each entry in the table requires six bytes of storage. Sufficient room has been provided in the assembled version of SCELBAL presented in this book for storage of up to 20 variable names at one time during the execution of a higher level program.

The general operation of the routine is illustrated in the flow chart which starts below.

The source listing for the subroutine is presented following the flow chart.

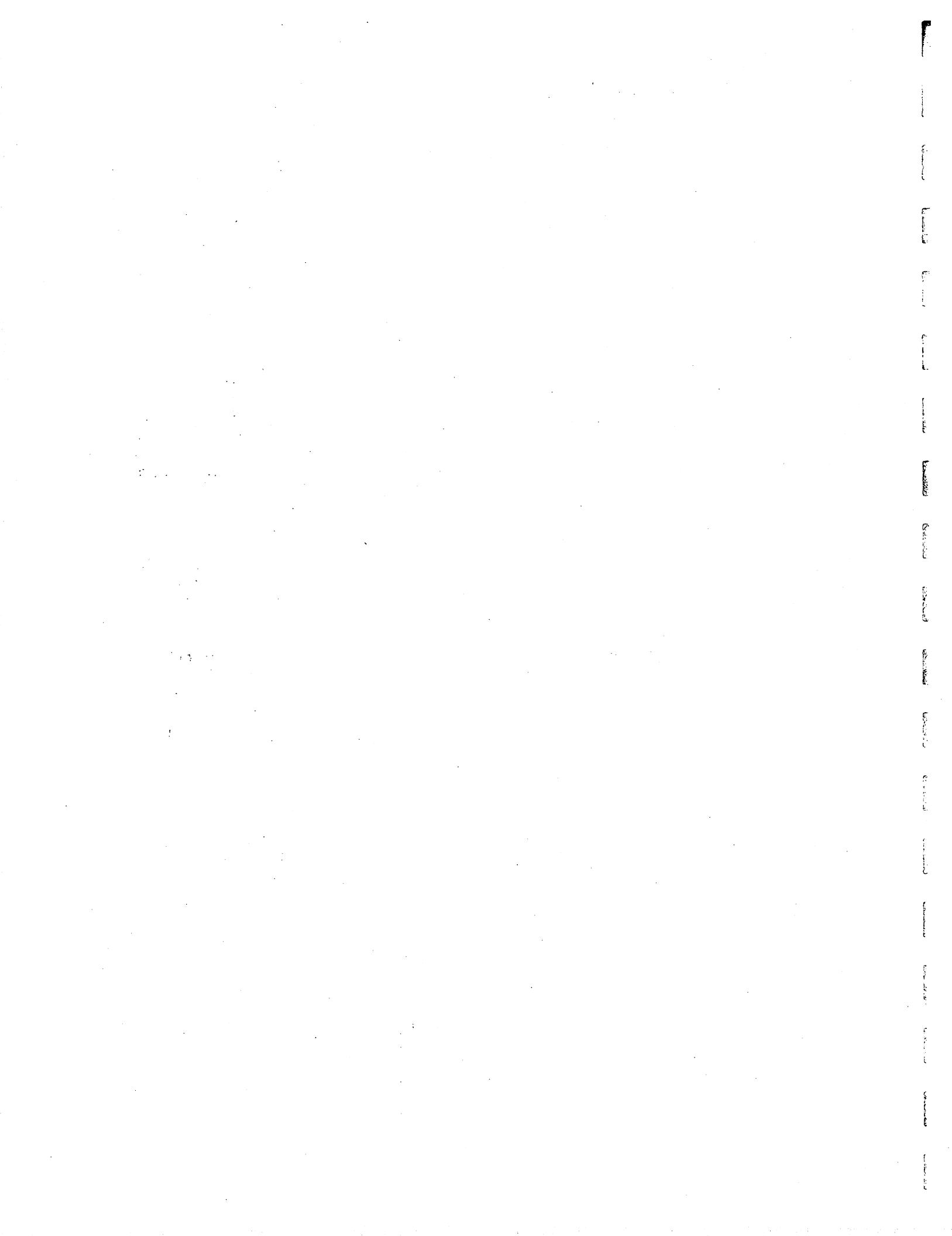




STOSYM,	LLI 201 LHI 027 LAM NDA JTZ STOSY1 LMI 000 LLI 204 LLM LHI 057 JMP FSTORE	Load L with address of ARRAY FLAG ** Load H with page of ARRAY FLAG Fetch the value of the ARRAY FLAG into the ACC Check to see if the flag is set indicating processing an Array variable value. Jump ahead if flag not set. If ARRAY FLAG was set, clear it for next time. Then load L with address of array address storage loc Fetch the array storage address as new pointer †† Set H to ARRAY VALUES page Store the array variable value and exit to caller.
STOSY1,	LLI 370 LHI 026 LMI 000 LLI 120 LDI 027 LEI 210 LAM CPI 001 JFZ STOSY2 LLI 122 LMI 000	Load L with address of TEMP CNTR ** Load H with page of TEMP CNTR Initialize the TEMP CNTR by clearing it Load L with starting address of SYMBOL BUFFER ** Load D with page of VARIABLES LOOK-UP table Load E with starting addr of VARIABLES LOOK-UP Table. Fetch the (cc) for the SYMBOL BUFFER into The ACC and see if length of variable name is just one Character. If not, skip next couple of instructions. Else, set pointer to second character location in the SYMBOL BUFFER and set it to zero

STOSY2,	LLI 121 LHI 026 CAL SWITCH LAM INL LBM INL CAL SWITCH CPM JFZ STOSY3 INL LAB CPM JTZ STOSY5	Load L with address of first character in the SYMBOL ** BUFFER. Load H with page of the buffer. Exchange pointer to buffer for pointer to VARIABLES LOOK-UP table. Fetch first char in a name from the Table. Advance the pointer to second char in a name. Fetch the second character into register B. Advance the pointer to first byte of a value in the table. Exchange table pointer for pointer to SYMBOL BUFF Compare first character in buffer against first character In table entry. If no match, try next entry in the table. If match, advance pointer to second character in buffer. Move second character obtained from table into ACC. Compare second characters in table and buffer. If same, have found the variable name in the table.
STOSY3,	CAL AD4DE LLI 370 LHI 026 LBM INB LMB LLI 077 LHI 027 LAB CPM JFZ STOSY2 LLI 077 LHI 027 LBM INB LMB LAB CPI 025 JFS BIGERR LLI 121 LHI 026 LBI 002 CAL MOVEIT	Add four to pointer in registers D&E to skip over value Portion of entry in table. Load L with address of TEMP ** CNTR. Load H with page of TEMP CNTR. Fetch the counter Increment the counter Restore it to storage Set L to address of VARIABLES CNTR (indicates ** Number of variables currently in table.) Set H too. Move the TEMP CNTR value into the ACC. (Number of Entries checked.) Compare with number of entries in The table. If have not checked all entries, try next one. If have checked all entries, load L with address of the ** VARIABLES CNTR. Set H too. Fetch the counter Value and increment it to account for New variable name that will now be Added to the table. Save the new value. Place the new counter value into the accumulator And check to see that adding new variable name to the Table will not cause table overflow. Big Error if it does! If room available in table, set L to address of first ** Character in the SYMBOL BUFFER. Set H too. Set a counter for number of characters to transfer. Move the variable name from buffer to table.
STOSY5,	CAL SWITCH CAL FSTORE JMP CLESYM	Exchange buffer pointer for table pointer. Transfer new mathematical value into the table. Clear the SYMBOL BUFFER and exit to calling routine.
SAVESY,	LLI 120 LHI 026 LDH LEI 144 JMP MOVECP	The subroutines below are used by some of the routines in this chapter as well as other parts of the program. Load L with the address of the start of the SYMBOL ** BUFFER. Load H with the page of the buffer. Load register D with the page of the AUX SYMBOL BUFFER and set register E to start of that buffer. Transfer SYMBOL BF contents to AUX SYMBOL BF

RESTSY,	LLI 144 LHI 026 LDH LEI 120	Load L with address of start of AUX SYMBOL BUFF ** Load H with page of AUX SYMBOL BUFFER Set D to page of SYMBOL BUFFER (same as H) Load E with start of SYMBOL BUFFER
MOVECP,	LBM INB JMP MOVEIT	Load (cc) for source string (first byte in source buffer) Add one to (cc) to include (cc) byte itself Move the source string to destination buffer



EVALUATING MATHEMATICAL EXPRESSIONS

This and the next several chapters will present the routines associated with EVALUATING mathematical expressions. While it will take a considerable number of pages of text to present the details and source listings of the routines, the essential concepts of this process remain quite simple and straightforward.

The reader who has studied the preceding chapter may recall that when a portion of a statement line contained a mathematical expression that needed to be evaluated, the routine would set up pointers to the starting and ending characters of the expression and then call a subroutine labeled EVAL. The EVAL routine, which is presented in this chapter, is able to process the string of characters making up a mathematical expression. In doing so, it calls on several other subroutines that will have separate chapters devoted to their details. However, the EVAL routine is the primary expression processing routine that ties the supportive subroutines for this process together.

Mathematical expressions that are to be evaluated by SCELBAL are assumed to consist of strings of characters that represent symbols joined by operators. Symbols in this context mean either actual numerical values such as 123.456 or 995 or 1.14159E+15; or they may be characters representing a variable name such as X. Operators are mathematical operating signs such as "+" (addition), "-" (subtraction or minus), "*" (multiplication), "/" (division), " \uparrow " (exponentiation), and such signs as "=" (equal), "<" (less than) and ">" (greater than). Two special operator signs are the right and left parenthesis "(" which may be used to group or nest portions of mathematical expressions, denote the argument part of a function, or be used to indicate a subscripted variable.

A typical mathematical expression that might appear in a SCELBAL program is illustrated here:

$$X \uparrow 2 + 4 * X - 16$$

In this expression, X is a symbol (name of a variable) as are 2, 4 and 16 (actual numerical values). Four mathematical operators are used in the above expression, \uparrow , *, + and - in that order.

The process of evaluating an expression to obtain a mathematical (numerical) value consists of scanning the expression to break it up into symbols and operators, and then performing the required operations in the proper order. The requirement of performing the operations IN THE PROPER ORDER is essential. The proper evaluation of mathematical expressions requires the following of precise rules for performing certain operations. For instance, the example expression just presented is meant to be read as, and evaluated in the following fashion.

"Raise the value represented by X to the second power. To this quantity add four times the quantity X. From this new total subtract the value 16."

A person who did not know the order in which operations were to be performed according to custom, or a computer that was not instructed otherwise, might just as easily interpret the example expression to mean.

"Raise X to the power of 2 plus four times X minus 16."

The order in which to perform various types of operations is defined by establishing a hierarchy for the various types of mathematical operators. The portion of SCELBAL that establishes the hierarchy and actually determines when various mathematical operations are to be performed has been given the label PARSER in accordance with the task it performs. This routine will be discussed and described in detail in the next chapter.

The EVALuating routine presented in this

chapter essentially serves to perform the following tasks. It breaks the mathematical expression being processed up into component parts consisting of symbols (whether a variable name or a numerical value) and mathematical operators. Characters making up a symbol are stored in the SYMBOL BUFFER. Whenever a mathematical operator is detected, a TOKEN VALUE is assigned to represent the operator similar to the manner in which a token value was assigned when the SYNTAX subroutine identified a STATEMENT KEYWORD. This TOKEN VALUE assigned for the mathematical operator is passed on to another subroutine called the PARSER (to be described in the next chapter) which will either store the symbol and operator for future use or perform the indicated operations depending on the precedence of the operator being processed. This process of obtaining symbols and operators continues until the entire expression has been scanned.

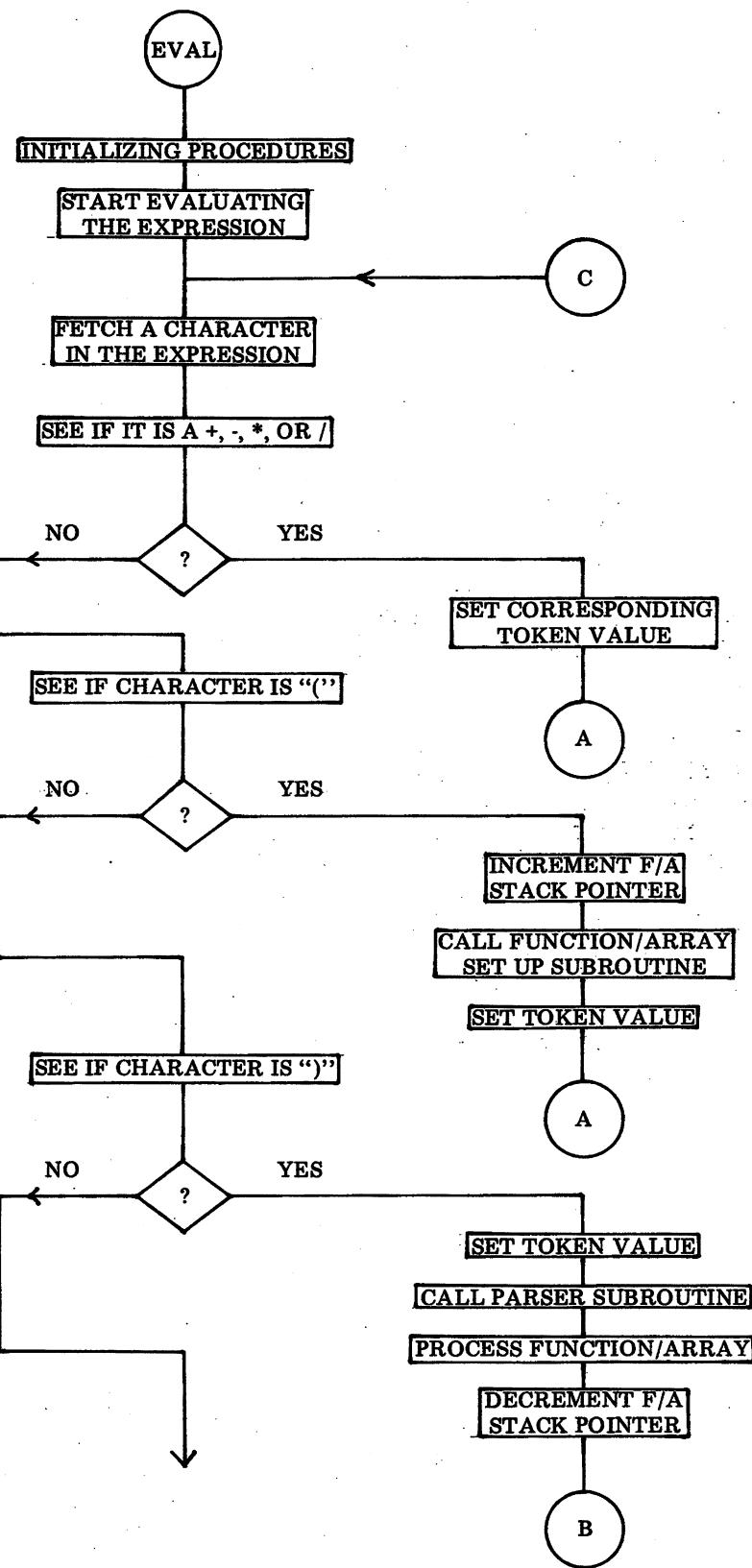
For reference purposes, a list of the TOKEN VALUES assigned to the various mathematical operators is presented below. Note that the first part of the table assigns a TOKEN VALUE to single operators. The latter part of the table assigns values to some special combinations of operators which may occur in IF statements. Later chapters will illustrate how these TOKEN VALUES are used to direct the operations of other SCELBAL expression handling routines.

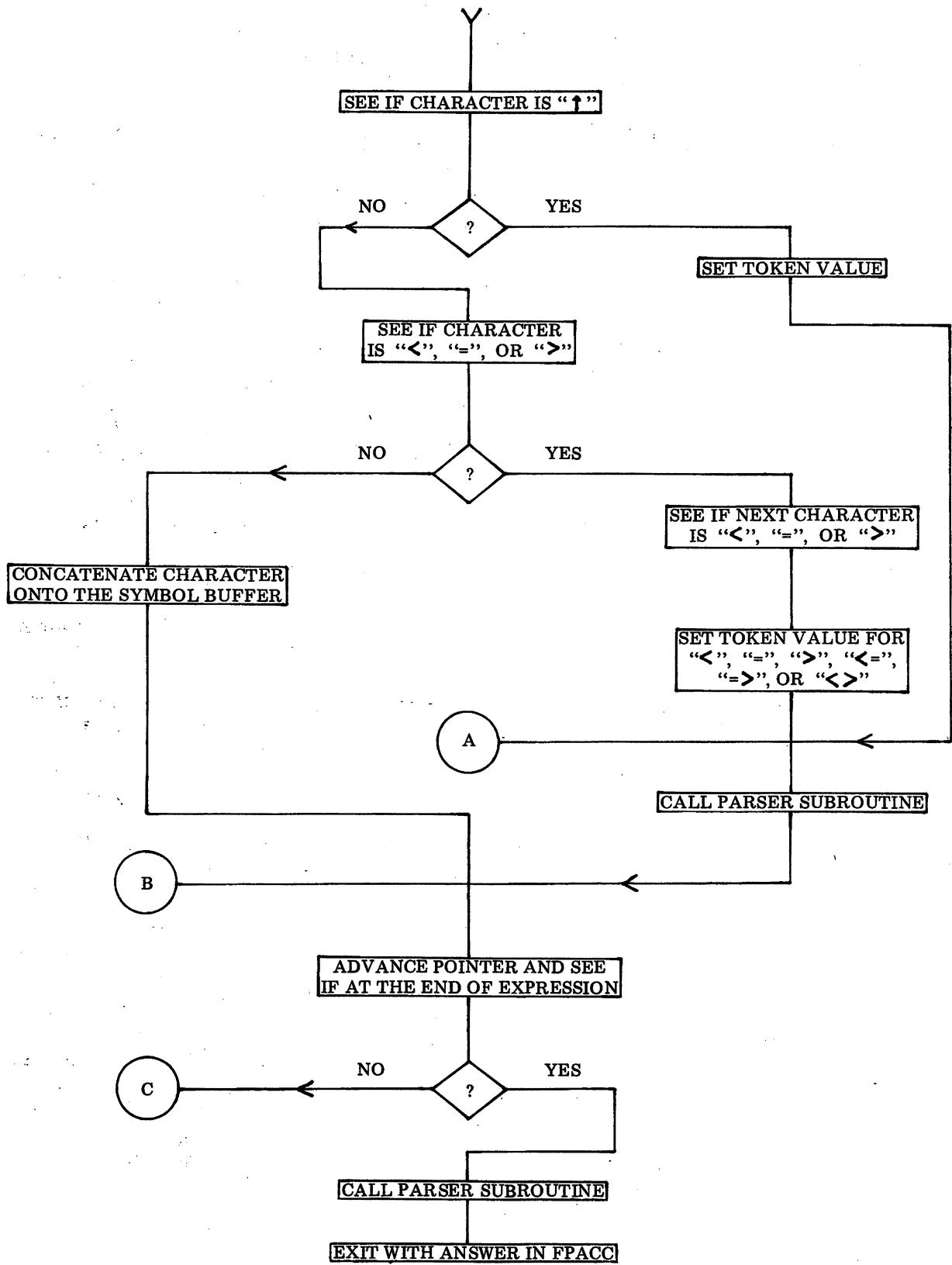
OPERATOR	TOKEN VALUE
EOS	000
+	001
-	002
*	003
/	004
↑	005
(006
)	007
<	011
=	012
>	013
<=	014
=>	015
<>	016

The presence of a parenthesis in a mathematical expression requires special consideration. As will be detailed in following chapters, a parenthesis may indicate grouping of terms, or the argument portion of a function, or the subscripted part of an array variable. When a parenthesis is detected by the EVAL subroutine, it will call on appropriate subroutines to determine what action is to be taken as a function of where the parenthesis occurs in an expression.

The overall operation of the EVAL routine is summarized in the flow chart shown on the next several pages. The source listing starts below.

EVAL,	LLI 227	Load L with address of ARITHMETIC STACK pointer
	LHI 001	** Set H to page of ARITHMETIC STACK pointer
	LMI 224	Initialize ARITH STACK pointer value to addr minus 4
	INL	Advance memory pointer to FUN/ARRAY STACK pntr
	LHI 026	** Set H to page of FUN/ARRAY STACK pointer
	LMI 000	Initialize FUN/ARRAY STACK pointer to start of stack
	CAL CLESYM	Initialize the SYMBOL BUFFER to empty condition
	LLI 210	Load L with address of OPERATOR STACK pointer
	LMI 000	Initialize OPERATOR STACK pointer value
	LLI 276	Set L to address of EVAL pointer (start of expression)
	LBM	Fetch the EVAL pointer value into register B
	LLI 200	Set up a working pointer register in this location
	LMB	And initialize EVAL CURRENT pointer





SCAN1,	LLI 200 CAL GETCHR JFZ SCAN10 CPI 253 JFZ SCAN2 LLI 176 LMI 001 JMP SCANFN	Load L with address of EVAL CURRENT pointer Fetch a character in the expression being evaluated If character is a space, jump out of this section See if character is a "+" sign If not, continue checking for an operator If yes, set pointer to PARSER TOKEN storage location Place TOKEN value for "+" sign in PARSER TOKEN Go to PARSER subroutine entry point
SCAN2,	CPI 255 JFZ SCAN4 LLI 120 LAM NDA JFZ SCAN3 LLI 176 LAM CPI 007 JFZ SCAN3 CPI 003 JFZ SYNERR CPI 005 JFZ SYNERR LLI 120 LMI 001 INL LMI 260	See if character is a minus (" - ") sign If not, continue checking for an operator If yes, check the length of the symbol stored in the SYMBOL BUFFER by fetching the (cc) byte And testing to see if (cc) is zero If length not zero, then not a unary minus indicator Else, check to see if last operator was a right parenthesis By fetching the value in the PARSER TOKEN storage Location and seeing if it is token value for ")" If last operator was ")" then do not have a unary minus Check to see if last operator was "*" If yes, then have a syntax error Check to see if last operator was exponentiation If yes, then have a syntax error If none of the above, then minus sign is unary, put Character string representing the Value zero in the SYMBOL BUFFER in string format (Character count (cc) followed by ASCII code for zero)
SCAN3,	LLI 176 LMI 002	Set L to address of PARSER TOKEN storage location Set PARSER TOKEN value for minus operator
SCANFN,	CAL PARSER JMP SCAN10	Call the PARSER subroutine to process current symbol And operator. Then jump to continue processing.
SCAN4,	CPI 252 JFZ SCAN5 LLI 176 LMI 003 JMP SCANFN	See if character fetched from expression is "*" If not, continue checking for an operator If yes, set pointer to PARSER TOKEN storage location Place TOKEN value for "*" (multiplication) operator in PARSER TOKEN and go to PARSER subroutine entry
SCAN5,	CPI 257 JFZ SCAN6 LLI 176 LMI 004 JMP SCANFN	See if character fetched from expression is "/" If not, continue checking for an operator If yes, set pointer to PARSER TOKEN storage location Place TOKEN value for "/" (division) operator in PARSER TOKEN and go to PARSER subroutine entry
SCAN6,	CPI 250 JFZ SCAN7 LLI 230 LBM INB	See if character fetched from expression is "(" If not, continue checking for an operator If yes, load L with address of FUN/ARRAY STACK Pointer. Fetch the value in the stack pointer. Increment It to indicate number of "(" operators encountered.

	LMB CAL FUNARR LLI 176 LMI 006 JMP SCANFN	Restore the updated stack pointer back to memory Call subroutine to process possible FUNCTION or ARRAY variable subscript. Then set pointer to PARSER TOKEN storage and set value for "(" operator Go to PARSER subroutine entry point.
SCAN7,	CPI 251 JFZ SCAN8 LLI 176 LMI 007 CAL PARSER CAL PRIGHT LLI 230 LHI 026 LBM DCB LMB JMP SCAN10	See if character fetched from expression is ")" If not, continue checking for an operator If yes, load L with address of PARSER TOKEN Set PARSER TOKEN value to reflect ")" Call the PARSER subroutine to process current symbol Call subroutine to handle FUNCTION or ARRAY Load L with address of FUN/ARRAY STACK pointer ** Set H to page of FUN/ARRAY STACK pointer Fetch the value in the stack pointer. Decrement it To account for left parenthesis just processed. Restore the updated value back to memory. Jump to continue processing expression.
SCAN8,	CPI 336 JFZ SCAN9 LLI 176 LMI 005 JMP SCANFN	See if character fetched from expression is "↑" If not, continue checking for an operator If yes, load L with address of PARSER TOKEN Put in value for exponentiation Go to PARSER subroutine entry point.
SCAN9,	CPI 274 JFZ SCAN11 LLI 200 LBM INB LMB CAL GETCHR CPI 275 JTZ SCAN13 CPI 276 JTZ SCAN15 LLI 200 LBM DCB LMB LLI 176 LMI 011 JMP SCANFN	See if character fetched is the "less than" sign If not, continue checking for an operator If yes, set L to the EVAL CURRENT pointer Fetch the pointer Increment it to point to the next character Restore the updated pointer value Fetch the next character in the expression Is the character the "==" sign? If so, have "less than or equal" combination Is the character the "greater than" sign? If so, have "less than or greater than" combination Else character is not part of the operator. Set L back To the EVAL CURRENT pointer. Fetch the pointer Value and decrement it back one character in the Expression. Restore the original pointer value. Have just the "less than" operator. Set L to the PARSER TOKEN storage location and set the value for The "less than" sign then go to PARSER entry point.
SCAN11,	CPI 275 JFZ SCAN12 LLI 200 LBM INB LMB CAL GETCHR	See if character fetched is the "=" sign If not, continue checking for an operator If yes, set L to the EVAL CURRENT pointer Fetch the pointer Increment it to point to the next character Restore the updated pointer value Fetch the next character in the expression

	CPI 274 JTZ SCAN13 CPI 276 JTZ SCAN14 LLI 200 LBM DCB LMB LLI 176 LMI 012 JMP SCANFN	Is the character the "less than" sign? If so, have "less than or equal" combination Is the character the "greater than" sign? If so, have "equal or greater than" combination Else character is not part of the operator. Set L back To the EVAL CURRENT pointer. Fetch the pointer Value and decrement it back one character in the Expression. Restore the original pointer value. Just have "=" operator. Set L to the PARSER TOKEN Storage location and set the value for the "=" sign. Go to the PARSER entry point.
SCAN12,	CPI 276 JFZ SCAN16 LLI 200 LBM INB LMB CAL GETCHR CPI 274 JTZ SCAN15 CPI 275 JTZ SCAN14 LLI 200 LBM DCB LMB LLI 176 LMI 013 JMP SCANFN	See if character fetched is the "greater than" sign If not, go append the character to the SYMBOL BUFF If so, set L to the EVAL CURRENT pointer Fetch the pointer Increment it to point to the next character Restore the updated pointer value Fetch the next character in the expression Is the character the "less than" sign? If so, have "less than or greater than" combination Is the character the "=" sign? If so, have the "equal to or greater than" combination Else character is not part of the operator. Set L back To the EVAL CURRENT pointer. Fetch the pointer Value and decrement it back one character in the Expression. Restore the original pointer value. Have just the "greater than" operator. Set L to the PARSER TOKEN storage location and set the value for The "greater than" sign then go to PARSER entry
SCAN13,	LLI 176 LMI 014 JMP SCANFN	When have "less than or equal" combination set L to PARSER TOKEN storage location and set the value. Then go to the PARSER entry point.
SCAN14,	LLI 176 LMI 015 JMP SCANFN	When have "equal to or greater than" combination set L To PARSER TOKEN storage location and set the value. Then go to the PARSER entry point.
SCAN15,	LLI 176 LMI 016 JMP SCANFN	When have "less than or greater than" combination set L to PARSER TOKEN storage location and set value. Then go to the PARSER entry point.
SCAN16,	CAL CONCTS	Concatenate the character to the SYMBOL BUFFER
SCAN10,	LLI 200 LHI 026 LBM INB LMB LLI 277	Set L to the EVAL CURRENT pointer storage location ** Set H to page of EVAL CURRENT pointer Fetch the EVAL CURRENT pointer value into B Increment the pointer value to point to next character In the expression and restore the updated value. Set L to EVAL FINISH storage location.

	LAM	Fetch the EVAL FINISH value into the accumulator.
	DCB	Set B to last character processed in the expression.
	CPB	See if last character was at EVAL FINISH location.
	JFZ SCAN1	If not, continue processing the expression. Else, jump
	JMP PARSEP	To final evaluation procedure and test. (Directs routine
	HLT	To a dislocated section.) Safety Halt in unused byte.
PARSEP,	LLI 176	Load L with PARSER TOKEN storage location. Set
	LMI 000	The value indicating end of expression. Call the
	CAL PARSER	PARSER subroutine for final time for the expression.
	LLI 227	Change L to point to the ARITH STACK pointer.
	LHI 001	** Set H to the page of the ARITH STACK pointer.
	LAM	Fetch the ARITH STACK pointer value.
	CPI 230	Should indicate only one value (answer) in stack.
	RTZ	Exit with answer in FPACC if ARITH STACK is O.K.
	JMP SYNERR	Else have a syntax error!

THE PARSER ROUTINE

The PARSER routine is a most important part of the mathematical expression evaluating process. The primary purpose of the routine is to arrange numbers and operators in an expression so that they may be performed in the proper order according to a set of rules. At appropriate times, the routine will call on other subroutines to perform mathematical operations.

The rules used to evaluate an expression are established according to standard mathematical practices by establishing a hierarchy among the various mathematical operators and following a consistent left to right pattern for evaluating expressions. In SCELBAL, the operating sign precedence is defined as follows.

Parenthesis, when used to enclose a group of operators and symbols (versus being used to separate the argument of a function or to indicate a subscripted variable), have the highest precedence. That is, whenever a right hand parenthesis is encountered, all of the operations signified by operators between it and the initiating left hand parenthesis, must be performed before any further processing is attempted.

Individual operators are assigned precedence according to the following hierarchy. Exponentiation has highest precedence. Next are the multiplication and division operators (having equal precedence to each other). Then comes the plus or minus operator. The lowest operator precedence is assigned to the equal, less than, or greater than operators (or combinations).

How do the rules of precedence enable the PARSER routine to correctly analyze mathematical expressions? They enable the program to determine whether to perform an operation between two symbols (numbers) joined by an operator, or whether to hold the values until more data is obtained! The process involves the use of stacking operations as will

be explained now.

The reader may recall from the previous chapter that each time the PARSER subroutine is called by EVAL, the routine will have placed a symbol (either a variable name or a number) in the SYMBOL BUFFER (unless the end of the expression had been reached which is a special case). Additionally, an operator TOKEN VALUE will have been set up for use by the PARSER routine.

The contents of the SYMBOL BUFFER are converted to a number in floating point format (using subroutines that will be presented in a later chapter). This number (which will reside in a special set of registers called the FPACC) will be considered as the top-most entry in an ARITHMETIC STACK for the purposes of the following discussion. The primary task of the PARSER is to obtain the precedence value of the operator currently being processed and determine whether or not an actual mathematical operation should be performed. This simple decision of whether or not to perform an operation is made by comparing the precedence of the current operator against any previous operator(s) it has received. If the precedence of the current operator is greater than the previous operator, then the operator is saved on an OPERATOR STACK. Remember, the numerical value of the symbol being processed has already been placed on the top of an ARITHMETIC STACK. Both of these stacks are configured as push-down, pop-up stacks (first in, last out). If the precedence of the operator just received is equal to or less than the previous operator (on the top of the OPERATOR STACK), then the operation indicated by the operator sign on the top of the OPERATOR STACK is performed between the two top-most numbers in the ARITHMETIC STACK. After this is done, the operator is removed from the OPERATOR STACK. The two values in the top of the ARITHMETIC STACK are replaced by the answer just obtained by per-

forming the operation. (It is important to note that the number in the top of the arithmetic stack operates on the number beneath it in the stack. For instance, for division the number in the top of the stack will be the divisor, the next number down will be the dividend. At the end of the operation, both the divisor and dividend will be removed from the arithmetic stack. The quotient obtained from the division process will be on the top of the arithmetic stack.) After cases where a precedence test results in an operation being performed, the precedence test is repeated against the next entry in the OPERATOR STACK (unless the stack is empty). Remember, since the operator for the operation just performed will be removed from the stack, any previous operator(s) stored in the stack will be popped-up to place a new operator in the top position. When a point is reached where the precedence fails (that is, the precedence of the current operator is greater than the sign at the top of the OPERATOR STACK), then the current operator sign is placed on the top of the stack. The routine then returns to the EVAL routine which will get the next symbol/operator pair!

The above explanation of the primary purpose of the PARSER routine may seem a bit complicated when first read. Indeed, the PARSER routine is perhaps the most complicated portion of SCELBAL. The actual operation of the major portion of the routine just described may be made somewhat clearer by following the evaluation of an example expression on a step-by-step basis.

Suppose the program is evaluating the mathematical expression:

$X \uparrow 2 + 4 * X - 16$

When the EVAL routine (presented in the preceding chapter) starts processing the expression from left to right it will first pick up the symbol X and the operator " \uparrow " which it will pass to the PARSER routine. Since the expression is just starting to be processed, both the ARITHMETIC STACK and

the OPERATOR STACK will be empty.

When the PARSER routine receives the symbol X it will determine that it is a variable name. It will call on a routine to ascertain the current value of X from a VARIABLES TABLE. This value will be placed (using floating point format) in the top of the ARITHMETIC STACK.

The TOKEN VALUE for the operator sign passed to the PARSER routine will be used to assign a precedence value to the operator using a precedence look-up table. The precedence of the operator will then be compared to the precedence of the operator currently at the top of the OPERATOR STACK. Since, at this point, the OPERATOR STACK will be empty, the current operator sign will be placed on the top of the OPERATOR STACK. Thus, at this point, the ARITHMETIC STACK and the OPERATOR STACK would have the following contents:

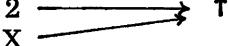
AS	OS
X	↑

(Remember, the value shown as being the top-most entry on the ARITHMETIC STACK in this discussion will actually be stored in the floating point accumulator (FPACC). This view simplifies the concept being explained.)

The PARSER routine at this point would return control back to the EVAL routine which would proceed to bring the next symbol and operator in the expression into appropriate buffers. For the example being presented this would mean the number 2 would be placed in the SYMBOL BUFFER. The token value for the operator "+" would be placed in the TOKEN VALUE register.

When the PARSER routine was again called upon, it would proceed to convert the number 2 into floating point format and store it as the top-most entry in the ARITHMETIC

STACK. The precedence for the “+” operator would be obtained and compared against that of the top-most entry in the OPERATOR STACK. At this point the two stacks would appear as:

AS	OS
2 	X 

The precedence of the current operator (plus sign) would be lower than that of the exponentiation sign on the top of the OPERATOR STACK. At this point, the operation dictated by the operator in the top of the OPERATOR STACK is performed on the top two numbers in the ARITHMETIC STACK (as indicated by the arrows in the above diagram). At the completion of this operation, the numerical result of the operation will be stored on the top of the ARITHMETIC STACK in place of the two original values that were operated on. The OPERATOR STACK will now be empty because the operator is removed from the stack once the operation has been performed. Since there are no more operators on the stack to compare against, the current “+” operator will be placed on the top of the OPERATOR STACK. The two stacks will now appear as shown here:

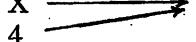
AS	OS
X ↑ 2	+

The program will then return back to the EVAL routine to obtain the next symbol and operator in the expression being processed. The next time the PARSER subroutine is entered the number 4 will be in the SYMBOL BUFFER and the token for the operator “*” (multiplication) will be in the TOKEN VALUE register. Since the precedence of the “*” sign is higher than the “+” sign on the top of the OPERATOR STACK, the new sign will be placed on the top of the stack. The

two stacks will now contain:

AS	OS
4 X ↑ 2	* +

The program will return back to the EVAL routine which will proceed to obtain the symbol X and the operator “-” from the expression. The value for X will be placed on the top of the ARITHMETIC STACK by the PARSER. The two stacks will now contain:

AS	OS
X  4 X ↑ 2	* +

Since the minus sign operator obtained by the routine has a lower precedence than the multiplication sign in the top of the OPERATOR STACK, the multiplication operation is performed between the two top entries in the ARITHMETIC STACK as indicated in the above diagram. At the completion of this operation, the two stacks will contain:

AS	OS
4 * X X ↑ 2	* +

At this point the current operator is compared with the sign which has just been popped-up to the top of the OPERATOR STACK. The current operator, being the minus sign, has the same precedence as the plus sign. This means the operation at the top of the operator stack must be performed. (Remember, if the precedence test results in the current operator being less

than OR EQUAL to the precedence of the operator in the top of the stack, that the operation is performed!) This operation is signified by the arrows in the diagram just presented. At the conclusion of this operation, the two stacks will hold:

AS	OS
$X^2 + 4 * X$	-

Once again the program will return to the EVAL routine which will proceed to pick up the final symbol in the expression (16) and then find the end of the expression. When the end of the expression is found, a special token value of zero is set up in place of an operator sign. This special zero token value has a precedence lower than any operator. When the symbol value is placed on the ARITHMETIC STACK by the PARSER routine the two stacks will register:

AS	OS
$X^2 + 4 * X$	16

Since the zero token value has a lower precedence than any operator, it means that any operators on the OPERATOR STACK will have to be performed to complete the evaluation of the expression. In the example there is only one operator left on the stack. This operation is performed. The OPERATOR STACK will then be empty. The ARITHMETIC STACK will contain the final value of the complete expression:

AS	OS
$X^2 + 4 * X - 16$	empty

The PARSER has performed its primary task!

In performing its primary task as just explained in detail, the PARSER routine has several subsections that perform related tasks. One such section is able to look-up the values of variable names in the VARIABLES TABLE and obtain the current value for the variable if the name is already present in the table. If it is not found in the table, the symbolic name is entered in the table and the initial value of zero is assigned to the variable.

Another subsection of the PARSER routine is a subsection that directs the program to perform specific mathematical operations when the PARSER has determined that they should be executed. This portion of the program uses the TOKEN VALUE assigned to the operator sign to determine which mathematical subroutines to call in order to execute the operation. The operation is performed using the top two entries in the ARITHMETIC STACK. Some of these operations, such as addition, subtraction, multiplication and division are performed by simply calling on appropriate parts of a floating point arithmetic package which is an integral part of SCELBAL. (This package is discussed in a separate chapter.)

However, a special group of operations involving the equal, less than, and greater than operators, are slightly more complex and are processed by individual routines that are presented as subsections in this chapter. These special operators have a very low precedence in the precedence hierarchy. These operators are used to actually perform comparison operations between the two top values in the ARITHMETIC STACK. If the comparison condition specified (such as less than, greater than etc., or combinations of these conditions) is found to be TRUE, then the result left in the ARITHMETIC STACK will be the value one. If the condition is not satisfied, the value zero will be left in the ARITHMETIC STACK. Thus, the PARSER is able to process conditional expressions such as those made in IF statements!

The handling of the unary minus sign by the EVAL and PARSER is a special case that should be understood by the reader. The unary minus sign is considered to be simply the case when a number is being negated (instead of subtracted). The EVAL and PARSER handle the unary minus sign by subtracting the value to be negated from zero. For instance, the evaluation of an expression such as:

$A + -B$

will actually be processed as:

$A + (0 - B)$

The reader may review the preceding chapter to see that whenever the EVAL routine picks up a unary minus sign in an expression, it will load the SYMBOL BUFFER with the value zero so that the PARSER will perform the negation on the next symbol that is passed to it. Because of the method used to handle the unary minus case, expressions are prohibited from containing double operators such as:

$A * -B$ or $A \uparrow -B$

because they would be processed as:

$A * 0 - B$ or $A \uparrow 0 - B$

(A times zero minus B or A raised to the zero power, with B subtracted from the result.)

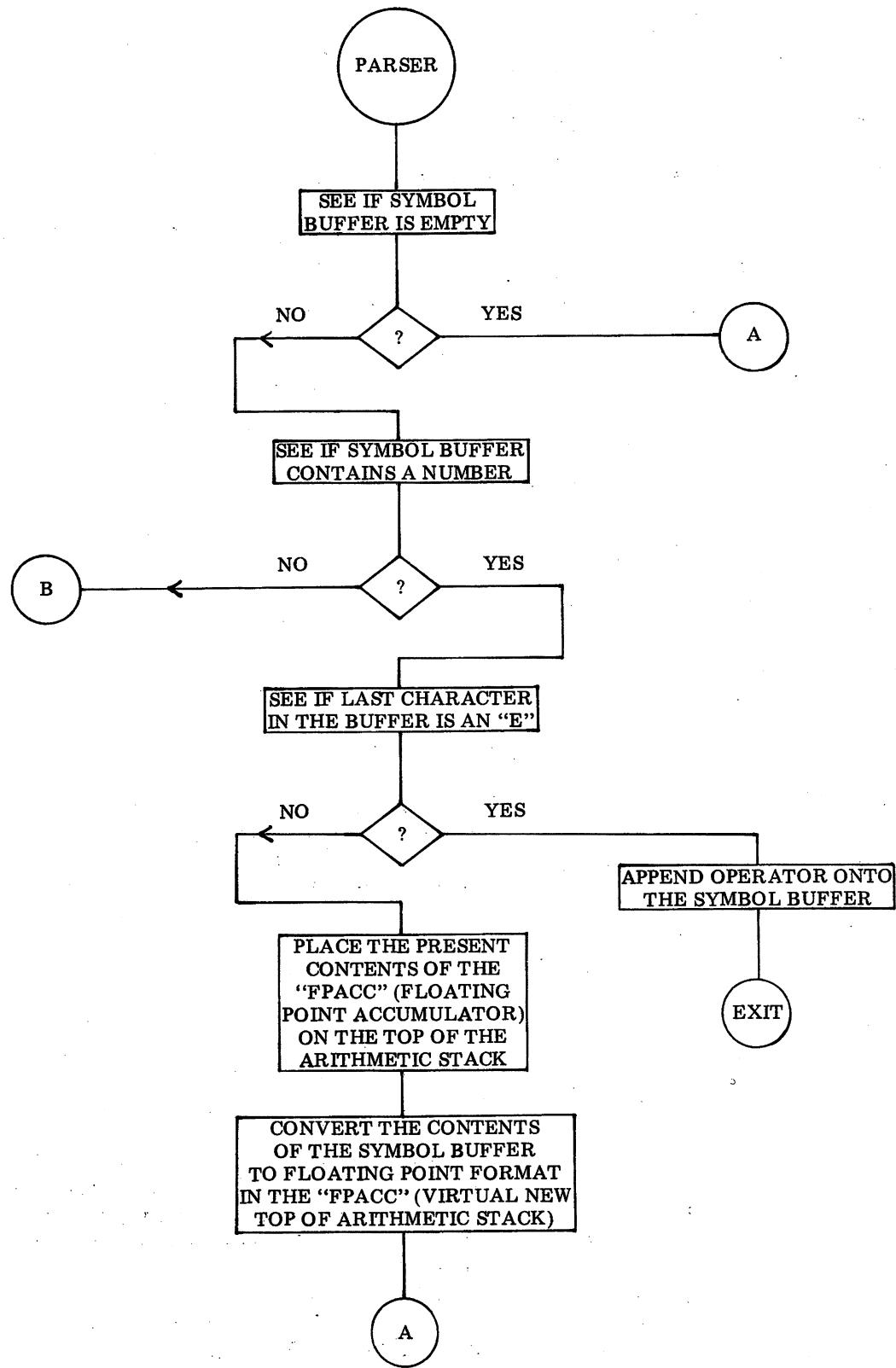
Thus, when using the unary minus sign with such operators, it is necessary to enclose the value to be negated in parenthesis thus:

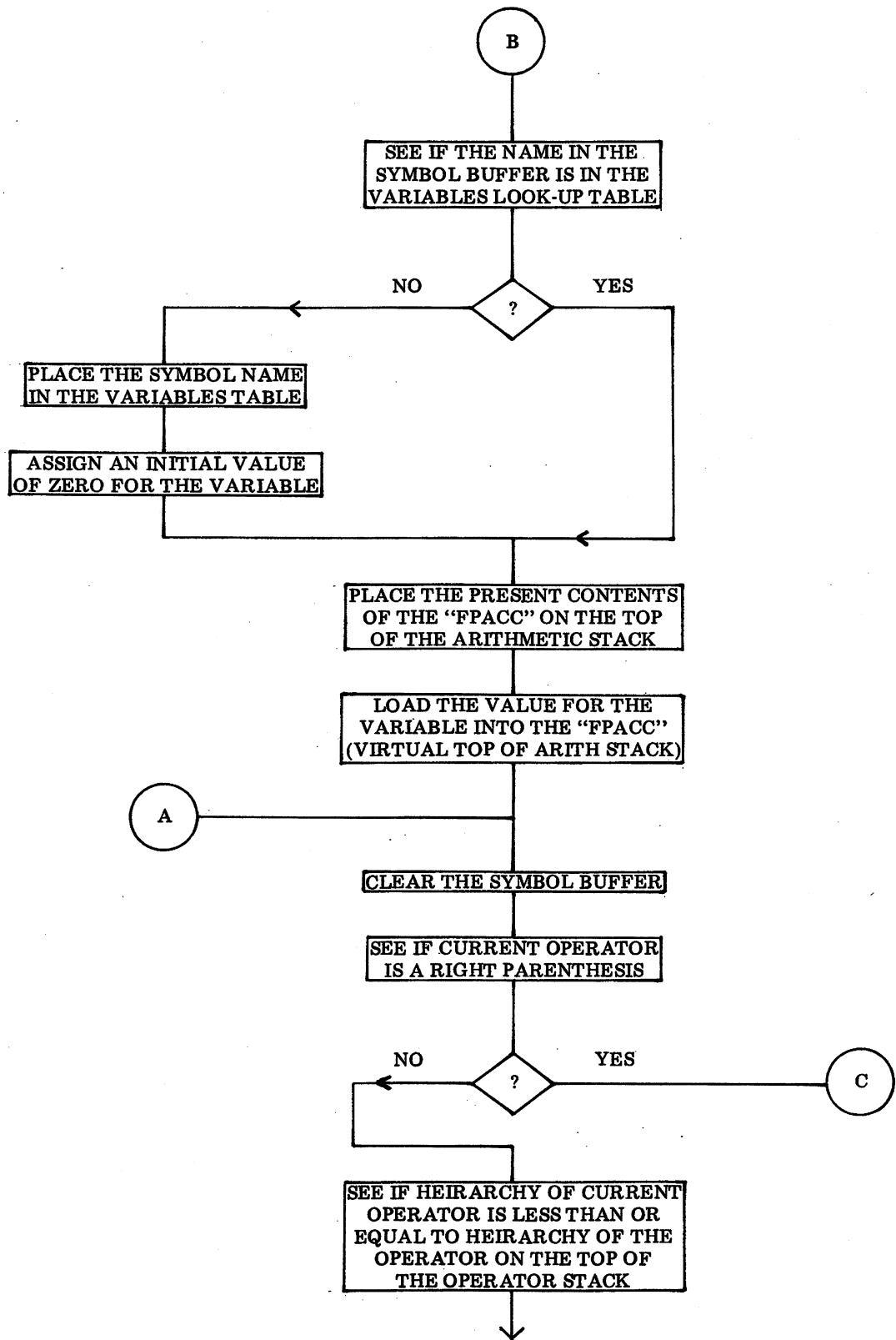
$A * (-B)$ or $A \uparrow (-B)$

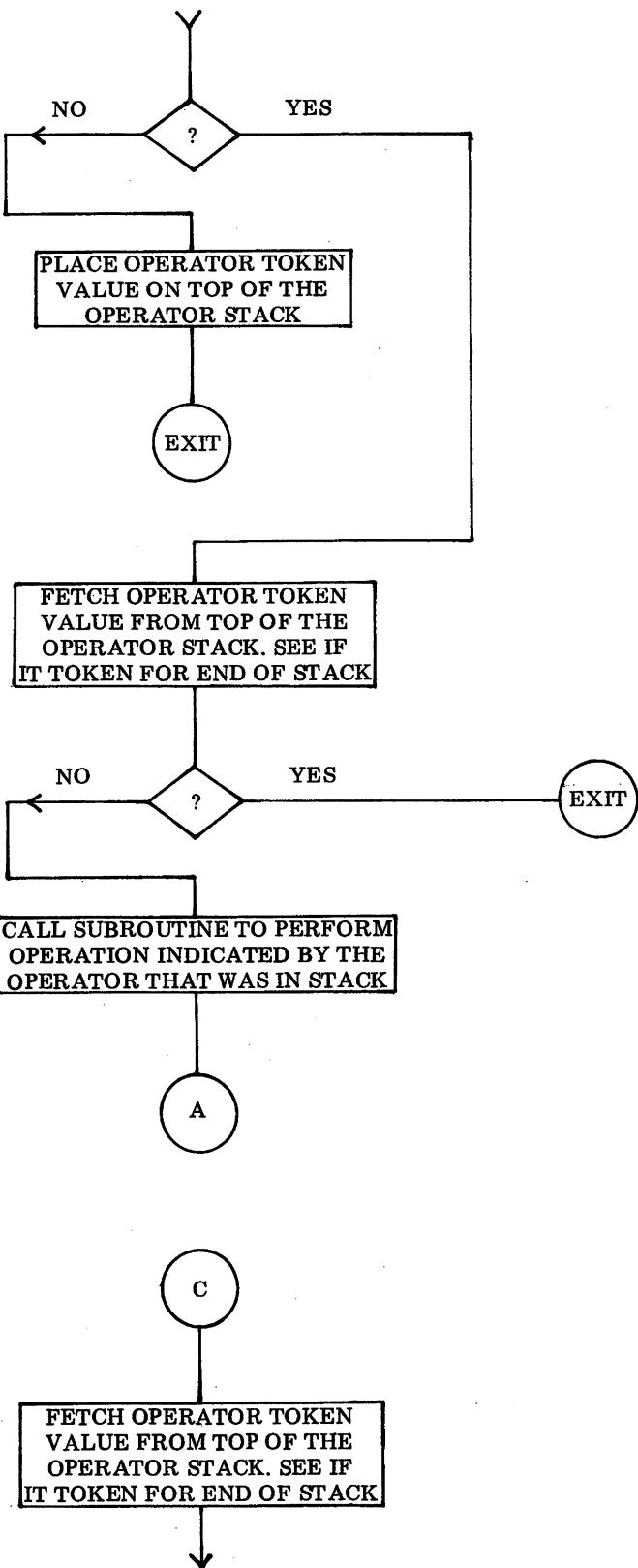
Expressions so stated can then be handled correctly by the EVAL and PARSER subroutines. (The reader may review the EVAL routine to see that incorrect use of the unary minus sign in expressions will result in a syntax error message being generated.)

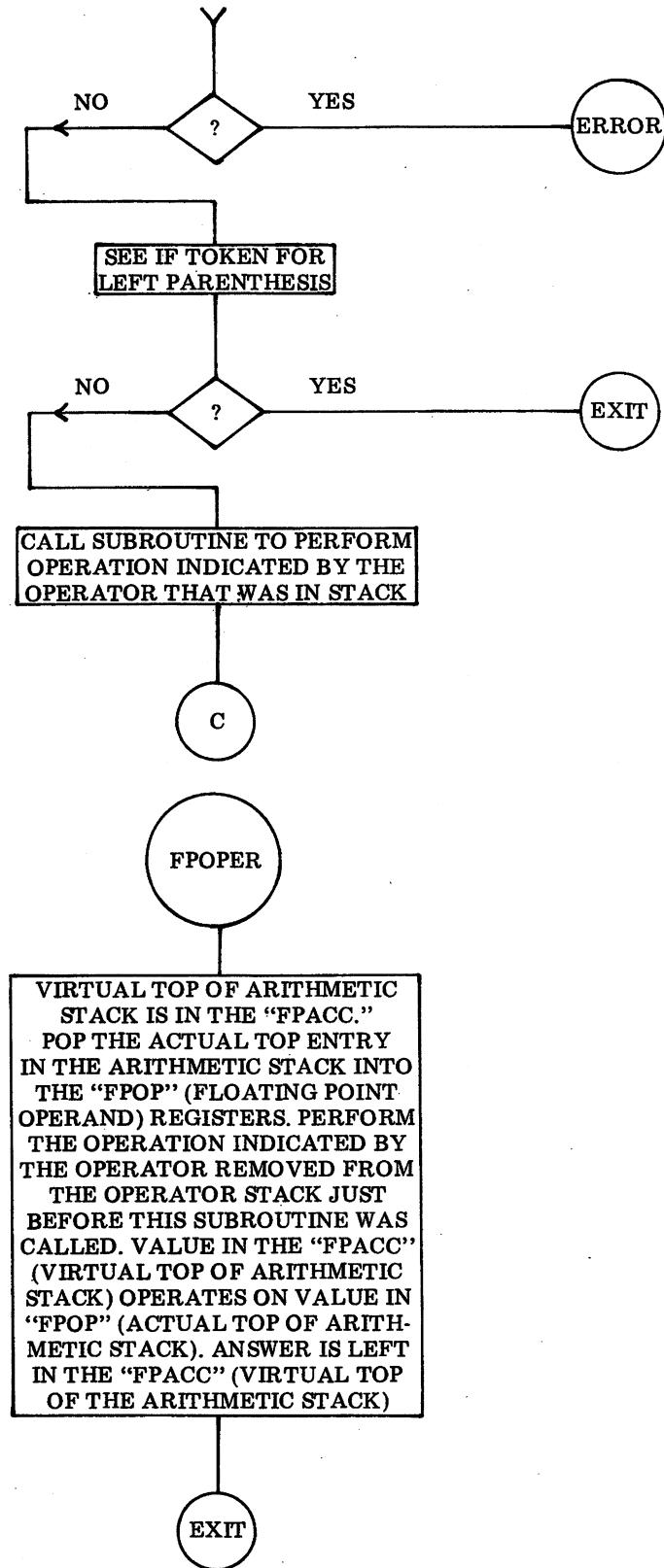
The flow of operations handled by the PARSER is illustrated by the flow chart presented on the next several pages. The source listing starts below.

PARSER,	LLI 120 LHI 026 LAM NDA JTZ PARSE INL LAM CPI 256 JTZ PARNUM CPI 260 JTS LOOKUP CPI 272 JFS LOOKUP	Load L with starting address of SYMBOL BUFFER ** Load H with page of SYMBOL BUFFER Fetch the (cc) for contents of SYMBOL BUFFER Into the ACC and see if buffer is empty If empty then no need to convert contents If not empty, advance buffer pointer Fetch the first character in the buffer See if it is ASCII code for decimal sign If yes, consider contents of buffer to be a number If not decimal sign, see if first character represents A decimal digit, if not, should have a variable Continue to test for a decimal digit If not, go look up the variable name
PARNUM,	DCL LAM CPI 001 JTZ NOEXPO ADL LLA LAM CPI 305	If SYMBOL BUFFER contains number, decrement Buffer pointer back to (cc) and fetch it to ACC See if length of string in buffer is just one If so, cannot have number with scientific notation If not, add length to buffer pointer to Point to last character in the buffer Fetch the last character in buffer and see if it Represents letter E for Exponent









	JFZ NOEXPO LLI 200 CAL GETCHR JMP CONCTS	If not, cannot have number with scientific notation If yes, have part of a scientific number, set pointer to Get the operator that follows the E and append it to The SYMBOL BUFFER and return to EVAL routine
NOEXPO,	LLI 227 LHI 001 LAM ADI 004 LMA LLA CAL FSTORE LLI 120 LHI 026 CAL DINPUT JMP PARSE	Load L with address of ARITHMETIC STACK pointer ** Load H with page of ARITHMETIC STACK pointer Fetch AS pointer value to ACC and add four to account For the number of bytes required to store a number in Floating point format. Restore pointer to memory. Then, change L to point to entry position in the AS Place contents of the FPACC onto top of the AS Change L to point to start of the SYMBOL BUFFER ** Set H to page of the SYMBOL BUFFER Convert number in the buffer to floating point format In the FPACC then jump to check operator sign.
LOOKUP,	LLI 370 LHI 026 LMI 000 LLI 120 LDI 027 LEI 210 LAM CPI 001 JFZ LOOKU1 LLI 122 LMI 000	Load L with address of LOOK-UP COUNTER ** Load H with page of the counter Initialize the counter to zero Load L with starting address of the SYMBOL BUFFER ** Load D with page of the VARIABLES TABLE Load E with start of the VARIABLES TABLE Fetch the (cc) for the string in the SYMBOL BUFFER See if the name length is just one character. If not, Should be two so proceed to look-up routine. Else, Change L to second character byte in the buffer and set It to zero to provide compatibility with entries in table
LOOKU1,	LLI 121 LHI 026 CAL SWITCH LAM INL LBM INL CAL SWITCH CPM JFZ LOOKU2 INL LAB CPM JTZ LOOKU4	Load L with addr of first character in the SYMBOL ** BUFFER. Set H to page of the SYMBOL BUFFER. Exchange contents of D&E with H&L so that can Fetch the first character of a name in the VARIABLES TABLE. Advance the table pointer and save the Second byte of name in B. Then advance the pointer Again to reach first byte of floating point formatted Number in table. Now exchange D&E with H&L and Compare first byte in table against first char in buffer If not the same, go try next entry in table. If same, Advance pointer to next char in buffer. Transfer the Character in B (second byte in table entry) to the ACC Compare it against second character in the buffer. If match, have found the name in the VARIABLES tbl.
LOOKU2,	CAL AD4DE LLI 370 LHI 026 LBM INB LMB LLI 077	Call subroutine to add four to the pointer in D&E to Advance the table pointer over value bytes. Then set ** Up H and L to point to LOOK-UP COUNTER. Fetch counter value (counts number of entries tested In the VARIABLES TABLE), increment it And restore it back to memory Load L with address of SYMBOL VARIABLES counter

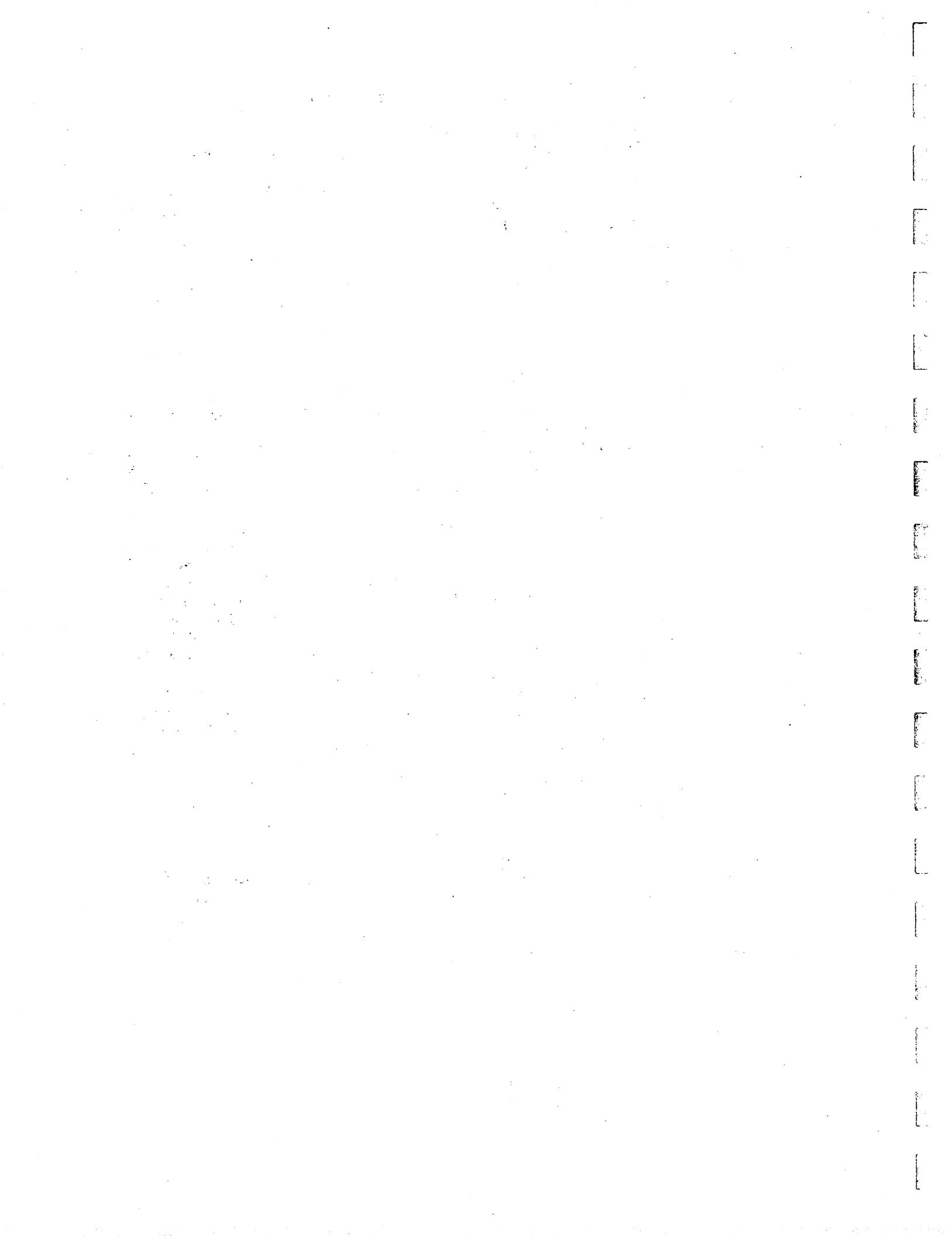
	LHI 027	** Do same for H. (Counts number of names in table.)
	LAB	Place LOOK-UP COUNTER value in the accumulator.
	CPM	Compare it with number of entries in the table.
	JFZ LOOKU1	If have not reached end of table, keep looking for name.
	LLI 077	If reach end of table without match, need to add name
	LHI 027	** To table. First set H & L to the SYMBOL
	LBM	VARIABLES counter. Fetch the counter value and
	INB	Increment to account for new name being added to the
	LMB	Table. Restore the updated count to memory. Also,
	LAB	Move the new counter value to the accumulator and
	CPI 025	Check to see that table size is not exceeded. If try to
	JFS BIGERR	Go over 20 (decimal) entries then have BiG error.
	LLI 121	Else, set L to point to first character in the SYMBOL
	LHI 026	** BUFFER and set H to proper page. Set the number
	LBI 002	Of bytes to be transferred into register B as a counter.
	CAL MOVEIT	Move the symbol name from the buffer to the
	LLE	VARIABLES TABLE. Now set up H & L with value
	LHD	Contained in D & E after moving ops (points to first
	XRA	Byte of the value to be associated with the symbol
	LMA	Name.) Clear the accumulator and place zero in all four
	INL	Bytes associated with the variable name entered
	LMA	In the VARIABLES TABLE
	INL	In order to
	LMA	Assign an
	INL	Initial value
	LMA	To the variable name
	LAL	Then transfer the address in L to the accumulator
	SUI 004	Subtract four to reset the pointer to start of zeroing ops
	LEA	Restore the address in D & E to be in same state as if
	LDH	Name was found in the table in the LOOKUP routine
LOOKU4,	CAL SAVEHL	Save current address to VARIABLES TABLE
	LLI 227	Load L with address of ARITHMETIC STACK pointer
	LHI 001	** Load H with page of the pointer
	LAM	Fetch the AS pointer value to the accumulator
	ADI 004	Add four to account for next floating point formatted
	LMA	Number to be stored in the stack. Restore the stack
	LLA	Pointer to memory and set it up in register L too.
	CAL FSTORE	Place the value in the FPACC on the top of the
	CAL RESTHL	ARITHMETIC STACK. Restore the VARIABLES
	CAL SWITCH	TABLE pointer to H&L and move it to D&E. Now load
	CAL FLOAD	The VARIABLE value from the table to the FPACC.
PARSE,	CAL CLESYM	Clear the SYMBOL BUFFER
	LLI 176	Load L with address of PARSER TOKEN VALUE
	LAM	And fetch the token value into the accumulator
	CPI 007	Is it token value for right parenthesis ")" ? If so, have
	JTZ PARSE2	Special case where must perform ops til find a "(" !
	ADI 240	Else, form address to HEIRARCHY IN table and
	LLA	Set L to point to HEIRARCHY IN VALUE in the table
	LBM	Fetch the hierarchy value from the table to register B

	LLI 210	Set L to OPERATOR STACK pointer storage location
	LCM	Fetch the OS pointer into CPU register C
	CAL INDEXC	Add OS pointer to address of OS pointer storage loc
	LAM	Fetch the token value for the operator at top of the OS
	ADI 257	And form address to HEIRARCHY OUT table
	LLA	Set L to point to HEIRARCHY OUT VALUE in the
	LAB	Table. Move the HEIRARCHY IN value to the ACC.
	CPM	Compare the HEIRARCHY IN with the HEIRARCHY
	JTZ PARSE1	OUT value. If heirarchy of current operator equal to or
	JTS PARSE1	Less than operator on top of OS stack, perform
	LLI 176	Operation indicated in top of OS stack. Else, fetch the
	LBM	Current operator token value into register B.
	LLI 210	Load L with address of the OPERATOR STACK pptr
	LCM	Fetch the stack pointer value
	INC	Increment it to account for new entry on the stack
	LMC	Restore the stack pointer value to memory
	CAL INDEXC	Form pointer to next entry in OPERATOR STACK
	LMB	Place the current operator token value on top of the OS
	RET	Exit back to the EVAL routine.
PARSE1,	LLI 210	Load L with address of the OPERATOR STACK pptr
	LAM	Fetch the stack pointer value to the accumulator
	ADL	Add in the value of the stack pointer address to form
	LLA	Address that points to top entry in the OS
	LAM	Fetch the token value at the top of the OS to the ACC
	NDA	Check to see if the token value is zero for end of stack
	RTZ	Exit back to the EVAL routine if stack empty
	LLI 210	Else, reset L to the OS pointer storage location
	LCM	Fetch the pointer value
	DCC	Decrement it to account for operator removed from
	LMC	The OPERATOR STACK and restore the pointer value
	CAL FPOPER	Perform the operation obtained from the top of the OS
	JMP PARSE	Continue to compare current operator against top of OS
PARSE2,	LLI 210	Load L with address of the OPERATOR STACK pptr
	LHI 026	** Load H with page of the pointer
	LAM	Fetch the stack pointer value to the accumulator
	ADL	Add in the value of the stack pointer address to form
	LLA	Address that points to top entry in the OS
	LAM	Fetch the token value at the top of the OS to the ACC
	NDA	Check to see if the token value is zero for end of stack
	JTZ PARNER	If end of stack, then have a parenthesis error condx
	LLI 210	Else, reset L to the OS pointer storage location
	LCM	Fetch the pointer value
	DCC	Decrement it to account for operator removed from
	LMC	The OPERATOR STACK and restore the pointer value
	CPI 006	Check to see if token value is "(" to close parenthesis
	RTZ	If so, exit back to EVAL routine.
	CAL FPOPER	Else, perform the op obtained from the top of the OS
	JMP PARSE2	Continue to process data in parenthesis

FPOPER,	LLI 371 LHI 026 LMA LLI 227 LHI 001 LAM LLA CAL OPLOAD LLI 227 LAM SUI 004 LMA LLI 371 LHI 026 LAM CPI 001 JTZ FPADD CPI 002 JTZ FPSUB CPI 003 JTZ FPMULT CPI 004 JTZ FPDIV CPI 005 JTZ INTEXP CPI 011 JTZ LT CPI 012 JTZ EQ CPI 013 JTZ GT CPI 014 JTZ LE CPI 015 JTZ GE CPI 016 JTZ NE	Load L with address of TEMP OP storage location ** Load H with page of TEMP OP storage location Store OP (from top of OPERATOR STACK) Change L to address of ARITHMETIC STACK pointer ** Load H with page of AS pointer Fetch AS pointer value into ACC Set L to top of ARITHMETIC STACK Transfer number from ARITHMETIC STACK to FPOP Restore pointer to AS pointer Fetch the pointer value to the ACC and subtract four To remove top value from the ARITHMETIC STACK Restore the updated AS pointer to memory Set L to address of TEMP OP storage location ** Set H to page of TEMP OP storage location Fetch the operator token value to the ACC Find out which kind of operation indicated Perform addition if have plus operator If not plus, see if minus Perform subtraction if have minus operator If not minus, see if multiplication Perform multiplication if have multiplication operator If not multiplication, see if division Perform division if have division operator If not division, see if exponentiation Perform exponentiation if have exponentiation operator If not exponentiation, see if "less than" operator Perform comparison for "less than" op if indicated If not "less than" see if have "equal" operator Perform comparison for "equal" op if indicated If not "equal" see if have "greater than" operator Perform comparison for "greater than" op if indicated If not "greater than" see if have "less than or equal" op Perform comparison for the combination op if indicated See if have "equal to or greater than" operator Perform comparison for the combination op if indicated See if have "less than or greater than" operator Perform comparison for the combination op if indicated
PARNER,	LLI 230 LHI 026 LMI 000 LAI 311 LCI 250 JMP ERROR	If cannot find operator, expression is not balanced ** Set H and L to address of F/A STACK pointer Clear the F/A STACK pointer to re-initialize Load ASCII code for letter I into the accumulator And code for "(" character into register C Go display I(for "Imbalanced Parenthesis) error msg
LT,	CAL FPSUB LLI 126 LAM NDA JTS CTRUE JMP CFALSE	Subtract contents of FPACC from FPOP to compare Set L to point to the MSW of the FPACC (Contains Result of the subtraction.) Fetch the MSW of the FPACC to the accumulator and test to see if result is Positive or negative. Set up the FPACC as a function Of the result obtained.

EQ,	CAL FPSUB LLI 126 LAM NDA JTZ CTRUE JMP CFALSE	Subtract contents of FPACC from FPOP to compare Set L to point to the MSW of the FPACC (Contains Result of the subtraction.) Fetch the MSW of the FPACC to the accumulator and test to see if result is Equal. Set up the FPACC as a function Of the result obtained.
GT,	CAL FPSUB LLI 126 LAM NDA JTZ CFALSE JFS CTRUE JMP CFALSE	Subtract contents of FPACC from FPOP to compare Set L to point to the MSW of the FPACC (Contains Result of the subtraction.) Fetch the MSW of the FPACC to the accumulator and test to see if result is Positive, Negative, or Equal. Set up the FPACC As a function Of the result obtained.
LE,	CAL FPSUB LLI 126 LAM NDA JTZ CTRUE JTS CTRUE JMP CFALSE	Subtract contents of FPACC from FPOP to compare Set L to point to the MSW of the FPACC (Contains Result of the subtraction.) Fetch the MSW of the FPACC to the accumulator and test to see if result is Positive, Negative, or Equal. Set up the FPACC As a function Of the result obtained
GE,	CAL FPSUB LLI 126 LAM NDA JFS CTRUE JMP CFALSE	Subtract contents of FPACC from FPOP to compare Set L to point to the MSW of the FPACC (Contains Result of the subtraction.) Fetch the MSW of the FPACC to the accumulator and test to see if result is Positive or Negative. Set up the FPACC As a function of the result obtained
NE,	CAL FPSUB LLI 126 LAM NDA JTZ CFALSE	Subtract contents of FPACC from FPOP to compare Set L to point to the MSW of the FPACC (Contains Result of the subtraction.) Fetch the MSW of the FPACC to the accumulator and test to see if result is Equal. Set up the FPACC as a function of the result.
CTRUE, FPONE,	LLI 004 JMP FLOAD	Load L with address of floating point value +1.0 Load FPACC with value +1.0 and exit to caller
CFALSE,	LLI 127 LMI 000 JMP FPZERO	Load L with address of FPACC Exponent register Set the FPACC Exponent to zero and then set the Mantissa portion of the FPACC to zero. Exit to caller.
AD4DE,	LAE ADI 004 LEA RET	Subroutine to add four to the value in register E. Move contents of E to the ACC and add four. Restore the updated value back to register E. Return to the calling routine.

INTEXP,	LLI 126 LHI 001 LAM LLI 003 LMA NDA JTZ FPONE CTS FPCOMP CAL FPFIX LLI 124 LBM LLI 013 LMB LLI 134 LEI 014 LHI 001 LDH LBI 004 CAL MOVEIT CAL FPONE LLI 003 LAM NDA JTS DVLOOP	Load L with address of MSW of FPACC (Floating Point ** ACCumulator). Load H with page of FPACC. Fetch MSW of the FPACC into the accumulator. Load L with address of EXP TEMP storage location Store the FPACC MSW value in EXP TEMP location Test contents of the MSW of the FPACC. If zero, then Set FPACC equal to +1.0 (any nr to zero power = 1.0!) If MSW indicates negative number, complement The FPACC. Then convert floating point number to Fixed point. Load L with address of LSW of fixed nr Fetch the LSW into CPU register B. Set L to address of EXPONENT COUNTER Place the fixed value in the EXP CNTR to indicate Number of multiplications needed (power). Now set L To LSW of FPOP and E to address of FP TEMP (LSW) ** Set H to floating point working area page. Set D to same page address. Set transfer (precision) counter. Call subroutine to move Contents of FPOP into FP TEMP registers to save Original value of FPOP. Now set FPACC to +1.0. Load L with pointer to original value of FPACC (Stored in FP TEMP) MSW and fetch contents to ACC. Test to see if raising to a negative power. If so, divide Instead of multiply!
MULLOOP,	LLI 014 CAL FACXOP CAL FPMULT LLI 013 LBM DCB LMB JFZ MULLOOP RET	Load L with address of LSW of FP TEMP (original Value in FPOP). Move FP TEMP into FPOP. Multiply FPACC by FPOP. Result left in FPACC. Load L with address of EXPONENT COUNTER. Fetch the counter value Decrement it Restore it to memory If counter not zero, continue exponentiation process When have raised to proper power, return to caller.
DVLOOP,	LLI 014 CAL FACXOP CAL FPDIV LLI 013 LBM DCB LMB JFZ DVLOOP RET	Load L with address of LSW of FP TEMP (original Value in FPOP). Move FP TEMP into FPOP. Divide FPACC by FPOP. Result left in FPACC. Load L with address of EXPONENT COUNTER. Fetch the counter value Decrement it Restore to memory If counter not zero, continue exponentiation process When have raised to proper power, return to caller.



FUNCTION AND OPTIONAL ARRAY HANDLING ROUTINES

When a mathematical expression is being evaluated by SCELBAL the presence of a parenthesis sign can indicate one of three possible conditions. The parenthesis may simply be used to group parts of a mathematical formula such as in the example:

$$((X + 2) * (X - 3))/(X + 4)$$

When parentheses are used in this manner, they are processed by the appropriate portions of the EVAL and PARSER routines previously described.

A second way in which parentheses may be used is when they isolate the argument portion of a function, such as in the examples illustrated here:

INT(X)

or

RND(0)

or

TAB(12)

The third case in which a parenthesis may be used is to indicate the subscripted part of an array variable:

A(1), A(2),...,A(8)

such as would occur for an array that had a DIMension of eight.

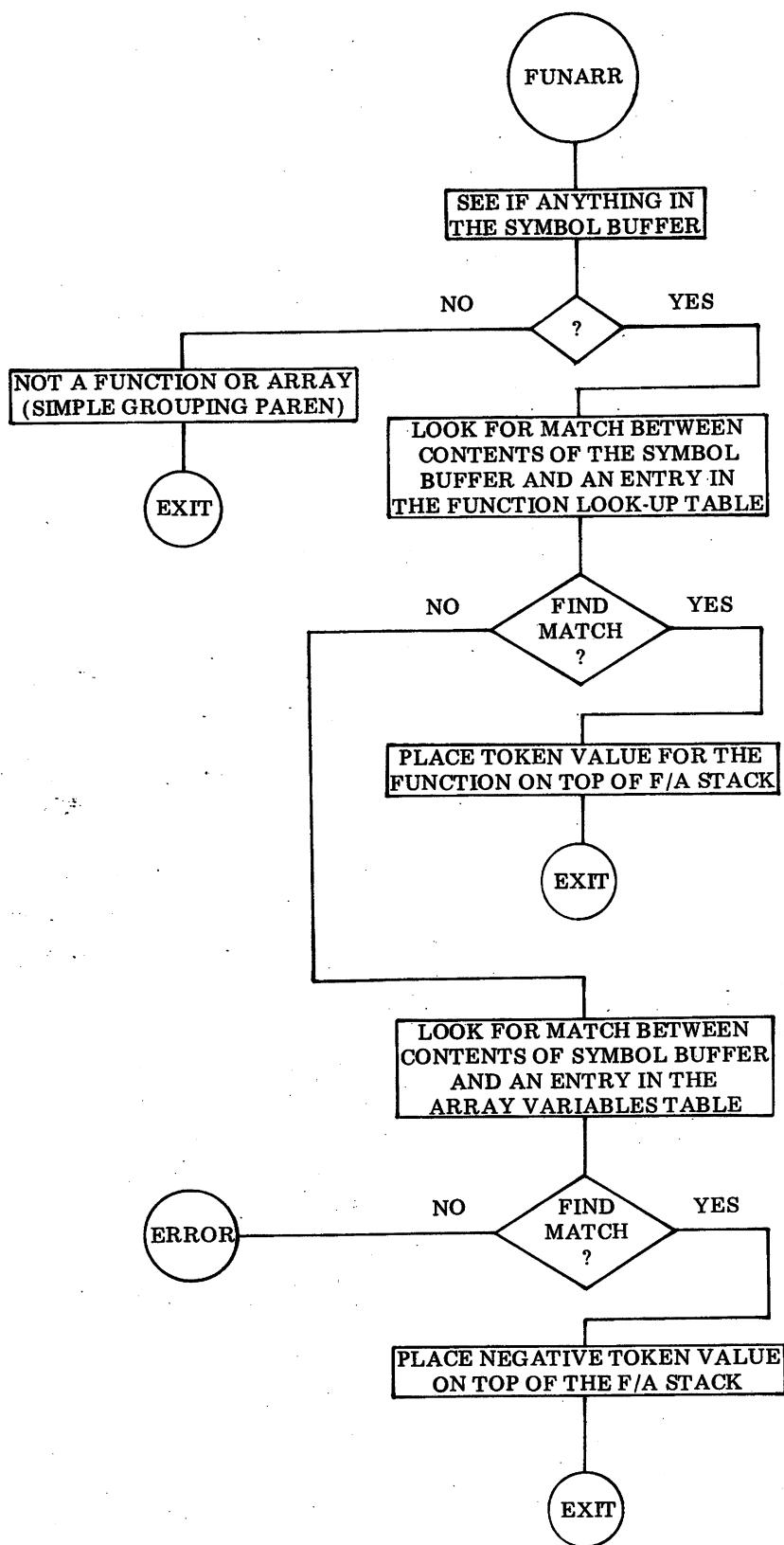
SCELBAL must be capable of distinguishing the purpose of a parenthesis whenever one is encountered and taking appropriate action once that purpose has been ascertained.

The process of determining the purpose of a parenthesis is handled by a subroutine to be presented shortly referred to by the label FUNARR (FUNction or ARRray handler). This subroutine is called by the EVAL routine

presented previously whenever it encounters a left hand ("(") parenthesis sign while processing an expression. The flow chart on the next page illustrates the basic operation of the FUNARR subroutine.

Essentially, the subroutine first determines whether the parenthesis is simply being used to group mathematical terms by checking to see if there is anything in the SYMBOL BUFFER. If there is anything in the symbol buffer it should either be the name of a function or the symbolic name for an array variable. A check for a function name is made by scanning a FUNCTION LOOK-UP table for a match between an entry in it and the character string in the SYMBOL BUFFER. Upon finding a match, a FUNCTION TOKEN VALUE is set up in a stack called the F/A STACK. This token value for a function will always be positive in value. (It is simply the position of the function name in the name table!) If the data in the SYMBOL BUFFER does not represent a function name, and if the user desires to utilize the optional array handling capability of SCELBAL, another subroutine (labeled FUNAR2) is called upon to see if the character in the SYMBOL BUFFER is an array variable by looking for a match with it in the ARRAY VARIABLES TABLE (discussed previously in the chapter describing the optional DIM statement). If the name is found in the table, a negative token value (corresponding to the position of the array name in the table) is established and placed in the F/A STACK.

The routine that handles the processing of subscripted array names is left out of the program if the user does not desire to incorporate the optional DIM statement and associated capability in SCELBAL. If it is left out, the reference instruction to it is changed to a no-operation instruction (indicated in the listing by the @@ notation) so that the routine will issue an error message if the program user attempts to subscript a variable when array capability is not implemented.



FUNARR,	LLI 120 LHI 026 LAM NDA RTZ LLI 202 LHI 027 LMI 000	Load L with starting address of SYMBOL BUFFER ** Load H with page of SYMBOL BUFFER Fetch the (cc) for contents of buffer to the ACC See if (cc) is zero, if so buffer is empty, return to Caller as have simple grouping parenthesis sign Else set L to TEMP COUNTER location ** Set H to TEMP COUNTER page Initialize TEMP COUNTER to zero
FUNAR1,	LLI 202 LHI 027 LBM INB LMB LCI 002 LLI 274 LHI 026 CAL TABADR LDI 026 LEI 120 CAL STRCP JTZ FUNAR4 LLI 202 LHI 027 LAM CPI 010 JFZ FUNAR1 LLI 202 LHI 027 LMI 000 JMP FUNAR2	Load L with address of TEMP COUNTER ** Load H with page of TEMP COUNTER Fetch the counter value to register B Increment the counter Restore the updated value to memory Initialize C to a value of two for future ops Load L with starting address (less four) of FUNCTION ** LOOK-UP TABLE. Set H to table page. Find address of next entry in the table ** Load D with page of SYMBOL BUFFER Load E with starting address of SYMBOL BUFFER Compare entry in FUNCTION LOOK-UP TABLE with Contents of SYMBOL BUFFER. If find match, go set Up the function token value. Else, set L to the TEMP ** COUNTER and set H to the proper page. Fetch the Current counter value and see if have tried all eight Possible functions in the table. If not, go back and check the next entry. If have tried all of the entries in the table, set L ** As well as H to the address of the TEMP COUNTER And reset it to zero. Now go see if have subscripted @@ Array (unless array capability not in program).
FAERR,	LLI 230 LHI 026 LMI 000 LAI 306 LCI 301 JMP ERROR	Load L with address of F/A STACK pointer ** Load H with page of F/A STACK pointer Clear the F/A STACK pointer to reset on an error Load the ASCII code for letter F into the ACC Load the ASCII code for letter A into register C Go display the FA error message
FUNAR4,	LLI 202 LHI 027 LBM LLI 230 LHI 026 LCM CAL INDEXC LMB JMP CLESYM	Load L with address of TEMP COUNTER ** Set H to page of TEMP COUNTER Load value in counter to register B. This is FUNCTION TOKEN VALUE. Change L to F/A STACK pointer. ** Load H with page of F/A STACK pointer. Fetch the F/A STACK pointer value into register C. Form the address to the top of the F/A STACK. Store the FUNCTION TOKEN VALUE in the F/A STACK. Then exit by clearing the SYMBOL BUFFER.

TABADR,	LAB	Move the TEMP COUNTER value from B to ACC
TABAD1,	RLC	Multiply by four using this loop to form value equal
	DCC	To number of bytes per entry (4) times current entry
	JFZ TABAD1	In the FUNCTION LOOK-UP TABLE.
	ADL	Add this value to the starting address of the table.
	LLA	Form pointer to next entry in table
	RFC	If no carry return to caller
	INH	Else, increment H before
	RET	Returning to caller
The following routine is only installed if the user desires to utilize single dimension array capability. This and associated array routines, if installed, will be in a separate area in memory apart from the standard SCELBAL routines.		
FUNAR2,	LLI 202	Load L with address of TEMP COUNTER
	LHI 027	** Load H with page of counter
	LBM	Fetch the counter value
	INB	Increment the value
	LMB	Restore the value to memory
	LCI 002	Initialize register C to a value of two for future ops
	LLI 114	Load L with address of start of ARRAY VARIABLES
	LHI 027	** TABLE (less four). Set H to page of the table.
	CAL TABADR	Calculate address of start of next name in table.
	LDI 026	** Load D with page of the SYMBOL BUFFER
	LEI 120	Set E to starting address of the SYMBOL BUFFER
	CAL STRCP	Compare name in ARRAY VARIABLES table to the
	JTZ FUNAR3	Contents of the SYMBOL BUFFER. If match, go set up
	LLI 202	Array token value. Else, reset L to address of TEMP
	LHI 027	** COUNTER. Set H to page of TEMP COUNTER.
	LAM	Fetch the counter value into the accumulator.
	LLI 075	Change L to number of arrays storage location.
	CPM	Compare number of entries checked against number
	JFZ FUNAR2	Possible. Keep searching table if not finished.
	JMP FAERR	If finished and no match than have F/A error condx.
FUNAR3,	LLI 202	Load L with address of TEMP COUNTER
	LHI 027	** Load H with page of counter.
	XRA	Clear the accumulator. Subtract the value in the TEMP
	SBM	COUNTER from zero to obtain two's complement.
	LMA	Place this back in counter location as ARRAY TOKEN
	JMP FUNAR4	VALUE (negative). Go place the value on F/A STACK.

The routines just presented take care of determining what type of purpose a parenthesis is being used for when the left hand parenthesis sign is encountered in an expression. There is, of course, still more to do!

The information enclosed in a set of parenthesis will either be argument portion of a function, the subscript of an array variable, or the terms that make up a mathematical expression when the parenthesis is used

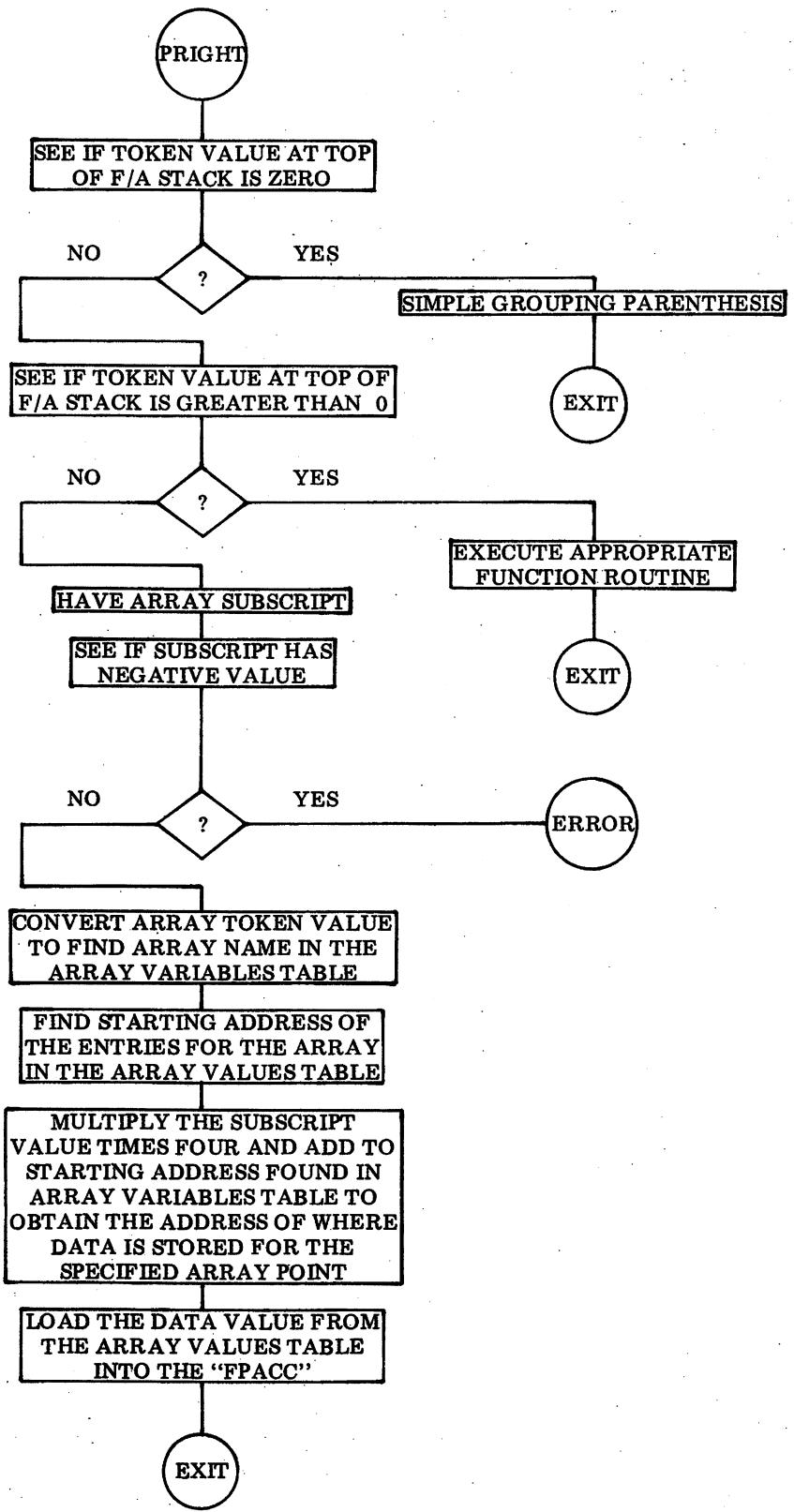
for grouping purposes. The latter case is taken care of between the EVAL and PARSER routines previously described as they simply proceed to evaluate all the terms enclosed by the current parenthesis before proceeding any further with the process of scanning the expression. Handling the cases involving functions or array variables is initiated when the EVAL routine detects a right hand ("") parenthesis sign and calls on the subroutine to be described next labeled PRIGHT.

The flow chart on the following page illustrates the key tasks of the PRIGHT subroutine and a supporting (optional) subroutine labeled PRIGH1. The routine portion starting with the label PRIGH1 is only used if array capability is implemented in a version of SCELBAL.

The source listings for these routines start here:

PRIGHT, LLI 230	Load L with address of F/A STACK pointer
LHI 026	** Load H with page of F/A STACK pointer
LAM	Fetch the pointer value into the ACC
ADL	Form pointer to top of the F/A STACK
LLA	Set L to point to top of the F/A STACK
LAM	Fetch the contents of the top of the F/A STACK into
LMI 000	The ACC then clear the top of the F/A STACK
LLI 203	Load L with address of F/A STACK TEMP storage
LHI 027	** Location. Set H to page of F/A STACK TEMP
LMA	Store value from top of F/A STACK into temp loc.
NDA	Test to see if token value in top of stack was zero
RTZ	If so, just had simple grouping parenthesis!
JTS PRIGH1	@@ If token value minus, indicates array subscript
CPI 001	For positive token value, look for appropriate function
JTZ INTX	If token value for INTeger function, go do it.
CPI 002	Else, see if token value for SiGN function.
JTZ SGNX	If so, go do it.
CPI 003	Else, see if token value for ABSolute function
JTZ ABSX	If so, go do it.
CPI 004	If not, see if token value for SQUare Root function
JTZ SQRX	If so, go do it.
CPI 005	If not, see if token value for TAB function
JTZ TABX	If so, go do it.
CPI 006	If not, see if token value for RaNDom function
JTZ RNDX	If so, go find a random number.
CPI 007	If not, see if token value for CHaRacter function
JTZ CHRX	If so, go perform the function.
CPI 010	Else, see if token for user defined machine language
JTZ UDEFX	†† Function. If so, perform the User DEfined Function
HLT	Safety halt. Program should not reach this location!

The following routine is only installed if the user desires to utilize single dimension array capability. This and associated array routines, if installed, will be in a separate area in memory apart from the standard SCELBAL routines. (Starts at top of page following the flow chart.)



PRIGH1,	LLI 126 LHI 001 LAM NDA JTS OUTRNG CAL FPFIX LLI 124 LAM SUI 001 RLC RLC LCA LLI 203 LHI 027 LAM XRI 377 RLC RLC ADI 120 LHI 027 LLA INL INL LAM ADC LLA LHI 057 JMP FLOAD	Load L with address of the MSW in the FPACC ** Set H to page of FPACC Fetch MSW of FPACC into the ACC. Test to see if value in FPACC is positive. If not, go display error message. If O.K. then convert floating point to fixed point Load L with address of LSW of converted value Fetch the LSW of the value into the ACC Subtract one from the value to establish proper Origin for future ops. Now rotate the value twice To effectively multiply by four. Save the Calculated result in CPU register C Load L with address of F/A STACK TEMP ** Load H with page of F/A STACK TEMP Fetch the value into the accumulator Complement the value Rotate the value twice to multiply by four (the number Of bytes per entry in the ARRAY VARIABLES table). Add the starting address of the ARRAY VARIABLES ** TABLE to form pointer. Set page address in H. Point to the name in the ARRAY VARIABLES Increment the pointer value twice to move over the Name in the table and point to starting address for the Array values in the ARRAY VALUES table. Fetch this Address to the ACC. Now add in the figure calculated To reach desired subscripted data storage location. Set †† The pointer to that location. Load the floating point Value stored there into the FPACC and exit to caller.
OUTRNG,	LAI 317 LCI 322 JMP ERROR	Load the ASCII code for letter O into the accumulator Load the ASCII code for letter R into register C Go display Out of Range (OR) error message.

The reader has just observed how the PRIGHT subroutine is used to direct the program to a specific routine if a right parenthesis indicates that a FUNCTION is to be executed.

The capabilities of the various FUNCTION routines were described briefly in an early chapter. Their use will be described in more detail in a later use. The actual implementation of these FUNCTION subroutines are quite straightforward for the most part and their operation can be easily followed by studying the commented source listings that follow.

There is one special FUNCTION to which a name has been assigned in the FUNCTION LOOK-UP TABLE but which will not be presented. The name given this function (which the user may readily alter) is UDF for User Defined Function. The reason the routine is not presented is because the routine is precisely what it has been named. The user is free to create whatever type of machine language subroutine the user might desire to have available in the higher level language. (How about special I/O handling capability or a frequently used mathematical function?) This user created routine may be installed wherever there is available memory in the user's system. (Small routines may be

placed at the end of page 31 in the assembled version provided in this manual.) The user should make sure the address to the start of the user defined subroutine is substituted for the dummy address provided for the jump instruction to the label UDEFX shown in the

listing. The user defined function routine should conclude with a RET instruction. Typical techniques that might be used in such a user created routine might be gleaned from studying the listings for the function routines that are provided as presented below.

INTX,	LLI 126 LHI 001 LAM NDA JFS INT1 LLI 014 CAL FSTORE CAL FPFIX LLI 123 LMI 000 CAL FPFLT LLI 014 CAL OPLOAD CAL FPSUB LLI 126 LAM NDA JTZ INT2 LLI 014 CAL FLOAD LLI 024 CAL FACXOP CAL FPADD	Load L with address of MSW of the FPACC ** Load H with the page of the FPACC Fetch the MSW of the FPACC into the accumulator Test the sign of the number in the FPACC. If Positive jump ahead to integerize If negative, load L with address of FP TEMP registers Store the value in the FPACC in FP TEMP Convert the value in FPACC from floating point to Fixed point. Load L with address of FPACC Extension register and clear it. Convert fixed binary back to FP to integerize Load L with address of FP TEMP registers Load the value in FP TEMP into FPOP Subtract integerized value from original Set L to address of MSW of FPACC Fetch the MSW of the FPACC into the accumulator See if original value and integerized value the same If so, have integer value in FP TEMP Else, load L with address of FP TEMP registers Restore FPACC to original (non-integerized) value Set L to register containing small value Set up to add small value to original value in FPACC Perform the addition
INT1,	CAL FPFIX LLI 123 LMI 000 JMP FPFLT	Convert the number in FPACC from floating point To fixed point. Load L with address of FPACC Extension register and clear it. Now convert the number Back to floating point to integerize it and exit to caller.
INT2,	LLI 014 JMP FLOAD	Load L with address of FP TEMP registers. Transfer Number from FP TEMP (orig) to FPACC and return.
ABSX,	LLI 126 LHI 001 LAM NDA JTS FPCOMP RET	Load L with address of MSW of the FPACC ** Set H to page of the FPACC Fetch the MSW of the FPACC into the accumulator Test the sign of the number to see if it is positive. If negative, complement the number before returning. Else, just return with absolute value in the FPACC.

SGNX,	LLI 126 LHI 001 LAM NDA RTZ JFS FPONE LLI 024 JMP FLOAD	Load L with address of MSW of the FPACC ** Load H with the page of the FPACC Fetch the MSW of the FPACC into the accumulator Test to see if the FPACC is zero Return to caller if FPACC is zero If FPACC is positive, load +1.0 into FPACC and exit If FPACC is negative, set up to load -1.0 into the FPACC and exit to caller
CHRX,	CAL FPFIX LLI 124 LAM CAL ECHO LLI 177 LHI 026 LMI 377 RET	Convert contents of FPACC from floating point to Fixed point. Load L with address of LSW of fixed Value. Fetch this byte into the accumulator. Display the value. Set L to address of the TAB FLAG ** Set H to page of the TAB FLAG Set TAB FLAG (to inhibit display of FP value) Exit to caller.
TABX, TAB1,	CAL FPFIX LLI 124 LAM LLI 043 SUM LLI 177 LHI 026 LMI 377 JTS BACKSP RTZ	Convert contents of FPACC from floating point to Fixed point. Load L with address of LSW of fixed Value. Fetch this byte into the accumulator. Load L with address of COLUMN COUNTER Subtract value in COLUMN COUNTER from desired TAB position. Load L with address of the TAB FLAG. ** Set H to page of the TAB FLAG. Set TAB FLAG (to inhibit display of FP value) If beyond TAB point desired, simulate back spacing Return to caller if at desired TAB location
TABC,	LCA LAI 240	Else, put difference count in register C Place ASCII code for space in ACC
TABLOP,	CAL ECHO DCC JFZ TABLOP RET	Display space on output device Decrement displacement counter If have not reached TAB position, continue to space Else, return to calling routine.
BACKSP,	LAI 215 CAL ECHO CAL ECHO LLI 043 LHI 001 LMI 001 LLI 124 LAM NDA RTS RTZ JMP TAB1	Load ASCII code for carriage-return into the ACC Display the carriage-return Repeat to provide extra time if TTY Load L with address of COLUMN COUNTER ** Set H to page of COLUMN COUNTER Set COLUMN COUNTER to first column Set L to address containing desired TAB position Fetch the desired TAB position value Test to see if it is Negative or zero In which case return to caller Else, proceed to perform the TAB operation.

SQRX,	LLI 014 LHI 001 CAL FSTORE LLI 126 LAM NDA JTS SQRERR JTZ CFALSE LLI 017 LAM NDA JTS NEGEXP RAR LBA LAI 000 RAL LMA JMP SQREXP	Load L with address of FP TEMP registers ** Set H to page of FP TEMP. Move contents of FPACC [Argument of SQR(X)] into FP TEMP for storage. Load L with MSW of FPACC Fetch the MSW into the accumulator Check the sign of the number in the FPACC If number negative, cannot take square root If number is zero, return with zero value in FPACC Load L with address of FP TEMP Exponent register Fetch the Exponent value into the ACC Check sign of the Exponent If Exponent less than zero, process negative Exponent If Exponent positive, rotate right to divide by two And save the result in CPU register B Clear the accumulator without disturbing Carry bit Rotate Carry bit into the ACC to save remainder Store the remainder back in FP TEMP Exponent reg. Jump to continue processing
NEGEXP,	LBA XRA SUB NDA RAR LBA LAI 000 ACA LMA JTZ NOREMD INB	For negative Exponent, form two's complement by Placing the positive value in CPU register B, clearing The accumulator, and then subtracting B from the ACC Clear the Carry bit after the complementing operation Rotate the value right to divide by two Save the result in CPU register B Clear the accumulator without disturbing Carry bit Add Carry bit to the accumulator as remainder Store the remainder back in FP TEMP Exponent reg If remainder was zero skip ahead. If not, increment the Result of the divide by two ops to compen for negative
NOREMD,	XRA SUB LBA	Clear the accumulator Subtract the quotient of the divide by two op to Form two's complement and save the result in register B
SQREXP,	LLI 013 LMB LLI 004 LEI 034 LDH LBI 004 CAL MOVEIT CAL CFALSE LLI 044 CAL FSTORE	Load L with address of TEMP register Store Exponent quotient from above ops in TEMP Load L with address of FP registers containing +1.0 Load E with address of SQR APPROX working registers Set D to same page as H Set up register B as a number of bytes to move counter Transfer value +1.0 into SQR APPROX registers Now clear the FPACC registers Load L with address of LAST SQR APPROX temp regs. Initialize the LAST SQR APPROX regs to value of zero
SQRLOP,	LLI 034 CAL FLOAD LLI 014 CAL OPLOAD CAL FPDIV	Load L with address of SQR APPROX working registers Transfer SQR APPROX into the FPACC Load L with address of SQR ARG storage registers Transfer SQR ARG into the FPOP Divde SQR ARG by SQR APPROX (Form X/A)

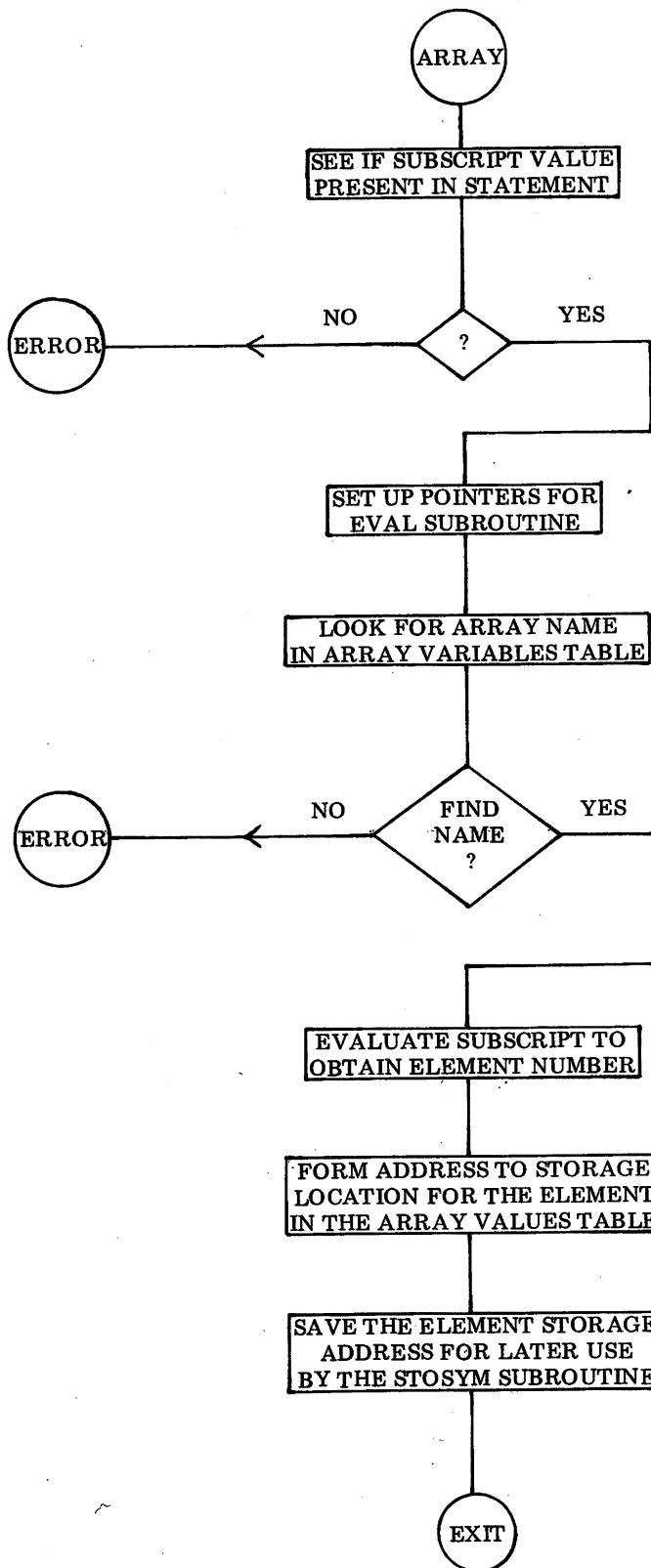
	LLI 034 CAL OPLOAD CAL FPADD LLI 127 LBM DCB LMB LLI 034 CAL FSTORE LLI 044 CAL OPLOAD CAL FPSUB LLI 127 LAM CPI 367 JTS SQRCNV LLI 034 LDH LEI 044 LBI 004 CAL MOVEIT JMP SQRLOP	Load L with address of SQR APPROX registers Transfer SQR APPROX into the FPOP Add to form value (X/A + A) Load L with address of FPACC Exponent register Fetch Exponent value into CPU register B Subtract one to effectively divide FPACC by two Restore to memory. (Now have ((X/A + A)/2) Load L with address of SQR APPROX registers Store contents of FPACC as new SQR APPROX Load L with address of LAST SQR APPROX registers Transfer LAST SQR APPROX into the FPOP Subtract (LAST SQR APPROX - SQR APPROX) Load L with address of FPACC Exponent Fetch the Exponent into the accumulator See if difference less than 2 to the minus ninth If so, approximation has converged Else, load L with address of SQR APPROX Set D to same page as H And E with address of LAST SQR APPROX Set up register B as a number of bytes to move counter Transfer SQR APPROX into LAST SQR APPROX Continue ops until approximation converges
SQRCNV,	LLI 013 LAM LLI 037 ADM LMA LLI 034 JMP FLOAD	Load L with address of TEMP register. Fetch the Exponent quotient store there into accumulator. Change L to point to SQR APPROX exponent. Add SQR APPROX exponent to quotient value. Store sum back in SQR APPROX Exponent register. Load L with address of SQR APPROX. Transfer the SQR APPROX into FPACC as answer and exit.
SQRERR,	LAI 323 LCI 321 JMP ERROR	Load ASCII code for letter S into the accumulator. Load ASCII code for letter Q into CPU register C. Display the SQquare root (SQ) error message.
RNDX,	LLI 064 LHI 001 CAL FLOAD LLI 050 CAL OPLOAD CAL FPMULT LLI 060 CAL OPLOAD CAL FPADD LLI 064 CAL FSTORE LLI 127 LAM SUI 020 LMA	Load L with address of SEED storage registers ** Set H to page for floating point working registers Transfer SEED into the FPACC Load L with address of random constant A Transfer random constant A into the FPOP Multiply to form (SEED * A) Load L with address of random constant C Transfer random constant C into the FPOP Add to form (SEED * A) + C Load L with address of SEED storage registers Store [(SEED * A) + C] in former SEED registers Load L with address of FPACC Exponent register Fetch Exponent value into the accumulator Subtract 16 (decimal) to effectively divide by 65,536 Now FPACC = [(SEED * A) + C]/65,536]

CAL FPFIX	Convert floating to fixed point to obtain integer part
LLI 123	Load L with address of FPACC Extension register
LMI 000	Clear the FPACC Extension register
LLI 127	Load L with address of FPACC Exponent
LMI 000	Clear the FPACC Exponent register
CAL FPFLT	Fetch INT(((SEED * A) + C)/65,536) into the FPACC
LLI 127	Load L with address of FPACC Exponent
LAM	Fetch FPACC Exponent into the accumulator
ADI 020	Add 16 (decimal) to effectively multiply by 65,536
LMA	(65,536 * INT[((SEED * A) + C)/65,536]) in FPACC
LLI 064	Load L with address of [(SEED * A) + C]
CAL OPLOAD	Transfer it into FPOP. Subtract FPACC to form
CAL FPSUB	[(SEED * A) + C] MOD 65,536
LLI 064	Load L with address of former SEED registers
CAL FSTORE	Store SEED MOD 65,536 in place of [(SEED * A) + C]
LLI 127	Load L with address of FPACC Exponent
LAM	Fetch FPACC Exponent into the ACC and subtract
SUI 020	16 (decimal) to form (SEED MOD 65,536)/65,536
LMA	So that random number in FPACC is between
RET	0.0 and +1.0 and exit to calling routine

The final routine to be discussed in this chapter is labeled ARRAY. It is part of the optional group of routines that are included if SCELBAL is to be implemented with single dimension array handling capability. This routine is actually a special extension of the LET statement routine. It is used to locate the address in the ARRAY VALUES TABLE at which a value assigned to an element of an array is to be stored.

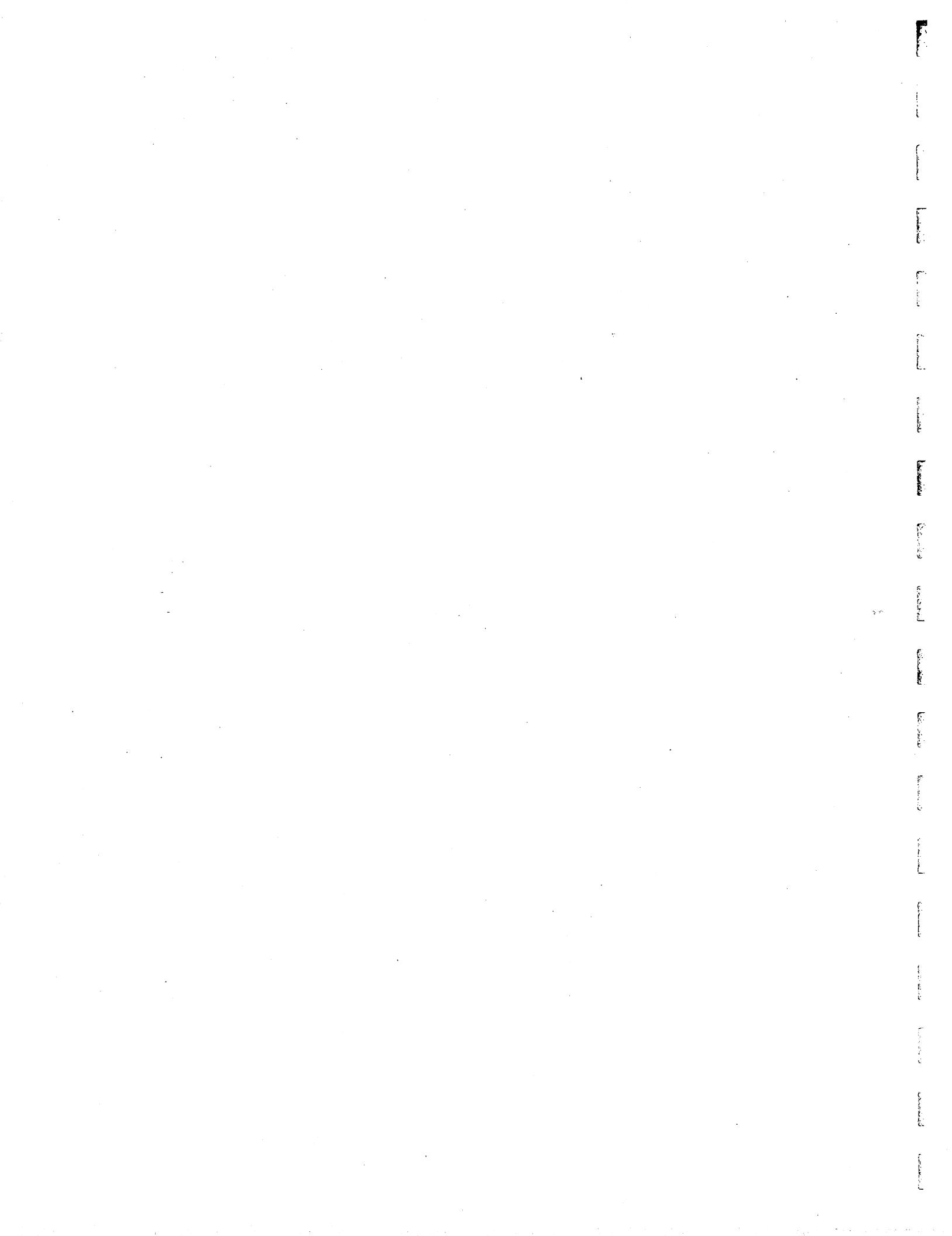
The key portions of the ARRAY routine are illustrated in the flow chart on the following page. The reader may wish to refer to the description of the optional DIMension statement routine in an earlier chapter. A discussion of the organization of the ARRAY VARIABLES and ARRAY VALUES tables is presented there which will be helpful in following the operation of the following routine.

ARRAY,	CAL RESTSY	Transfer contents of AUX SYMBOL BUFFER into the
	JMP ARRAY2	SYMBOL BUFFER. (Entry when have actual LET)
ARRAY1,	LLI 202	Load L with address of SCAN pointer
	JMP ARRAY3	Proceed to process. (Entry point for IMPLIED LET)
ARRAY2,	LLI 203	Load L with address of LET pointer
ARRAY3,	LHI 026	** Set H to pointer page
	LBM	Fetch pointer to location where "(" found in statement
	INB	Line. Increment it to point to next character in the line.
	LLI 276	Load L with address of EVAL pointer and load it with
	LMB	The starting address for the EVAL routine
	LLI 206	Change L to address of ARRAY SETUP pointer
	LMB	And also store address in that location



ARRAY4,	LLI 206 CAL GETCHR CPI 251 JTZ ARRAY5 LLI 206 CAL LOOP JFZ ARRAY4 LAI 301 LCI 306 JMP ERROR	Load L with address of ARRAY SETUP pointer Fetch character pointed to by ARRAY SETUP pntr See if character is ")" ? If so, then have located End of the subscript. If not, reset L to the ARRAY SETUP pointer. Increment the Pointer and test for the end of the statement line. If not end of line, continue looking for right paren. If reach end of line before right parenthesis than load ASCII code for letters A and F and display message Indicating Array Format (AF) error condition
ARRAY5,	LLI 206 LBM DCB LLI 277 LMB LLI 207 LMI 000	Load L with address of ARRAY SETUP pointer Fetch pointer (pointing to ")" sign) into register B Decrement it to move back to end of subscript number Load L with address of EVAL FINISH pointer location Place the pointer value in the EVAL FINISH pointer Load L with address of LOOP COUNTER Initialize LOOP COUNTER to value of zero
ARRAY6,	LLI 207 LHI 026 LBM INB LMB LCI 002 LLI 114 LHI 027 CAL TABADR LEI 120 LDI 026 CAL STRCP JTZ ARRAY7 LLI 207 LHI 026 LAM LLI 075 LHI 027 CPM JFZ ARRAY6 JMP FAERR	Load L with address of LOOP COUNTER ** Load H with page of LOOP COUNTER Fetch the counter value Increment it Restore the counter value to memory Set up counter in register C for future ops Load L with address of start of ARRAY VARIABLES ** Table (less four). Set H to page of the table. Calculate the address of next entry in the table Load register E with starting address of SYMBOL BUFF ** Set D to page of SYMBOL BUFFER Compare entry in table against contents of SYMBOL BF If match, have found array name in the table. Else, set L to address of the LOOP COUNTER ** Set H to page of the LOOP COUNTER Fetch the counter value to the ACC Change L to the counter containing number of arrays ** Set H to the proper page Compare number of arrays to count in LOOP CNTR If more entries in the table, continue looking for match If no matching name in table then have an error condx.
ARRAY7,	CAL EVAL CAL FPFX LLI 207 LHI 026 LBM LCI 002 LLI 114 LHI 027 CAL TABADR INL	Call subroutine to evaluate subscript expression Convert the subscript value obtained to fixed format Load L with address of LOOP COUNTER ** Set H to page of the LOOP COUNTER Fetch the value in the LOOP COUNTER into the ACC Set up counter in register C for future ops Load L with address of ARRAY VARIABLES ** Table (less four). Set H to page of the table. Calculate the address of entry in the table Advance the ARRAY VARIABLES table pointer twice

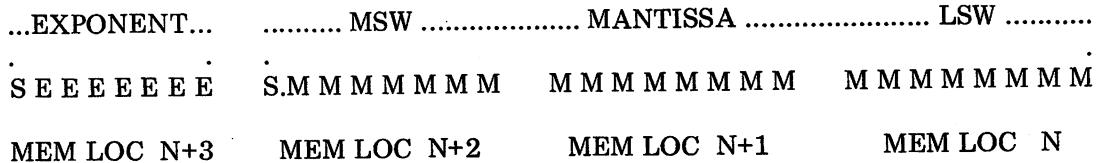
INL	To advance pointer over array name.
LCM	Fetch array base address in ARRAY VALUES table
LLI 124	Load L with address of subscript value
LHI 001	** Set H to page of subscript value
LAM	Fetch the subscript value into the accumulator
SUI 001	Subtract one from subscript value to allow for zero
RLC	Origin. Now multiply by four
RLC	Using rotates (number of bytes required for each entry
ADC	In the ARRAY VALUES table). Add in base address to
LLI 204	The calculated value to form final address in the
LHI 027	** ARRAY VALUES table. Now set H & L to TEMP
LMA	ARRAY ELEMENT storage location & store the addr.
LLI 201	Change L to point to ARRAY FLAG
LMI 377	Set the ARRAY FLAG for future use
RET	Exit to calling routine



MATHEMATICAL ROUTINES

Essentially all mathematical operations in SCELBAL are performed by a group of subroutines utilizing triple-precision binary floating point techniques. That is, the mantissa portion of a binary number is stored in three consecutive memory registers in order to provide 23 bits of magnitude plus a sign bit in which to represent the magnitude of the significant digits of a number. In order to allow for the raising of numbers to a power, a fourth byte is used to maintain the expo-

nent of a number. That is, the power to which the mantissa is to be raised. The exponent portion of a number may thus have a magnitude of 7 bits. The eighth bit available in a register is used to maintain the sign of the exponent. Thus, each number stored in floating point format in SCELBAL requires four consecutive bytes in memory for storage. One byte for the exponent and three bytes for the significant digits or mantissa. The format is illustrated in the following diagram.



Twenty-three binary bits can represent decimal numbers from 0.0 to 8,388,847. This is thus the largest value that the mantissa portion may represent in SCELBAL. (While the floating point routines can manipulate numbers up to this size, the input routine for SCELBAL limits the maximum decimal number that may be inputted to about half this value. As a general rule, the operator should restrict decimal inputs to six significant digits for the mantissa portion of a number.)

The seven bits available for the exponent portion of a number in the floating point routines allow a decimal number to be raised to approximately the 38th power of ten. (While the reader at first glance might think that seven bits would provide for an exponent range to 127 decimal, such is not the case. This is because raising a number by a power of ten decimal requires raising a binary number by between the third and fourth power when using the base 2 (remember, two to the third power is 8, which is less than 10).)

Thus, instead of 7 bits allowing for an exponent of up to 127 decimal, it can only represent about one third that amount.

The reader should note that if numbers being manipulated by SCELBAL should exceed the absolute magnitudes indicated above that the results of such calculations will be in error. This is because the binary exponent register would change sign on an overflow/underflow condition. This type of error is most likely to occur if a user should raise a large number to a relatively higher power, or multiply two large numbers such as 100E+22 times 50E+24. The range of powers (plus or minus 38 decimal) that SCELBAL can handle is quite adequate for most applications. Extending this range would require increasing the number of registers (precision) used to hold numbers and would significantly decrease the overall operating speed of the language. The triple-precision plus exponent format was chosen as a suitable compromise between other options.

Various portions of the floating point package to be described in this chapter are called upon by many of the routines described previously. Most of the mathematical operations are performed between two floating point multiple-byte registers named the FLOATING POINT ACCUMULATOR (abbreviated FPACC) and FLOATING POINT OPERAND (abbreviated FPOP).

The first section of the floating point section of SCELBAL consists of a group of subroutines that may be called upon separately to perform the following operations.

FLOATING POINT FIX (FPFIX). This subroutine will convert a number stored in floating point format back to binary fixed point format provided that the floating point number is in a range that can be converted to fixed point. (That is, will not require more than 23 decimal bits for storage.) Thus a number such as 5 decimal, which would appear in binary floating point format as:

0.101 E+11

would be converted to the fixed binary format:

101

The reader may note that converting floating point to fixed point is merely a matter of rotating the floating point value to the left until the binary exponent has a value of zero. Thus the above floating point number:

0.101 E+11

would be rotated to the left three places.

A floating point number such as:

0.101 E+10000

could not be properly positioned as a fixed point binary number in a triple-precision register (8 bits per register) format because it would have to be rotated to the left 32 decimal positions.

FLOATING POINT ZERO (FPZERO). subroutine simply sets the FPACC to a value of zero. It is used to initialize or clear out the floating point accumulator.

FLOATING POINT NORMALIZE (FPNORM). This is the reverse procedure of for the case when a binary fixed point value is being changed to floating point notation. The fixed point value is simply rotated to the right while the binary exponent value is incremented until all significant digits are to the right of an implied decimal point. Thus, the fixed point value:

101

would be converted to:

0.101 E+11

Normalization is also used after other floating point operations to standardize the mantissa to be in the range greater than or equal to ONE HALF (1/2) but less than ONE. Thus, if a number such as 0.1 decimal which would appear as:

0.00011001100... E+0

in binary was normalized it would be shifted to the left while the binary exponent was decremented until it appeared as:

0.11001100... E+11

This normalization or standardization process is valuable primarily because the process aids in maintaining the maximum number of significant digits throughout a series of complex operations.

FLOATING POINT ADDITION (FPADD). This subroutine simply adds the floating point binary number in the FPACC to the floating point binary number in the FPOP and leaves the result of the addition in the FPACC.

FLOATING POINT SUBTRACTION (FPSUB). This subroutine subtracts the value in the FPACC from the value in the FPOP and

leaves the result in the FPACC.

The source listings for the five floating

point operations just described (FPFIX, FPZERO, FPNORM, FPADD and FPSUB) are presented next.

Following subroutine converts number stored as floating point in FPACC to fixed point.

FPFIX,	LLI 126 LHI 001 LAM LLI 100 LMA NDA CTS FPCOMP LLI 127 LAI 027 LBM INB DCB JTS FPZERO SUB JTS FIXERR LCA	Set L to point to MSW of FPACC ** Set H to point to page of FPACC Fetch MSW of FPACC Change pointer to SIGN indicator on same page Place MSW of FPACC into SIGN indicator Now test sign bit of MSW of FPACC Two's complement value in FPACC if negative Change pointer to FPACC Exponent register Set accumulator to 23 (decimal) for number of bits Load FPACC Exponent into CPU register B Exercise the value in register B To set CPU flags If FPACC Exponent is negative set FPACC to zero Subtract value of FPACC Exponent from 23 decimal If Exp larger than 23 decimal cannot convert Else place result in register C as counter for number Of rotate ops. Set pointer to MSW of FPACC Set precision counter (number of bytes in mantissa) Rotate FPACC right the number of places indicated By count in register C to effectively rotate all the Significant bits to the left of the floating point decimal Point. Go check original sign & negate answer if req'd.
PPFIXL,	LLI 126 LBI 003 CAL ROTATTR DCC JFZ FPFIXL JMP RESIGN	

Following subroutine clears the FPACC to the zero condition.

FPZERO,	LLI 126 XRA LMA DCL LMA DCL LMA DCL RET	Set L to point to MSW of FPACC Clear the accumulator Set the MSW of FPACC to zero Decrement the pointer Set the next significant word of FPACC to zero Decrement the pointer Set the LSW of FPACC to zero Decrement the pointer Set the auxiliary FPACC byte to zero Exit to calling routine
---------	---	---

The next instruction is a special entry point to the FPNORM subroutine that is used when a number is converted from fixed to floating point. The FPNORM label is the entry point when a number already in floating point format is to be normalized.

FPFLT,	LBI 027	For fixed to float set CPU register B to 23 decimal
FPNORM,	LAB LHI 001 LLI 127 NDA JTZ NOEXCO LMB	Get CPU register B into ACC to check for special case ** Set H to page of FPACC Set L to FPACC Exponent byte Set CPU flags to test what was in CPU register B If B was zero then do standard normalization Else set Exponent of FPACC to 23 decimal
NOEXCO,	DCL LAM LLI 100 LMA NDA JFS ACZERT LBI 004 LLI 123 CAL COMPLM	Change pointer to MSW of FPACC Fetch MSW of FPACC into accumulator Change pointer to SIGN indicator storage location Place the MSW of FPACC there for future reference Set CPU flags to test MSW of FPACC If sign bit not set then jump ahead to do next test If sign bit set, number in FPACC is negative. Set up For two's complement operation And negate the value in the FPACC to make it positive
ACZERT,	LLI 126	Reset pointer to MSW of FPACC
	LBI 004	Set precision counter to number of bytes in FPACC
LOOK0,	LAM NDA JFZ ACNONZ DCL DCB JFZ LOOK0 LLI 127 XRA LMA RET	Plus one. Fetch a byte of the FPACC. Set CPU flags If find anything then FPACC is not zero Else decrement pointer to NSW of FPACC Decrement precision counter Continue checking to see if FPACC contains anything Until precision counter is zero. If reach here then Reset pointer to FPACC Exponent. Clear the ACC and Clear out the FPACC Exponent. Value of FPACC is zip! Exit to calling routine
ACNONZ,	LLI 123 LBI 004 CAL ROTATL LAM NDA JTS ACCSET INL LBM DCB LMB JMP ACNONZ	If FPACC has any value set pointer to LSW minus one Set precision counter to number of bytes in FPACC Plus one for special cases. Rotate the contents of the FPACC to the LEFT. Pointer will be set to MSW after Rotate ops. Fetch MSW and see if have anything in Most significant bit position. If so, have rotated enough If not, advance pointer to FPACC Exponent. Fetch The value of the Exponent and decrement it by one To compensate for the rotate left of the mantissa Restore the new value of the Exponent Continue rotating ops to normalize the FPACC
ACCSET,	LLI 126 LBI 003 CAL ROTATR	Set pntr to FPACC MSW. Now must provide room for Sign bit in normalized FPACC. Set precision counter. Rotate the FPACC once to the right now.
RESIGN,	LLI 100 LAM NDA RFS	Set the pointer to SIGN indicator storage location Fetch the original sign of the FPACC Set CPU flags If original sign of FPACC was positive, can exit now.

FPCOMP,	LLI 124 LBI 003 JMP COMPLM	However, if original sign was negative, must now restore The FPACC to negative by performing two's complement on FPACC. Return to calling rtn via COMPLM.
		Floating point ADDITION. Adds contents of FPACC to FPOP and leaves result in FPACC. Routine first checks to see if either register contains zero. If so addition result is already present!
FPADD,	LLI 126 LHI 001 LAM NDA JFZ NONZAC	Set L to point to MSW of FPACC ** Do same for register H Fetch MSW of FPACC to accumulator Set CPU flags after loading op If accumulator non-zero then FPACC has some value
MOVOP,	LLI 124 LDH LEL LLI 134 LBI 004 JMP MOVEIT	But, if accumulator was zero then normalized FPACC Must also be zero. Thus answer to addition is simply the Value in FPOP. Set up pointers to transfer contents of FPOP to FPACC by pointing to the LSW of both Registers and perform the transfer. Then exit to calling Routine with answer in FPACC via MOVEIT.
NONZAC,	LLI 136 LAM NDA RTZ	If FPACC was non-zero then check to see if FPOP has Some value by obtaining MSW of FPOP Set CPU flags after loading op. If MSW zero then Normalized FPOP must be zero. Answer is in FPACC!
		If neither FPACC or FPOP was zero then must perform addition operation. Must first check to see if two numbers are within significant range. If not, largest number is answer. If numbers within range, then must align exponents before performing the addition of the mantissa.
CKEQEX,	LLI 127 LAM LLI 137 CPM JTZ SHACOP LBA LAM SBB JFS SKPNEG LBA XRA SBB	Set pointer to FPACC Exponent storage location. Fetch the Exponent value to the accumulator. Change the pointer to the FPOP Exponent Compare the values of the exponents. If they are the Same then can immediately proceed to add operations. If not the same, store FPACC Exponent size in regis B Fetch the FPOP Exponent size into the ACC Subtract the FPACC Exponent from the FPOP Exp. If result is positive jump over the next few instructions If result was negative, store the result in B Clear the accumulator Subtract register B to negate the original value
SKPNEG,	CPI 030 JTS LINEUP LAM LLI 127	See if difference is less than 24 decimal. If so, can align exponents. Go do it. If not, find out which number is largest. Fetch FPOP Exponent into ACC. Change pointer to FPACC Exp.

	SUM RTS LLI 124 JMP MOVOP	Subtract FPACC from FPOP. If result is negative then FPACC was larger. Return with answer in FPACC. If result was positive, larger value in FPOP. Set pointers To transfer FPOP into FPACC and then exit to caller.
LINEUP,	LAM LLI 127 SUM JTS SHIFTO LCA LLI 127 CAL SHLOOP DCC JFZ MORACC JMP SHACOP	Fetch FPOP Exponent into accumulator. Change pointer to FPACC Exponent. Subtract FPACC Exponent from FPOP Exponent. If Result is negative FPACC is larger. Go shift FPOP. If result positive FPOP larger, must shift FPACC. Store Difference count in C. Reset pointer to FPACC Exp Call the Shift LOOP to rotate FPACC mantissa RIGHT And INCREMENT Exponent. Decr difference counter Continue rotate operations until diff counter is zero Go do final alignment and perform addition process
SHIFTO,	LCA	Routine to shift FPOP. Set difference count into reg. C
MOROP,	LLI 137 CAL SHLOOP INC JFZ MOROP	Set pointer to FPOP Exponent. Call the Shift LOOP to rotate FPOP mantissa RIGHT And INCREMENT Exponent. Then incr difference cntr Continue rotate operations until diff counter is zero
SHACOP,	LLI 123 LMI 000 LLI 127 CAL SHLOOP LLI 137 CAL SHLOOP LDH LEI 123 LBI 004 CAL ADDER LBI 000 JMP FPNORM	Set pointer to FPACC LSW minus one to provide extra Byte for addition ops. Clear that location to zero. Change pointer to FPACC Exponent Rotate FPACC mantissa RIGHT & Increment Exponent Change pointer to FPOP Exponent Rotate FPOP mantissa RIGHT & Increment Exponent Rotate ops provide room for overflow. Now set up Pointers to LSW minus one for both FPACC & FPOP (FPOP already set after SHLOOP). Set precision counter Call quad precision ADDITION subroutine. Set CPU register B to indicate standard normalization Go normalize the result and exit to caller.
SHLOOP,	LBM INB LMB DCL LBI 004	Shifting loop. First fetch Exponent currently being Pointed to and Increment the value by one. Return the updated Exponent value to memory. Decrement the pointer to mantissa portion MSW Set precision counter
FSHIFT,	LAM NDA JFS ROTATR	Fetch MSW of mantissa Set CPU flags after load ops If MSB not a one can do normal rotate ops
BRING1,	RAL JMP ROTR	If MSB is a one need to set up carry bit for the negative Number case. Then make special entry to ROTATR sub

The following subroutine moves the contents of a string of memory locations from the address pointed to by CPU registers H & L to the address specified by the contents of registers D & E when the routine is entered. The process continues until the counter in register B is zero.

MOVEIT,	LAM	Fetch a word from memory string A
	INL	Advance A string pointer
	CAL SWITCH	Switch pointer to string B
	LMA	Put word from string A into string B
	INL	Advance B string pointer
	CAL SWITCH	Switch pointer back to string A
	DCB	Decrement loop counter
	RTZ	Return to calling routine when counter reaches zero
	JMP MOVEIT	Else continue transfer operations

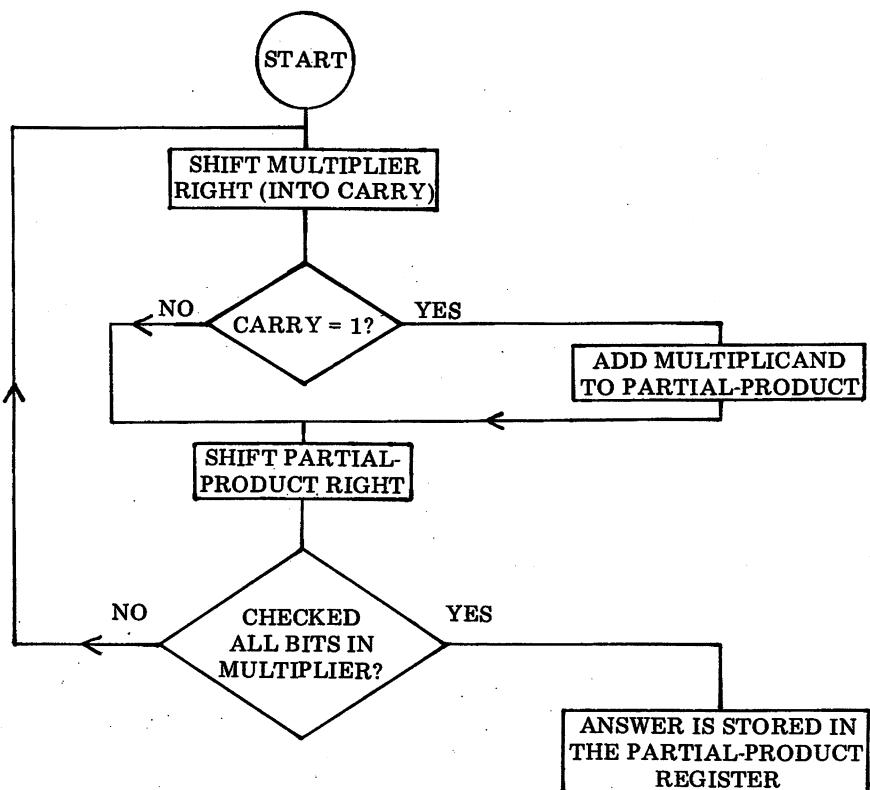
The following subroutine SUBTRACTS the contents of the FLOATING POINT ACCUMULATOR from the contents of the FLOATING POINT OPERAND and leaves the result in the FPACC. The routine merely negates the value in the FPACC and then goes to the FPADD subroutine just presented.

FSUB,	LLI 124	Set L to address of LSW of FPACC
	LHI 001	** Set H to page of FPACC
	LBI 003	Set precision counter
	CAL COMPLM	Two's complement the value in the FPACC
	JMP FPADD	Now go add the negated value to perform subtraction!

FLOATING POINT MULTIPLICATION

The next section of the floating point package is a routine that performs floating point multiplication. A conventional floating point multiplication algorithm is utilized to perform this function. The essence of the algorithm is illustrated in the flow chart shown on the next page. Prior to implementing this algorithm the routine performs several initializing procedures. It checks the signs of the multiplier and multiplicand and negates the values if they are negative. If the signs of the two numbers to be multiplied are

different, the final answer will be negated. The exponents of the two numbers are then added. Finally the two mantissas are multiplied using a double width (six byte) partial-product register. The final answer in this register is then rounded off to the 23 most significant binary bits as the final answer. This answer is left in the FPACC at the conclusion of the routine (after being negated if the signs of the original numbers were different). The listing for the floating point multiplication subroutine is presented next.



The first part of the FLOATING POINT MULTIPLICATION subroutine calls a subroutine to check the original signs of the numbers that are to be multiplied and perform working register clearing functions. Next the exponents of the numbers to be multiplied are added together.

FPMULT,	CAL CKSIGN	Call routine to set up registers & ck signs of numbers
ADDEXP,	LLI 137	Set pointer to FPOP Exponent
	LAM	Fetch FPOP Exponent into the accumulator
	LLI 127	Change pointer to FPACC Exponent
	ADM	Add FPACC Exponent to FPOP Exponent
	ADI 001	Add one more to total for algorithm compensation
	LMA	Store result in FPACC Exponent location

SETMCT,	LLI 102	Change pointer to bit counter storage location
	LMI 027	Initialize bit counter to 23 decimal

Next portion of the FPMULT routine is the implementation of the algorithm illustrated in the flow chart above. This portion multiplies the values of the two mantissas. The final value is rounded off to leave the 23 most significant bits as the answer that is stored back in the FPACC.

MULTIP,	LLI 126 LBI 003 CAL ROTATR CTC ADOPPP LLI 146 LBI 006 CAL ROTATR LLI 102 LCM DCC LMC JFZ MULTIP LLI 146 LBI 006 CAL ROTATR LLI 143 LAM RAL NDA CTS MROUND LLI 123 LEL LDH LLI 143 LBI 004	Set pointer to MSW of FPACC mantissa Set precision counter Rotate FPACC (multiplier) RIGHT into carry bit If carry is a one, add multiplicand to partial-product Set pointer to partial-product most significant byte Set precision counter (p-p register is double length) Shift partial-product RIGHT Set pointer to bit counter storage location Fetch current value of bit counter Decrement the value of the bit counter Restore the updated bit counter to its storage location If have not multiplied for 23 (decimal) bits, keep going If have done 23 (decimal) bits, set pntr to p-p MSW Set precision counter (for double length) Shift partial-product once more to the RIGHT Set pointer to access 24'th bit in partial-product Fetch the byte containing the 24'th bit Position the 24'th bit to be MSB in the accumulator Set the CPU flags after to rotate operation and test to See if 24'th bit of p-p is a ONE. If so, must round-off Now set up pointers To perform transfer Of the multiplication results From the partial-product location To the FPACC
EXMLDV,	CAL MOVEIT LBI 000 CAL FPNORM LLI 101 LAM NDA RFZ JMP FPCOMP	Perform the transfer from p-p to FPACC Set up CPU register B to indicate regular normalization Normalize the result of multiplication Now set the pointer to the original SIGNS indicator Fetch the indicator Exercise the CPU flags If indicator is non-zero, answer is positive, can exit here. If not, answer must be negated, exit via 2's complement.
CKSIGN,	LLI 140 LHI 001 LBI 010 XRA	The following portions of the FPMULT routine set up working locations in memory by clearing locations for an expanded FPOP area and the partial-product storage area. Next, the signs of the two numbers to be multiplied are examined. Negative numbers are negated in preparation for the multiplication algorithm. A SIGNS Indicator register is set up during this process to indicate whether the final result of the multiplication should be positive or negative. (Negative if original signs of the two numbers to be multiplied are different.)

CLRNX,	LMA INL DCB JFZ CLRNX	Now clear out locations for the partial-product Working registers Until the loop counter Is zero
CLROPL,	LBI 004 LLI 130	Set a loop counter Set up pointer
CLRNX1,	LMA INL DCB JFZ CLRNX1 LLI 101 LMI 001 LLI 126 LAM NDA JTS NEGFPA	Clear out some extra registers so that the FPOP may be extended in length Perform clearing ops until loop counter Is zero Set pointer to M/D SIGNS indicator storage location Set initial value of SIGNS indicator to plus one Change pointer to MSW of FPACC Fetch MSW of mantissa into accumulator Test flags If MSB in MSW of FPACC is a one, number is negative
OPSGNT,	LLI 136 LAM NDA RFS LLI 101 LCM DCC LMC LLI 134 LBI 003 JMP COMPLM	Set pointer to MSW of FPOP Fetch MSW of mantissa into accumulator Test flags Return to caller if number in FPOP is positive Else change pointer to M/D SIGNS indicator Fetch the value in the SIGNS indicator Decrement the value by one Restore the new value back to storage location Set pointer to LSW of FPOP Set precision counter Two's complement value of FPOP & return to caller
NEGFPA,	LLI 101 LCM DCC LMC LLI 124 LBI 003 CAL COMPLM JMP OPSGNT	Set pointer to M/D SIGNS indicator Fetch the value in the SIGNS indicator Decrement the value by one Restore the new value back to storage location Set pointer to LSW of FPACC Set precision counter Two's complement value of FPACC Proceed to check sign of FPOP
		The following subroutine adds the double length (six register) multiplicand in FPOP to the partial-product register when called on by the multiplication algorithm.
ADOPPP,	LEI 141 LDH LLI 131 LBI 006 JMP ADDER	Pointer to LSW of partial-product On same page as FPOP LSW of FPOP which contains extended multiplicand Set precision counter (double length working registers) Add multiplicand to partial-product & return to caller

MROUND,	LBI 003	Set up precision counter
	LAI 100	Prepare to add one to 24'th bit of partial-product
	ADM	Add one to the 24'th bit of the partial-product
CROUND,	LMA	Restore the updated byte to memory
	INL	Advance the memory pointer to next most significant
	LAI 000	Byte of partial-product, then clear ACC without
	ACM	Disturbing carry bit. Now perform add with carry to
	DCB	Propagate any rounding in the partial-product registers.
	JFZ CROUND	If counter is not zero continue propagating any carry
	LMA	Restore final byte to memory
	RET	Exit to calling routine

FLOATING POINT DIVISION

The next part of the floating point group of routines is that which performs floating point division. A flow chart on the next page illustrates the conventional algorithm that is the main portion of this routine.

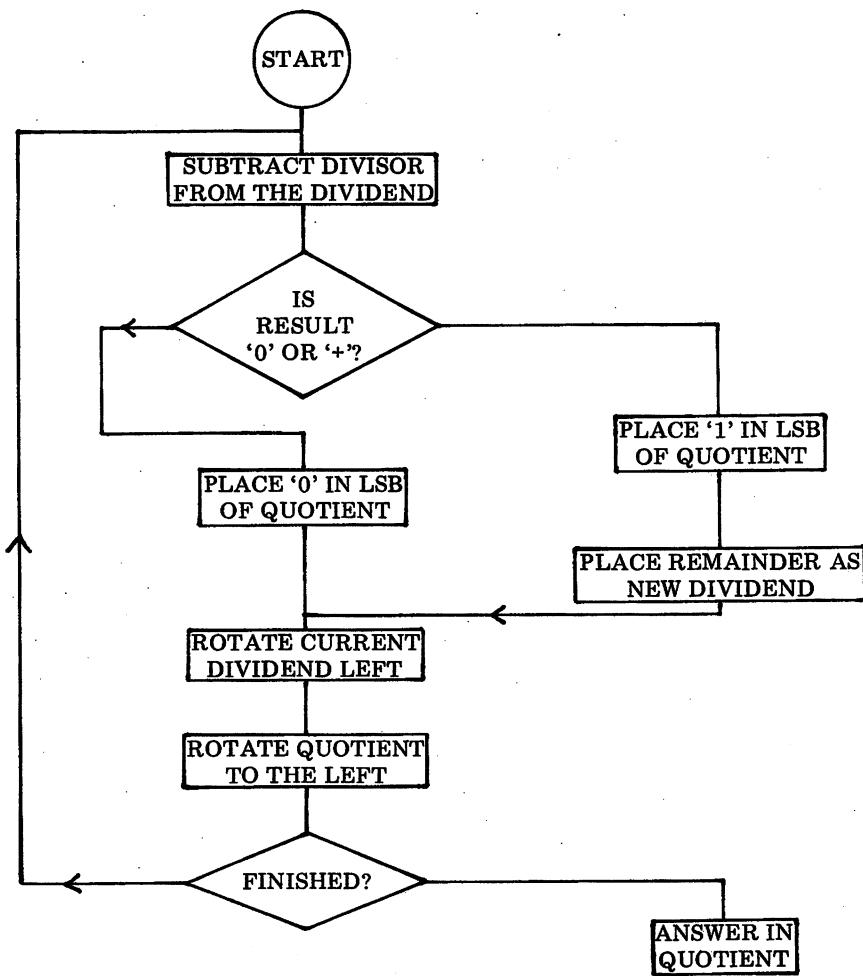
The division subroutine begins in the same manner used for floating point multiplication. Working registers are initialized and the signs of the two numbers (dividend and divisor) are tested. Negative numbers are negated before performing the division. The final answer is negated if the signs of the original numbers are different. Prior to attempting division, a check is made to see

if the divisor is zero. If so, an error message is displayed to the operator. If not, division is accomplished by first subtracting the exponent of the divisor from that of the dividend. The mantissas are then multiplied using the algorithm illustrated in the flow chart.

At the conclusion of the division process, a check is made to see if rounding-off is required. If so, this function is performed. The final answer is left in the FPACC at the conclusion of the routine (after being negated if the signs of the original numbers were different). The listing for the floating point division subroutine is presented next.

The first part of the FLOWING POINT DIVISION subroutine calls a subroutine to check the original signs of the numbers and perform initialization procedures. Next a test is made to see if the divisor is zero. An error message is displayed in such a case. Next the exponent of the divisor is subtracted from the dividend exponent.

FPDIV,	CAL CKSIGN	Call routine to set up registers & ck signs of numbers
	LLI 126	Set pointer to MSW of FPACC (divisor)
	LAM	Fetch MSW of FPACC to accumulator
	NDA	Exercise CPU flags
	JTZ DVERR	If MSW of FPACC is zero go display 'DZ' error message
SUBEXP,	LLI 137	Set pointer to FPOP (dividend) Exponent
	LAM	Get FPOP Exponent into accumulator
	LLI 127	Change pointer to FPACC (divisor) Exponent



SUM	Subtract divisor exponent from dividend exponent	
ADI 001	Add one for algorithm compensation	
LMA	Place result in FPACC Exponent	
SETDCT,	LLI 102	Set pointer to bit counter storage location
	LMI 027	Initialize bit counter to 23 decimal
Main division algorithm for mantissas		
DIVIDE,	CAL SETSUB	Go subtract divisor from dividend
	JTS NOGO	If result is negative then place a zero bit in quotient
	LEI 134	If result zero or positive then move remainder after
	LLI 131	Subtraction from working area to become new dividend
	LBI 003	Set up moving pointers and initialize precision counter
	CAL MOVEIT	Perform the transfer
	LAI 001	Place a one into least significant bit of accumulator
	RAR	And rotate it out into the carry bit

	JMP QUOROT	Proceed to rotate the carry bit into the current quotient
NOGO,	XRA	When result is negative, put a zero in the carry bit, then:
QUOROT,	LLI 144	Set up pointer to LSW of quotient register
	LBI 003	Set precision counter
	CAL ROTL	Rotate carry bit into quotient by using special entry to ROTATL subroutine. Now set up pointer to dividend LSW and set precision counter
	LLI 134	Rotate the current dividend to the left
	LBI 003	Set pointer to bit counter storage location
	CAL ROTATL	Fetch the value of the bit counter
	LLI 102	Decrement the value by one
	LCM	Restore the new counter value to storage
	DCC	If bit counter is not zero, continue division process
	LMC	After 23 (decimal) bits, do subtraction once more for Possible rounding. Jump ahead if no rounding required.
	JFZ DIVIDE	If rounding required set pointer to LSW of quotient
	CAL SETSUB	Fetch LSW of quotient to accumulator
	JTS DVEXIT	Add one to 23'rd bit of quotient
	LLI 144	Restore updated LSW of quotient
	LAM	Clear accumulator without disturbing carry bit
	ADI 001	Advance pointer to next significant byte of quotient
	LMA	Propagate any carry as part of rounding process
	LAI 000	Restore the updated byte of quotient
	INL	Clear ACC again without disturbing carry bit
	ACM	Advance pointer to MSW of quotient
	LMA	Propagate any carry to finish rounding process
	LAI 000	Restore the updated byte of quotient
	INL	If most significant bit of quotient is zero, go finish up
	ACM	If not, set precision counter
	LMA	And rotate quotient to the right to clear the sign bit
	JFS DVEXIT	Set pointer to FPACC Exponent
	LBI 003	Fetch FPACC exponent
	CAL ROTATR	Increment the value to compensate for the rotate right
	LLI 127	Restore the updated exponent value
	LBM	
	INB	
	LMB	
DVEXIT,	LLI 1443	Set up pointers
	LEI 1243	To transfer the quotient into the FPACC
	LBI 0034	Set precision counter
	JMP EXMLDV	And exit through FPMULT routine at EXMLDV
		Subroutine to subtract divisor from dividend. Used by main DIVIDE subroutine.
SETSUB,	LEI 131	Set pointer to LSW of working area
	LDH	On same page as FPACC
	LLI 124	Set pointer to LSW of FPACC (divisor)
	LBI 003	Set precision counter
	CAL MOVEIT	Perform transfer

LEI 131	Reset pointer to LSW of working area (now divisor)
LLI 134	Reset pointer to LSW of FPOP (dividend)
LBI 003	Set precision counter
CAL SUBBER	Subtract divisor from dividend
LAM	Get MSW of the result of the subtraction operations
NDA	Exercise CPU flags
RET	Return to caller with status

FLOATING POINT UTILITY SUBROUTINES

The following section presents a group of so-called "utility" subroutines. These subroutines perform a variety of minor functions required by the floating point package. Many of these subroutines are also used by other

portions of SCELBAL. The specific purpose of each routine will be explained in the comments portion of the source listing which is presented below.

N'th precision addition subroutine. Length of multi-byte numbers specified by contents of CPU register B upon entry. Number starting at location pointed to by H & L (least significant byte) is added to number starting at address specified by contents of D & E.

ADDER, ADDMOR,	NDA LAM CAL SWITCH ACM LMA DCB RTZ INL CAL SWITCH INL JMP ADDMOR	Initialize the carry bit to zero upon entry Fetch byte from register group A Switch memory pointer to register group B Add byte from A to byte from B with carry Leave result in register group B Decrement number of bytes (precision) counter Return to caller when all bytes in group processed Else advance pointer for register group B Switch memory pointer back to register group A Advance the pointer for register group A Continue the multi-byte addition operation
----------------	--	---

N'th precision two's complement (negate) subroutine. Performs a two's complement on the multi-byte register starting at the address pointed to by H & L (least significant byte) upon entry.

COMPLM, MORCOM,	LAM XRI 377 ADI 001 LMA RAR LDA DCB	Fetch the least significant byte of the number to ACC Exclusive OR to complement the byte Add one to form two's complement of byte Restore the negated byte to memory Save the carry bit In CPU register D Decrement number of bytes (precision) counter
-----------------	---	--

	RTZ	Return to caller when all bytes in number processed
	INL	Else advance the pointer
	LAM	Fetch the next byte of the number to ACC
	XRI 377	Exclusive OR to complement the byte
	LEA	Save complemented value in register E temporarily
	LAD	Restore previous carry status to ACC
	RAL	And rotate it out to the carry bit
	LAI 000	Clear ACC without disturbing carry status
	ACE	Add in any carry to complemented value
	JMP MORCOM	Continue the two's complement procedure as req'd
		N'th precision rotate left subroutine. Rotates a multi-byte number left starting at the address initially specified by the contents of CPU registers H & L upon subroutine entry (LSW). First entry point will clear the carry bit before beginning rotate operations. Second entry point does not clear the carry bit.
ROTATL,	NDA	Clear the carry bit at this entry point
ROTL,	LAM	Fetch a byte from memory
	RAL	Rotate it left (bring carry into LSB, push MSB to carry)
	LMA	Restore rotated word to memory
	DCB	Decrement precision counter
	RTZ	Exit to caller when finished
	INL	Else advance pointer to next byte
	JMP ROTL	Continue rotate left operations
		N'th precision rotate right subroutine. Opposite of above subroutine.
ROTATR,	NDA	Clear the carry bit at this entry point
ROTR,	LAM	Fetch a byte from memory
	RAR	Rotate it right (carry into MSB, LSB to carry)
	LMA	Restore rotated word to memory
	DCB	Decrement precision counter
	RTZ	Exit to caller when finished
	DCL	Else decrement pointer to next byte
	JMP ROTR	Continue rotate right operations
		N'th precision subtraction subroutine. Number starting at location pointed to by D & E (least significant byte) is subtracted from number starting at address specified by contents of H & L.
SUBBER,	NDA	Initialize the carry bit to zero upon entry
SUBTRA,	LAM	Feth byte from register group A
	CAL SWITCH	Switch memory pointer to register group B
	SBM	Subtract byte from group B from that in group A
	LMA	Leave result in register group B
	DCB	Decrement number of bytes (precision) counter
	RTZ	Return to caller when all bytes in group processed

	INL CAL SWITCH INL JMP SUBTRA	Else advance pointer for register group B Switch memory pointer back to register group A Advance the pointer for register group A Continue the multi-byte subtraction operation
		The next subroutine will transfer the four byte register string (generally a number in floating point format) from the starting address pointed to by CPU registers H & L when the subroutine is entered to the FPACC (floating point accumulator registers).
FLOAD,	LDI 001 LEI 124 LBI 004 JMP MOVEIT	** Set page address of FPACC Set address of least significant byte of FPACC Set precision counter to four bytes (mantissa bytes Plus Exponent) and exit via the transfer routine
		The next several subroutines are used to perform floating point register loading and transfer operations.
FSTORE,	LEL LDH LLI 124 LHI 001 JMP SETIT	Transfer contents of register L to E Transfer contents of register H to D Set L to least significant byte of FPACC mantissa ** Set page to FPACC storage area Go transfer FPACC contents to area pointed to by D&E
OPLOAD,	LDI 001 LEI 134	** Set page to FPOP storage area Set pointer to least significant byte of FPOP
SETIT,	LBI 004 JMP MOVEIT	Set precision counter. Transfer from H & L area to Locations pointed to by D & E
		The next subroutine performs a double transfer operation. It first transfers the contents of the FPACC into the FPOP. It then transfers new data (as pointed to by H & L upon entry to the subroutine) into the FPACC.
FACXOP,	CAL SAVEHL LLI 124 LHI 001 CAL OPLOAD CAL RESTHL JMP FLOAD	Save contents of H & L upon entry to subroutine Set pointer to FPACC LSW ** Set pointer to page of FPACC Transfer FPACC to FPOP Recover original subroutine entry values for H & L Transfer registers pointed to by H & L into the FPACC
		Subroutine to save the contents of CPU registers D, E, H and L in a temporary storage area in memory.
SAVEHL,	LAH LBL LLI 200 LHI 001 LMA INL	Transfer value in H to ACC And value in L to B Now set L to start of temporary storage locations ** And set H to storage area page Save A (entry value of H) in memory Advance pointer

LMB	Save B (entry value of L) in memory
INL	Advance pointer
LMD	Save D in memory
INL	Advance pointer
LME	Save E in memory
LHA	Restore entry value of H
LLB	Restore entry value of L
RET	Exit to calling routine
	Subroutine to restore the contents of CPU registers D, E, H and L from temporary storage in memory.
RESTHL, LLI 200	Set L to start of temporary storage locations
LHI 001	** Set H to storage area page
LAM	Fetch stored value for H in ACC
INL	Advance pointer
LBM	Fetch stored value for L into B
INL	Advance pointer
LDM	Fetch stored value for D
INL	Advance pointer
LEM	Fetch stored value for E
LHA	Restore saved value for H
LLB	Restore saved value for L
LAM	Leave stored value for E in ACC
RET	Exit to calling routine
	Subroutine to exchange the contents of H & L with D & E.
SWITCH, LCH	Transfer register H to C temporarily
LHD	Place value of D into H
LDC	Now put former H from C into D
LCL	Transfer register L to C temporarily
LLE	Place value of E into L
LEC	Now put former L from C into E
RET	Exit to calling routine

CONVERSION OF FIXED AND FLOATING POINT DECIMAL TO FLOATING POINT BINARY

The next section of the floating point package is used to convert strings of ASCII characters representing fixed or floating point numbers to floating point binary numbers.

The ASCII character strings which are to be inputted to this portion of the floating

point package will be residing in a buffer, such as the SYMBOL or TOKEN buffer, after having been evaluated by other portions of SCELBAL as representing numbers.

Such numbers may be in the form of fixed point decimal numbers such as:

1234.56

or floating point decimal numbers such as:

654.321 E-15

The next portion of the floating point program effectively inputs these character strings representing decimal numbers and converts them to a normalized floating point binary number for further processing by SCELBAL.

This is accomplished in a two part process. First the ASCII character string representing the mantissa portion of a decimal number is converted to a normalized binary floating point number. Next, any decimal exponent associated with the mantissa, as in the case when a floating point decimal number is being inputted, is processed. This conversion is accomplished by raising the binary floating

point representation of the mantissa by a power of ten for each digit in the decimal exponent. (This is readily accomplished as will be observed shortly by calling on the subroutine FPMULT presented earlier in this chapter.) Or, by multiplying the floating point representation of the mantissa by one tenth (dividing by ten) for each digit in the decimal exponent when it represents a minus power.

The decimal to binary conversion routine must also examine the signs of the decimal numbers (mantissas and exponents) and take appropriate steps to negate the binary representations as necessary.

All of these tasks are handled by the next section of the package as may be observed by studying the following source listing.

The following subroutine is used to input decimal number strings (stored as ASCII characters in a buffer) to the floating point input routine. Each time the subroutine is called it fetches one ASCII character from the buffer location pointed to by the contents of D & E (upon entry) as augmented by an indexing register.

GETINP,	LHI 001 LLI 220 LCM INC DCC JFZ NOT0 LLE LHD LCM INC CAL INDEXC LMI 000	** Set H to page of GETINP character counter Set L to address of GETINP character counter Load counter value into CPU register C Exercise the counter in order To set CPU flags. If counter is non-zero, then indexing Register (GETINP counter) is all set so jump ahead. But, if counter zero, then starting to process a new Character string. Transfer char string buffer pointer into H & L and fetch the string's character count value (cc) Increment the (cc) by one to take account of (cc) byte Add contents of regis C to H & L to point to end of the Character string in buffer and place a zero byte marker
NOT0,	LLI 220 LHI 001 LCM INC LMC LLE LHD CAL INDEXC	Set L back to address of GETINP counter which is used ** As an indexing value. Set H to correct page. Fetch the value of GETINP counter into register C Increment the value in C Restore the updated value for future use Bring the base address of the character string buffer into CPU registers H & L Add contents of register C to form indexed address of

	LAM NDA LHI 001 RFZ LLI 220 LMI 000 RET	Next character to be fetched as input. Fetch the next Character. Exercise the CPU flags. ** Restore page pointer to floating point working area If character is non-zero, not end of string, exit to caller If zero character, must reset GETINP counter for next String. Reset pointer and clear GETINP counter to zero Then exit to calling routine
		Following subroutine causes register C to be used as an indexing register. Value in C is added to address in H and L to form new address.
INDEXC,	LAL ADC LLA RFC INH RET	Place value from register L into accumulator Add quantity in register C Restore updated value back to L Exit to caller if no carry from addition But, if have carry then must increment register H Before returning to calling routine
		Main Decimal INPUT subroutine to convert strings of ASCII characters representing decimal fixed or floating point numbers to binary floating point numbers.
DINPUT,	LEL LDH LHI 001 LLI 150 XRA LBI 010	Save entry value of register L in E. (Pointer to buffer Containing ASCII character string.) Do same for H to D. ** Set H to page of floating point working registers Set L to start of decimal-to-binary working area Clear the accumulator Set up a loop counter
CLRNX2,	LMA INL DCB JFZ CLRNX2 LLI 103 LBI 004	Deposit zero in working area to initialize Advance the memory pointer Decrement the loop counter Clear working area until loop counter is zero Set pointer to floating point temporary registers and Indicators working area. Set up a loop counter.
CLRNX3,	LMA INL DCB JFZ CLRNX3 CAL GETINP CPI 253 JTZ NINPUT CPI 255 JFZ NOTPLM LLI 103 LMA	Deposit zero in working area to initialize Advance the memory pointer Decrement the loop counter Clear working area until loop counter is zero Fetch a character from the ASCII char string buffer (Typically the SYMBOL/TOKEN buffer). See if it is Code for + sign. Jump ahead if code for + sign. See if code for minus (-) sign. Jump ahead if not code for minus sign. If code for Minus sign, set pointer to MINUS flag storage location. Set the MINUS flag to indicate a minus number
NINPUT,	CAL GETINP	Fetch another character from the ASCII char string

NOTPLM,	CPI 256 JTZ PERIOD CPI 305 JTZ FNDEXP CPI 240 JTZ NINPUT NDA JTZ ENDINP CPI 260 JTS NUMERR CPI 272 JFS NUMERR LLI 156 LCA LAI 370 NDM JFZ NINPUT LLI 105 LBM INB LMB CAL DECBIN JMP NINPUT	See if character represents a period (decimal point) in Input string. Jump ahead if yes. If not period, see if code for E as in Exponent Jump ahead if yes. Else see if code for space. Ignore space character, go fetch another character. If none of the above see if zero byte Indicating end of input char string. If yes, jump ahead. If not end of string, check to see If character represents A valid decimal number (0 to 9) Display error message if not a valid digit at this point! For valid digit, set pointer to MSW of temporary Decimal to binary holding registers. Save character in C. Form mask for sizing in accumulator. Now see if Holding register has enough room for the conversion of Another digit. Ignore the input if no more room. If have room in register then set pointer to input digit Counter location. Fetch the present value. Increment it to account for incoming digit. Restore updated count to storage location. Call the DECimal to BINary conversion routine to add In the new digit in holding registers. Continue inputting.
PERIOD,	LBA LLI 106 LAM NDA JFZ NUMERR LLI 105 LMA INL LMB JMP NINPUT	Save character code in register B Set pointer to PERIOD indicator storage location Fetch value in PERIOD indicator Exercise CPU flags If already have a period then display error message If not, change pointer to digit counter storage location Clear the digit counter back to zero Advance pointer to PERIOD indicator Set the PERIOD indicator Continue processing the input character string
FNDEXP,	CAL GETINP CPI 253 JTZ EXPINP CPI 255 JFZ NOEXPS LLI 104 LMA	Get next character in Exponent See if it is code for + sign Jump ahead if yes. If not + sign, see if minus sign If not minus sign then jump ahead For minus sign, set pointer to EXP SIGN indicator Set the EXP SIGN indicator for a minus exponent
EXPINP,	CAL GETINP	Fetch the next character in the decimal exponent
NOEXPS,	NDA JTZ ENDINP CPI 260 JTS NUMERR CPI 272 JFS NUMERR	Exercise the CPU flags If character inputted was zero, then end of input string If not end of string, check to see If character represents A valid decimal number (0 to 9) Display error message if not a valid digit at this point!

	NDI 017	Else trim the ASCII code to BCD
	LBA	And save in register B
	LLI 157	Set pointer to input exponent storage location
	LAI 003	Set accumulator equal to three
	CPM	See if any previous digit in exponent greater than three
	JTS NUMERR	Display error message if yes
	LCM	Else save any previous value in register C
	LAM	And also place any previous value in accumulator
	NDA	Clear the carry bit with this instruction
	RAL	Single precision multiply by ten algorithm
	RAL	Two rotate lefts equals times four
	ADC	Adding in the digit makes total times five
	RAL	Rotating left again equals times ten
	ADB	Now add in digit just inputted
	LMA	Restore the value to exponent storage location
	JMP EXPINP	Go get any additional exponent input
ENDINP,	LLI 103	Set pointer to mantissa SIGN indicator
	LAM	Fetch the SIGN indicator to the accumulator
	NDA	Exercise the CPU flags
	JTZ FININP	If SIGN indicator is zero, go finish up as nr is positive
	LLI 154	But, if indicator is non-zero, number is negative
	LBI 003	Set pntr to LSW of storage registers, set precision cntr
	CAL COMPLM	Negate the triple-precision number in holding registers
FININP,	LLI 153	Set pointer to input storage LSW minus one
	XRA	Clear the accumulator
	LMA	Clear the LSW minus one location
	LDH	Set register D to floating point working page
	LEI 123	Set E to address of FPACC LSW minus one
	LBI 004	Set precision counter
	CAL MOVEIT	Move number from input register to FPACC
	CAL FPFLT	Now convert the binary fixed point to floating point
	LLI 104	Set pointer to Exponent SIGN indicator location
	LAM	Fetch the value of the EXP SIGN indicator
	NDA	Exercise the CPU flags
	LLI 157	Reset pointer to input exponent storage location
	JTZ POSEXP	If EXP SIGN indicator zero, exponent is positive
	LAM	Else, exponent is negative so must negate
	XRI 377	The value in the input exponent storage location
	ADI 001	By performing this two's complement
	LMA	Restore the negated value to exponent storage location
POSEXP,	LLI 106	Set pointer to PERIOD indicator storage location
	LAM	Fetch the contents of the PERIOD indicator
	NDA	Exercise the CPU flags
	JTZ EXPOK	If PERIOD indicator clear, no decimal point involved
	LLI 105	If have a decimal point, set pointer to digit counter
	XRA	Storage location. Clear the accumulator.
	SUM	And get a negated value of the digit counter in ACC

EXPOK,	LLI 157 ADM LMA JTS MINEXP RTZ	Change pointer to input exponent storage location Add this value to negated digit counter value Restore new value to storage location If new value is minus, skip over next subroutine If new value is zero, no further processing required
FPX10,	LLI 210 LHI 001 CAL FACXOP CAL FPMULT LLI 157 LCM DCC LMC JFZ FPX10 RET	Following subroutine will multiply the floating point binary number stored in FPACC by ten times the value stored in the decimal exponent storage location. Set pointer to registers containing floating point ** Binary representation of 10 (decimal). Transfer FPACC to FPOP and 10 (dec) to FPACC Multiply FPOP (formerly FPACC) by 10 (decimal) Set pointer to decimal exponent storage location Fetch the exponent value Decrement Restore to storage If exponent value is not zero, continue multiplication When exponent is zero can exit. Conversion completed.
MINEXP, FPD10,	LLI 214 LHI 001 CAL FACXOP CAL FPMULT LLI 157 LBM INB LMB JFZ FPD10 RET	Following subroutine will multiply the floating point binary number stored in FPACC by 0.1 times the value (negative) stored in the decimal exponent storage location. Set pointer to registers containing floating point ** Binary representation of 0.1 (decimal). Transfer FPACC to FPOP and 0.1 (dec) to FPACC Multiply FPOP (formerly FPACC) by 0.1 (decimal) Set pointer to decimal exponent storage location Fetch the exponent value Increment Restore to storage If exponent value is not zero, continue multiplication When exponent is zero can exit. Conversion completed.
DECBIN,	CAL SAVEHL LLI 153 LAC NDI 017 LMA LEI 150 LLI 154 LDH LBI 003 CAL MOVEIT	Following subroutine is used to convert decimal characters to binary fixed point format in a triple-precision format. Save entry value of D, E, H and L in memory Set pointer to temporary storage location Restore character inputted to accumulator Trim ASCII code to BCD Store temporarily Set pointer to working area LSW of multi-byte register Set another pointer to LSW of conversion register Make sure D set to page of working area Set precision counter Move original value of conversion register to working

LLI 154	Register. Reset pointer to LSW of conversion register.
LBI 003	Set precision counter
CAL ROTATL	Rotate register left. (Multiplies value by two.)
LLI 154	Reset pointer to LSW.
LBI 003	Set precision counter
CAL ROTATL	Multiply by two again (total now times four).
LEI 154	Set pointer to LSW of conversion register.
LLI 150	Set pointer to LSW of working register (original value).
LBI 003	Set precision counter.
CAL ADDER	Add original value to rotated value (now times five).
LLI 154	Reset pointer to LSW
LBI 003	Set precision counter
CAL ROTATL	Multiply by two once more (total now times ten).
LLI 152	Set pointer to clear working register locations
XRA	Clear the accumulator
LMA	Clear MSW of working register
DCL	Decrement pointer
LMA	Clear next byte
LLI 153	Set pointer to current digit storage location
LAM	Fetch the current digit
LLI 150	Change pointer to LSW of working register
LMA	Deposit the current digit in LSW of working register
LEI 154	Set pointer to conversion register LSW
LBI 003	Set precision counter
CAL ADDER	Add current digit to conversion register to complete
JMP RESTHL	Conversion. Exit to caller by restoring CPU registers.

CONVERSION OF FLOATING POINT BINARY TO FIXED AND FLOATING POINT DECIMAL

The final section of the SCELBAL floating point package performs essentially the reverse of the portion just presented. It will convert a number from floating point binary format into fixed or floating point decimal format for display on the user's output device.

Selecting between fixed point and floating point decimal output is automatically determined by the conversion routine. If the number stored in binary floating point format can be represented in 23 binary bits or less, and is greater than one, the number will be displayed in fixed point format with the decimal point positioned as required. If the number is not within this range, it will be outputted in decimal floating point format as a mantissa raised to the appropriate decimal power of ten.

The routine operates in essentially the reverse manner of the input routine. First the floating point binary number is converted to a fixed point binary number (representing the mantissa digits of its decimal equivalent) and an associated binary exponent portion representing the powers of ten to which the decimal mantissa is to be raised (for numbers requiring an exponent). These binary representations are then converted and displayed as decimal digits with the output being the ASCII code for each digit in the number. The output routine also takes care of inserting a decimal point and minus signs if appropriate.

The source listing for this final section of the floating point package is presented next.

The first portion of the FPOUT subroutine performs initializing operations and then determines whether the output is to be in fixed or floating point format.

FPOUT,	LHI 001 LLI 157 LMI 000 LLI 126 LAM NDA JTS OUTNEG LAI 240 JMP AHEAD1	** Set H to working area for floating point routines Set pointer to decimal exponent storage location Initialize storage location to zero Change pointer to FPACC (number to be outputted) And fetch MSW of FPACC Test the contents of MSW of FPACC If most significant bit of MSW is a one, have a minus nr. Else number is positive, set ASCII code for space for a Positive number and go display a space
OUTNEG,	LLI 124 LBI 003 CAL COMPLM LAI 255	If number in FPACC is negative must negate in order To display. Set pptr to LSW of FPACC & set prec. cntr. Negate the number in the FPACC to make it positive But load ACC with ASCII code for minus sign
AHEAD1,	CAL ECHO LLI 110 LAM NDA JTZ OUTFLT LLI 127 LAI 027 LBM INB DCB JTS OUTFLT SUB JTS OUTFLT JMP OUTFIX	Call user display driver to output space or minus sign Set pointer to FIXED/FLOAT indicator Fetch value of FIXED/FLOAT indicator Test contents of indicator. If contents are zero, calling Routine has directed floating point output format. If indicator non-zero, fixed point format requested if Possible. Point to FPACC Exponent. Put 23 decimal in Accumulator. Fetch FPACC Exponent into register B And exercise the register to test its Original contents. If FPACC Exponent is negative in Value then go to floating point output format. If value Is positive, subtract value from 23 (decimal). If result Negative, number is too big to use fixed format. Else, can use fixed format so skip next routine
OUTFLT,	LLI 110 LMI 000 LAI 260 CAL ECHO LAI 256 CAL ECHO	Set pointer to FIXED/FLOAT indicator. Clear indicator to indicate floating point output format Load ASCII code for '0' into accumulator Call user display driver to output '0' as first character in Number string. Now load ASCII code for decimal point. Call user display driver to output '.' as second character.
OUTFIX,	LLI 127 LAI 377 ADM LMA	Set pointer to FPACC Exponent Load accumulator with minus one Add value in FPACC Exponent Restore compensated exponent value

Next portion of routine establishes the value for the decimal exponent that will be outputted by processing the binary exponent value in the FPACC.

DECEXT,	JFS DECEXD LAI 004 ADM JFS DECOUT LLI 210 LHI 001 CAL FAXOP CAL FPMULT LLI 157 LCM DCC LMC	If compensated exponent value is zero or positive Then go multiply FPACC by 0.1 (decimal). Else, Add four to the exponent value. If exponent now zero or positive, ready to output If exponent negative, multiply FPACC by 10 (decimal) ** Set pointer to registers holding 10 (dec) in binary Floating point format. Set up for multiplication. Perform the multiplication. Answer in FPACC. Set pointer to decimal exponent storage location. Each time the FPACC is multiplied by ten, need to Decrement the value in the decimal exponent storage Location. (This establishes decimal exponent value!)
DECREP,	LLI 127 LAM NDA JMP DECEXT	Reset pointer to FPACC Exponent Fetch value in exponent Test value Repeat process as required
DECEXD,	LLI 214 LHI 001 CAL FAXOP CAL FPMULT LLI 157 LBM INB LMB JMP DECREP	If exponent is positive, multiply FPACC by 0.1 ** Set pointer to registers holding 0.1 (dec) in binary Floating point format. Set up for multiplication. Perform the multiplication. Answer in FPACC. Set pointer to decimal exponent storage location. Each time the FPACC is multiplied by one tenth, need To increment the value in the decimal exponent storage Location. (This establishes decimal exponent value!) Repeat process as required
		The next section outputs the mantissa (or fixed point number) by converting the value remaining in the FPACC (after the decimal exponent equivalent has been extracted from the original value if required by the pre- vious routines) to a string of decimal digits.
DECOUT,	LEI 164 LDH LLI 124 LBI 003 CAL MOVEIT LLI 167 LMI 000 LLI 164 LBI 003 CAL ROTATL CAL OUTX10	Set pointer to LSW of output working register Set D to same page value as H Set pointer to LSW of FPACC Set precision counter Move value in FPACC to output working register Set pointer to MSW plus one of output working register Clear that location to zero Set pointer to LSW of output working register Set precision counter Rotate register left once to compensate for sign bit Multiply output register by 10, overflow into MSW+1
COMPEN,	LLI 127 LBM INB LMB JTZ OUTDIG	Set pointer back to FPACC Exponent Compensate for any remainder in the binary exponent By performing a rotate right on the output working Register until the binary exponent becomes zero Go output decimal digits when this loop is finished

	LLI 167 LBI 004 CAL ROTATR JMP COMPEN	Binary exponent compensating loop. Set pointer to Working register MSW+1. Set precision counter. Rotate working register to the right. Repeat loop as required.
OUTDIG,	LLI 107 LMI 007 LLI 167 LAM NDA JTZ ZERODG	Set pointer to output digit counter storage location Initialize to value of seven Change pointer to output working register MSW+1 Fetch MSW+1 byte containing BCD of digit to be Displayed. Test the contents of this byte. If zero jump to ZERODG routine.
OUTDGS,	LLI 167 LAM NDA JFZ OUTDGX LLI 110 LAM NDA JTZ OUTZER LLI 157 LCM DCC INC JFS OUTZER LLI 166 LAM NDI 340 JFZ OUTZER RET	Reset pointer to working register MSW+1 Fetch BCD of digit to be outputted Exercise CPU flags If not zero, go display the digit If zero, change pointer to FIXED/FLOAT indicator Fetch the indicator into the accumulator Test value of indicator If in floating point mode, go display the digit Else change pointer to decimal exponent storage Location, which, for fixed point, will have a positive Value for all digits before the decimal point. Decrement And increment to exercise flags. See if count is positive. If positive, must display any zero digit. If not, change pointer to MSW of working register And test to see if any significant digits coming up By forming a mask and testing for presence of bits If more significant digits coming up soon, display the Zero digit. Else, exit to calling routine. Finished.
OUTZER,	XRA	Clear the accumulator to restore zero digit value
OUTDGX,	ADI 260 CAL ECHO	Add 260 (octal) to BCD code in ACC to form ASCII Code and call the user's display driver subroutine
DECRDG,	LLI 110 LAM NDA JFZ CKDECP LLI 107 LCM DCC LMC JTZ EXPOUT	Set pointer to FIXED/FLOAT indicator storage Fetch the indicator to the accumulator Exercise the CPU flags If indicator non-zero, doing fixed point output Else, get output digit counter Decrement the digit counter & restore to storage When digit counter is zero, go take care of exponent
PUSHIT,	CAL OUTX10 JMP OUTDGS	Else push next BCD digit out of working register And continue the outputting process
CKDECP,	LLI 157 LCM	For fixed point output, decimal exponent serves as Counter for number of digits before decimal point

	DCC	Fetch the counter and decrement it to account for
	LMC	Current digit being processed. Restore to storage.
	JFZ NODECP	If count does not go to zero, jump ahead.
	LAI 256	When count reaches zero, load ASCII code for period
	CAL ECHO	And call user's display driver to display decimal point
NODECP,	LLI 107	Set pointer to output digit counter storage location
	LCM	Fetch the digit counter
	DCC	Decrement the value
	LMC	Restore to storage
	RTZ	If counter reaches zero, exit to caller. Finished.
	JMP PUSHIT	Else continue to output the number.
ZERODG,	LLI 157	If first digit of floating point number is a zero, set
	LCM	Pointer to decimal exponent storage location.
	DCC	Decrement the value to compensate for skipping
	LMC	Display of first digit. Restore to storage.
	LLI 166	Change pointer to MSW of output working register
	LAM	Fetch MSW of output working register
	NDA	Test the contents
	JFZ DECRDG	If non-zero, continue outputting
	DCL	Else decrement pointer to next byte in working register
	LAM	Fetch its contents
	NDA	Test
	JFZ DECRDG	If non-zero, continue outputting
	DCL	Else decrement pointer to LSW of working register
	LAM	Fetch its contents
	NDA	Test
	JFZ DECRDG	If non-zero, continue outputting
	LLI 157	If decimal mantissa is zero, set pointer to decimal
	LMA	Exponent storage and clear it.
	JMP DECRDG	Finish outputting
		Following routine multiplies the binary number in the
		output working register by ten to push the most significant
		digit out to the MSW+1 byte.
OUTX10,	LLI 167	Set pointer to working register MSW+1
	LMI 000	Clear it in preparation for receiving next digit pushed
	LLI 164	Into it. Change pointer to working register LSW.
	LDH	Set up register D to same page as H.
	LEI 160	Set second pointer to LSW of second working register
	LBI 004	Set precision counter
	CAL MOVEIT	Move first working register into second
	LLI 164	Reset pointer to LSW of first working register
	LBI 004	Set precision counter
	CAL ROTATL	Rotate contents of first working register left (X 2)
	LLI 164	Reset pointer to LSW
	LBI 004	Reset precision counter
	CAL ROTATL	Rotate contents left again (X 4)
	LLI 160	Set pointer to LSW of original value in 2'nd register

LEI 164	Set pointer to LSW of rotated value
LBI 004	Set precision counter
CAL ADDER	Add rotated value to original value (X 5)
LLI 164	Reset pointer to LSW of first working register
LBI 004	Set precision counter
CAL ROTATL	Rotate contents left again (X 10)
RET	Exit to calling routine

The final group of routines in the floating point output section take care of outputting the decimal exponent portion of floating point numbers.

EXPOUT,	LLI 157 LAM NDA RTZ LAI 305 CAL ECHO LAM NDA JTS EXOUTN LAI 253 JMP AHEAD2	Set pointer to decimal exponent storage location Fetch value to the accumulator Test the value If zero, then no exponent portion. Exit to caller. Else, load ACC with ASCII code for letter E. Display E for Exponent via user's display driver rtn Get decimal exponent value back into ACC Test again If value is negative, skip ahead If positive, load ASCII code for + sign Jump to display the + sign
EXOUTN,	XRI 377 ADI 001 LMA LAI 255	When decimal exponent is negative, must negate Value for display purposes. Perform two's complement And restore the negated value to storage location Load ASCII code for minus sign
AHEAD2,	CAL ECHO LBI 000 LAM	Display the ASCII character in ACC Clear register B Fetch the decimal exponent value back into ACC
SUB12,	SUI 012 JTS TOMUCH LMA INB JMP SUB12	Subtract 10 (decimal) from value in ACC Break out of loop when accumulator goes negative Else restore value to storage location Increment register B as a counter Repeat loop to form tens value of decimal exponent
TOMUCH,	LAI 260 ADB CAL ECHO LAM ADI 260 CAL ECHO RET	Load base ASCII value for digit into the accumulator Add to the count in B to form tens digit of decimal Exponent. Display via user's driver subroutine Fetch remainder of decimal exponent value Add in ASCII base value to form final digit Display second digit of decimal exponent Finished outputting. Return to caller.

I/O ROUTINES

Because of the wide variety of I/O devices that individual system owners may have connected to their computers, SCELBAL was designed so that individual users could provide their own actual I/O routines. In order to allow this, the reader may have noted in the previous chapters that all references to I/O routines are vectored to one of four locations in the program. Each one of these locations contains a jump or call instruction that the user must complete by supplying the actual address to the user supplied I/O routine. The four locations referred to are discussed here.

The location in the program labeled CINPUT (located at the address 03 221 in the assembled version of the program presented in this publication) is the vector instruction for the user provided OPERATOR INPUT DEVICE. This device would typically be an electronic keyboard or similar device on which the operator would type in commands to the SCELBAL executive and enter statements or programs into the user program buffer. SCELBAL expects all inputs to the program itself to be in the form of ASCII encoded characters with the eighth bit always marking. A list of the octal codes for ASCII encoded characters utilized by the program is shown on the next page. This routine should also provide a duplicate of the character received on the system's output device so that the user may verify the characters inputted to the program.

The vector point for sending data from the program to the system's display device is located in the subroutine labeled ECHO (at address 03 213 in the assembled version of the program). The output device would typically be an electro-mechanical printing device or other suitable display mechanism on which data from the program may be displayed. SCELBAL has the ASCII code for the character to be displayed in the accumulator when this vector point is encountered. It expects the user provided output driver

routine to display the character corresponding to the ASCII code on the system's display device. Of course, if the user's display mechanism uses some other type of code, it is possible for the user to insert an appropriate conversion routine in the output routine. (This also applies for inputs.)

There are several extremely important considerations for the reader to bear in mind when preparing to implement the actual I/O driving routines to be used with SCELBAL. (The following two considerations refer to I/O operations involving the system device through which the operator communicates with the program. They do not apply to the I/O routines associated with the system's bulk storage device which will be discussed further on in this chapter.)

1. Only CPU register B and the accumulator may be used by the I/O routines. All the other CPU registers must contain their original values when I/O operations have been completed.

2. For the 8008 version of SCELBAL, the I/O routines themselves may only utilize a maximum of two levels of nesting! This is because, when called, the 8008 internal stack may at times be loaded to the point where pushing the stack down more than two times would result in the loss of stack information.

Consideration number one above causes no real concern for readers who implement SCELBAL on an 8080 system. The 8080, which has the CPU's stack implemented in RAM memory, can easily save CPU registers C through L on the stack if required while performing an I/O operation. The registers may then be restored from the stack when the I/O operation is completed.

For 8008 users, the consideration will be fairly easy for most users to cope with if their I/O device has a parallel type interface with the computer such as commonly found

in devices that utilize a UART device. With such an interface it is generally quite easy to perform the necessary transfer functions using just the accumulator and a CPU register. (Just remember to use register B!)

Users with a serial interface may find the restriction somewhat challenging, especially if restriction number two above also applies. As an aid to those that might find themselves in such a situation, an example input and output routine designed to operate with a serial electro-mechanical keyboard and printer, that

satisfies both conditions above, will be provided starting on the next page.

Consideration number two must be strictly adhered to when SCELBAL is operating in an 8008 system. Naturally, for an 8080 based unit with its stack residing in RAM memory, the restriction does not apply provided that the user allocates sufficient room for the stack in memory. Recommendations of suitable areas in memory that may be reserved for 8080 stack use are made in the chapter that contains the object code listing of the SCELBAL program for the 8080 CPU.

CHARACTER	BINARY	OCTAL
A	11 000 001	301
B	11 000 010	302
C	11 000 011	303
D	11 000 100	304
E	11 000 101	305
F	11 000 110	306
G	11 000 111	307
H	11 001 000	310
I	11 001 001	311
J	11 001 010	312
K	11 001 011	313
L	11 001 100	314
M	11 001 101	315
N	11 001 110	316
O	11 001 111	317
P	11 010 000	320
Q	11 010 001	321
R	11 010 010	322
S	11 010 011	323
T	11 010 100	324
U	11 010 101	325
V	11 010 110	326
W	11 010 111	327
X	11 011 000	330
Y	11 011 001	331
Z	11 011 010	332
[11 011 011	333
\	11 011 100	334
]	11 011 101	335
↑	11 011 110	336
←	11 011 111	337
SPACE	11 100 000	240

CHARACTER	BINARY	OCTAL
!	10 100 001	241
"	10 100 010	242
#	10 100 011	243
\$	10 100 100	244
%	10 100 101	245
&	10 100 110	246
,	10 100 111	247
(10 101 000	250
)	10 101 001	251
*	10 101 010	252
+	10 101 011	253
,	10 101 100	254
-	10 101 101	255
.	10 101 110	256
/	10 101 111	257
0	10 110 000	260
1	10 110 001	261
2	10 110 010	262
3	10 110 011	263
4	10 110 100	264
5	10 110 101	265
6	10 110 110	266
7	10 110 111	267
8	10 111 000	270
9	10 111 001	271
:	10 111 010	272
;	10 111 011	273
<	10 111 100	274
=	10 111 101	275
>	10 111 110	276
?	10 111 111	277
Control 'C'	10 000 011	203

TABLE OF ASCII CODES WITH PARITY BIT MARKING AS USED BY SCELBAL

		Routine to receive serial data from an INPUT device connected to bit B7 of an input port. Incoming characters assumed to be in format: 1 start bit, eight data bits (1 to 8) and 2 stop bits. Timing loops in example shown for characters coming in at a rate of 10 characters per second and assuming 8008 CPU clock set at 500 Khz. Received character will be in the accumulator when routine is finished. This routine will automatically echo the character received to an OUTPUT device connected to bit B0 of an output port. To disable the echo replace output instructions with NOPs such as LAA. This routine uses only register B and the accumulator and does not push the CPU stack down more than two levels as it operates.
RCV,	INP +†† NDA JTS RCV XRA LBI 104	Sample the current input on the serial line from input Device. Check to see if the line has gone to logic zero Condition indicating a possible START bit. If not, loop To look for the start bit. If have start bit, clear the ACC. Set a counter up in register B to cause time delay equal
MORE1,	DCB JFZ MORE1 OUT +†† CAL TIMER CAL NEXBIT CAL NEXBIT CAL NEXBIT CAL NEXBIT CAL NEXBIT CAL NEXBIT CAL NEXBIT CAL NEXBIT CAL NEXBIT	To about half a bit. Fall into the first timing loop and Time it out until counter in B is zero. Now start the Echo process by sending logic zero to output device. Call subroutine to provide time delay equal to one bit. Input the first bit. Input the second bit. Input the third bit. Input the fourth bit. Input the fifth bit. Input the sixth bit. Input the seventh bit. Input the eighth bit.
STOP,	LAI 001 OUT +†† LAB RLC LBI 314	Set up stop bit for the output device. Send a logic one to the output device. Fetch the character from B to the ACC. Format character to compensate for R.R.C by NEXBIT. Set up a counter in register B to cause time delay equal
MORE3,	DCB JFZ MORE3 RET	To about one and a half bits for STOP bits. Fall into the Timing loop and time out until counter is zero. Now return to calling routine with character in ACC.
NEXBIT,	INP +†† NDI 200 RLC OUT +†† RRC ADB RRC	Input a character to bit B7 from the selected input port. Mask off bits b6 through B0 to leave just bit B7. Position the bit in B7 to bit B0 to prepare to Echo bit. Output bit B0 to the output device. Restore the bit back to B7. Add previous bits in character stored in register B. Rotate all bits to make room for next incoming bit.

TIMER,	LBI 213	Set up a counter in register B to cause time delay equal
MORE2,	DCB JFZ MORE2 LBA RET	To about one bit. Fall into the timing loop and Time out until counter is zero. Now save the contents of the ACC in register B. Return to main inputting routine.
		Routine to send data in serial format to an OUTPUT device connected to bit B0 of an output port. Character assumed to have same format and is sent at same rate as in the example input routine. Routine expects ASCII encoded character to be in the accumulator when the routine is entered. This routine uses only register B and the accumulator and does not push the CPU stack down more than two levels during its execution.
PRINT,	NDA RAL OUT ††† RAR CAL TIMER CAL BITOUT CAL BITOUT CAL BITOUT CAL BITOUT CAL BITOUT CAL BITOUT CAL BITOUT CAL BITOUT CAL BITOUT LBA LAI 001 OUT ††† LAB CAL TIMER LBI 103 JMP MORE3	Clear the carry flag prior to set up for sending START Bit. Rotate the carry status into bit B0. Now output a Logic zero level for START bit to output device. Restore the original ASCII character in the ACC. Provide one bit delay for sending of the START bit. Output the first bit. Output the second bit. Output the third bit. Output the fourth bit. Output the fifth bit. Output the sixth bit. Output the seventh bit. Output the eighth bit. Save contents of the ACC in register B. Set bit B0 to a logic one for sending STOP bit. Send a logic one from bit B0 to the output device. Restore the character from register B to the ACC. Provide time delay for the two STOP bits. Finish providing time delay for the STOP bits. Exit from the PRINT routine when finished timing out.
BITOUT,	OUT ††† RRC CAL TIMER RET	Output status of B0 to output device. Position the next bit in the ACC to bit position B0. Provide one bit time delay. Return to main outputting routine.

It is important to reiterate, as illustrated in the example INPUT subroutine, that the input routine provided by the user for use with SCELBAL should reflect the character inputted on the system's output device. If this is not done, the operator will not be able to

see the information as it is inputted. This may be done in the manner illustrated in the example program (where the character is reflected to the output device on a bit-by-bit basis as it is received) or it may be accomplished by simply having the input subroutine

jump to the output subroutine when a character has been completely received. The latter technique, however, generally slows down the overall inputting speed to a level that is unpleasant for many operators if an electro-mechanical I/O device is being used. This is because the operator must wait an extra fraction of a second for the character to be sent to the output device.

NOTE: The example I/O routines presented serve only as guide lines for the special case mentioned where serial I/O devices are being utilized with an 8008 equipped computer. The actual values used in timing loops, and other parameters would vary depending on the individual system's I/O arrangements. Many reader's will not require such elaborate I/O subroutines.

The two types of I/O subroutines discussed to this point are essential to the operation of SCELBAL as they provide the means for the operator to communicate with the program. There are two more types of I/O routines that might be considered optional by some users. These two routines may be created by the user to provide the capability of saving a program that has been placed in the user program buffer on an external bulk storage device, and vice versa.

The reader who desires to save user programs on a bulk storage device should note that the vector to such a routine is located in the EXECutive portion of SCELBAL in the subsection headed by the label NOSCR (at address 11 104 in the assembled version of the program). This vector is taken when an operator specifies the EXECutive command SAVE.

In order to implement SAVE capability the user need only provide a routine that will effectively dump the contents of the user program buffer and the contents of a pair of memory words on the system's bulk storage device. The pair of words that should be saved is the pair that holds the pointer to the end of the user program buffer! In the assembled version of SCELBAL provided in this manual

that register pair is located at 26 364 and 26 365.

Thus, for whatever type of bulk storage device the reader is utilizing, the reader need simply create a routine that will first write out the contents of the user program buffer. (It starts at location 33 000 in the assembled version of SCELBAL provided herein. It ends at the point indicated by the contents of the "end of user program buffer pointer." That point will vary depending on the particular size of a user's program.) Then, the routine should write out the contents of the "end of user program buffer pointer" (which was just used to determine how much of the user program buffer should be written on the bulk storage device)!

The details of such a routine will be entirely a function of the type of bulk storage device the system utilizes. However, for most systems, the creation of such a routine should be quite easy and consist of a series of calls to standard driver routines for the particular device being utilized.

The fourth I/O routine referred to in SCELBAL is the routine that would read in a high level program from the bulk storage device into the user program buffer area and set the "end of user program buffer pointer" to the appropriate value. In essence, all this subroutine does is read back in what the subroutine discussed above wrote on the bulk storage medium, placing it in the appropriate addresses in memory. (The user program buffer and the two bytes of the pointer.)

Reference to this routine is made in the subsection of the EXECutive part of the program labeled NOSCR (at address 11 122 in the assembled version of the program). This routine would be executed when the user issued the LOAD directive.

The two user provided routines for handling the bulk storage device are free to use all of the CPU registers. Additionally, the developer of these routines need only ensure that the use of the CPU stack (by subroutine nest-

ing operations) is kept within the capabilities of the 8008, or, in the case of the 8080, within the boundaries of the memory area reserved for the CPU stack.

Both of the routines associated with the bulk storage device operations should end by directing program operation back to the start of the EXECutive since these operations are essentially independent events. (The EXECutive starts at location 10 266 in the assembled object code listing shown in this book.)

If the reader does not desire to implement the SAVE and LOAD commands, the two vector locations (11 104 and 11 122) can be used to direct the program back to the start of the EXECutive in case a user inadvertently should enter one of those commands.

I/O routines may be placed on page 00 in the system if desired. All of the locations on that page were left for such use in the version of SCELBAL illustrated in this publication. If that page is not suitable, the I/O routines may be tucked into some of the unused memory locations available in the assembled version illustrated on pages 31 and 32 (if the routines are relatively short). Alternately, the reader may reduce the amount of area dedicated to the storage of the user's program (USER PROGRAM BUFFER). If this is done it is recommended that the upper portion of the buffer area be used for that purpose. Then the programmer need only change the "end of buffer page" value (page 12 location 122) so that the buffer area is limited to protect the installed I/O routines.

SCELBAL ASSEMBLED FOR OPERATION IN AN 8008 BASED SYSTEM

This chapter presents an assembled version of SCELBAL for operation in an 8008 based microcomputer. This version may be loaded into a system along with the user provided I/O subroutines to provide the user with SCELBAL capability.

The user may elect, by choosing the proper machine codes at key locations, to load the program as an 8 K version that does not have the optional DIM statement capability. This version of the program will leave room for about 1,250 bytes in the user program buffer. Or, the user may load the program as a 12 K version with DIM capability. (Leaving about 4,500 bytes for program storage.) Alternately, by changing a few specially marked locations, the user may elect to have the program operate in 8 K of memory with DIM capability. However, this version is not recommended because it will leave only about 500 bytes for storage of a high level language user program. (It is mentioned as an option because some prospective users may desire to run small programs that require the DIM capability.) Finally, the user may opt to place the DIM routines (by changing the associated pointers, etc.) in the upper pages of available RAM memory in any system having more than 8 K of memory (such as a 10 K, 16 K, 32K system) and using the area between the locations used by the main SCELBAL routines and the optional DIM routines as a user program buffer.

The reader who has studied this book to this point should have no difficulty understanding what is involved in selecting the options just mentioned. Many readers may well elect to make other alterations and may, of course, do so at their own discretion. Let it be said, that the version presented is just one way in which the program may be assembled for operation!

The reader should pay careful attention in the following object code listing to all locations marked by a double asterisk (**),

double at sign (@@), or double cross (††). The convention established in the earlier chapters for those special indicators will be reviewed here.

A double asterisk (**) is of importance only to those readers who might elect to change the memory pages used for the storage of pointers, counters, temporary buffers and look-up tables. The pages used for these purposes in the version of SCELBAL presented are pages 01, 26 and 27. Readers who take on the task of re-assigning these pages will probably have elected to completely re-assemble SCELBAL and should be equipped (mentally and with suitable hardware!) to take on such a task.

A double cross (††) denotes an elective value on the part of the user. These locations generally refer to the starting addresses of user provided routines (such as I/O drivers), or the assignment of the starting and ending address of the user program buffer area. (For the version presented the user program buffer is assumed to start on page 33 and end on page 54. The ending address would be changed to page 37 if an 8 K system was being used and the DIM capability left out. Or, page 34 for an 8 K system with DIM capability provided, etc.)

Locations marked with a @@ should be replaced with the machine code for a no-operation instruction, such as LAA, if the user will not be using the optional DIM statement capability. Alternately, some of these locations relating to addressing values would be altered if the user elected to change the storage areas for the DIM and associated array handling subroutines.

It is suggested that user I/O subroutines be placed on page 00 if possible. Alternately, they may be placed in the upper regions of available memory. If this is done, the ending address of the user program buffer should be altered accordingly.

Hopefully, all this information makes plenty of sense to the serious reader who has read this publication and is ready to implement SCELBAL.

One final word before presenting the object code is in order. Do not attempt to skip over the machine code listings provided for the special pages 01, 26 and 27. The values in the look-up tables must be in memory along with the initial values of many of the locations on those pages when the program

is first started. (Those locations where the initial values are irrelevant are denoted by XXX.) The format of the object code listing for these special pages will be slightly different than the rest of the listing in that the mnemonics column will contain comments relating to the use of the locations (since the locations will contain "data" versus actual instructions.)

An assembled listing for an 8008 version of SCELBAL will now be presented.

070	01 000	XXX	Not Assigned
071	01 001	XXX	Not Assigned
072	01 002	XXX	Not Assigned
073	01 003	XXX	Not Assigned
074	01 004	000	Stores floating point
075	01 005	000	constant
076	01 006	100	value +1.0
	01 007	001	Not Assigned
	01 010	XXX	Not Assigned
	01 011	XXX	Not Assigned
	01 012	XXX	Exponent Counter
	01 013	000	Stores floating point
	01 014	000	number
	01 015	000	temporarily
	01 016	000	Not Assigned
	01 017	000	Not Assigned
	01 020	XXX	Not Assigned
	01 021	XXX	Not Assigned
	01 022	XXX	Not Assigned
	01 023	XXX	Not Assigned
	01 024	000	Stores floating point
	01 025	000	constant
	01 026	300	value -1.0
	01 027	001	Scratch Pad Area
	01 030	000	
	01 047	000	Scratch Pad Area
	01 050	001	Stores random
	01 051	120	number generator
	01 052	162	constant
	01 053	002	value
	01 054	XXX	Not Assigned
	01 055	XXX	Not Assigned
	01 056	XXX	Not Assigned
	01 057	XXX	Not Assigned

01 060	003	Stores random number generator constant value
01 061	150	
01 062	157	
01 063	014	
01 064	000	Scratch Pad Area
.	.	.
01 077	000	Scratch Pad Area
01 100	000	Sign Indicator
01 101	000	Sign Indicator
01 102	000	Bits Counter
01 103	000	Sign Indicator
01 104	000	Sign Indicator
01 105	000	Input Digit Counter
01 106	000	Temp Storage
01 107	000	Output Digit Counter
01 110	000	FP Mode Indicator
01 111	XXX	Not Assigned
.	.	.
01 117	XXX	Not Assigned
01 120	000	FPACC Extension
01 121	000	FPACC Extension
01 122	000	FPACC Extension
01 123	000	FPACC Extension
01 124	000	FPACC LSW
01 125	000	FPACC NSW
01 126	000	FPACC MSW
01 127	000	FPACC Exponent
01 130	000	FPOP Extension
01 131	000	FPOP Extension
01 132	000	FPOP Extension
01 133	000	FPOP Extension
01 134	000	FPOP LSW
01 135	000	FPOP NSW
01 136	000 ^f	FPOP MSW
01 137	000	FPOP Exponent
01 140	000	Floating point working area
.	.	.
01 167	000	Floating point working area
01 170	XXX	Not Assigned
.	.	.
01 177	XXX	Not Assigned
01 200	000	Temporary register
01 201	000	storage
01 202	000	area (D, E, H & L)
01 203	000	

01 204	XXX		Not Assigned
01 205	XXX		Not Assigned
01 206	XXX		Not Assigned
01 207	XXX		Not Assigned
01 210	000		Stores floating
01 211	000		point
01 212	120		constant
01 213	004		value +10.0
01 214	147		Stores floating
01 215	146		point
01 216	146		constant
01 217	375		value +0.1
01 220	000		GETINP Counter
01 221	XXX		Not Assigned
01 222	XXX		Not Assigned
01 223	XXX		Not Assigned
01 224	XXX		Not Assigned
01 225	XXX		Not Assigned
01 226	XXX		Not Assigned
01 227	000		Arithmetic Stack Pointer
01 230	000	004	Arithmetic Stack
01 272		1323	Arithmetic Stack
01 273			
01 274	301	S	
01 275	326	A	
01 276	305	V	
01 277	000 004	E	
01 300	000 314	(cc) FOR LOAD	Arithmetic Stack
01 301	000 317	L	FPACC
01 302	000 301	O	temporary
01 303	000 304	A	storage
01 304	000	D	location
01 305	000		STEP value
01 306	000		temporary
01 307	000		storage
01 310	000		location
01 311	000		FOR/NEXT Limit
01 312	000		temporary
01 313	000		storage
01 314	000		location
01 315	000		Array pointer
01 316	000		temporary
01 317	000		storage
			location

EE NOTE
PAGE A-5

Executive & special messages
look-up table and storage area.

01 320	004	(cc) for THEN
01 321	324	T
01 322	310	H
01 323	305	E
01 324	316	N

01 325 002 (cc) for TO
 01 326 324 T
 01 327 317 O
 01 330 004 (cc) for STEP
 01 331 323 S
 01 332 324 T
 01 333 305 E
 01 334 320 P
 01 335 004 (cc) for LIST
 01 336 314 L
 01 337 311 I
 01 340 323 S
 01 341 324 T
 01 342 003 (cc) for RUN
 01 343 322 R
 01 344 325 U
 01 345 316 N
 01 346 003 (cc) for SCR
 01 347 323 S
 01 350 303 C
 01 351 322 R
 01 352 013 (cc) for READY message
 01 353 224 Ctrl T
 01 354 215 Carriage-return
 01 355 212 Line-feed
 01 356 322 R
 01 357 305 E
 01 360 301 A
 01 361 304 D
 01 362 331 Y
 01 363 215 Carriage-return
 01 364 212 Line-feed
 01 365 212 Line-feed
 01 366 011 (cc) for AT LINE message
 01 367 240 Space
 01 370 301 A
 01 371 324 T
 01 372 240 Space
 01 373 314 L
 01 374 311 I
 01 375 316 N
 01 376 305 E
 01 377 240 Space

CHANGING "BELL" ASCH 207
 207

End of page 01.

01 363 104
 01 364 075
 01 365 212

02 000	106 255 002		SYNTAX,	CAL CLESYM
02 003	066 340			LLI 340
02 005	056 026	**		LHI 026
02 007	076 000			LMI 000
02 011	066 201			LLI 201
02 013	076 001			LMI 001
02 015	066 201		SYNTAX1,	LLI 201
02 017	106 240 002			CAL GETCHR
02 022	150 044 002			JTZ SYNTX2
02 025	074 260			CPI 260
02 027	160 061 002			JTS SYNTX3
02 032	074 272			CPI 272
02 034	120 061 002			JFS SYNTX3
02 037	066 340			LLI 340
02 041	106 314 002			CAL CONCT1
02 044	066 201		SYNTAX2,	LLI 201
02 046	106 003 003			CAL LOOP
02 051	110 015 002			JFZ SYNTX1
02 054	066 203			LLI 203
02 056	076 000			LMI 000
02 060	007			RET
02 061	066 201		SYNTAX3,	LLI 201
02 063	317			LBM
02 064	066 202			LLI 202
02 066	371			LMB
02 067	066 202		SYNTAX4,	LLI 202
02 071	106 240 002			CAL GETCHR
02 074	150 171 002			JTZ SYNTX6
02 077	074 275			CPI 275
02 101	150 210 002			JTZ SYNTX7
02 104	074 250			CPI 250
02 106	150 215 002			JTZ SYNTX8
02 111	106 310 002			CAL CONCTS
02 114	066 203			LLI 203
02 116	076 001			LMI 001
02 120	056 027	**		LHI 027
02 122	066 000			LLI 000
02 124	036 026	**	SYNTAX5,	LDI 026
02 126	046 120			LEI 120
02 130	106 332 002			CAL STRCP
02 133	053			RTZ
02 134	106 356 022			CAL SWITCH
02 137	060		SYNTAXL,	INL
02 140	307			LAM
02 141	044 300			NDI 300

02 143	110 137 002		JFZ SYNTXL
02 146	106 356 022		CAL SWITCH
02 151	066 203		LLI 203
02 153	056 026	**	LHI 026
02 155	317		LBM
02 156	010		INB
02 157	371		LMB
02 160	106 356 022		CAL SWITCH
02 163	301		LAB
02 164	074 015		CPI 015
02 166	110 124 002		JFZ SYNTX5
02 171	066 202		SYNTX6, LLI 202
02 173	056 026	**	LHI 026
02 175	106 003 003		CAL LOOP
02 200	110 067 002		JFZ SYNTX4
02 203	066 203		LLI 203
02 205	076 377		LMI 377
02 207	007		RET
02 210	066 203		SYNTX7, LLI 203
02 212	076 015		LMI 015
02 214	007		RET
02 215	066 203		SYNTX8, LLI 203
02 217	076 016		LMI 016
02 221	007		RET
02 222	006 302		BIGERR, LAI 302
02 224	026 307		LCI 307
02 226	106 202 003		ERROR, CAL ECHO
02 231	302		LAC
02 232	106 202 003		CAL ECHO
02 235	104 322 012		JMP FINERR
02 240	307		GETCHR, LAM
02 241	074 120		CPI 120
02 243	120 222 002		JFS BIGERR
02 246	360		LLA
02 247	056 026	**	LHI 026
02 251	307		LAM
02 252	074 240		CPI 240
02 254	007		RET
02 255	066 120		CLESYM, LLI 120
02 257	056 026	**	LHI 026
02 261	076 000		LMI 000
02 263	007		RET
02 264	074 301		CONCTA, CPI 301

02 266	160 276 002	JTS CONCTN
02 271	074 333	CPI 333
02 273	<u>160</u> 310 002	JTS CONCTS
02 276	074 260	CONCTN, CPI 260
02 300	160 327 002	JTS CONCTE
02 303	074 272	CPI 272
02 305	120 327 002	JFS CONCTE
02 310	066 120	CONCTS, LLI 120
02 312	056 026	** LHI 026
02 314	327	CONCT1, LCM
02 315	020	INC
02 316	372	LMC
02 317	310	LBA
02 320	106 036 023	CAL INDEXC
02 323	371	LMB
02 324	006 000	LAI 000
02 326	007	RET
02 327	104 152 011	CONCTE, JMP SYNERR
02 332	307	STRCP, LAM
02 333	106 356 022	CAL SWITCH
02 336	317	LBM
02 337	271	CPB
02 340	013	RFZ
02 341	106 356 022	CAL SWITCH
02 344	106 377 002	STRCPL, CAL ADV
02 347	307	LAM
02 350	106 356 022	CAL SWITCH
02 353	106 377 002	CAL ADV
02 356	277	STRCPE, CPM
02 357	013	RFZ
02 360	106 356 022	CAL SWITCH
02 363	011	DCB
02 364	110 344 002	JFZ STRCPL
02 367	007	RET
02 370	307	STRCPC, LAM
02 371	106 356 022	CAL SWITCH
02 374	104 356 002	JMP STRCPE
02 377	060	ADV, INL
03 000	013	RFZ
03 001	050	INH
03 002	007	RET

03 003	317	LOOP,	LBM
03 004	010		INB
03 005	371		LMB
03 006	066 000		LLI 000
03 010	307		LAM
03 011	011		DCB
03 012	271		CPB
03 013	007		RET
03 014	026 000	STRIN,	LCI 000
03 016	106 221 003	STRIN1,	CAL CINPUT
03 021	074 377		CPI 377
03 023	110 045 003		JFZ NOTDEL
03 026	006 334		LAI 334
03 030	106 202 003		CAL ECHO
03 033	021		DCC
03 034	160 014 003		JTS STRIN
03 037	106 164 003		CAL DEC
03 042	104 016 003		JMP STRIN1
03 045	074 203	NOTDEL,	CPI 203
03 047	150 313 012		JTZ CTRLC
03 052	074 215		CPI 215
03 054	150 102 003		JTZ STRINF
03 057	074 212		CPI 212
03 061	150 016 003		JTZ STRIN1
03 064	106 377 002		CAL ADV
03 067	020		INC
03 070	370		LMA
03 071	302		LAC
03 072	074 120		CPI 120
03 074	120 222 002		JFS BIGERR
03 077	104 016 003		JMP STRIN1
03 102	312	STRINF,	LBC
03 103	106 113 003		CAL SUBHL
03 106	372		LMC
03 107	106 141 003		CAL CRLF
03 112	007		RET
03 113	306	SUBHL,	LAL
03 114	221		SUB
03 115	360		LLA
03 116	003		RFC
03 117	051		DCH
03 120	007		RET
03 121	327	TEXTC,	LCM
03 122	307		LAM
03 123	240		NDA

03 124	053		RTZ
03 125	106 377 002	TEXTCL,	CAL ADV
03 130	307		LAM
03 131	106 202 003		CAL ECHO
03 134	021		DCC
03 135	110 125 003		JFZ TEXTCL
03 140	007		RET
03 141	006 215	CRLF,	LAI 215
03 143	106 202 003		CAL ECHO
03 146	006 212		LAI 212
03 150	106 202 003		CAL ECHO
03 153	066 043		LLI 043
03 155	056 001	**	LHI 001
03 157	076 001		LMI 001
03 161	353		LHD
03 162	364		LLE
03 163	007		RET
03 164	061	DEC,	DCL
03 165	060		INL
03 166	110 172 003		JFZ DECN0
03 171	051		DCH
03 172	061	DECN0,	DCL
03 173	007		RET
03 174	306	INDEXB,	LAL
03 175	201		ADB
03 176	360		LLA
03 177	003		RFC
03 200	050		INH
03 201	007		RET
03 202	335	ECHO,	LDH
03 203	346		LEL
03 204	066 043		LLI 043
03 206	056 001	**	LHI 001
03 210	317		LBM
03 211	010		INB
03 212	371 121 000		LMB 121 000
<i>TVT OUTPUT →</i>	03 213	106 ††† †††	CAL ††† †††
	03 216	353	LHD
	03 217	364	LLE
	03 220	007	RET
<i>KBD INPUT →</i>	03 221	104 100 000 ††† †††	CINPUT, JMP 100 000 ††† †††
	03 224	066 227	EVAL, LLI 227
	03 226	056 001	** LHI 001

03 230	076 224		LMI 224
03 232	060		INL
03 233	056 026	**	LHI 026
03 235	076 000		LMI 000
03 237	106 255 002		CAL CLESYM
03 242	066 210		LLI 210
03 244	076 000		LMI 000
03 246	066 276		LLI 276
03 250	317		LBM
03 251	066 200		LLI 200
03 253	371		LMB
03 254	066 200		SCAN1, LLI 200
03 256	106 240 002		CAL GETCHR
03 261	150 301 004		JTZ SCAN10
03 264	074 253		CPI 253
03 266	110 300 003		JFZ SCAN2
03 271	066 176		LLI 176
03 273	076 001		LMI 001
03 275	104 351 003		JMP SCANFN
03 300	074 255		SCAN2, CPI 255
03 302	110 357 003		JFZ SCAN4
03 305	066 120		LLI 120
03 307	307		LAM
03 310	240		NDA
03 311	110 345 003		JFZ SCAN3
03 314	066 176		LLI 176
03 316	307		LAM
03 317	074 007		CPI 007
03 321	150 345 003		JTZ SCAN3
03 324	074 003		CPI 003
03 326	150 152 011		JTZ SYNERR
03 331	074 005		CPI 005
03 333	150 152 011		JTZ SYNERR
03 336	066 120		LLI 120
03 340	076 001		LMI 001
03 342	060		INL
03 343	076 260		LMI 260
03 345	066 176		SCAN3, LLI 176
03 347	076 002		LMI 002
03 351	106 324 004		SCANFN, CAL PARSER
03 354	104 301 004		JMP SCAN10
03 357	074 252		SCAN4, CPI 252
03 361	110 373 003		JFZ SCAN5
03 364	066 176		LLI 176
03 366	076 003		LMI 003
03 370	104 351 003		JMP SCANFN

03 373	074 257	SCAN5, CPI 257
03 375	110 007 004	JFZ SCAN6
04 000	066 176	LLI 176
04 002	076 004	LMI 004
04 004	104 351 003	JMP SCANFN
5/16		
04 007	074 250	SCAN6, CPI 250
04 011	110 033 004	JFZ SCAN7
04 014	066 230	LLI 230
04 016	317	LBM
04 017	010	INB
04 020	371	LMB
04 021	106 100 007	CAL FUNARR
04 024	066 176	LLI 176
04 026	076 006	LMI 006
04 030	104 351 003	JMP SCANFN
04 033	074 251	SCAN7, CPI 251
04 035	110 064 004	JFZ SCAN8
04 040	066 176	LLI 176
04 042	076 007	LMI 007
04 044	106 324 004	CAL PARSER
04 047	106 003 007	CAL PRIGHT
04 052	066 230	LLI 230
04 054	056 026	LHI 026
04 056	317	LBM
04 057	011	DCB
04 060	371	LMB
04 061	104 301 004	JMP SCAN10
04 064	074 336	SCAN8, CPI 336
04 066	110 100 004	JFZ SCAN9
04 071	066 176	LLI 176
04 073	076 005	LMI 005
04 075	104 351 003	JMP SCANFN
04 100	074 274	SCAN9, CPI 274
04 102	110 143 004	JFZ SCAN11
04 105	066 200	LLI 200
04 107	317	LBM
04 110	010	INB
04 111	371	LMB
04 112	106 240 002	CAL GETCHR
04 115	074 275	CPI 275
04 117	150 251 004	JTZ SCAN13
04 122	074 276	CPI 276
04 124	150 267 004	JTZ SCAN15
04 127	066 200	LLI 200
04 131	317	LBM
04 132	011	DCB
04 133	371	LMB

04 134	066 176	LLI 176
04 136	076 011	LMI 011
04 140	104 351 003	JMP SCANFN
04 143	074 275	SCAN11, CPI 275
04 145	110 206 004	JFZ SCAN12
04 150	066 200	LLI 200
04 152	317	LBM
04 153	010	INB
04 154	371	LMB
04 155	106 240 002	CAL GETCHR
04 160	074 274	CPI 274
04 162	150 251 004	JTZ SCAN13
04 165	074 276	CPI 276
04 167	150 260 004	JTZ SCAN14
04 172	066 200	LLI 200
04 174	317	LBM
04 175	011	DCB
04 176	371	LMB
04 177	066 176	LLI 176
04 201	076 012	LMI 012
04 203	104 351 003	JMP SCANFN
04 206	074 276	SCAN12, CPI 276
04 210	110 276 004	JFZ SCAN16
04 213	066 200	LLI 200
04 215	317	LBM
04 216	010	INB
04 217	371	LMB
04 220	106 240 002	CAL GETCHR
04 223	074 274	CPI 274
04 225	150 267 004	JTZ SCAN15
04 230	074 275	CPI 275
04 232	150 260 004	JTZ SCAN14
04 235	066 200	LLI 200
04 237	317	LBM
04 240	011	DCB
04 241	371	LMB
04 242	066 176	LLI 176
04 244	076 013	LMI 013
04 246	104 351 003	JMP SCANFN
04 251	066 176	SCAN13, LLI 176
04 253	076 014	LMI 014
04 255	104 351 003	JMP SCANFN
04 260	066 176	SCAN14, LLI 176
04 262	076 015	LMI 015
04 264	104 351 003	JMP SCANFN
04 267	066 176	SCAN15, LLI 176

04 271	076 016	LMI 016
04 273	104 351 003	JMP SCANFN
04 276	106 310 002	SCAN16, CAL CONCTS
04 301	066 200	SCAN10, LLI 200
04 303	056 026	LHI 026
04 305	317	LBM
04 306	010	INB
04 307	371	LMB
04 310	066 277	LLI 277
04 312	307	LAM
04 313	011	DCB
04 314	271	CPB
04 315	110 254 003	JFZ SCAN1
04 320	104 300 031	JMP PARSEP
04 323	000	HLT
04 324	066 120	PARSER, LLI 120
04 326	056 026	** LHI 026
04 330	307	LAM
04 331	240	NDA
04 332	150 231 005	JTZ PARSE
04 335	060	INL
04 336	307	LAM
04 337	074 256	CPI 256
04 341	150 356 004	JTZ PARNUM
04 344	074 260	CPI 260
04 346	160 033 005	JTS LOOKUP
04 351	074 272	CPI 272
04 353	120 033 005	JFS LOOKUP
Was 03		
—> 04 356	061	PARNUM, DCL
04 357	307	LAM
04 360	074 001	CPI 001
04 362	150 005 005	JTZ NOEXPO
04 365	206	ADL
04 366	360	LLA
04 367	307	LAM
04 370	074 305	CPI 305
04 372	110 005 005	JFZ NOEXPO
04 375	066 200	LLI 200
04 377	106 240 002	CAL GETCHR
05 002	104 310 002	JMP CONCTS
05 005	066 227	NOEXPO, LLI 227
05 007	056 001	** LHI 001
05 011	307	LAM
05 012	004 004	ADI 004
05 014	370	LMA
05 015	360	LLA

05 016	106 255 022		CAL FSTORE
05 021	066 120		LLI 120
05 023	056 026	**	LHI 026
05 025	106 044 023		CAL DINPUT
05 030	104 231 005		JMP PARSE
05 033	066 370		LOOKUP, LLI 370
05 035	056 026	**	LHI 026
05 037	076 000		LMI 000
05 041	066 120		LLI 120
05 043	036 027	**	LDI 027
05 045	046 210		LEI 210
05 047	307		LAM
05 050	074 001		CPI 001
05 052	110 061 005		JFZ LOOKU1
05 055	066 122		LLI 122
05 057	076 000		LMI 000
05 061	066 121		LOOKU1, LLI 121
05 063	056 026	**	LHI 026
05 065	106 356 022		CAL SWITCH
05 070	307		LAM
05 071	060		INL
05 072	317		LBM
05 073	060		INL
05 074	106 356 022		CAL SWITCH
05 077	277		CPM
05 100	110 111 005		JFZ LOOKU2
05 103	060		INL
05 104	301		LAB
05 105	277		CPM
05 106	150 201 005		JTZ LOOKU4
05 111	106 256 006		LOOKU2, CAL AD4DE
05 114	066 370		LLI 370
05 116	056 026	**	LHI 026
05 120	317		LBM
05 121	010		INB
05 122	371		LMB
05 123	066 077		LLI 077
05 125	056 027	**	LHI 027
05 127	301		LAB
05 130	277		CPM
05 131	110 061 005		JFZ LOOKU1
05 134	066 077		LLI 077
05 136	056 027	**	LHI 027
05 140	317		LBM
05 141	010		INB
05 142	371		LMB
05 143	301		LAB
05 144	074 025		CPI 025

05 146	120 222 002	JFS BIGERR
05 151	066 121	LLI 121
05 153	056 026	LHI 026
05 155	016 002	LBI 002
05 157	106 013 021	CAL MOVEIT
05 162	364	LLE
05 163	353	LHD
05 164	250	XRA
05 165	370	LMA
05 166	060	INL
05 167	370	LMA
05 170	060	INL
05 171	370	LMA
05 172	060	INL
05 173	370	LMA
05 174	306	LAL
05 175	024 004	SUI 004
05 177	340	LEA
05 200	335	LDH
05 201	106 317 022	LOOKU4, CAL SAVEHL
05 204	066 227	LLI 227
05 206	056 001	** LHI 001
05 210	307	LAM
05 211	004 004	ADI 004
05 213	370	LMA
05 214	360	LLA
05 215	106 255 022	CAL FSTORE
05 220	106 337 022	CAL RESTHL
05 223	106 356 022	CAL SWITCH
05 226	106 244 022	CAL FLOAD
05 231	106 255 002	PARSE, CAL CLESYM
05 234	066 176	LLI 176
05 236	307	LAM
05 237	074 007	CPI 007
05 241	150 332 005	JTZ PARSE2
05 244	004 240	ADI 240
05 246	360	LLA
05 247	317	LBM
05 250	066 210	LLI 210
05 252	327	LCM
05 253	106 036 023	CAL INDEXC
05 256	307	LAM
05 257	004 257	ADI 257
05 261	360	LLA
05 262	301	LAB
05 263	277	CPM
05 264	150 307 005	JTZ PARSE1
05 267	160 307 005	JTS PARSE1
05 272	066 176	LLI 176

05 274	317		LBM
05 275	066 210		LLI 210
05 277	327		LCM
05 300	020		INC
05 301	372		LMC
05 302	106 036 023		CAL INDEXC
05 305	371		LMB
05 306	007		RET
05 307	066 210	PARSE1,	LLI 210
05 311	307		LAM
05 312	206		ADL
05 313	360		LLA
05 314	307		LAM
05 315	240		NDA
05 316	053		RTZ
05 317	066 210		LLI 210
05 321	327		LCM
05 322	021		DCC
05 323	372		LMC
05 324	106 364 005		CAL FPOPER
05 327	104 231 005		JMP PARSE
05 332	066 210	PARSE2,	LLI 210
05 334	056 026	**	LHI 026
05 336	307		LAM
05 337	206		ADL
05 340	360		LLA
05 341	307		LAM
05 342	240		NDA
05 343	150 104 006		JTZ PARNER
05 346	066 210		LLI 210
05 350	327		LCM
05 351	021		DCC
05 352	372		LMC
05 353	074 006		CPI 006
05 355	053		RTZ
05 356	106 364 005		CAL FPOPER
05 361	104 332 005		JMP PARSE2
05 364	066 371	FPOPER,	LLI 371
05 366	056 026	**	LHI 026
05 370	370		LMA
05 371	066 227		LLI 227
05 373	056 001	**	LHI 001
05 375	307		LAM
05 376	360		LLA
05 377	106 266 022		CAL OPLOAD
06 002	066 227		LLI 227
06 004	307		LAM
06 005	024 004		SUI 004

06 007	370		LMA
06 010	066 371		LLI 371
06 012	056 026	**	LHI 026
06 014	307		LAM
06 015	074 001		CPI 001
06 017	150 211 020		JTZ FPADD
06 022	074 002		CPI 002
06 024	150 032 021		JTZ FPSUB
06 027	074 003		CPI 003
06 031	150 046 021		JTZ FPMULT
06 034	074 004		CPI 004
06 036	150 322 021		JTZ FPDIV
06 041	074 005		CPI 005
06 043	150 263 006		JTZ INTEXP
06 046	074 011		CPI 011
06 050	150 121 006		JTZ LT
06 053	074 012		CPI 012
06 055	150 136 006		JTZ EQ
06 060	074 013		CPI 013
06 062	150 153 006		JTZ GT
06 065	074 014		CPI 014
06 067	150 173 006		JTZ LE
06 072	074 015		CPI 015
06 074	150 213 006		JTZ GE
06 077	074 016		CPI 016
06 101	150 230 006		JTZ NE
06 104	066 230		PARNER, LLI 230
06 106	056 026	**	LHI 026
06 110	076 000		LMI 000
06 112	006 311		LAI 311
06 114	026 250		LCI 250
06 116	104 226 002		JMP ERROR
06 121	106 032 021		LT, CAL FPSUB
06 124	066 126		LLI 126
06 126	307		LAM
06 127	240		NDA
06 130	160 242 006		JTS CTRUE
06 133	104 247 006		JMP CFALSE
06 136	106 032 021		EQ, CAL FPSUB
06 141	066 126		LLI 126
06 143	307		LAM
06 144	240		NDA
06 145	150 242 006		JTZ CTRUE
06 150	104 247 006		JMP CFALSE
06 153	106 032 021		GT, CAL FPSUB
06 156	066 126		LLI 126
06 160	307		LAM
06 161	240		NDA

06 162	150 247 006	JTZ CFALSE
06 165	120 242 006	JFS CTRUE
06 170	104 247 006	JMP CFALSE
06 173	106 032 021	LE, CAL FPSUB
06 176	066 126	LLI 126
06 200	307	LAM
06 201	240	NDA
06 202	150 242 006	JTZ CTRUE
06 205	160 242 006	JTS CTRUE
06 210	104 247 006	JMP CFALSE
06 213	106 032 021	GE, CAL FPSUB
06 216	066 126	LLI 126
06 220	307	LAM
06 221	240	NDA
06 222	120 242 006	JFS CTRUE
06 225	104 247 006	JMP CFALSE
06 230	106 032 021	NE, CAL FPSUB
06 233	066 126	LLI 126
06 235	307	LAM
06 236	240	NDA
06 237	150 247 006	JTZ CFALSE
06 242	066 004	CTRUE, FPONE, LLI 004
06 244	104 244 022	JMP FLOAD
06 247	066 127	CFALSE, LLI 127
06 251	076 000	LMI 000
06 253	104 051 020	JMP FPZERO
06 256	304	AD4DE, LAE
06 257	004 004	ADI 004
06 261	340	LEA
06 262	007	RET
06 263	066 126	INTEXP, LLI 126
06 265	056 001	LHI 001
06 267	307	LAM
06 270	066 003	LLI 003
06 272	370	LMA
06 273	240	NDA
06 274	150 242 006	JTZ FPONE
06 277	162 202 020	CTS FPCOMP
06 302	106 000 020	CAL FPFIX
06 305	066 124	LLI 124
06 307	317	LBM
06 310	066 013	LLI 013
06 312	371	LMB
06 313	066 134	LLI 134

06 315	046 014		LEI 014
06 317	056 001	**	LHI 001
06 321	335		LDH
06 322	016 004		LBI 004
06 324	106 013 021		CAL MOVEIT
06 327	106 242 006		CAL FPONE
06 332	066 003		LLI 003
06 334	307		LAM
06 335	240		NDA
06 336	160 362 006		JTS DVLOOP
06 341	066 014		MULLOOP, LLI 014
06 343	106 277 022		CAL FACXOP
06 346	106 046 021		CAL FPMULT
06 351	066 013		LLI 013
06 353	317		LBM
06 354	011		DCB
06 355	371		LMB
06 356	110 341 006		JFZ MULLOOP
06 361	007		RET
06 362	066 014		DVLOOP, LLI 014
06 364	106 277 022		CAL FACXOP
06 367	106 322 021		CAL FPDIV
06 372	066 013		LLI 013
06 374	317		LBM
06 375	011		DCB
06 376	371		LMB
06 377	110 362 006		JFZ DVLOOP
07 002	007		RET
07 003	066 230		PRIGHT, LLI 230
07 005	056 026	**	LHI 026
07 007	307		LAM
07 010	206		ADL
07 011	360		LLA
07 012	307		LAM
07 013	076 000		LMI 000
07 015	066 203		LLI 203
07 017	056 027	**	LHI 027
07 021	370		LMA
07 022	240		NDA
07 023	053		RTZ
07 024	160 000 055	@@	JTS PRIGH1
07 027	074 001		CPI 001
07 031	150 243 007		JTZ INTX
07 034	074 002		CPI 002
07 036	150 360 007		JTZ SGNX
07 041	074 003		CPI 003
07 043	150 346 007		JTZ ABSX
07 046	074 004		CPI 004

07 050	150 000 032		JTZ SQRX
07 053	074 005		CPI 005
07 055	150 017 010		JTZ TABX
07 060	074 006		CPI 006
07 062	150 240 032		JTZ RNDX
07 065	074 007		CPI 007
07 067	150 377 007		JTZ CHRX
07 072	074 010		CPI 010
07 074	150 ††† †††	†† USER DEFINED	JTZ UDEFX
07 077	000		HLT
07 100	066 120		FUNARR, LLI 120
07 102	056 026	**	LHI 026
07 104	307		LAM
07 105	240		NDA
07 106	053		RTZ
07 107	066 202		LLI 202
07 111	056 027	**	LHI 027
07 113	076 000		LMI 000
07 115	066 202		FUNAR1, LLI 202
07 117	056 027	**	LHI 027
07 121	317		LBM
07 122	010		INB
07 123	371		LMB
07 124	026 002		LCI 002
07 126	066 274		LLI 274
07 130	056 026	**	LHI 026
07 132	106 230 007		CAL TABADR
07 135	036 026	**	LDI 026
07 137	046 120		LEI 120
07 141	106 332 002		CAL STRCP
07 144	150 207 007		JTZ FUNAR4
07 147	066 202		LLI 202
07 151	056 027	**	LHI 027
07 153	307		LAM
07 154	074 010		CPI 010
07 156	110 115 007		JFZ FUNAR1
07 161	066 202		LLI 202
07 163	056 027	**	LHI 027
07 165	076 000		LMI 000
07 167	104 054 055	@@,	JMP FUNAR2
07 172	066 230		FAERR, LLI 230
07 174	056 026	**	LHI 026
07 176	076 000		LMI 000
07 200	006 306		LAI 306
07 202	026 301		LCI 301
07 204	104 226 002		JMP ERROR
07 207	066 202		FUNAR4, LLI 202

07 211	056 027	**	LHI 027
07 213	317		LBM
07 214	066 230		LLI 230
07 216	056 026	**	LHI 026
07 220	327		LCM
07 221	106 036 023		CAL INDEXC
07 224	371		LMB
07 225	104 255 002		JMP CLESYM
07 230	301		TABADR, LAB
07 231	002		TABAD1, RLC
07 232	021		DCC
07 233	110 231 007		JFZ TABAD1
07 236	206		ADL
07 237	360		LLA
07 240	003		RFC
07 241	050		INH
07 242	007		RET
07 243	066 126		INTX, LLI 126
07 245	056 001	**	LHI 001
07 247	307		LAM
07 250	240		NDA
07 251	120 327 007		JFS INT1
07 254	066 014		LLI 014
07 256	106 255 022		CAL FSTORE
07 261	106 000 020		CAL FPFIX
07 264	066 123		LLI 123
07 266	076 000		LMI 000
07 270	106 064 020		CAL FPFLT
07 273	066 014		LLI 014
07 275	106 266 022		CAL OPLOAD
07 300	106 032 021		CAL FPSUB
07 303	066 126		LLI 126
07 305	307		LAM
07 306	240		NDA
07 307	150 341 007		JTZ INT2
07 312	066 014		LLI 014
07 314	106 244 022		CAL FLOAD
07 317	066 024		LLI 024
07 321	106 277 022		CAL FACXOP
07 324	106 211 020		CAL FPADD
07 327	106 000 020		INT1, CAL FPFIX
07 332	066 123		LLI 123
07 334	076 000		LMI 000
07 336	104 064 020		JMP FPFLT
07 341	066 014		INT2, LLI 014
07 343	104 244 022		JMP FLOAD

07 346	066 126		ABSX,	LLI 126
07 350	056 001	**		LHI 001
07 352	307			LAM
07 353	240			NDA
07 354	160 202 020			JTS FPCOMP
07 357	007			RET
07 360	066 126		SGNX,	LLI 126
07 362	056 001	**		LHI 001
07 364	307			LAM
07 365	240			NDA
07 366	053			RTZ
07 367	120 242 006			JFS FPONE
07 372	066 024			LLI 024
07 374	104 244 022			JMP FLOAD
07 377	106 000 020		CHRX,	CAL FPFIX
10 002	066 124			LLI 124
10 004	307			LAM
10 005	106 202 003			CAL ECHO
10 010	066 177			LLI 177
10 012	056 026	**		LHI 026
10 014	076 377			LMI 377
10 016	007			RET
10 017	106 000 020		TABX,	CAL FPFIX
10 022	066 124		TAB1,	LLI 124
10 024	307			LAM
10 025	066 043			LLI 043
10 027	227			SUM
10 030	066 177			LLI 177
10 032	056 026	**		LHI 026
10 034	076 377			LMI 377
10 036	160 217 031			JTS BACKSP
10 041	053			RTZ
10 042	320		TABC,	LCA
10 043	006 240			LAI 240
10 045	106 202 003		TABLOP,	CAL ECHO
10 050	021			DCC
10 051	110 045 010			JFZ TABLOP
10 054	007			RET
10 055	066 201		STOSYM,	LLI 201
10 057	056 027	**		LHI 027
10 061	307			LAM
10 062	240			NDA
10 063	150 100 010			JTZ STOSY1
10 066	076 000			LMI 000
10 070	066 204			LLI 204
10 072	367			LLM

✓ C:\USERS\HP\1073
 WHEN USING ISX

10 073	056 057	††	LHI 057
10 075	104 255 022		JMP FSTORE
10 100	066 370		STOSY1, LLI 370
10 102	056 026	**	LHI 026
10 104	076 000		LMI 000
10 106	066 120		LLI 120
10 110	036 027	**	LDI 027
10 112	046 210		LEI 210
10 114	307		LAM
10 115	074 001		CPI 001
10 117	110 126 010		JFZ STOSY2
10 122	066 122		LLI 122
10 124	076 000		LMI 000
10 126	066 121		STOSY2, LLI 121
10 130	056 026	**	LHI 026
10 132	106 356 022		CAL SWITCH
10 135	307		LAM
10 136	060		INL
10 137	317		LBM
10 140	060		INL
10 141	106 356 022		CAL SWITCH
10 144	277		CPM
10 145	110 156 010		JFZ STOSY3
10 150	060		INL
10 151	301		LAB
10 152	277		CPM
10 153	150 227 010		JTZ STOSY5
10 156	106 256 006		STOSY3, CAL AD4DE
10 161	066 370		LLI 370
10 163	056 026	**	LHI 026
10 165	317		LBM
10 166	010		INB
10 167	371		LMB
10 170	066 077		LLI 077
10 172	056 027	**	LHI 027
10 174	301		LAB
10 175	277		CPM
10 176	110 126 010		JFZ STOSY2
10 201	066 077		LLI 077
10 203	056 027	**	LHI 027
10 205	317		LBM
10 206	010		INB
10 207	371		LMB
10 210	301		LAB
10 211	074 025		CPI 025
10 213	120 222 002		JFS BIGERR
10 216	066 121		LLI 121
10 220	056 026	**	LHI 026

10 222	016 002		LBI 002
10 224	106 013 021		CAL MOVEIT
10 227	106 356 022		STOSY5, CAL SWITCH
10 232	106 255 022		CAL FSTORE
10 235	104 255 002		JMP CLESYM
10 240	066 120		SAVESY, LLI 120
10 242	056 026	**	LHI 026
10 244	335		LDH
10 245	046 144		LEI 144
10 247	104 261 010		JMP MOVECP
10 252	066 144		RESTSY, LLI 144
10 254	056 026	**	LHI 026
10 256	335		LDH
10 257	046 120		LEI 120
10 261	317		MOVECP, LBM
10 262	010		INB
10 263	104 013 021		JMP MOVEIT
START! →	10 266	066 352	EXEC, LLI 352
	10 270	056 001	LHI 001
	10 272	106 121 003	CAL TEXTC
	10 275	066 000	EXEC1, LLI 000
	10 277	056 026	LHI 026
	10 301	106 014 003	CAL STRIN
	10 304	307	LAM
	10 305	240	NDA
	10 306	150 275 010	JTZ EXEC1
	10 311	066 335	LLI 335
	10 313	056 001	LHI 001
	10 315	036 026	LDI 026
	10 317	046 000	LEI 000
	10 321	106 332 002	CAL STRCP
	10 324	110 354 010	JFZ NOLIST
	10 327	066 000	LLI 000
	10 331	056 033	LHI 033
	10 333	307	LIST, LAM
	10 334	240	NDA
	10 335	150 266 010	JTZ EXEC
	10 340	106 121 003	CAL TEXTC
	10 343	106 377 002	CAL ADV
	10 346	106 141 003	CAL CRLF
	10 351	104 333 010	JMP LIST
	10 354	066 342	NOLIST, LLI 342
	10 356	056 001	** LHI 001

10 360	046 000		LEI 000
10 362	036 026	**	LDI 026
10 364	046 000		LEI 000
10 366	106 332 002		CAL STRCP
10 371	150 070 013		JTZ RUN
10 374	036 026	**	LDI 026
10 376	046 000		LEI 000
11 000	066 346		LLI 346
11 002	056 001	**	LHI 001
11 004	106 332 002		CAL STRCP
11 007	110 071 011		JFZ NOSCR
11 012	056 026	**	LHI 026
11 014	066 364		LLI 364
11 016	076 033	††	LMI 033
11 020	060		INL
11 021	076 000		LMI 000
11 023	066 077		LLI 077
11 025	056 027	**	LHI 027
11 027	076 001		LMI 001
11 031	066 075		LLI 075
11 033	076 000	@@.	LMI 000
11 035	066 120	@@.	LLI 120
11 037	076 000	@@.	LMI 000
11 041	066 210		LLI 210
11 043	076 000		LMI 000
11 045	060		INL
11 046	076 000		LMI 000
11 050	056 033	††	LHI 033
11 052	066 000		LLI 000
11 054	076 000		LMI 000
11 056	056 057	@@.	LHI 057
11 060	076 000	@@.	SCRLOP, LMI 000
11 062	060	@@.	INL
11 063	110 060 011	@@.	JFZ SCRLOP
11 066	104 266 010		JMP EXEC
11 071	046 272		NOSCR, LEI 272
11 073	036 001	**	LDI 001
11 075	056 026	**	LHI 026
11 077	066 000		LLI 000
11 101	106 332 002		CAL STRCP
11 104	150 ††† †††	††	JTZ SAVE
11 107	066 277		LLI 277
11 111	056 001	**	LHI 001
11 113	036 026	**	LDI 026
11 115	046 000		LEI 000
11 117	106 332 002		CAL STRCP
11 122	150 ††† †††	††	JTZ LOAD
11 125	066 360		LLI 360
11 127	056 026	**	LHI 026

11 131	076 033	††	LMI 033
11 133	060		INL
11 134	076 000		LMI 000
11 136	106 000 002		CAL SYNTAX
11 141	066 203		LLI 203
11 143	056 026	**	LHI 026
11 145	307		LAM
11 146	240		NDA
11 147	120 161 011		JFS SYNTOK
11 152	006 323		SYNERR, LAI 323
11 154	026 331		LCI 331
11 156	104 226 002		JMP ERROR
11 161	066 340		SYNTOK, LLI 340
11 163	307		LAM
11 164	240		NDA
11 165	150 211 013		JTZ DIRECT
11 170	066 360		LLI 360
11 172	076 033	††	LMI 033
11 174	060		INL
11 175	076 000		LMI 000
11 177	066 201		GETAUX, LLI 201
11 201	056 026	**	LHI 026
11 203	076 001		LMI 001
11 205	066 350		LLI 350
11 207	076 000		LMI 000
11 211	066 201		GETAU0, LLI 201
11 213	106 123 012		CAL GETCHP
11 216	150 242 011		JTZ GETAU1
11 221	074 260		CPI 260
11 223	160 267 011		JTS GETAU2
11 226	074 272		CPI 272
11 230	120 267 011		JFS GETAU2
11 233	066 350		LLI 350
11 235	056 026	**	LHI 026
11 237	106 314 002		CAL CONCT1
11 242	066 201		GETAU1, LLI 201
11 244	056 026	**	LHI 026
11 246	317		LBM
11 247	010		INB
11 250	371		LMB
11 251	066 360		LLI 360
11 253	056 026	**	LHI 026
11 255	327		LCM
11 256	060		INL
11 257	367		LLM
11 260	352		LHC

11 261	307		LAM
11 262	011		DCB
11 263	271		CPB
11 264	110 211 011		JFZ GETAU0
11 267	066 360		GETAU2, LLI 360
11 271	056 026	**	LHI 026
11 273	337		LDM
11 274	060		INL
11 275	367		LLM
11 276	353		LHD
11 277	307		LAM
11 300	240		NDA
11 301	110 336 011		JFZ NOTEND
11 304	104 005 012		JMP NOSAME

Note open addresses.
This space available
for patching.

11 336	066 350		NOTEND, LLI 350
11 340	056 026	**	LHI 026
11 342	036 026	**	LDI 026
11 344	046 340		LEI 340
11 346	106 332 002		CAL STRCP
11 351	160 073 012		JTS CONTIN
11 354	110 005 012		JFZ NOSAME
11 357	066 360		LLI 360
11 361	056 026	**	LHI 026
11 363	327		LCM
11 364	060		INL
11 365	367		LLM
11 366	352		LHC
11 367	317		LBM
11 370	010		INB
11 371	106 144 012		CAL REMOVE
11 374	066 203		LLI 203
11 376	056 026	**	LHI 026
12 000	307		LAM
12 001	240		NDA
12 002	150 266 010		JTZ EXEC
12 005	066 360		NOSAME, LLI 360
12 007	056 026	**	LHI 026
12 011	337		LDM
12 012	060		INL
12 013	347		LEM
12 014	066 000		LLI 000
12 016	056 026	**	LHI 026
12 020	317		LBM
12 021	010		INB

12 022	106 205 012		CAL INSERT
12 025	066 360		LLI 360
12 027	056 026	**	LHI 026
12 031	337		LDM
12 032	060		INL
12 033	347		LEM
12 034	066 000		LLI 000
12 036	056 026	**	LHI 026
12 040	106 046 012		CAL MOVEC
12 043	104 275 010		JMP EXEC1
<i>WIS</i> 312 602	<i>317</i>		
12 046	317		MOVEC, LBM
12 047	010		INB
<i>K#3</i> <i>SK 2</i>			
12 050	307		MOVEPG, LAM
12 051	106 377 002		CAL ADV
12 054	106 356 022		CAL SWITCH
12 057	370		LMA
12 060	106 377 002		CAL ADV
12 063	106 356 022		CAL SWITCH
12 066	011		DCB
12 067	110 050 012		JFZ MOVEPG
12 072	007		RET
<i>REMARKS</i> <i>1/1/84</i>			
12 073	066 360		CONTIN, LLI 360
12 075	056 026	**	LHI 026
12 077	337		LDM
12 100	060		INL
12 101	347		LEM
12 102	353		LHD
12 103	364		LLE
12 104	317		LBM
12 105	010		INB
12 106	106 305 012		CAL ADBDE
12 111	066 360		LLI 360
12 113	056 026	**	LHI 026
12 115	373		LMD
12 116	060		INL
12 117	374		LME
12 120	104 177 011		JMP GETAUX
<i>1/1/84</i>			
12 123	056 026	**	GETCHP, LHI 026
12 125	317		LBM
12 126	066 360		LLI 360
12 130	337		LDM
12 131	060		INL
12 132	347		LEM
12 133	106 305 012		CAL ADBDE
12 136	353		LHD
12 137	364		LLE
12 140	307		LAM

12 141	074 240		CPI 240
12 143	007		RET
12 144	106 174 003		REMOVE, CAL INDEXB
12 147	327		LCM
12 150	106 113 003		CAL SUBHL
12 153	372		LMC
12 154	302		LAC
12 155	240		NDA
12 156	150 167 012		JTZ REMOV1
12 161	106 377 002		CAL ADV
12 164	104 144 012		JMP REMOVE
12 167	066 364		REMOV1, LLI 364
12 171	056 026	**	LHI 026
12 173	337		LDM
12 174	060		INL
12 175	307		LAM
12 176	221		SUB
12 177	370		LMA
12 200	003		RFC
12 201	061		DCL
12 202	031		DCD
12 203	373		LMD
12 204	007		RET
12 205	066 364		INSERT, LLI 364
12 207	056 026	**	LHI 026
12 211	307		LAM
12 212	060		INL
12 213	367		LLM
12 214	350		LHA
12 215	106 174 003		CAL INDEXB
12 220	305		LAH
12 221	074 054	††	CPI 054
12 223	120 222 002		JFS BIGERR
12 226	106 113 003		CAL SUBHL
12 231	327		INSER1, LCM
12 232	106 174 003		CAL INDEXB
12 235	372		LMC
12 236	106 113 003		CAL SUBHL
12 241	106 277 012		CAL CPHLDE
12 244	150 255 012		JTZ INSER3
12 247	106 164 003		CAL DEC
12 252	104 231 012		JMP INSER1
12 255	066 000		INSER3, INCLIN, LLI 000
12 257	056 026	**	LHI 026
12 261	317		LBM
12 262	010		INB

12 263	066 364		LLI 364
12 265	337		LDM
12 266	060		INL
12 267	347		LEM
12 270	106 305 012		CAL ADBDE
12 273	374		LME
12 274	061		DCL
12 275	373		LMD
12 276	007		RET
12 277	305		CPHLDE, LAH
12 300	273		CPD
12 301	013		RFZ
12 302	306		LAL
12 303	274		CPE
12 304	007		RET
12 305	304		ADBDE, LAE
12 306	201		ADB
12 307	340		LEA
12 310	003		RFC
12 311	030		IND
12 312	007		RET
12 313	006 336		CTRLC, LAI 336
12 315	026 303		LCI 303
12 317	104 226 002		JMP ERROR
12 322	066 340		FINERR, LLI 340
12 324	056 026	**	LHI 026
12 326	307		LAM
12 327	240		NDA
12 330	150 351 012		JTZ FINER1
12 333	066 366		LLI 366
12 335	056 001	**	LHI 001
12 337	106 121 003		CAL TEXTC
12 342	066 340		LLI 340
12 344	056 026	**	LHI 026
12 346	106 121 003		CAL TEXTC
12 351	106 141 003		FINER1, CAL CRLF
12 354	104 266 010		JMP EXEC
12 357	006 304		DVERR, LAI 304
12 361	026 332		LCI 332
12 363	104 226 002		JMP ERROR
12 366	006 306		FIXERR, LAI 306
12 370	026 330		LCI 330
12 372	104 226 002		JMP ERROR

12 375	006 311		NUMERR,	LAI 311
12 377	026 316			LCI 316
13 001	066 220			LLI 220
13 003	056 001	**		LHI 001
13 005	076 000			LMI 000
13 007	104 226 002			JMP ERROR
13 012	036 026	**	INSTR,	LDI 026
13 014	046 000			LEI 000
13 016	106 064 013		INSTR1,	CAL ADVDE
13 021	106 317 022			CAL SAVEHL
13 024	317			LBM
13 025	106 377 002			CAL ADV
13 030	106 370 002			CAL STRCPC
13 033	150 337 022			JTZ RESTHL
13 036	106 337 022			CAL RESTHL
13 041	066 000			LLI 000
13 043	056 026	**		LHI 026
13 045	307			LAM
13 046	274			CPE
13 047	150 061 013			JTZ INSTR2
13 052	106 337 022			CAL RESTHL
13 055	104 016 013			JMP INSTR1
13 060	000			HLT
13 061	046 000		INSTR2,	LEI 000
13 063	007			RET
13 064	040		ADVDE,	INE
13 065	013			RFZ
13 066	030			IND
13 067	007			RET
13 070	066 073		RUN,	LLI 073
13 072	056 027	**		LHI 027
13 074	076 000			LMI 000
13 076	066 205			LLI 205
13 100	076 000			LMI 000
13 102	066 360			LLI 360
13 104	056 026	**		LHI 026
13 106	076 033	††		LMI 033
13 110	060			INL
13 111	076 000			LMI 000
13 113	104 156 013			JMP SAMLIN
13 116	066 360		NXTLIN,	LLI 360
13 120	056 026	**		LHI 026
13 122	337			LDM
13 123	060			INL
13 124	347			LEM

13 125	353		LHD
13 126	364		LLE
13 127	317		LBM
13 130	010		INB
13 131	106 305 012		CAL ADBDE
13 134	066 360		LLI 360
13 136	056 026	**	LHI 026
13 140	373		LMD
13 141	060		INL
13 142	374		LME
13 143	066 340		LLI 340
13 145	056 026	**	LHI 026
13 147	307		LAM
13 150	240		NDA
13 151	150 266 010		JTZ EXEC
13 154	300		LAA
13 155	300		LAA
13 156	066 360		SAMLIN, LLI 360
13 160	056 026	**	LHI 026
13 162	327		LCM
13 163	060		INL
13 164	367		LLM
13 165	352		LHC
13 166	036 026	**	LDI 026
13 170	046 000		LEI 000
13 172	106 046 012		CAL MOVEC
13 175	066 000		LLI 000
13 177	056 026	**	LHI 026
13 201	307		LAM
13 202	240		NDA
13 203	150 266 010		JTZ EXEC
13 206	106 000 002		CAL SYNTAX
13 211	066 203		DIRECT, LLI 203
13 213	056 026	**	LHI 026
13 215	307		LAM
13 216	074 001		CPI 001
13 220	150 116 013		JTZ NXTLIN
13 223	074 002		CPI 002
13 225	150 027 016		JTZ IF
13 230	074 003		CPI 003
13 232	150 031 015		JTZ LET
13 235	074 004		CPI 004
13 237	150 174 015		JTZ GOTO
13 242	074 005		CPI 005
13 244	150 345 013		JTZ PRINT
13 247	074 006		CPI 006
13 251	150 365 016		JTZ INPUT
13 254	074 007		CPI 007
13 256	150 164 017		JTZ FOR

13 261	074 010	CPI 010
13 263	150 013 030	JTZ NEXT
13 266	074 011	CPI 011
13 270	150 236 016	JTZ GOSUB
13 273	074 012	CPI 012
13 275	150 304 016	JTZ RETURN
13 300	074 013	CPI 013
13 302	150 365 055	JTZ DIM
13 305	074 014	CPI 014
13 307	150 266 010	JTZ EXEC
13 312	074 015	CPI 015
13 314	150 013 015	JTZ LETO
13 317	074 016	CPI 016
13 321	110 152 011	JFZ SYNERR
13 324	106 153 055	CAL ARRAY1
13 327	066 206	LLI 206
13 331	056 026	LHI 026
13 333	317	LBM
13 334	066 202	LLI 202
13 336	371	LMB
13 337	106 240 010	CAL SAVESY
13 342	104 042 015	JMP LET1
13 345	066 202	PRINT, LLI 202
13 347	056 026	LHI 026
13 351	307	LAM
13 352	066 000	LLI 000
13 354	277	CPM
13 355	160 366 013	JTS PRINT1
13 360	106 141 003	CAL CRLF
13 363	104 116 013	JMP NXTLIN
13 366	106 255 002	PRINT1, CAL CLESYM
13 371	066 202	LLI 202
13 373	056 026	LHI 026
13 375	317	LBM
13 376	010	INB
13 377	066 203	LLI 203
14 001	371	LMB
14 002	066 203	PRINT2, LLI 203
14 004	106 240 002	CAL GETCHR
14 007	074 247	CPI 247
14 011	150 203 014	JTZ QUOTE
14 014	074 242	CPI 242
14 016	150 203 014	JTZ QUOTE
14 021	074 254	CPI 254
14 023	150 043 014	JTZ PRINT3
14 026	074 273	CPI 273
14 030	150 043 014	JTZ PRINT3
14 033	066 203	LLI 203

VOL 76
 11/00
 PM

14 035	106 003 003	CAL LOOP
14 040	110 002 014	JFZ PRINT2
14 043	066 202	PRINT3, LLI 202
14 045	317	LBM
14 046	010	INB
14 047	066 276	LLI 276
14 051	371	LMB
14 052	066 203	LLI 203
14 054	317	LBM
14 055	011	DCB
14 056	066 277	LLI 277
14 060	371	LMB
14 061	066 367	LLI 367
14 063	307	LAM
14 064	240	NDA
14 065	150 075 014	JTZ PRINT4
14 070	076 000	LMI 000
14 072	104 125 014	JMP PRINT6
14 075	106 224 003	PRINT4, CAL EVAL
14 100	066 177	LLI 177
14 102	056 026	** LHI 026
14 104	307	LAM
14 105	240	NDA
14 106	066 110	LLI 110
14 110	056 001	** LHI 001
14 112	076 377	LMI 377
14 114	152 314 014	PRINT5, CTZ PFPOUT
14 117	066 177	LLI 177
14 121	056 026	** LHI 026
14 123	076 000	LMI 000
14 125	066 203	PRINT6, LLI 203
14 127	106 240 002	CAL GETCHR
14 132	074 254	CPI 254
14 134	152 357 014	CTZ PCOMMA
14 137	066 203	LLI 203
14 141	056 026	** LHI 026
14 143	317	LBM
14 144	066 202	LLI 202
14 146	371	LMB
14 147	066 000	LLI 000
14 151	301	LAB
14 152	277	CPM
14 153	160 366 013	JTS PRINT1
14 156	066 000	LLI 000
14 160	106 240 002	CAL GETCHR
14 163	074 254	CPI 254
14 165	150 116 013	JTZ NXTLIN

14 170	074 273	CPI 273
14 172	150 116 013	JTZ NXTLIN
14 175	106 141 003	CAL CRLF
14 200	104 116 013	JMP NXTLIN
14 203	066 367	QUOTE, LLI 367
14 205	370	LMA
14 206	106 255 002	CAL CLESYM
14 211	066 203	LLI 203
14 213	317	LBM
14 214	010	INB
14 215	066 204	LLI 204
14 217	371	LMB
14 220	066 204	QUOTE1, LLI 204
14 222	106 240 002	CAL GETCHR
14 225	066 367	LLI 367
14 227	277	CPM
14 230	150 263 014	JTZ QUOTE2
14 233	106 202 003	CAL ECHO
14 236	066 204	LLI 204
14 240	106 003 003	CAL LOOP
14 243	110 220 014	JFZ QUOTE1
14 246	006 311	QUOTER, LAI 311
14 250	026 321	LCI 321
14 252	066 367	LLI 367
14 254	056 026	LHI 026
14 256	076 000	LMI 000
14 260	104 226 002	JMP ERROR
14 263	066 204	QUOTE2, LLI 204
14 265	317	LBM
14 266	066 202	LLI 202
14 270	371	LMB
14 271	301	LAB
14 272	066 000	LLI 000
14 274	277	CPM
14 275	110 366 013	JFZ PRINT1
14 300	106 141 003	CAL CRLF
14 303	066 367	LLI 367
14 305	056 026	LHI 026
14 307	076 000	LMI 000
14 311	104 116 013	JMP NXTLIN
14 314	066 126	PFPOUT, LLI 126
14 316	056 001	LHI 001
14 320	307	LAM
14 321	240	NDA
14 322	150 336 014	JTZ ZERO
14 325	060	INL

11/26 10
11/26 05
11/26 10

14 326	307	LAM
14 327	240	NDA
14 330	150 350 014	JTZ FRAC
14 333	104 165 024	JMP FPOUT
14 336	006 240	ZERO, LAI 240
14 340	106 202 003	CAL ECHO
14 343	006 260	LAI 260
14 345	104 202 003	JMP ECHO
14 350	066 110	FRAC, LLI 110
14 352	076 000	LMI 000
14 354	104 165 024	JMP FPOUT
14 357	066 000	PCOMMA, LLI 000
14 361	307	LAM
14 362	066 203	LLI 203
14 364	227	SUM
14 365	063	RTS
14 366	066 043	LLI 043
14 370	056 001	** LHI 001
14 372	307	LAM
14 373	044 360	NDI 360
14 375	004 020	ADI 020
14 377	227	SUM
15 000	320	LCA
15 001	006 240	LAI 240
15 003	106 202 003	PCOM1, CAL ECHO
15 006	021	DCC
15 007	110 003 015	JFZ PCOM1
15 012	007	RET
15 013	106 240 010	LETO, CAL SAVSYM
15 016	066 202	LLI 202
15 020	056 026	** LHI 026
15 022	317	LBM
15 023	066 203	LLI 203
15 025	371	LMB
15 026	104 141 015	JMP LET5
15 031	106 255 002	LET, CAL CLESYM
15 034	066 144	LLI 144
15 036	056 026	** LHI 026
15 040	076 000	LMI 000
15 042	066 202	LET1, LLI 202
15 044	056 026	** LHI 026
15 046	317	LBM
15 047	010	INB
15 050	066 203	LLI 203

15 052	371	LMB
15 053	066 203	LET2, LLI 203
15 055	106 240 002	CAL GETCHR
15 060	150 122 015	JTZ LET4
15 063	074 275	CPI 275
15 065	150 141 015	JTZ LET5
15 070	074 250	CPI 250
15 072	110 113 015	JFZ LET3
15 075	106 145 056 ⁷¹	CAL ARRAY
15 100	066 206	LLI 206
15 102	056 026	LHI 026
15 104	317	LBM
15 105	066 203	LLI 203
15 107	371	LMB
15 110	104 122 015	JMP LET4
15 113	066 144	LET3, LLI 144
15 115	056 026	LHI 026
15 117	106 314 002	CAL CONCT1
15 122	066 203	LET4, LLI 203
15 124	106 003 003	CAL LOOP
15 127	110 053 015	JFZ LET2
15 132	006 314	LETERR, LAI 314
15 134	026 305	LCI 305
15 136	104 226 002	JMP ERROR
15 141	066 203	LET5, LLI 203
15 143	056 026	LHI 026
15 145	317	LBM
15 146	010	INB
15 147	066 276	LLI 276
15 151	371	LMB
15 152	066 000	LLI 000
15 154	317	LBM
15 155	066 277	LLI 277
15 157	371	LMB
15 160	106 224 003	CAL EVAL
15 163	106 252 010	CAL RESTSY
15 166	106 055 010	CAL STOSYM
15 171	104 116 013	JMP NXTLIN
15 174	066 350	GOTO, LLI 350
15 176	056 026	LHI 026
15 200	076 000	LMI 000
15 202	066 202	LLI 202
15 204	317	LBM
15 205	010	INB
15 206	066 203	LLI 203

15 210	371	LMB
15 211	066 203	GOTO1, LLI 203
15 213	106 240 002	CAL GETCHR
15 216	150 240 015	JTZ GOTO2
15 221	074 260	CPI 260
15 223	160 250 015	JTS GOTO3
15 226	074 272	CPI 272
15 230	120 250 015	JFS GOTO3
15 233	066 350	LLI 350
15 235	106 314 002	CAL CONCT1
15 240	066 203	GOTO2, LLI 203
15 242	106 003 003	CAL LOOP
15 245	110 211 015	JFZ GOTO1
15 250	066 360	GOTO3, LLI 360
15 252	056 026	** LHI 026
15 254	076 033	†† LMI 033
15 256	060	INL
15 257	076 000	LMI 000
15 261	106 255 002	GOTO4, CAL CLESYM
15 264	066 204	LLI 204
15 266	076 001	LMI 001
15 270	066 204	GOTO5, LLI 204
15 272	106 123 012	CAL GETCHP
15 275	150 315 015	JTZ GOTO6
15 300	074 260	CPI 260
15 302	160 340 015	JTS GOTO7
15 305	074 272	CPI 272
15 307	120 340 015	JFS GOTO7
15 312	106 310 002	CAL CONCTS
15 315	066 204	GOTO6, LLI 204
15 317	056 026	** LHI 026
15 321	317	LBM
15 322	010	INB
15 323	371	LMB
15 324	066 360	LLI 360
15 326	327	LCM
15 327	060	INL
15 330	367	LLM
15 331	352	LHC
15 332	307	LAM
15 333	011	DCB
15 334	271	CPB
15 335	110 270 015	JFZ GOTO5
15 340	066 120	GOTO7, LLI 120

15 342	056 026	**	LHI 026
15 344	036 026	**	LDI 026
15 346	046 350		LEI 350
15 350	106 332 002		CAL STRCP
15 353	150 156 013		JTZ SAMLIN
15 356	066 360		LLI 360
15 360	056 026	**	LHI 026
15 362	337		LDM
15 363	060		INL
15 364	347		LEM
15 365	353		LHD
15 366	364		LLE
15 367	317		LBM
15 370	010		INB
15 371	106 305 012		CAL ADBDE
15 374	066 360		LLI 360
15 376	056 026	**	LHI 026
16 000	373		LMD
16 001	060		INL
16 002	374		LME
16 003	066 364		LLI 364
16 005	303		LAD
16 006	277		CPM
16 007	110 261 015		JFZ GOTO4
16 012	060		INL
16 013	304		LAE
16 014	277		CPM
16 015	110 261 015		JFZ GOTO4
16 020	006 325		GOTOER, LAI 325
16 022	026 316		LCI 316
16 024	104 226 002		JMP ERROR
16 027	066 202		IF, LLI 202
16 031	056 026	**	LHI 026
16 033	317		LBM
16 034	010		INB
16 035	066 276		LLI 276
16 037	371		LMB
16 040	106 255 002		CAL CLESYM
16 043	066 320		LLI 320
16 045	056 001	**	LHI 001
16 047	106 012 013		CAL INSTR
16 052	304		LAE
16 053	240		NDA
16 054	110 102 016		JFZ IF1
16 057	066 013		LLI 013
16 061	056 027	**	LHI 027
16 063	106 012 013		CAL INSTR
16 066	304		LAE
16 067	240		NDA

16 070	110 102 016	JFZ IF1
16 073	006 311	IFERR, LAI 311
16 075	026 306	LCI 306
16 077	104 226 002	JMP ERROR
16 102	066 277	IF1, LLI 277
16 104	056 026	** LHI 026
16 106	041	DCE
16 107	374	LME
16 110	106 224 003	CAL EVAL
16 113	066 126	LLI 126
16 115	056 001	** LHI 001
16 117	307	LAM
16 120	240	NDA
16 121	150 116 013	JTZ NXTLIN
16 124	066 277	LLI 277
16 126	056 026	** LHI 026
16 130	307	LAM
16 131	004 005	ADI 005
16 133	066 202	LLI 202
16 135	370	LMA
16 136	310	LBA
16 137	010	INB
16 140	066 204	LLI 204
16 142	371	LMB
16 143	066 204	IF2, LLI 204
16 145	106 240 002	CAL GETCHR
16 150	110 166 016	JFZ IF3
16 153	066 204	LLI 204
16 155	106 003 003	CAL LOOP
16 160	110 143 016	JFZ IF2
16 163	104 073 016	JMP IFERR
16 166	074 260	IF3, CPI 260
16 170	160 200 016	JTS IF4
16 173	074 272	CPI 272
16 175	160 174 015	JTS GOTO
16 200	066 000	IF4, LLI 000
16 202	307	LAM
16 203	066 204	LLI 204
16 205	227	SUM
16 206	310	LBA
16 207	010	INB
16 210	327	LCM
16 211	066 000	LLI 000
16 213	371	LMB
16 214	362	LLC
16 215	036 026	** LDI 026

16 217	046 001		LEI 001
16 221	106 013 021		CAL MOVEIT
16 224	066 202		LLI 202
16 226	076 001		LMI 001
16 230	106 067 002		CAL SYNTAX4
16 233	104 211 013		JMP DIRECT
16 236	066 340		GOSUB, LLI 340
16 240	056 026	**	LHI 026
16 242	337		LDM
16 243	030		IND
16 244	031		DCD
16 245	150 255 016		JTZ GOSUB1
16 250	066 360		LLI 360
16 252	337		LDM
16 253	060		INL
16 254	347		LEM
16 255	066 073		GOSUB1, LLI 073
16 257	056 027	**	LHI 027
16 261	307		LAM
16 262	004 002		ADI 002
16 264	074 021		CPI 021
16 266	120 347 016		JFS GOSERR
16 271	370		LMA
16 272	066 076		LLI 076
16 274	206		ADL
16 275	360		LLA
16 276	373		LMD
16 277	060		INL
16 300	374		LME
16 301	104 174 015		JMP GOTO
16 304	066 073		RETURN, LLI 073
16 306	056 027	**	LHI 027
16 310	307		LAM
16 311	024 002		SUI 002
16 313	160 356 016		JTS RETERR
16 316	370		LMA
16 317	004 002		ADI 002
16 321	066 076		LLI 076
16 323	206		ADL
16 324	360		LLA
16 325	337		LDM
16 326	030		IND
16 327	031		DCD
16 330	150 266 010		JTZ EXEC
16 333	060		INL
16 334	347		LEM
16 335	066 360		LLI 360
16 337	056 026	**	LHI 026

16 341	373	LMD
16 342	060	INL
16 343	374	LME
16 344	104 116 013	JMP NXTLIN
16 347	006 307	GOSERR, LAI 307
16 351	026 323	LCI 323
16 353	104 226 002	JMP ERROR
16 356	006 322	RETERR, LAI 322
16 360	026 324	LCI 324
16 362	104 226 002	JMP ERROR
16 365	106 255 002	INPUT, CAL CLESYM
16 370	066 202	LLI 202
16 372	317	LBM
16 373	010	INB
16 374	066 203	LLI 203
16 376	371	LMB
16 377	066 203	INPUT1, LLI 203
17 001	106 240 002	CAL GETCHR
17 004	150 042 017	JTZ INPUT3
17 007	074 254	CPI 254
17 011	150 063 017	JTZ INPUT4
17 014	074 250	CPI 250
17 016	110 037 017	JFZ INPUT2
17 021	106 160 055	CAL ARRAY2
17 024	066 206	LLI 206
17 026	056 026	LHI 026
17 030	317	LBM
17 031	066 203	LLI 203
17 033	371	LMB
17 034	104 042 017	JMP INPUT3
17 037	106 310 002	INPUT2, CAL CONCTS
17 042	066 203	INPUT3, LLI 203
17 044	106 003 003	CAL LOOP
17 047	110 377 016	JFZ INPUT1
17 052	106 104 017	CAL INPUTX
17 055	106 055 010	CAL STOSYM
17 060	104 116 013	JMP NXTLIN
17 063	106 104 017	INPUT4, CAL INPUTX
17 066	106 055 010	CAL STOSYM
17 071	056 026	LHI 026
17 073	066 203	LLI 203
17 075	317	LBM
17 076	066 202	LLI 202
17 100	371	LMB

17 101	104 365 016	JMP INPUT
17 104	066 120	INPUTX, LLI 120
17 106	307	LAM
17 107	206	ADL
17 110	360	LLA
17 111	307	LAM
17 112	074 244	CPI 244
17 114	110 140 017	JFZ INPUTN
17 117	066 120	LLI 120
17 121	317	LBM
17 122	011	DCB
17 123	371	LMB
17 124	106 157 017	CAL FP0
17 127	106 221 003	CAL CINPUT
17 132	066 124	LLI 124
17 134	370	LMA
17 135	104 064 020	JMP FPFLT
17 140	066 144	INPUTN, LLI 144
17 142	056 026	** LHI 026
17 144	006 277	LAI 277
17 146	106 202 003	CAL ECHO
17 151	106 014 003	CAL STRIN
17 154	104 044 023	JMP DINPUT
17 157	056 001	FPO, LHI 001
17 161	104 247 006	JMP CFALSE
17 164	066 144	FOR, LLI 144
17 166	056 026	** LHI 026
17 170	076 000	LMI 000
17 172	066 146	LLI 146
17 174	076 000	LMI 000
17 176	066 205	LLI 205
17 200	056 027	** LHI 027
17 202	317	LBM
17 203	010	INB
17 204	371	LMB
17 205	066 360	LLI 360
17 207	056 026	** LHI 026
17 211	337	LDM
17 212	060	INL
17 213	347	LEM
17 214	301	LAB
17 215	002	RLC
17 216	002	RLC
17 217	004 134	ADI 134
17 221	360	LLA
17 222	056 027	** LHI 027
17 224	373	LMD

END
1 JUL 76

3 JUL

17 225	060		INL
17 226	374		LME
17 227	066 325		LLI 325
17 231	056 001	**	LHI 001
17 233	106 012 013		CAL INSTR
17 236	304		LAE
17 237	240		NDA
17 240	110 252 017		JFZ FOR1
17 243	006 306	FORERR,	LAI 306
17 245	026 305		LCI 305
17 247	104 226 002		JMP ERROR
17 252	066 202	FOR1,	LLI 202
17 254	056 026	**	LHI 026
17 256	317		LBM
17 257	010		INB
17 260	066 204		LLI 204
17 262	371		LMB
17 263	066 203		LLI 203
17 265	374		LME
17 266	066 204	FOR2,	LLI 204
17 270	106 240 002		CAL GETCHR
17 273	150 310 017		JTZ FOR3
17 276	074 275		CPI 275
17 300	150 323 017		JTZ FOR4
17 303	066 144		LLI 144
17 305	106 314 002		CAL CONCT1
17 310	066 204	FOR3,	LLI 204
17 312	106 003 003		CAL LOOP
17 315	110 266 017		JFZ FOR2
17 320	104 243 017		JMP FORERR
17 323	066 204	FOR4,	LLI 204
17 325	317		LBM
17 326	010		INB
17 327	066 276		LLI 276
17 331	371		LMB
17 332	066 203		LLI 203
17 334	317		LBM
17 335	011		DCB
17 336	066 277		LLI 277
17 340	371		LMB
17 341	106 224 003		CAL EVAL
17 344	106 252 010		CAL RESTSY
17 347	066 144		LLI 144
17 351	056 026	**	LHI 026
17 353	307		LAM
17 354	074 001		CPI 001
17 356	110 246 031		JFZ FOR5
17 361	066 146		LLI 146

17 363 076 000
17 365 104 246 031

LMI 000
JMP FOR5

Note open addresses.
This space available
for patching.

20 000	066 126		FPFIX, LLI 126
20 002	056 001	**	LHI 001
20 004	307		LAM
20 005	066 100		LLI 100
20 007	370		LMA
20 010	240		NDA
20 011	162 202 020		CTS FPCOMP
20 014	066 127		LLI 127
20 016	006 027		LAI 027
20 020	317		LBM
20 021	010		INB
20 022	011		DCB
20 023	160 051 020		JTS FPZERO
20 026	221		SUB
20 027	160 366 012		JTS FIXERR
20 032	320		LCA
20 033	066 126		FPFIXL, LLI 126
20 035	016 003		LBI 003
20 037	106 211 022		CAL ROTATR
20 042	021		DCC
20 043	110 033 020		JFZ FPFLX
20 046	104 175 020		JMP RESIGN
20 051	066 126		FPZERO, LLI 126
20 053	250		XRA
20 054	370		LMA
20 055	061		DCL
20 056	370		LMA
20 057	061		DCL
20 060	370		LMA
20 061	061		DCL
20 062	370		LMA
20 063	007		RET
20 064	016 027		FPFLT, LBI 027
20 066	301		FPNORM, LAB
20 067	056 001	**	LHI 001
20 071	066 127		LLI 127
20 073	240		NDA
20 074	150 100 020		JTZ NOEXCO
20 077	371		LMB
20 100	061		NOEXCO, DCL
20 101	307		LAM
20 102	066 100		LLI 100

20 104	370		LMA
20 105	240		NDA
20 106	120 120 020		JFS ACZERT
20 111	016 004		LBI 004
20 113	066 123		LLI 123
20 115	106 150 022		CAL COMPLM
20 120	066 126	ACZERT,	LLI 126
20 122	016 004		LBI 004
20 124	307	LOOK0,	LAM
20 125	240		NDA
20 126	110 143 020		JFZ ACNONZ
20 131	061		DCL
20 132	011		DCB
20 133	110 124 020		JFZ LOOK0
20 136	066 127		LLI 127
20 140	250		XRA
20 141	370		LMA
20 142	007		RET
20 143	066 123	ACNONZ,	LLI 123
20 145	016 004		LBI 004
20 147	106 177 022		CAL ROTATL
20 152	307		LAM
20 153	240		NDA
20 154	160 166 020		JTS ACCSET
20 157	060		INL
20 160	317		LBM
20 161	011		DCB
20 162	371		LMB
20 163	104 143 020		JMP ACNONZ
20 166	066 126	ACCSET,	LLI 126
20 170	016 003		LBI 003
20 172	106 211 022		CAL ROTATR
20 175	066 100	RESIGN,	LLI 100
20 177	307		LAM
20 200	240		NDA
20 201	023		RFS
20 202	066 124	FPCOMP,	LLI 124
20 204	016 003		LBI 003
20 206	104 150 022		JMP COMPLM
20 211	066 126	FPADD,	LLI 126
20 213	056 001	**	LHI 001
20 215	307		LAM
20 216	240		NDA
20 217	110 235 020		JFZ NONZAC
20 222	066 124	MOVOP,	LLI 124
20 224	335		LDH
20 225	346		LEL
20 226	066 134		LLI 134
20 230	016 004		LBI 004
20 232	104 013 021		JMP MOVEIT

20 235	066 136	NONZAC,	LLI 136
20 237	307		LAM
20 240	240		NDA
20 241	053		RTZ
20 242	066 127	CKEQEX,	LLI 127
20 244	307		LAM
20 245	066 137		LLI 137
20 247	277		CPM
20 250	150 341 020		JTZ SHACOP
20 253	310		LBA
20 254	307		LAM
20 255	231		SBB
20 256	120 264 020		JFS SKPNEG
20 261	310		LBA
20 262	250		XRA
20 263	231		SBB
20 264	074 030	SKPNEG,	CPI 030
20 266	160 303 020		JTS LINEUP
20 271	307		LAM
20 272	066 127		LLI 127
20 274	227		SUM
20 275	063		RTS
20 276	066 124		LLI 124
20 300	104 222 020		JMP MOVOP
20 303	307	LINEUP,	LAM
20 304	066 127		LLI 127
20 306	227		SUM
20 307	160 327 020		JTS SHIFTO
20 312	320		LCA
20 313	066 127	MORACC,	LLI 127
20 315	106 374 020		CAL SHLOOP
20 320	021		DCC
20 321	110 313 020		JFZ MORACC
20 324	104 341 020		JMP SHACOP
20 327	320	SHIFTO,	LCA
20 330	066 137	MOROP,	LLI 137
20 332	106 374 020		CAL SHLOOP
20 335	020		INC
20 336	110 330 020		JFZ MOROP
20 341	066 123	SHACOP,	LLI 123
20 343	076 000		LMI 000
20 345	066 127		LLI 127
20 347	106 374 020		CAL SHLOOP
20 352	066 137		LLI 137
20 354	106 374 020		CAL SHLOOP
20 357	335		LDH
20 360	046 123		LEI 123
20 362	016 004		LBI 004
20 364	106 127 022		CAL ADDER
20 367	016 000		LBI 000
20 371	104 066 020		JMP FPNORM

20 374	317		SHLOOP,	LBM
20 375	010			INB
20 376	371			LMB
20 377	061			DCL
21 000	016 004			LBI 004
21 002	307		FSHIFT,	LAM
21 003	240			NDA
21 004	120 211 022			JFS ROTATR
21 007	022		BRING1,	RAL
21 010	104 212 022			JMP ROTR
21 013	307		MOVEIT,	LAM
21 014	060			INL
21 015	106 356 022			CAL SWITCH
21 020	370			LMA
21 021	060			INL
21 022	106 356 022			CAL SWITCH
21 025	011			DCB
21 026	053			RTZ
21 027	104 013 021			JMP MOVEIT
21 032	066 124		FSUB,	LLI 124
21 034	056 001	**		LHI 001
21 036	016 003			LBI 003
21 040	106 150 022			CAL COMPLM
21 043	104 211 020			JMP FPADD
21 046	106 166 021		FPMULT,	CAL CKSIGN
21 051	066 137		ADDEXP,	LLI 137
21 053	307			LAM
21 054	066 127			LLI 127
21 056	207			ADM
21 057	004 001			ADI 001
21 061	370			LMA
21 062	066 102		SETMCT,	LLI 102
21 064	076 027			LMI 027
21 066	066 126		MULTIP,	LLI 126
21 070	016 003			LBI 003
21 072	106 211 022			CAL ROTATR
21 075	142 270 021			CTC ADOPPP
21 100	066 146			LLI 146
21 102	016 006			LBI 006
21 104	106 211 022			CAL ROTATR
21 107	066 102			LLI 102
21 111	327			LCM
21 112	021			DCC
21 113	372			LMC
21 114	110 066 021			JFZ MULTIP
21 117	066 146			LLI 146
21 121	016 006			LBI 006
21 123	106 211 022			CAL ROTATR
21 126	066 143			LLI 143

21 130	307		LAM
21 131	022		RAL
21 132	240		NDA
21 133	162 302 021		CTS MROUND
21 136	066 123		LLI 123
21 140	346		LEL
21 141	335		LDH
21 142	066 143		LLI 143
21 144	016 004		LBI 004
21 146	106 013 021	EXMLDV,	CAL MOVEIT
21 151	016 000		LBI 000
21 153	106 066 020		CAL FPNORM
21 156	066 101		LLI 101
21 160	307		LAM
21 161	240		NDA
21 162	013		RFZ
21 163	104 202 020		JMP FPCOMP
21 166	066 140	CKSIGN,	LLI 140
21 170	056 001		LHI 001
21 172	016 010		LBI 010
21 174	250		XRA
21 175	370	CLRNEX,	LMA
21 176	060		INL
21 177	011		DCB
21 200	110 175 021		JFZ CLRNEX
21 203	016 004	CLROPL,	LBI 004
21 205	066 130		LLI 130
21 207	370	CLRNX1,	LMA
21 210	060		INL
21 211	011		DCB
21 212	110 207 021		JFZ CLRNX1
21 215	066 101		LLI 101
21 217	076 001		LMI 001
21 221	066 126		LLI 126
21 223	307		LAM
21 224	240		NDA
21 225	160 251 021		JTS NEGFPA
21 230	066 136	OPSGNT,	LLI 136
21 232	307		LAM
21 233	240		NDA
21 234	023		RFS
21 235	066 101		LLI 101
21 237	327		LCM
21 240	021		DCC
21 241	372		LMC
21 242	066 134		LLI 134
21 244	016 003		LBI 003
21 246	104 150 022		JMP COMPLM
21 251	066 101	NEGFPA,	LLI 101

21 253	327		LCM
21 254	021		DCC
21 255	372		LMC
21 256	066 124		LLI 124
21 260	016 003		LBI 003
21 262	106 150 022		CAL COMPLM
21 265	104 230 021		JMP OPSGNT
21 270	046 141	ADOPPP,	LEI 141
21 272	335		LDH
21 273	066 131		LLI 131
21 275	016 006		LBI 006
21 277	104 127 022		JMP ADDER
21 302	016 003	MROUND,	LBI 003
21 304	006 100		LAI 100
21 306	207		ADM
21 307	370	CROUND,	LMA
21 310	060		INL
21 311	006 000		LAI 000
21 313	217		ACM
21 314	011		DCB
21 315	110 307 021		JFZ CROUND
21 320	370		LMA
21 321	007		RET
21 322	106 166 021	FPDIV,	CAL CKSIGN
21 325	066 126		LLI 126
21 327	307		LAM
21 330	240		NDA
21 331	150 357 012		JTZ DVERR
21 334	066 137	SUBEXP,	LLI 137
21 336	307		LAM
21 337	066 127		LLI 127
21 341	227		SUM
21 342	004 001		ADI 001
21 344	370		LMA
21 345	066 102	SETDCT,	LLI 102
21 347	076 027		LMI 027
21 351	106 101 022	DIVIDE,	CAL SETSUB
21 354	160 376 021		JTS NOGO
21 357	046 134		LEI 134
21 361	066 131		LLI 131
21 363	016 003		LBI 003
21 365	106 013 021		CAL MOVEIT
21 370	006 001		LAI 001
21 372	032		RAR
21 373	104 377 021		JMP QUOROT
21 376	250	NOGO,	XRA
21 377	066 144	QUOROT,	LLI 144
22 001	016 003		LBI 003
22 003	106 200 022		CAL ROTL
22 006	066 134		LLI 134

22 010	016 003	LBI 003
22 012	106 177 022	CAL ROTATL
22 015	066 102	LLI 102
22 017	327	LCM
22 020	021	DCC
22 021	372	LMC
22 022	110 351 021	JFZ DIVIDE
22 025	106 101 022	CAL SETSUB
22 030	160 070 022	JTS DVEXIT
22 033	066 144	LLI 144
22 035	307	LAM
22 036	004 001	ADI 001
22 040	370	LMA
22 041	006 000	LAI 000
22 043	060	INL
22 044	217	ACM
22 045	370	LMA
22 046	006 000	LAI 000
22 050	060	INL
22 051	217	ACM
22 052	370	LMA
22 053	120 070 022	JFS DVEXIT
22 056	016 003	LBI 003
22 060	106 211 022	CAL ROTATR
22 063	066 127	LLI 127
22 065	317	LBM
22 066	010	INB
22 067	371	LMB
22 070	066 14 43	DVEXIT, LLI 14 43
22 072	046 12 43	LEI 12 43
22 074	016 00 34	LBI 00 34
22 076	104 146 021	JMP EXMLDV
22 101	046 131	SETSUB, LEI 131
22 103	335	LDH
22 104	066 124	LLI 124
22 106	016 003	LBI 003
22 110	106 013 021	CAL MOVEIT
22 113	046 131	LEI 131
22 115	066 134	LLI 134
22 117	016 003	LBI 003
22 121	106 223 022	CAL SUBBER
22 124	307	LAM
22 125	240	NDA
22 126	007	RET
22 127	240	ADDER, NDA
22 130	307	ADDMOR, LAM
22 131	106 356 022	CAL SWITCH
22 134	217	ACM
22 135	370	LMA
22 136	011	DCB

22 137	053		RTZ
22 140	060		INL
22 141	106 356 022		CAL SWITCH
22 144	060		INL
22 145	104 130 022		JMP ADDMOR
22 150	307	COMPLM,	LAM
22 151	054 377		XRI 377
22 153	004 001		ADI 001
22 155	370	MORCOM,	LMA
22 156	032		RAR
22 157	330		LDA
22 160	011		DCB
22 161	053		RTZ
22 162	060		INL
22 163	307		LAM
22 164	054 377		XRI 377
22 166	340		LEA
22 167	303		LAD
22 170	022		RAL
22 171	006 000		LAI 000
22 173	214		ACE
22 174	104 155 022		JMP MORCOM
22 177	240	ROTATL,	NDA
22 200	307	ROTL,	LAM
22 201	022		RAL
22 202	370		LMA
22 203	011		DCB
22 204	053		RTZ
22 205	060		INL
22 206	104 200 022		JMP ROTL
22 211	240	ROTATR,	NDA
22 212	307	ROTR,	LAM
22 213	032		RAR
22 214	370		LMA
22 215	011		DCB
22 216	053		RTZ
22 217	061		DCL
22 220	104 212 022		JMP ROTR
22 223	240	SUBBER,	NDA
22 224	307	SUBTRA,	LAM
22 225	106 356 022		CAL SWITCH
22 230	237		SBM
22 231	370		LMA
22 232	011		DCB
22 233	053		RTZ
22 234	060		INL
22 235	106 356 022		CAL SWITCH

22 240	060		INL
22 241	104 224 022		JMP SUBTRA
22 244	036 001	**	FLOAD, LDI 001
22 246	046 124		LEI 124
22 250	016 004		LBI 004
22 252	104 013 021		JMP MOVEIT
22 255	346		FSTORE, LEL
22 256	335		LDH
22 257	066 124		LLI 124
22 261	056 001	**	LHI 001
22 263	104 272 022		JMP SETIT
22 266	036 001	**	OPLOAD, LDI 001
22 270	046 134		LEI 134
22 272	016 004		SETIT, LBI 004
22 274	104 013 021		JMP MOVEIT
22 277	106 317 022		FACXOP, CAL SAVEHL
22 302	066 124		LLI 124
22 304	056 001	**	LHI 001
22 306	106 266 022		CAL OPLOAD
22 311	106 337 022		CAL RESTHL
22 314	104 244 022		JMP FLOAD
22 317	305		SAVEHL, LAH
22 320	316		LBL
22 321	066 200		LLI 200
22 323	056 001	**	LHI 001
22 325	370		LMA
22 326	060		INL
22 327	371		LMB
22 330	060		INL
22 331	373		LMD
22 332	060		INL
22 333	374		LME
22 334	350		LHA
22 335	361		LLB
22 336	007		RET
22 337	066 200		RESTHL, LLI 200
22 341	056 001	**	LHI 001
22 343	307		LAM
22 344	060		INL
22 345	317		LBM
22 346	060		INL
22 347	337		LDM
22 350	060		INL
22 351	347		LEM
22 352	350		LHA

22 353	361		LLB	
22 354	307		LAM	
22 355	007		RET	
22 356	325	SWITCH,	LCH	
22 357	353		LHD	
22 360	332		LDC	
22 361	326		LCL	
22 362	364		LLE	
22 363	342		LEC	
22 364	007		RET	
22 365	056 001	**	GETINP,	LHI 001
22 367	066 220			LLI 220
22 371	327			LCM
22 372	020			INC
22 373	021			DCC
22 374	110 010 023			JFZ NOT0
22 377	364			LLE
23 000	353			LHD
23 001	327			LCM
23 002	020			INC
23 003	106 036 023			CAL INDEXC
23 006	076 000			LMI 000
23 010	066 220		NOT0,	LLI 220
23 012	056 001	**		LHI 001
23 014	327			LCM
23 015	020			INC
23 016	372			LMC
23 017	364			LLE
23 020	353			LHD
23 021	106 036 023			CAL INDEXC
23 024	307			LAM
23 025	240			NDA
23 026	056 001	**		LHI 001
23 030	013			RFZ
23 031	066 220			LLI 220
23 033	076 000			LMI 000
23 035	007			RET
23 036	306	INDEXC,	LAL	
23 037	202		ADC	
23 040	360		LLA	
23 041	003		RFC	
23 042	050		INH	
23 043	007		RET	
23 044	346	DINPUT,	LEL	
23 045	335		LDH	
23 046	056 001	**	LHI 001	

23 050	066 150		LLI 150
23 052	250		XRA
23 053	016 010		LBI 010
23 055	370	CLRNX2,	LMA
23 056	060		INL
23 057	011		DCB
23 060	110 055 023		JFZ CLRNX2
23 063	066 103		LLI 103
23 065	016 004		LBI 004
23 067	370	CLRNX3,	LMA
23 070	060		INL
23 071	011		DCB
23 072	110 067 023		JFZ CLRNX3
23 075	106 365 022		CAL GETINP
23 100	074 253		CPI 253
23 102	150 115 023		JTZ NINPUT
23 105	074 255		CPI 255
23 107	110 120 023		JFZ NOTPLM
23 112	066 103		LLI 103
23 114	370		LMA
23 115	106 365 022	NINPUT,	CAL GETINP
23 120	074 256	NOTPLM,	CPI 256
23 122	150 201 023		JTZ PERIOD
23 125	074 305		CPI 305
23 127	150 221 023		JTZ FNDEXP
23 132	074 240		CPI 240
23 134	150 115 023		JTZ NINPUT
23 137	240		NDA
23 140	150 311 023		JTZ ENDINP
23 143	074 260		CPI 260
23 145	160 375 012		JTS NUMERR
23 150	074 272		CPI 272
23 152	120 375 012		JFS NUMERR
23 155	066 156		LLI 156
23 157	320		LCA
23 160	006 370		LAI 370
23 162	247		NDM
23 163	110 115 023		JFZ NINPUT
23 166	066 105		LLI 105
23 170	317		LBM
23 171	010		INB
23 172	371		LMB
23 173	106 056 024		CAL DECBIN
23 176	104 115 023		JMP NINPUT
23 201	310	PERIOD,	LBA
23 202	066 106		LLI 106
23 204	307		LAM
23 205	240		NDA

23 206	110 375 012	JFZ NUMERR
23 211	066 105	LLI 105
23 213	370	LMA
23 214	060	INL
23 215	371	LMB
23 216	104 115 023	JMP NINPUT
23 221	106 365 022	FNDEXP, CAL GETINP
23 224	074 253	CPI 253
23 226	150 241 023	JTZ EXPINP
23 231	074 255	CPI 255
23 233	110 244 023	JFZ NOEXPS
23 236	066 104	LLI 104
23 240	370	LMA
23 241	106 365 022	EXPINP, CAL GETINP
23 244	240	NOEXPS, NDA
23 245	150 311 023	JTZ ENDINP
23 250	074 260	CPI 260
23 252	160 375 012	JTS NUMERR
23 255	074 272	CPI 272
23 257	120 375 012	JFS NUMERR
23 262	044 017	NDI 017
23 264	310	LBA
23 265	066 157	LLI 157
23 267	006 003	LAI 003
23 271	277	CPM
23 272	160 375 012	JTS NUMERR
23 275	327	LCM
23 276	307	LAM
23 277	240	NDA
23 300	022	RAL
23 301	022	RAL
23 302	202	ADC
23 303	022	RAL
23 304	201	ADB
23 305	370	LMA
23 306	104 241 023	JMP EXPINP
23 311	066 103	ENDINP, LLI 103
23 313	307	LAM
23 314	240	NDA
23 315	150 327 023	JTZ FININP
23 320	066 154	LLI 154
23 322	016 003	LBI 003
23 324	106 150 022	CAL COMPLM
23 327	066 153	FININP, LLI 153
23 331	250	XRA
23 332	370	LMA

23 333	335		LDH
23 334	046 123		LEI 123
23 336	016 004		LBI 004
23 340	106 013 021		CAL MOVEIT
23 343	106 064 020		CAL FPFLT
23 346	066 104		LLI 104
23 350	307		LAM
23 351	240		NDA
23 352	066 157		LLI 157
23 354	150 365 023		JTZ POSEXP
23 357	307		LAM
23 360	054 377		XRI 377
23 362	004 001		ADI 001
23 364	370		LMA
23 365	066 106	POSEXP,	LLI 106
23 367	307		LAM
23 370	240		NDA
23 371	150 000 024		JTZ EXPOK
23 374	066 105		LLI 105
23 376	250		XRA
23 377	227		SUM
24 000	066 157	EXPOK,	LLI 157
24 002	207		ADM
24 003	370		LMA
24 004	160 033 024		JTS MINEXP
24 007	053		RTZ
24 010	066 210	FPX10,	LLI 210
24 012	056 001	**	LHI 001
24 014	106 277 022		CAL FACXOP
24 017	106 046 021		CAL FPMULT
24 022	066 157		LLI 157
24 024	327		LCM
24 025	021		DCC
24 026	372		LMC
24 027	110 010 024		JFZ FPX10
24 032	007		RET
24 033	066 214	MINEXP, FPD10,	LLI 214
24 035	056 001	**	LHI 001
24 037	106 277 022		CAL FACXOP
24 042	106 046 021		CAL FPMULT
24 045	066 157		LLI 157
24 047	317		LBM
24 050	010		INB
24 051	371		LMB
24 052	110 033 024		JFZ FPD10
24 055	007		RET

24 056	106 317 022		DECBIN,	CAL SAVEHL
24 061	066 153			LLI 153
24 063	302			LAC
24 064	044 017			NDI 017
24 066	370			LMA
24 067	046 150			LEI 150
24 071	066 154			LLI 154
24 073	335			LDH
24 074	016 003			LBI 003
24 076	106 013 021			CAL MOVEIT
24 101	066 154			LLI 154
24 103	016 003			LBI 003
24 105	106 177 022			CAL ROTATL
24 110	066 154			LLI 154
24 112	016 003			LBI 003
24 114	106 177 022			CAL ROTATL
24 117	046 154			LEI 154
24 121	066 150			LLI 150
24 123	016 003			LBI 003
24 125	106 127 022			CAL ADDER
24 130	066 154			LLI 154
24 132	016 003			LBI 003
24 134	106 177 022			CAL ROTATL
24 137	066 152			LLI 152
24 141	250			XRA
24 142	370			LMA
24 143	061			DCL
24 144	370			LMA
24 145	066 153			LLI 153
24 147	307			LAM
24 150	066 150			LLI 150
24 152	370			LMA
24 153	046 154			LEI 154
24 155	016 003			LBI 003
24 157	106 127 022			CAL ADDER
24 162	104 337 022			JMP RESTHL
24 165	056 001	**	FPOUT,	LHI 001
24 167	066 157			LLI 157
24 171	076 000			LMI 000
24 173	066 126			LLI 126
24 175	307			LAM
24 176	240			NDA
24 177	160 207 024			JTS OUTNEG
24 202	006 240			LAI 240
24 204	104 220 024			JMP AHEAD1
24 207	066 124		OUTNEG,	LLI 124
24 211	016 003			LBI 003
24 213	106 150 022			CAL COMPLM
24 216	006 255			LAI 255

24 220	106 202 003	AHEAD1,	CAL ECHO
24 223	066 110		LLI 110
24 225	307		LAM
24 226	240		NDA
24 227	150 253 024		JTZ OUTFLT
24 232	066 127		LLI 127
24 234	006 027		LAI 027
24 236	317		LBM
24 237	010		INB
24 240	011		DCB
24 241	160 253 024		JTS OUTFLT
24 244	221		SUB
24 245	160 253 024		JTS OUTFLT
24 250	104 271 024		JMP OUTFIX
24 253	066 110	OUTFLT,	LLI 110
24 255	076 000		LMI 000
24 257	006 260		LAI 260
24 261	106 202 003		CAL ECHO
24 264	006 256		LAI 256
24 266	106 202 003		CAL ECHO
24 271	066 127	OUTFIX,	LLI 127
24 273	006 377		LAI 377
24 275	207		ADM
24 276	370		LMA
24 277	120 336 024	DECEXT,	JFS DECEXD
24 302	006 004		LAI 004
24 304	207		ADM
24 305	120 360 024		JFS DECOUT
24 310	066 210		LLI 210
24 312	056 001	**	LHI 001
24 314	106 277 022		CAL FACXOP
24 317	106 046 021		CAL FPMULT
24 322	066 157		LLI 157
24 324	327		LCM
24 325	021		DCC
24 326	372		LMC
24 327	066 127	DECREP,	LLI 127
24 331	307		LAM
24 332	240		NDA
24 333	104 277 024		JMP DECEXT
24 336	066 214	DECEXD,	LLI 214
24 340	056 001	**	LHI 001
24 342	106 277 022		CAL FACXOP
24 345	106 046 021		CAL FPMULT
24 350	066 157		LLI 157
24 352	317		LBM

24 353	010	INB
24 354	371	LMB
24 355	104 327 024	JMP DECREP
24 360	046 164	DECOUT, LEI 164
24 362	335	LDH
24 363	066 124	LLI 124
24 365	016 003	LBI 003
24 367	106 013 021	CAL MOVEIT
24 372	066 167	LLI 167
24 374	076 000	LMI 000
24 376	066 164	LLI 164
25 000	016 003	LBI 003
25 002	106 177 022	CAL ROTATL
25 005	106 223 025	CAL OUTX10
25 010	066 127	COMPEN, LLI 127
25 012	317	LBM
25 013	010	INB
25 014	371	LMB
25 015	150 032 025	JTZ OUTDIG
25 020	066 167	LLI 167
25 022	016 004	LBI 004
25 024	106 211 022	CAL ROTATTR
25 027	104 010 025	JMP COMPEN
25 032	066 107	OUTDIG, LLI 107
25 034	076 007	LMI 007
25 036	066 167	LLI 167
25 040	307	LAM
25 041	240	NDA
25 042	150 165 025	JTZ ZERODG
25 045	066 167	OUTDGS, LLI 167
25 047	307	LAM
25 050	240	NDA
25 051	110 105 025	JFZ OUTDGX
25 054	066 110	LLI 110
25 056	307	LAM
25 057	240	NDA
25 060	150 104 025	JTZ OUTZER
25 063	066 157	LLI 157
25 065	327	LCM
25 066	021	DCC
25 067	020	INC
25 070	120 104 025	JFS OUTZER
25 073	066 166	LLI 166
25 075	307	LAM
25 076	044 340	NDI 340
25 100	110 104 025	JFZ OUTZER
25 103	007	RET

25 104	250	OUTZER,	XRA
25 105	004 260	OUTDGX,	ADI 260
25 107	106 202 003		CAL ECHO
25 112	066 110	DECRDG,	LLI 110
25 114	307		LAM
25 115	240		NDA
25 116	110 137 025		JFZ CKDECP
25 121	066 107		LLI 107
25 123	327		LCM
25 124	021		DCC
25 125	372		LMC
25 126	150 300 025		JTZ EXPOUT
25 131	106 223 025	PUSHIT,	CAL OUTX10
25 134	104 045 025		JMP OUTDGS
25 137	066 157	CKDECP,	LLI 157
25 141	327		LCM
25 142	021		DCC
25 143	372		LMC
25 144	110 154 025		JFZ NODECP
25 147	006 256		LAI 256
25 151	106 202 003		CAL ECHO
25 154	066 107	NODECP,	LLI 107
25 156	327		LCM
25 157	021		DCC
25 160	372		LMC
25 161	053		RTZ
25 162	104 131 025		JMP PUSHIT
25 165	066 157	ZERODG,	LLI 157
25 167	327		LCM
25 170	021		DCC
25 171	372		LMC
25 172	066 166		LLI 166
25 174	307		LAM
25 175	240		NDA
25 176	110 112 025		JFZ DECRDG
25 201	061		DCL
25 202	307		LAM
25 203	240		NDA
25 204	110 112 025		JFZ DECRDG
25 207	061		DCL
25 210	307		LAM
25 211	240		NDA
25 212	110 112 025		JFZ DECRDG
25 215	066 157		LLI 157
25 217	370		LMA
25 220	104 112 025		JMP DECRDG

25 223	066 167	OUTX10,	LLI 167
25 225	076 000		LMI 000
25 227	066 164		LLI 164
25 231	335		LDH
25 232	046 160		LEI 160
25 234	016 004		LBI 004
25 236	106 013 021		CAL MOVEIT
25 241	066 164		LLI 164
25 243	016 004		LBI 004
25 245	106 177 022		CAL ROTATL
25 250	066 164		LLI 164
25 252	016 004		LBI 004
25 254	106 177 022		CAL ROTATL
25 257	066 160		LLI 160
25 261	046 164		LEI 164
25 263	016 004		LBI 004
25 265	106 127 022		CAL ADDER
25 270	066 164		LLI 164
25 272	016 004		LBI 004
25 274	106 177 022		CAL ROTATL
25 277	007		RET
25 300	066 157	EXPOUT,	LLI 157
25 302	307		LAM
25 303	240		NDA
25 304	053		RTZ
25 305	006 305		LAI 305
25 307	106 202 003		CAL ECHO
25 312	307		LAM
25 313	240		NDA
25 314	160 324 025		JTS EXOUTN
25 317	006 253		LAI 253
25 321	104 333 025		JMP AHEAD2
25 324	054 377	EXOUTN,	XRI 377
25 326	004 001		ADI 001
25 330	370		LMA
25 331	006 255		LAI 255
25 333	106 202 003	AHEAD2,	CAL ECHO
25 336	016 000		LBI 000
25 340	307		LAM
25 341	024 012	SUB12,	SUI 012
25 343	160 353 025		JTS TOMUCH
25 346	370		LMA
25 347	010		INB
25 350	104 341 025		JMP SUB12
25 353	006 260	TOMUCH,	LAI 260
25 355	201		ADB

25 356	106 202 003	CAL ECHO
25 361	307	LAM
25 362	004 260	ADI 260
25 364	106 202 003	CAL ECHO
25 367	007	RET

Note open addresses.
This space available
for patching.

NOTE: Pages 26 and 27 in memory are used for temporary data registers, pointers, counters and look-up tables. The following data should be placed on those pages. An entry marked XXX indicates the initial contents of the location are irrelevant to the program's operation.

26 000	000	(cc) for INPUT LINE BUFF
26 001	XXX	These locations used as the INPUT LINE BUFFER storage area
.	.	
26 117	XXX	These locations used as the SYMBOL BUFFER storage area
26 120	000	
.	.	
26 143	000	These locations used as the AUXILIARY SYMBOL BUFFER storage area
26 144	000	
.	.	
26 175	000	TEMP SCAN storage register
26 176	000	TAB FLAG
26 177	000	EVAL CURRENT temp. reg.
26 200	000	SYNTAX LINE NUMBER
26 201	000	SCAN temporary register
26 202	000	STATEMENT TOKEN
26 203	000	Temporary working register
26 204	000	Temporary working register
26 205	000	ARRAY pointer
26 206	000	ARRAY pointer
26 207	000	OPERATOR STACK pointer
26 210	000	These locations used as the OPERATOR STACK storage
26 211	XXX	area
.	.	
26 227	XXX	FUN/ARRAY STACK pointer
26 230	000	These locations used as the FUNCTION/ARRAY STACK storage
26 231	XXX	area
.	.	
26 237	XXX	

Heirarchy table (for out of stack ops).
Used by PARSER routine.

26 240	000	EOS
26 241	003	Plus sign
26 242	003	Minus sign
26 243	004	Multiplication sign
26 244	004	Division sign
26 245	005	Exponentiation sign
26 246	006	Left parenthesis
26 247	001	Right parenthesis
26 250	002	Not assigned
26 251	002	Less than sign
26 252	002	Equal sign
26 253	002	Greater than sign
26 254	002	Less than or equal combo
26 255	002	Equal to or greater than
26 256	002	Less than or greater than

Heirarchy table (for into stack ops).
Used by PARSER routine.

26 257	000	EOS
26 260	003	Plus sign
26 261	003	Minus sign
26 262	004	Multiplication sign
26 263	004	Division sign
26 264	005	Exponentiation sign
26 265	001	Left parenthesis
26 266	001	Right parenthesis
26 267	002	Not assigned
26 270	002	Less than sign
26 271	002	Equal sign
26 272	002	Greater than sign
26 273	002	Less than or equal combo
26 274	002	Equal to or greater than
26 275	002	Less than or greater than
26 276	000	EVAL (start) pointer
26 277	000	EVAL FINISH pointer

FUNCTION NAMES TABLE

26 300	003	(cc) for INT
26 301	311	I
26 302	316	N
26 303	324	T
26 304	003	(cc) for SGN
26 305	323	S
26 306	307	G
26 307	316	N

26 310	003	(cc) for ABS
26 311	301	A
26 312	302	B
26 313	323	S
26 314	003	(cc) for SQR
26 315	323	S
26 316	321	Q
26 317	322	R
26 320	003	(cc) for TAB
26 321	324	T
26 322	301	A
26 323	302	B
26 324	003	(cc) for RND
26 325	322	R
26 326	316	N
26 327	304	D
26 330	003	(cc) for CHR
26 331	303	C
26 332	310	H
26 333	322	R
26 334	003	(cc) for UDF
26 335	325	U
26 336	304	D
26 337	306	F
26 340	000	These locations used as the LINE NUMBER BUFFER storage area
.	.	.
.	.	.
26 347	000	These locations used as the AUX LINE NUMBER BUFFER
26 350	000	storage area
.	.	.
.	.	.
26 357	000	USER PGM LINE pointer (pg)
26 360	000	USER PGM LINE pntr (low)
26 361	000	AUX PGM LINE pointer (pg)
26 362	000	AUX PGM LINE pntr (low)
26 363	000	END of USER PGM BFR (pg)
26 364	000	END of USER PGM BFR pntr
26 365	000	Parenthesis counter
26 366	000	QUOTE Indicator
26 367	000	Table counter
26 370	000	Not assigned
26 371	XXX	
.	.	.
.	.	.
26 377	XXX	Not assigned

End of page 26.

STATEMENT KEYWORD TABLE

27 000	003	(cc) for REM
27 001	322	R
27 002	305	E
27 003	315	M
27 004	002	(cc) for IF
27 005	311	I
27 006	306	F
27 007	003	(cc) for LET
27 010	314	L
27 011	305	E
27 012	324	T
27 013	004	(cc) for GOTO
27 014	307	G
27 015	317	O
27 016	324	T
27 017	317	O
27 020	005	(cc) for PRINT
27 021	320	P
27 022	322	R
27 023	311	I
27 024	316	N
27 025	324	T
27 026	005	(cc) for INPUT
27 027	311	I
27 030	316	N
27 031	320	P
27 032	325	U
27 033	324	T
27 034	003	(cc) for FOR
27 035	306	F
27 036	317	O
27 037	322	R
27 040	004	(cc) for NEXT
27 041	316	N
27 042	305	E
27 043	330	X
27 044	324	T
27 045	005	(cc) for GOSUB
27 046	307	G
27 047	317	O
27 050	323	S
27 051	325	U
27 052	302	B
27 053	006	(cc) for RETURN
27 054	322	R
27 055	305	E
27 056	324	T
27 057	325	U
27 060	322	R

27 061	316	N
27 062	003	(cc) for DIM
27 063	304	D
27 064	311	I
27 065	315	M
27 066	003	(cc) for END
27 067	305	E
27 070	316	N
27 071	304	D
27 072	000	End of Table
27 073	000	GOSUB STACK pointer
27 074	XXX	Not assigned
27 075	000	Number of arrays counter
27 076	000	ARRAY pointer
27 077	000	VARIABLES counter
27 100	000	These locations used as the GOSUB STACK storage area
.	.	These locations used as the
.	.	ARRAY VARIABLES TABLE storage area
27 117	000	These locations used as the FOR/NEXT STACK storage area
27 120	000	.
.	.	.
.	.	.
27 137	000	FOR/NEXT STACK pointer
27 140	000	ARRAY/VARIABLE flag
.	.	STOSYM counter
.	.	FUN/ARRAY STACK pointer
27 177	000	ARRAY VALUES pointer
27 200	000	Not assigned
27 201	000	Not assigned
27 202	000	These locations used as the
27 203	000	VARIABLES SYMBOL
27 204	000	TABLE
27 205	XXX	storage area
27 207	XXX	.
27 210	000	.
27 211	XXX	.
27 377	XXX	.

End of page 27.

Note open addresses at start of page 30.
These locations available for patching.

30 013	066 144		NEXT,	LLI 144
30 015	056 026	**		LHI 026
30 017	076 000			LMI 000
30 021	066 202			LLI 202
30 023	317			LBM
30 024	010			INB
30 025	066 201			LLI 201
30 027	371			LMB
30 030	066 201		NEXT1,	LLI 201
30 032	106 240 002			CAL GETCHR
30 035	150 045 030			JTZ NEXT2
30 040	066 144			LLI 144
30 042	106 314 002			CAL CONCT1
30 045	066 201		NEXT2,	LLI 201
30 047	106 003 003			CAL LOOP
30 052	110 030 030			JFZ NEXT1
30 055	066 144			LLI 144
30 057	307			LAM
30 060	074 001			CPI 001
30 062	110 071 030			JFZ NEXT3
30 065	066 146			LLI 146
30 067	076 000			LMI 000
30 071	066 205		NEXT3,	LLI 205
30 073	056 027	**		LHI 027
30 075	307			LAM
30 076	002			RLC
30 077	002			RLC
30 100	004 136			ADI 136
30 102	056 027	**		LHI 027
30 104	360			LLA
30 105	036 026	**		LDI 026
30 107	046 145			LEI 145
30 111	016 002			LBI 002
30 113	106 370 002			CAL STR CPC
30 116	150 130 030			JTZ NEXT4
30 121	006 306		FORNXT,	LAI 306
30 123	026 316			LCI 316
30 125	104 226 002			JMP ERROR
30 130	066 360		NEXT4,	LLI 360
30 132	056 026	**		LHI 026
30 134	337			LDM
30 135	060			INL
30 136	347			LEM
30 137	060			INL
30 140	373			LMD
30 141	060			INL
30 142	374			LME
30 143	066 205			LLI 205
30 145	056 027	**		LHI 027
30 147	307			LAM
30 150	002			RLC

30 151	002		RLC
30 152	004 134		ADI 134
30 154	360		LLA
30 155	337		LDM
30 156	060		INL
30 157	347		LEM
30 160	066 360		LLI 360
30 162	056 026	**	LHI 026
30 164	373		LMD
30 165	060		INL
30 166	374		LME
30 167	353		LHD
30 170	364		LLE
30 171	036 026	**	LDI 026
30 173	046 000		LEI 000
30 175	106 046 012		CAL MOVEC
30 200	066 325		LLI 325
30 202	056 001	**	LHI 001
30 204	106 012 013		CAL INSTR
30 207	304		LAE
30 210	240		NDA
30 211	150 121 030		JTZ FORNXT
30 214	004 002		ADI 002
30 216	066 276		LLI 276
30 220	056 026	**	LHI 026
30 222	370		LMA
30 223	066 330		LLI 330
30 225	056 001	**	LHI 001
30 227	106 012 013		CAL INSTR
30 232	304		LAE
30 233	240		NDA
30 234	110 300 030		JFZ NEXT5
30 237	066 004		LLI 004
30 241	056 001	**	LHI 001
30 243	106 244 022		CAL FLOAD
30 246	066 304		LLI 304
30 250	106 255 022		CAL FSTORE
30 253	066 000		LLI 000
30 255	056 026	**	LHI 026
30 257	317		LBM
30 260	066 277		LLI 277
30 262	371		LMB
30 263	106 224 003		CAL EVAL
30 266	066 310		LLI 310
30 270	056 001	**	LHI 001
30 272	106 255 022		CAL FSTORE
30 275	104 351 030		JMP NEXT6
30 300	041		NEXT5, DCE
30 301	066 277		LLI 277
30 303	056 026	**	LHI 026

30 305	374		LME
30 306	106 224 003		CAL EVAL
30 311	066 310		LLI 310
30 313	056 001	**	LHI 001
30 315	106 255 022		CAL FSTORE
30 320	066 277		LLI 277
30 322	056 026	**	LHI 026
30 324	307		LAM
30 325	004 005		ADI 005
30 327	061		DCL
30 330	370		LMA
30 331	066 000		LLI 000
30 333	317		LBM
30 334	066 277		LLI 277
30 336	371		LMB
30 337	106 224 003		CAL EVAL
30 342	066 304		LLI 304
30 344	056 001	**	LHI 001
30 346	106 255 022		CAL FSTORE
30 351	066 144		NEXT6, LLI 144
30 353	056 026	**	LHI 026
30 355	076 000		LMI 000
30 357	066 034		LLI 034
30 361	056 027	**	LHI 027
30 363	106 012 013		CAL INSTR
30 366	304		LAE
30 367	240		NDA
30 370	066 202		LLI 202
30 372	056 026	**	LHI 026
30 374	370		LMA
30 375	150 121 030		JTZ FORNXT
31 000	004 003		ADI 003
31 002	066 203		LLI 203
31 004	370		LMA
31 005	066 203		NEXT7, LLI 203
31 007	106 240 002		CAL GETCHR
31 012	150 027 031		JTZ NEXT8
31 015	074 275		CPI 275
31 017	150 042 031		JTZ NEXT9
31 022	066 144		LLI 144
31 024	106 314 002		CAL CONCT1
31 027	066 203		NEXT8, LLI 203
31 031	106 003 003		CAL LOOP
31 034	110 005 031		JFZ NEXT7
31 037	104 121 030		JMP FORNXT
31 042	066 202		NEXT9, LLI 202
31 044	056 026	**	LHI 026

31 046	307		LAM
31 047	004 003		ADI 003
31 051	066 276		LLI 276
31 053	370		LMA
31 054	066 203		LLI 203
31 056	317		LBM
31 057	011		DCB
31 060	066 277		LLI 277
31 062	371		LMB
31 063	106 224 003		CAL EVAL
31 066	066 304		LLI 304
31 070	056 001	**	LHI 001
31 072	106 277 022		CAL FACXOP
31 075	106 211 020		CAL FPADD
31 100	066 314		LLI 314
31 102	056 001	**	LHI 001
31 104	106 255 022		CAL FSTORE
31 107	066 310		LLI 310
31 111	106 277 022		CAL FACXOP
31 114	106 032 021		CAL FPSUB
31 117	066 306		LLI 306
31 121	307		LAM
31 122	240		NDA
31 123	066 126		LLI 126
31 125	307		LAM
31 126	150 121 030		JTZ FORNXT
31 131	160 170 031		JTS NEXT11
31 134	240		NDA
31 135	160 177 031		JTS NEXT12
31 140	150 177 031		JTZ NEXT12
31 143	066 363		NEXT10, LLI 363
31 145	056 026	**	LHI 026
31 147	347		LEM
31 150	061		DCL
31 151	337		LDM
31 152	061		DCL
31 153	374		LME
31 154	061		DCL
31 155	373		LMD
31 156	066 205		LLI 205
31 160	056 027	**	LHI 027
31 162	317		LBM
31 163	011		DCB
31 164	371		LMB
31 165	104 116 013		JMP NXTLIN
31 170	240		NDA
31 171	120 177 031		JFS NEXT12
31 174	104 143 031		JMP NEXT10

31 177	066 314		NEXT12,	LLI 314
31 201	056 001	**		LHI 001
31 203	106 244 022			CAL FLOAD
31 206	106 252 010			CAL RESTSY
31 211	106 055 010			CAL STOSYM
31 214	104 116 013			JMP NXTLIN
31 217	006 215		BACKSP,	LAI 215
31 221	106 202 003			CAL ECHO
31 224	106 202 003			CAL ECHO
31 227	066 043			LLI 043
31 231	056 001	**		LHI 001
31 233	076 001			LMI 001
31 235	066 124			LLI 124
31 237	307			LAM
31 240	240			NDA
31 241	063			RTS
31 242	053			RTZ
31 243	104 022 010			JMP TAB1
31 246	066 205		FOR5,	LLI 205
31 250	056 027	**		LHI 027
31 252	307			LAM
31 253	002			RLC
31 254	002			RLC
31 255	004 136			ADI 136
31 257	340			LEA
31 260	335			LDH
31 261	066 145			LLI 145
31 263	056 026	**		LHI 026
31 265	016 002			LBI 002
31 267	106 013 021			CAL MOVEIT
31 272	106 055 010			CAL STOSYM
31 275	104 116 013			JMP NXTLIN
31 300	066 176		PARSEP,	LLI 176
31 302	076 000			LMI 000
31 304	106 324 004			CAL PARSER
31 307	066 227			LLI 227
31 311	056 001	**		LHI 001
31 313	307			LAM
31 314	074 230			CPI 230
31 316	053			RTZ
31 317	104 152 011			JMP SYNERR

Note open addresses.
This space available
for patching.

025 147 104 203 031

031 302 066 665

32 000	066 014	SQRX,	LLI 014
32 002	056 001		LHI 001
32 004	106 255 022		CAL FSTORE
32 007	066 126		LLI 126
32 011	307		LAM
32 012	240		NDA
32 013	160 217 032		JTS SQRERR
32 016	150 247 006		JTZ CFALSE
32 021	066 017		LLI 017
32 023	307		LAM
32 024	240		NDA
32 025	160 041 032		JTS NEGEXP
32 030	032		RAR
32 031	310		LBA
32 032	006 000		LAI 000
32 034	022		RAL
32 035	370		LMA
32 036	104 062 032		JMP SQREXP
 32 041	310	NEGEXP,	LBA
32 042	250		XRA
32 043	221		SUB
32 044	240		NDA
32 045	032		RAR
32 046	310		LBA
32 047	006 000		LAI 000
32 051	210		ACA
32 052	370		LMA
32 053	150 057 032		JTZ NOREMD
32 056	010		INB
 32 057	250	NOREMD,	XRA
32 060	221		SUB
32 061	310		LBA
 32 062	066 013	SQREXP,	LLI 013
32 064	371		LMB
32 065	066 004		LLI 004
32 067	046 034		LEI 034
32 071	335		LDH
32 072	016 004		LBI 004
32 074	106 013 021		CAL MOVEIT
32 077	106 247 006		CAL CFALSE
32 102	066 044		LLI 044
32 104	106 255 022		CAL FSTORE
 32 107	066 034	SQRLOP,	LLI 034
32 111	106 244 022		CAL FLOAD
32 114	066 014		LLI 014
32 116	106 266 022		CAL OPLOAD
32 121	106 322 021		CAL FPDIV

32 124	066 034		LLI 034
32 126	106 266 022		CAL OPLOAD
32 131	106 211 020		CAL FPADD
32 134	066 127		LLI 127
32 136	317		LBM
32 137	011		DCB
32 140	371		LMB
32 141	066 034		LLI 034
32 143	106 255 022		CAL FSTORE
32 146	066 044		LLI 044
32 150	106 266 022		CAL OPLOAD
32 153	106 032 021		CAL FPSUB
32 156	066 127		LLI 127
32 160	307		LAM
32 161	074 367		CPI 367
32 163	160 203 032		JTS SQRCNV
32 166	066 034		LLI 034
32 170	335		LDH
32 171	046 044		LEI 044
32 173	016 004		LBI 004
32 175	106 013 021		CAL MOVEIT
32 200	104 107 032		JMP SQRLOP
32 203	066 013	SQRCNV,	LLI 013
32 205	307		LAM
32 206	066 037		LLI 037
32 210	207		ADM
32 211	370		LMA
32 212	066 034		LLI 034
32 214	104 244 022		JMP FLOAD
32 217	006 323	SQRERR,	LAI 323
32 221	026 321		LCI 321
32 223	104 226 002		JMP ERROR

Note open addresses.

This space available
for patching.

32 240	066 064	RNDX,	LLI 064
32 242	056 001		LHI 001
32 244	106 244 022		CAL FLOAD
32 247	066 050		LLI 050
32 251	106 266 022		CAL OPLOAD
32 254	106 046 021		CAL FPMULT
32 257	066 060		LLI 060
32 261	106 266 022		CAL OPLOAD
32 264	106 211 020		CAL FPADD
32 267	066 064		LLI 064
32 271	106 255 022		CAL FSTORE
32 274	066 127		LLI 127

32 276	307	LAM
32 277	024 020	SUI 020
32 301	370	LMA
32 302	106 000 020	CAL FPFIX
32 305	066 123	LLI 123
32 307	076 000	LMI 000
32 311	066 127	LLI 127
32 313	076 000	LMI 000
32 315	106 064 020	CAL FPFLT
32 320	066 127	LLI 127
32 322	307	LAM
32 323	004 020	ADI 020
32 325	370	LMA
32 326	066 064	LLI 064
32 330	106 266 022	CAL OPLOAD
32 333	106 032 021	CAL FPSUB
32 336	066 064	LLI 064
32 340	106 255 022	CAL FSTORE
32 343	066 127	LLI 127
32 345	307	LAM
32 346	024 020	SUI 020
32 350	370	LMA
32 351	007	RET

Note open addresses
to end of page 32.

Pages 33 to remainder
of memory (or start of
optional ARRAY
handling routines) used as
USER PROGRAM BUFFER.

Optional ARRAY routines
assembled for operation in
the upper three pages of a
12 K system are listed here.

* = CHANGES TO INCORPORATE
3 MARKS OF MEMORY TO
PERMIT 32 X 8000 WORDS PROGRAM BUFFER
(See section 12.22)

* FOR OPENING OF SYSTEM CALLS IN 11 & 12
3 PAGES OF 16K SYSTEM CALLS IN 11 & 12
CHANGE NUMBER OF CALLS 56 = 11
56 = 12

55 000	066 126	PRIGH1,	LLI 126
55 002	056 001	**	LHI 001
55 004	307		LAM
55 005	240		NDA
55 006	160 136 055		JTS OUTRNG
55 011	106 000 020		CAL FPFIX
55 014	066 124		LLI 124
55 016	307		LAM
55 017	024 001		SUI 001
55 021	002		RLC
55 022	002		RLC
55 023	320		LCA
55 024	066 203		LLI 203

* MOVE 55 CCC - 57000
TO 71000 - 73000
USE 16008 OP SYSTEM CALLS
TO MOVE & CHANGE NUMBER OF CALLS

55 026	056 027	**	LHI 027
55 030	307		LAM
55 031	054 377		XRI 377
55 033	002		RLC
55 034	002		RLC
55 035	004 120		ADI 120
55 037	056 027	**	LHI 027
55 041	360		LLA
55 042	060		INL
55 043	060		INL
55 044	307		LAM
55 045	202		ADC
55 046	360		LLA
55 047	056 057	††	LHI 057
55 051	104 244 022		JMP FLOAD
55 054	066 202		FUNAR2, LLI 202
55 056	056 027	**	LHI 027
55 060	317		LBM
55 061	010		INB
55 062	371		LMB
55 063	026 002		LCI 002
55 065	066 114		LLI 114
55 067	056 027	**	LHI 027
55 071	106 230 007		CAL TABADR
55 074	036 026	**	LDI 026
55 076	046 120		LEI 120
55 100	106 332 002		CAL STRCP
55 103	150 124 055		JTZ FUNAR3
55 106	066 202		LLI 202
55 110	056 027	**	LHI 027
55 112	307		LAM
55 113	066 075		LLI 075
55 115	277		CPM
55 116	110 054 055	71	JFZ FUNAR2
55 121	104 172 007		JMP FAERR
55 124	066 202		FUNAR3, LLI 202
55 126	056 027	**	LHI 027
55 130	250		XRA
55 131	237		SBM
55 132	370		LMA
55 133	104 207 007		JMP FUNAR4
55 136	006 317		OUTRNG, LAI 317
55 140	026 322		LCI 322
55 142	104 226 002		JMP ERROR
55 145	106 252 010		ARRAY, CAL RESTSY
55 150	104 160 055	71	JMP ARRAY2

*NEW
71*

55 153	066 202		ARRAY1, LLI 202
55 155	104 162 055	71	JMP ARRAY3
55 160	066 203		ARRAY2, LLI 203
55 162	056 026	**	ARRAY3, LHI 026
55 164	317		LBM
55 165	010		INB
55 166	066 276		LLI 276
55 170	371		LMB
55 171	066 206		LLI 206
55 173	371		LMB
55 174	066 206		ARRAY4, LLI 206
55 176	106 240 002		CAL GETCHR
55 201	074 251		CPI 251
55 203	150 225 055	71	JTZ ARRAY5
55 206	066 206		LLI 206
55 210	106 003 003		CAL LOOP
55 213	110 174 055	71	JFZ ARRAY4
55 216	006 301		LAI 301
55 220	026 306		LCI 306
55 222	104 226 002		JMP ERROR
55 225	066 206		ARRAY5, LLI 206
55 227	317		LBM
55 230	011		DCB
55 231	066 277		LLI 277
55 233	371		LMB
55 234	066 207		LLI 207
55 236	076 000		LMI 000
55 240	066 207		ARRAY6, LLI 207
55 242	056 026	**	LHI 026
55 244	317		LBM
55 245	010		INB
55 246	371		LMB
55 247	026 002		LCI 002
55 251	066 114		LLI 114
55 253	056 027	**	LHI 027
55 255	106 230 007		CAL TABADR
55 260	046 120		LEI 120
55 262	036 026	**	LDI 026
55 264	106 332 002		CAL STRCP
55 267	150 312 055	71	JTZ ARRAY7
55 272	066 207		LLI 207
55 274	056 026	**	LHI 026
55 276	307		LAM
55 277	066 075		LLI 075
55 301	056 027	**	LHI 027
55 303	277		CPM

55 304	110 240 055	JFZ ARRAY6
55 307	104 172 007	JMP FAERR
55 312	106 224 003	ARRAY7, CAL EVAL
55 315	106 000 020	CAL FPFIX
55 320	066 207	LLI 207
55 322	056 026	LHI 026
55 324	317	LBM
55 325	026 002	LCI 002
55 327	066 114	LLI 114
55 331	056 027	LHI 027
55 333	106 230 007	CAL TABADR
55 336	060	INL
55 337	060	INL
55 340	327	LCM
55 341	066 124	LLI 124
55 343	056 001	LHI 001
55 345	307	LAM
55 346	024 001	SUI 001
55 350	002	RLC
55 351	002	RLC
55 352	202	ADC
55 353	066 204	LLI 204
55 355	056 027	LHI 027
55 357	370	LMA
55 360	066 201	LLI 201
55 362	076 377	LMI 377
55 364	007	RET
55 365	106 255 002	DIM, CAL CLESYM
55 370	066 202	LLI 202
55 372	317	LBM
55 373	010	INB
55 374	066 203	LLI 203
55 376	371	LMB
55 377	066 203	DIM1, LLI 203
56 001	106 240 002	CAL GETCHR
56 004	150 017 056	JTZ DIM2
56 007	074 250	CPI 250
56 011	150 032 056	JTZ DIM3
56 014	106 310 002	CAL CONCTS
56 017	066 203	DIM2, LLI 203
56 021	106 003 003	CAL LOOP
56 024	110 377 055	JFZ DIM1
56 027	104 337 056	JMP DIMERR
56 032	066 206	DIM3, LLI 206
56 034	076 000	LMI 000

CAL LOOP	3 206		DIM4, LLI 206
JFZ DIM9	3 026	**	LHI 026
JMP NXTLIN	7		LAM
	2		RLC
	2		RLC
LLI 203	4 114		ADI 114
LBM	5 027	**	LHI 027
LLI 202	9		LLA
LMB	5 120		LEI 120
JMP DIM	5 026	**	LDI 026
LAI 304	5 332 002		CAL STRCP
LCI 305	0 301 056		JTZ DIM9
JMP ERROR	6 206		LLI 206
	6 026	**	LHI 026
	7		LBM
	10		INB
	71		LMB
	66 075		LLI 075
	6 027	**	LHI 027
	7		LAM
	1		DCB
	1		CPB
	0 036 056 72		JFZ DIM4
	6 075		LLI 075
	6 027	**	LHI 027
	7		LBM
	0		INB
	1		LMB
	6 076		LLI 076
	1		LMB
	6 206		LLI 206
*136	6 026	**	LHI 026
	1		LMB
	7		LAM
	2		RLC
	2		RLC
VISIGER	4 114		ADI 114
	0		LEA
	6 027	**	LDI 027
	6 120		LLI 120
	6 026	**	LHI 026
	6 046 012		CAL MOVEC
	6 255 002		CAL CLESYM
	6 203		LLI 203
	6 026	**	LHI 026
	7		LBM
	0		INB
	6 204		LLI 204
	71		LMB
	6 204		DIM5, LLI 204

56 161	106 240 002	CAL GETCHR
56 164	150 211 056	JTZ DIM6
56 167	074 251	CPI 251
56 171	150 224 056	JTZ DIM7
56 174	074 260	CPI 260
56 176	160 337 056	JTS DIMERR
56 201	074 272	CPI 272
56 203	120 337 056	JFS DIMERR
56 206	106 310 002	CAL CONCTS
56 211	066 204	DIM6, LLI 204
56 213	106 003 003	CAL LOOP
56 216	110 157 056	JFZ DIM5
56 221	104 337 056	JMP DIMERR
56 224	066 120	DIM7, LLI 120
56 226	056 026	LHI 026
56 230	106 044 023	CAL DINPUT
56 233	106 000 020	CAL FPFIX
56 236	066 124	LLI 124
56 240	307	LAM
56 241	002	RLC
56 242	002	RLC
56 243	320	LCA
56 244	066 076	LLI 076
56 246	056 027	LHI 027
56 250	307	LAM
56 251	024 001	SUI 001
56 253	002	RLC
56 254	002	RLC
56 255	004 122	ADI 122
56 257	360	LLA
56 260	056 027	LHI 027
56 262	317	LBM
56 263	004 004	ADI 004
56 265	360	LLA
56 266	301	LAB
56 267	202	ADC
56 270	370	LMA
56 271	066 204	DIM8, LLI 204
56 273	056 026	LHI 026
56 275	317	LBM
56 276	066 203	LLI 203
56 300	371	LMB
56 301	066 203	DIM9, LLI 203
56 303	106 240 002	CAL GETCHR
56 306	074 254	CPI 254
56 310	150 326 056	JTZ DIM10
56 313	066 203	LLI 203

56 315	106 003 003	CAL LOOP
56 320	110 301 056 ⁷²	JFZ DIM9
56 323	104 116 013	JMP NXTLIN
56 326	066 203	DIM10, LLI 203
56 330	317	LBM
56 331	066 202	LLI 202
56 333	371	LMB
56 334	104 365 055 ⁷¹	JMP DIM
56 337	006 304	DIMERR, LAI 304
56 341	026 305	LCI 305
⁷² 56 343	104 226 002	JMP ERROR

Note open addresses

to end of page 56

^A CHANGE TO 73 (SCU LOGIC IN 15th)
 Page 57 reserved
 for use by the
 ARRAY VALUES TABLE.

(PAGE 73 IS LAST PAGE IN 15th)

SCR DOES CLEAR THE ARRAY VALUES TABLE

SCR DOES NOT CLEAR THE PROGRAM BUFFER
 ONLY THE FIRST BYTE (033 000)

SCELBAL ASSEMBLED FOR OPERATION IN AN 8080 BASED SYSTEM

This chapter presents an assembled version of SCELBAL for operation in an 8080 based microcomputer. This version may be loaded into a system along with the user provided I/O subroutines to provide the user with SCELBAL capability.

The user may elect, by choosing the proper machine codes at key locations, to load the program as an 8 K version that does not have the optional DIM statement capability. This version of the program will leave room for about 1,250 bytes in the user program buffer. Or, the user may load the program as a 12 K version with DIM capability. (Leaving about 4,500 bytes for program storage.) Alternately, by changing a few specially marked locations, the user may elect to have the program operate in 8 K of memory with DIM capability. However, this version is not recommended because it will leave only about 500 bytes for storage of a high level language user program. (It is mentioned as an option because some prospective users may desire to run small programs that require the DIM capability.) Finally, the user may opt to place the DIM routines (by changing the associated pointers, etc.) in the upper pages of available RAM memory in any system having more than 8 K of memory (such as a 10 K, 16 K, 32 K system) and using the area between the locations used by the main SCELBAL routines and the optional DIM routines as a user program buffer.

The reader who has studied this book to this point should have no difficulty understanding what is involved in selecting the options just mentioned. Many readers may well elect to make other alterations and may, of course, do so at their own discretion. Let it be said, that the version presented is just one way in which the program may be assembled for operation!

The reader should pay careful attention in the following object code listing to all locations marked by a double asterisk (**),

double at sign (@@), or double cross (††). The convention established in the earlier chapters for those special indicators will be reviewed here.

A double asterisk (**) is of importance only to those readers who might elect to change the memory pages used for the storage of pointers, counters, temporary buffers and look-up tables. The pages used for these purposes in the version of SCELBAL presented are pages 01, 26 and 27. Readers who take on the task of re-assigning these pages will probably have elected to completely re-assemble SCELBAL and should be equipped (mentally and with suitable hardware!) to take on such a task.

A double cross (††) denotes an elective value on the part of the user. These locations generally refer to the starting addresses of user provided routines (such as I/O drivers), or the assignment of the starting and ending address of the user program buffer area. (For the version presented the user program buffer is assumed to start on page 33 and end on page 54. The ending address would be changed to page 37 if an 8 K system was being used and the DIM capability left out. Or, page 34 for an 8 K system with DIM capability provided, etc.)

Locations marked with a @@ should be replaced with the machine code for a no-operation instruction, such as LAA, if the user will not be using the optional DIM statement capability. Alternately, some of these locations relating to addressing values would be altered if the user elected to change the storage areas for the DIM and associated array handling subroutines.

It is suggested that user I/O subroutines be placed on page 00 if possible. Alternately, they may be placed in the upper regions of available memory. If this is done, the ending address of the user program buffer should be altered accordingly.

The 8080 object code presented in this chapter was derived from the source listings presented in detail in earlier chapters with one small exception. Since the 8080 CPU requires an area in memory to be set aside as a stack, the start of the EXECutive routine (refer to the appropriate chapter as required) has been altered to include a stack initializing instruction. For the version presented herein, the 8080 stack is initialized to the address: PAGE 32 LOCATION 000, so that the top region of page 31 is used as the stack area. In order to compensate for the insertion of the stack initializing command at the start of the EXECutive routine, and still maintain the same address references for labels between the two versions of the program presented herein (8008 and 8080), a small subroutine was added (at PAGE 31 LOCATION 330). This subroutine simply contains a pointer initializing

command and call to the subroutine TEXTC. This subroutine has been labeled EXECSP in the following listing.

One final word before presenting the object code is in order. Do not attempt to skip over the machine code listings provided for the special pages 01, 26, and 27. The values in the look-up tables must be in memory along with the initial values of many of the locations on those pages when the program is first started. (Those locations where the initial values are irrelevant are denoted by XXX.) The format of the object code listing for these special pages will be slightly different than the rest of the listing in that the mnemonics column will contain comments relating to the use of the locations (since the locations will contain "data" versus actual instructions).

01 000	XXX	Not Assigned
01 001	XXX	Not Assigned
01 002	XXX	Not Assigned
01 003	XXX	Not Assigned
01 004	000	Stores floating point
01 005	000	constant
01 006	100	value +1.0
01 007	001	Not Assigned
01 010	XXX	Not Assigned
01 011	XXX	Not Assigned
01 012	XXX	Not Assigned
01 013	000	Exponent Counter
01 014	000	Stores floating point
01 015	000	number
01 016	000	temporarily
01 017	000	Not Assigned
01 020	XXX	Not Assigned
01 021	XXX	Not Assigned
01 022	XXX	Not Assigned
01 023	XXX	Not Assigned
01 024	000	Stores floating point
01 025	000	constant
01 026	300	value - 1.0
01 027	001	Scratch Pad Area
01 030	000	
.	.	
.	.	
01 047	000	Scratch Pad Area

01 050	001	Stores random number generator constant value
01 051	120	Not Assigned
01 052	162	Not Assigned
01 053	002	Not Assigned
01 054	XXX	Not Assigned
01 055	XXX	Not Assigned
01 056	XXX	Not Assigned
01 057	XXX	Not Assigned
01 060	003	Stores random number generator constant value
01 061	150	Scratch Pad Area
01 062	157	Scratch Pad Area
01 063	014	Scratch Pad Area
01 064	000	Scratch Pad Area
01 077	000	Scratch Pad Area
01 100	000	Sign Indicator
01 101	000	Sign Indicator
01 102	000	Bits Counter
01 103	000	Sign Indicator
01 104	000	Sign Indicator
01 105	000	Input Digit Counter
01 106	000	Temp Storage
01 107	000	Output Digit Counter
01 110	000	FP Mode Indicator
01 111	XXX	Not Assigned
01 117	XXX	Not Assigned
01 120	000	FPACC Extension
01 121	000	FPACC Extension
01 122	000	FPACC Extension
01 123	000	FPACC Extension
01 124	000	FPACC LSW
01 125	000	FPACC NSW
01 126	000	FPACC MSW
01 127	000	FPACC Exponent
01 130	000	FPOP Extension
01 131	000	FPOP Extension
01 132	000	FPOP Extension
01 133	000	FPOP Extension
01 134	000	FPOP LSW
01 135	000	FPOP NSW
01 136	000	FPOP MSW
01 137	000	FPOP Exponent
01 140	000	Floating point working area
01 167	000	Floating point working area
01 170	XXX	Not Assigned
01 177	XXX	Not Assigned
01 200	000	Temporary

01 201	000	register
01 202	000	storage
01 203	000	area (D, E, H & L)
01 204	XXX	Not Assigned
01 205	XXX	Not Assigned
01 206	XXX	Not Assigned
01 207	XXX	Not Assigned
01 210	000	Stores floating
01 211	000	point
01 212	120	constant
01 213	004	value +10.0
01 214	147	Stores floating
01 215	146	point
01 216	146	constant
01 217	375	value +0.1
01 220	000	GETINP Counter
01 221	XXX	Not Assigned
01 222	XXX	Not Assigned
01 223	XXX	Not Assigned
01 224	XXX	Not Assigned
01 225	XXX	Not Assigned
01 226	XXX	Not Assigned
01 227	000	Arithmetic Stack Pointer
01 230	000	Arithmetic Stack
01 277	000	Arithmetic Stack
01 300	000	FPACC
01 301	000	temporary
01 302	000	storage
01 303	000	location
01 304	000	STEP value
01 305	000	temporary
01 306	000	storage
01 307	000	location
01 310	000	FOR/NEXT Limit
01 311	000	temporary
01 312	000	storage
01 313	000	location
01 314	000	Array pointer
01 315	000	temporary
01 316	000	storage
01 317	000	location

Executive & special messages
look-up table and storage area.

01 320	004	(cc) for THEN
01 321	324	T

01 322	310	H
01 323	305	E
01 324	316	N
01 325	002	(cc) for TO
01 326	324	T
01 327	317	O
01 330	004	(cc) for STEP
01 331	323	S
01 332	324	T
01 333	305	E
01 334	320	P
01 335	004	(cc) for LIST
01 336	314	L
01 337	311	I
01 340	323	S
01 341	324	T
01 342	003	(cc) for RUN
01 343	322	R
01 344	325	U
01 345	316	N
01 346	003	(cc) for SCR
01 347	323	S
01 350	303	C
01 351	322	R
01 352	013	(cc) for READY message
01 353	224	Ctrl T
01 354	215	Carriage-return
01 355	212	Line-feed
01 356	322	R
01 357	305	E
01 360	301	A
01 361	304	D
01 362	331	Y
01 363	215	Carriage-return
01 364	212	Line-feed
01 365	212	Line-feed
01 366	011	(cc) for AT LINE message
01 367	240	Space
01 370	301	A
01 371	324	T
01 372	240	Space
01 373	314	L
01 374	311	I
01 375	316	N
01 376	305	E
01 377	240	Space

End of page 01.

02 000	315 255 002		SYNTAX,	CAL CLESYM
02 003	056 340			LLI 340
02 005	046 026	**		LHI 026
02 007	066 000			LMI 000
02 011	056 201			LLI 201
02 013	066 001			LMI 001
02 015	056 201		SYNTX1,	LLI 201
02 017	315 240 002			CAL GETCHR
02 022	312 044 002			JTZ SYNTX2
02 025	376 260			CPI 260
02 027	372 061 002			JTS SYNTX3
02 032	376 272			CPI 272
02 034	362 061 002			JFS SYNTX3
02 037	056 340			LLI 340
02 041	315 314 002			CAL CONCT1
02 044	056 201		SYNTX2,	LLI 201
02 046	315 003 003			CAL LOOP
02 051	302 015 002			JFZ SYNTX1
02 054	056 203			LLI 203
02 056	066 000			LMI 000
02 060	311			RET
02 061	056 201		SYNTX3,	LLI 201
02 063	106			LBM
02 064	056 202			LLI 202
02 066	160			LMB
02 067	056 202		SYNTX4,	LLI 202
02 071	315 240 002			CAL GETCHR
02 074	312 171 002			JTZ SYNTX6
02 077	376 275			CPI 275
02 101	312 210 002			JTZ SYNTX7
02 104	376 250			CPI 250
02 106	312 215 002			JTZ SYNTX8
02 111	315 310 002			CAL CONCTS
02 114	056 203			LLI 203
02 116	066 001			LMI 001
02 120	046 027	**		LHI 027
02 122	056 000			LLI 000
02 124	026 026	**	SYNTX5,	LDI 026
02 126	036 120			LEI 120
02 130	315 332 002			CAL STRCP
02 133	310			RTZ
02 134	315 356 022			CAL SWITCH
02 137	054		SYNTXL,	INL
02 140	176			LAM
02 141	346 300			NDI 300

02 143	302 137 002	JFZ SYNTXL
02 146	315 356 022	CAL SWITCH
02 151	056 203	LLI 203
02 153	046 026	LHI 026
02 155	106	LBM
02 156	004	INB
02 157	160	LMB
02 160	315 356 022	CAL SWITCH
02 163	170	LAB
02 164	376 015	CPI 015
02 166	302 124 002	JFZ SYNTX5
02 171	056 202	SYNTX6, LLI 202
02 173	046 026	** LHI 026
02 175	315 003 003	CAL LOOP
02 200	302 067 002	JFZ SYNTX4
02 203	056 203	LLI 203
02 205	066 377	LMI 377
02 207	311	RET
02 210	056 203	SYNTX7, LLI 203
02 212	066 015	LMI 015
02 214	311	RET
02 215	056 203	SYNTX8, LLI 203
02 217	066.016	LMI.016
02 221	311	RET
02 222	076 302	BIGERR, LAI 302
02 224	016 307	LCI 307
02 226	315 202 003	ERROR, CAL ECHO
02 231	171	LAC
02 232	315 202 003	CAL ECHO
02 235	303 322 012	JMP FINERR
02 240	176	GETCHR, LAM
02 241	376 120	CPI 120
02 243	362 222 002	JFS BIGERR
02 246	157	LLA
02 247	046 026	** LHI 026
02 251	176	LAM
02 252	376 240	CPI 240
02 254	311	RET
02 255	056 120	CLESYM, LLI 120
02 257	046 026	** LHI 026
02 261	066 000	LMI 000
02 263	311	RET
02 264	376 301	CONCTA, CPI 301

02 266	372 276 002		JTS CONCTN
02 271	376 333		CPI 333
02 273	372 310 002		JTS CONCTS
02 276	376 260	CONCTN,	CPI 260
02 300	372 327 002		JTS CONCTE
02 303	376 272		CPI 272
02 305	362 327 002		JFS CONCTE
02 310	056 120	CONCTS,	LLI 120
02 312	046 026	**	LHI 026
02 314	116	CONCT1,	LCM
02 315	014		INC
02 316	161		LMC
02 317	107		LBA
02 320	315 036 023		CAL INDEXC
02 323	160		LMB
02 324	076 000		LAI 000
02 326	311		RET
02 327	303 152 011	CONCTE,	JMP SYNERR
02 332	176	STRCP,	LAM
02 333	315 356 022		CAL SWITCH
02 336	106		LBM
02 337	270		CPB
02 340	300		RFZ
02 341	315 356 022		CAL SWITCH
02 344	315 377 002	STRCPL,	CAL ADV
02 347	176		LAM
02 350	315 356 022		CAL SWITCH
02 353	315 377 002		CAL ADV
02 356	276	STRCPE,	CPM
02 357	300		RFZ
02 360	315 356 022		CAL SWITCH
02 363	005		DCB
02 364	302 344 002		JFZ STRCPL
02 367	311		RET
02 370	176	STRCPC,	LAM
02 371	315 356 022		CAL SWITCH
02 374	303 356 002		JMP STRCPE
02 377	054	ADV,	INL
03 000	300		RFZ
03 001	044		INH
03 002	311		RET

03 003	106	LOOP,	LBM
03 004	004		INB
03 005	160		LMB
03 006	056 000		LLI 000
03 010	176		LAM
03 011	005		DCB
03 012	270		CPB
03 013	311		RET
03 014	016 000	STRIN,	LCI 000
03 016	315 221 003	STRIN1,	CAL CINPUT
03 021	376 377		CPI 377
03 023	302 045 003		JFZ NOTDEL
03 026	076 334		LAI 334
03 030	315 202 003		CAL ECHO
03 033	015		DCC
03 034	372 014 003		JTS STRIN
03 037	315 164 003		CAL DEC
03 042	303 016 003		JMP STRIN1
03 045	376 203	NOTDEL,	CPI 203
03 047	312 313 012		JTZ CTRL C
03 052	376 215		CPI 215
03 054	312 102 003		JTZ STRINF
03 057	376 212		CPI 212
03 061	312 016 003		JTZ STRIN1
03 064	315 377 002		CAL ADV
03 067	014		INC
03 070	167		LMA
03 071	171		LAC
03 072	376 120		CPI 120
03 074	362 222 002		JFS BIGERR
03 077	303 016 003		JMP STRIN1
03 102	101	STRINF,	LBC
03 103	315 113 003		CAL SUBHL
03 106	161		LMC
03 107	315 141 003		CAL CRLF
03 112	311		RET
03 113	175	SUBHL,	LAL
03 114	220		SUB
03 115	157		LLA
03 116	320		RFC
03 117	045		DCH
03 120	311		RET
03 121	116	TEXTC,	LCM
03 122	176		LAM
03 123	247		NDA

03 124	310		RTZ
03 125	315 377 002	TEXTCL,	CAL ADV
03 130	176		LAM
03 131	315 202 003		CAL ECHO
03 134	015		DCC
03 135	302 125 003		JFZ TEXTCL
03 140	311		RET
03 141	076 215	CRLF,	LAI 215
03 143	315 202 003		CAL ECHO
03 146	076 212		LAI 212
03 150	315 202 003		CAL ECHO
03 153	056 043		LLI 043
03 155	046 001	**	LHI 001
03 157	066 001		LMI 001
03 161	142		LHD
03 162	153		LLE
03 163	311		RET
03 164	055	DEC,	DCL
03 165	054		INL
03 166	302 172 003		JFZ DECN0
03 171	045		DCH
03 172	055	DECN0,	DCL
03 173	311		RET
03 174	175	INDEXB,	LAL
03 175	200		ADB
03 176	157		LLA
03 177	320		RFC
03 200	044		INH
03 201	311		RET
03 202	124	ECHO,	LDH
03 203	135		LEL
03 204	056 043		LLI 043
03 206	046 001	**	LHI 001
03 210	106		LBM
03 211	004		INB
03 212	160		LMB
03 213	315 ††† †††	††	CAL ††† †††
03 216	142		LHD
03 217	153		LLE
03 220	311		RET
03 221	303 ††† †††	††	CINPUT, JMP ††† †††
03 224	056 227	EVAL,	LLI 227
03 226	046 001	**	LHI 001

03 230	066 224		LMI 224
03 232	054		INL
03 233	046 026	**	LHI 026
03 235	066 000		LMI 000
03 237	315 255 002		CAL CLESYM
03 242	056 210		LLI 210
03 244	066 000		LMI 000
03 246	056 276		LLI 276
03 250	106		LBM
03 251	056 200		LLI 200
03 253	160		LMB
03 254	056 200		SCAN1, LLI 200
03 256	315 240 002		CAL GETCHR
03 261	312 301 004		JTZ SCAN10
03 264	376 253		CPI 253
03 266	302 300 003		JFZ SCAN2
03 271	056 176		LLI 176
03 273	066 001		LMI 001
03 275	303 351 003		JMP SCANFN
03 300	376 255		SCAN2, CPI 255
03 302	302 357 003		JFZ SCAN4
03 305	056 120		LLI 120
03 307	176		LAM
03 310	247		NDA
03 311	302 345 003		JFZ SCAN3
03 314	056 176		LLI 176
03 316	176		LAM
03 317	376 007		CPI 007
03 321	312 345 003		JTZ SCAN3
03 324	376 003		CPI 003
03 326	312 152 011		JTZ SYNERR
03 331	376 005		CPI 005
03 333	312 152 011		JTZ SYNERR
03 336	056 120		LLI 120
03 340	066 001		LMI 001
03 342	054		INL
03 343	066 260		LMI 260
03 345	056 176		SCAN3, LLI 176
03 347	066 002		LMI 002
03 351	315 324 004		SCANFN, CAL PARSER
03 354	303 301 004		JMP SCAN10
03 357	376 252		SCAN4, CPI 252
03 361	302 373 003		JFZ SCAN5
03 364	056 176		LLI 176
03 366	066 003		LMI 003
03 370	303 351 003		JMP SCANFN

03 373	376 257	SCAN5,	CPI 257
03 375	302 007 004		JFZ SCAN6
04 000	056 176		LLI 176
04 002	066 004		LMI 004
04 004	303 351 003		JMP SCANFN
04 007	376 250	SCAN6,	CPI 250
04 011	302 033 004		JFZ SCAN7
04 014	056 230		LLI 230
04 016	106		LBM
04 017	004		INB
04 020	160		LMB
04 021	315 100 007		CAL FUNARR
04 024	056 176		LLI 176
04 026	066 006		LMI 006
04 030	303 351 003		JMP SCANFN
04 033	376 251	SCAN7,	CPI 251
04 035	302 064 004		JFZ SCAN8
04 040	056 176		LLI 176
04 042	066 007		LMI 007
04 044	315 324 004		CAL PARSER
04 047	315 003 007		CAL PRIGHT
04 052	056 230		LLI 230
04 054	046 026	**	LHI 026
04 056	106		LBM
04 057	005		DCB
04 060	160		LMB
04 061	303 301 004		JMP SCAN10
04 064	376 336	SCAN8,	CPI 336
04 066	302 100 004		JFZ SCAN9
04 071	056 176		LLI 176
04 073	066 005		LMI 005
04 075	303 351 003		JMP SCANFN
04 100	376 274	SCAN9,	CPI 274
04 102	302 143 004		JFZ SCAN11
04 105	056 200		LLI 200
04 107	106		LBM
04 110	004		INB
04 111	160		LMB
04 112	315 240 002		CAL GETCHR
04 115	376 275		CPI 275
04 117	312 251 004		JTZ SCAN13
04 122	376 276		CPI 276
04 124	312 267 004		JTZ SCAN15
04 127	056 200		LLI 200
04 131	106		LBM
04 132	005		DCB
04 133	160		LMB

04 134	056 176	LLI 176
04 136	066 011	LMI 011
04 140	303 351 003	JMP SCANFN
04 143	376 275	SCAN11, CPI 275
04 145	302 206 004	JFZ SCAN12
04 150	056 200	LLI 200
04 152	106	LBM
04 153	004	INB
04 154	160	LMB
04 155	315 240 002	CAL GETCHR
04 160	376 274	CPI 274
04 162	312 251 004	JTZ SCAN13
04 165	376 276	CPI 276
04 167	312 260 004	JTZ SCAN14
04 172	056 200	LLI 200
04 174	106	LBM
04 175	005	DCB
04 176	160	LMB
04 177	056 176	LLI 176
04 201	066 012	LMI 012
04 203	303 351 003	JMP SCANFN
04 206	376 276	SCAN12, CPI 276
04 210	302 276 004	JFZ SCAN16
04 213	056 200	LLI 200
04 215	106	LBM
04 216	004	INB
04 217	160	LMB
04 220	315 240 002	CAL GETCHR
04 223	376 274	CPI 274
04 225	312 267 004	JTZ SCAN15
04 230	376 275	CPI 275
04 232	312 260 004	JTZ SCAN14
04 235	056 200	LLI 200
04 237	106	LBM
04 240	005	DCB
04 241	160	LMB
04 242	056 176	LLI 176
04 244	066 013	LMI 013
04 246	303 351 003	JMP SCANFN
04 251	056 176	SCAN13, LLI 176
04 253	066 014	LMI 014
04 255	303 351 003	JMP SCANFN
04 260	056 176	SCAN14, LLI 176
04 262	066 015	LMI 015
04 264	303 351 003	JMP SCANFN
04 267	056 176	SCAN15, LLI 176

04 271	066 016	LMI 016
04 273	303 351 003	JMP SCANFN
04 276	315 310 002	SCAN16, CAL CONCTS
04 301	056 200	SCAN10, LLI 200
04 303	046 026	** LHI 026
04 305	106	LBM
04 306	004	INB
04 307	160	LMB
04 310	056 277	LLI 277
04 312	176	LAM
04 313	005	DCB
04 314	270	CPB
04 315	302 254 003	JFZ SCAN1
04 320	303 300 031	JMP PARSEP
04 323	166	HLT
04 324	056 120	PARSER, LLI 120
04 326	046 026	** LHI 026
04 330	176	LAM
04 331	247	NDA
04 332	312 231 005	JTZ PARSE
04 335	054	INL
04 336	176	LAM
04 337	376 256	CPI 256
04 341	312 356 004	JTZ PARNUM
04 344	376 260	CPI 260
04 346	372 033 005	JTS LOOKUP
04 351	376 272	CPI 272
04 353	362 033 005	JFS LOOKUP
04 356	055	PARNUM, DCL
04 357	176	LAM
04 360	376 001	CPI 001
04 362	312 005 005	JTZ NOEXPO
04 365	205	ADL
04 366	157	LLA
04 367	176	LAM
04 370	376 305	CPI 305
04 372	302 005 005	JFZ NOEXPO
04 375	056 200	LLI 200
04 377	315 240 002	CAL GETCHR
05 002	303 310 002	JMP CONCTS
05 005	056 227	NOEXPO, LLI 227
05 007	046 001	** LHI 001
05 011	176	LAM
05 012	306 004	ADI 004
05 014	167	LMA
05 015	157	LLA

05 016	315 255 022		CAL FSTORE
05 021	056 120		LLI 120
05 023	046 026	**	LHI 026
05 025	315 044 023		CAL DINPUT
05 030	303 231 005		JMP PARSE
05 033	056 370		LOOKUP, LLI 370
05 035	046 026	**	LHI 026
05 037	066 000		LMI 000
05 041	056 120		LLI 120
05 043	026 027	**	LDI 027
05 045	036 210		LEI 210
05 047	176		LAM
05 050	376 001		CPI 001
05 052	302 061 005		JFZ LOOKU1
05 055	056 122		LLI 122
05 057	066 000		LMI 000
05 061	056 121		LOOKU1, LLI 121
05 063	046 026	**	LHI 026
05 065	315 356 022		CAL SWITCH
05 070	176		LAM
05 071	054		INL
05 072	106		LBM
05 073	054		INL
05 074	315 356 022		CAL SWITCH
05 077	276		CPM
05 100	302 111 005		JFZ LOOKU2
05 103	054		INL
05 104	170		LAB
05 105	276		CPM
05 106	312 201 005		JTZ LOOKU4
05 111	315 256 006		LOOKU2, CAL AD4DE
05 114	056 370		LLI 370
05 116	046 026	**	LHI 026
05 120	106		LBM
05 121	004		INB
05 122	160		LMB
05 123	056 077		LLI 077
05 125	046 027	**	LHI 027
05 127	170		LAB
05 130	276		CPM
05 131	302 061 005		JFZ LOOKU1
05 134	056 077		LLI 077
05 136	046 027	**	LHI 027
05 140	106		LBM
05 141	004		INB
05 142	160		LMB
05 143	170		LAB
05 144	376 025		CPI 025

05 146	362 222 002		JFS BIGERR
05 151	056 121		LLI 121
05 153	046 026	**	LHI 026
05 155	006 002		LBI 002
05 157	315 013 021		CAL MOVEIT
05 162	153		LLE
05 163	142		LHD
05 164	257		XRA
05 165	167		LMA
05 166	054		INL
05 167	167		LMA
05 170	054		INL
05 171	167		LMA
05 172	054		INL
05 173	167		LMA
05 174	175		LAL
05 175	326 004		SUI 004
05 177	137		LEA
05 200	124		LDH
05 201	315 317 022		LOOKU4, CAL SAVEHL
05 204	056 227		LLI 227
05 206	046 001	**	LHI 001
05 210	176		LAM
05 211	306 004		ADI 004
05 213	167		LMA
05 214	157		LLA
05 215	315 255 022		CAL FSTORE
05 220	315 337 022		CAL RESTHL
05 223	315 356 022		CAL SWITCH
05 226	315 244 022		CAL FLOAD
05 231	315 255 002		PARSE, CAL CLESYM
05 234	056 176		LLI 176
05 236	176		LAM
05 237	376 007		CPI 007
05 241	312 332 005		JTZ PARSE2
05 244	306 240		ADI 240
05 246	157		LLA
05 247	106		LBM
05 250	056 210		LLI 210
05 252	116		LCM
05 253	315 036 028		CAL INDEXC
05 256	176		LAM
05 257	306 257		ADI 257
05 261	157		LLA
05 262	170		LAB
05 263	276		CPM
05 264	312 307 005		JTZ PARSE1
05 267	372 307 005		JTS PARSE1
05 272	056 176		LLI 176

05 274	106		LBM
05 275	056 210		LLI 210
05 277	116		LCM
05 300	014		INC
05 301	161		LMC
05 302	315 036 023		CAL INDEXC
05 305	160		LMB
05 306	311		RET
05 307	056 210	PARSE1,	LLI 210
05 311	176		LAM
05 312	205		ADL
05 313	157		LLA
05 314	176		LAM
05 315	247		NDA
05 316	310		RTZ
05 317	056 210		LLI 210
05 321	116		LCM
05 322	015		DCC
05 323	161		LMC
05 324	315 364 005		CAL FPOPER
05 327	303 231 005		JMP PARSE
05 332	056 210	PARSE2,	LLI 210
05 334	046 026	**	LHI 026
05 336	176		LAM
05 337	205		ADL
05 340	157		LLA
05 341	176		LAM
05 342	247		NDA
05 343	312 104 006		JTZ PARNER
05 346	056 210		LLI 210
05 350	116		LCM
05 351	015		DCC
05 352	161		LMC
05 353	376 006		CPI 006
05 355	310		RTZ
05 356	315 364 005		CAL FPOPER
05 361	303 332 005		JMP PARSE2
05 364	056 371	FPOPER,	LLI 371
05 366	046 026	**	LHI 026
05 370	167		LMA
05 371	056 227		LLI 227
05 373	046 001	**	LHI 001
05 375	176		LAM
05 376	157		LLA
05 377	315 266 022		CAL OPLOAD
06 002	056 227		LLI 227
06 004	176		LAM
06 005	326 004		SUI 004

06 007	167		LMA
06 010	056 371		LLI 371
06 012	046 026	**	LHI 026
06 014	176		LAM
06 015	376 001		CPI 001
06 017	312 211 020		JTZ FPADD
06 022	376 002		CPI 002
06 024	312 032 021		JTZ FPSUB
06 027	376 003		CPI 003
06 031	312 046 021		JTZ FPMULT
06 034	376 004		CPI 004
06 036	312 322 021		JTZ FPDIV
06 041	376 005		CPI 005
06 043	312 263 006		JTZ INTEXP
06 046	376 011		CPI 011
06 050	312 121 006		JTZ LT
06 053	376 012		CPI 012
06 055	312 136 006		JTZ EQ
06 060	376 013		CPI 013
06 062	312 153 006		JTZ GT
06 065	376 014		CPI 014
06 067	312 173 006		JTZ LE
06 072	376 015		CPI 015
06 074	312 213 006		JTZ GE
06 077	376 016		CPI 016
06 101	312 230 006		JTZ NE
06 104	056 230		PARNER, LLI 230
06 106	046 026	**	LHI 026
06 110	066 000		LMI 000
06 112	076 311		LAI 311
06 114	016 250		LCI 250
06 116	303 226 002		JMP ERROR
06 121	315 032 021		LT, CAL FPSUB
06 124	056 126		LLI 126
06 126	176		LAM
06 127	247		NDA
06 130	372 242 006		JTS CTRUE
06 133	303 247 006		JMP CFALSE
06 136	315 032 021		EQ, CAL FPSUB
06 141	056 126		LLI 126
06 143	176		LAM
06 144	247		NDA
06 145	312 242 006		JTZ CTRUE
06 150	303 247 006		JMP CFALSE
06 153	315 032 021		GT, CAL FPSUB
06 156	056 126		LLI 126
06 160	176		LAM
06 161	247		NDA

06 162	312 247 006	JTZ CFALSE
06 165	362 242 006	JFS CTRUE
06 170	303 247 006	JMP CFALSE
06 173	315 032 021	LE, CAL FPSUB
06 176	056 126	LLI 126
06 200	176	LAM
06 201	247	NDA
06 202	312 242 006	JTZ CTRUE
06 205	372 242 006	JTS CTRUE
06 210	303 247 006	JMP CFALSE
06 213	315 032 021	GE, CAL FPSUB
06 216	056 126	LLI 126
06 220	176	LAM
06 221	247	NDA
06 222	362 242 006	JFS CTRUE
06 225	303 247 006	JMP CFALSE
06 230	315 032 021	NE, CAL FPSUB
06 233	056 126	LLI 126
06 235	176	LAM
06 236	247	NDA
06 237	312 247 006	JTZ CFALSE
06 242	056 004	CTRUE, FPONE, LLI 004
06 244	303 244 022	JMP FLOAD
06 247	056 127	CFALSE, LLI 127
06 251	066 000	LMI 000
06 253	303 051 020	JMP FPZERO
06 256	173	AD4DE, LAE
06 257	306 004	ADI 004
06 261	137	LEA
06 262	311	RET
06 263	056 126	INTEXP, LLI 126
06 265	046.001	** LHI 001
06 267	176	LAM
06 270	056 003	LLI 003
06 272	167	LMA
06 273	247	NDA
06 274	312 242 006	JTZ FPONE
06 277	374 202 020	CTS FPCOMP
06 302	315 000 020	CAL FPFIX
06 305	056 124	LLI 124
06 307	106	LBM
06 310	056 013	LLI 013
06 312	160	LMB
06 313	056 134	LLI 134

06 315	036 014		LEI 014	
06 317	046 001	**	LHI 001	
06 321	124		LDH	
06 322	006 004		LBI 004	
06 324	315 013 021		CAL MOVEIT	
06 327	315 242 006		CAL FPONE	
06 332	056 003		LLI 003	
06 334	176		LAM	
06 335	247		NDA	
06 336	372 362 006		JTS DVLOOP	
06 341	056 014		MULLOOP,	LLI 014
06 343	315 277 022		CAL FACXOP	
06 346	315 046 021		CAL FPMULT	
06 351	056 013		LLI 013	
06 353	106		LBM	
06 354	005		DCB	
06 355	160		LMB	
06 356	302 341 006		JFZ MULLOOP	
06 361	311		RET	
06 362	056 014		DVLOOP,	LLI 014
06 364	315 277 022		CAL FACXOP	
06 367	315 322 021		CAL FPDIV	
06 372	056 013		LLI 013	
06 374	106		LBM	
06 375	005		DCB	
06 376	160		LMB	
06 377	302 362 006		JFZ DVLOOP	
07 002	311		RET	
07 003	056 230		PRIGHT,	LLI 230
07 005	046 026	**	LHI 026	
07 007	176		LAM	
07 010	205		ADL	
07 011	157		LLA	
07 012	176		LAM	
07 013	066 000		LMI 000	
07 015	056 203		LLI 203	
07 017	046 027	**	LHI 027	
07 021	167		LMA	
07 022	247		NDA	
07 023	310		RTZ	
07 024	372 000 055	@@	JTS PRIGH1	
07 027	376 001		CPI 001	
07 031	312 243 007		JTZ INTX	
07 034	376 002		CPI 002	
07 036	312 360 007		JTZ SGNX	
07 041	376 003		CPI 003	
07 043	312 346 007		JTZ ABSX	
07 046	376 004		CPI 004	

07 050	312 000 032		JTZ SQRX	
07 053	376 005		CPI 005	
07 055	312 017 010		JTZ TABX	
07 060	376 006		CPI 006	
07 062	312 240 032		JTZ RNDX	
07 065	376 007		CPI 007	
07 067	312 377 007		JTZ CHRX	
07 072	376 010		CPI 010	
07 074	312 ††† †††	††	JTZ ††† †††	
07 077	166		HLT	
07 100	056 120		FUNARR,	LLI 120
07 102	046 026	**		LHI 026
07 104	176			LAM
07 105	247			NDA
07 106	310			RTZ
07 107	056 202			LLI 202
07 111	046 027	**		LHI 027
07 113	066 000			LMI 000
07 115	056 202		FUNAR1,	LLI 202
07 117	046 027	**		LHI 027
07 121	106			LBM
07 122	004			INB
07 123	160			LMB
07 124	016 002			LCI 002
07 126	056 274			LLI 274
07 130	046 026	**		LHI 026
07 132	315 230 007			CAL TABADR
07 135	026 026	**		LDI 026
07 137	036 120			LEI 120
07 141	315 332 002			CAL STRCP
07 144	312 207 007			JTZ FUNAR4
07 147	056 202			LLI 202
07 151	046 027	**		LHI 027
07 153	176			LAM
07 154	376 010			CPI 010
07 156	302 115 007			JFZ FUNAR1
07 161	056 202			LLI 202
07 163	046 027	**		LHI 027
07 165	066 000			LMI 000
07 167	303 054 055	@@		JMP FUNAR2
07 172	056 230		FAERR,	LLI 230
07 174	046 026	**		LHI 026
07 176	066 000			LMI 000
07 200	076 306			LAI 306
07 202	016 301			LCI 301
07 204	303 226 002			JMP ERROR
07 207	056 202		FUNAR4,	LLI 202

07 211	046 027	**	LHI 027
07 213	106		LBM
07 214	056 230		LLI 230
07 216	046 026	**	LHI 026
07 220	116		LCM
07 221	315 036 023		CAL INDEXC
07 224	160		LMB
07 225	303 255 002		JMP CLESYM
07 230	170	TABADR,	LAB
07 231	007	TABAD1,	RLC
07 232	015		DCC
07 233	302 231 007		JFZ TABAD1
07 236	205		ADL
07 237	157		LLA
07 240	320		RFC
07 241	044		INH
07 242	311		RET
07 243	056 126	INTX,	LLI 126
07 245	046 001	**	LHI 001
07 247	176		LAM
07 250	247		NDA
07 251	362 327 007		JFS INT1
07 254	056 014		LLI 014
07 256	315 255 022		CAL FSTORE
07 261	315 000 020		CAL FPFIX
07 264	056 123		LLI 123
07 266	066 000		LMI 000
07 270	315 064 020		CAL FPFLT
07 273	056 014		LLI 014
07 275	315 266 022		CAL OPLOAD
07 300	315 032 021		CAL FPSUB
07 303	056 126		LLI 126
07 305	176		LAM
07 306	247		NDA
07 307	312 341 007		JTZ INT2
07 312	056 014		LLI 014
07 314	315 244 022		CAL FLOAD
07 317	056 024		LLI 024
07 321	315 277 022		CAL FACXOP
07 324	315 211 020		CAL FPADD
07 327	315 000 020	INT1,	CAL FPFIX
07 332	056 123		LLI 123
07 334	066 000		LMI 000
07 336	303 064 020		JMP FPFLT
07 341	056 014	INT2,	LLI 014
07 343	303 244 022		JMP FLOAD

07 346	056 126		ABSX,	LLI 126
07 350	046 001	**		LHI 001
07 352	176			LAM
07 353	247			NDA
07 354	372 202 020			JTS FPCOMP
07 357	311			RET
07 360	056 126		SGNX,	LLI 126
07 362	046 001	**		LHI 001
07 364	176			LAM
07 365	247			NDA
07 366	310			RTZ
07 367	362 242 006			JFS FPONE
07 372	056 024			LLI 024
07 374	303 244 022			JMP FLOAD
07 377	315 000 020		CHRX,	CAL FPFIX
10 002	056 124			LLI 124
10 004	176			LAM
10 005	315 202 003			CAL ECHO
10 010	056 177			LLI 177
10 012	046 026	**		LHI 026
10 014	066 377			LMI 377
10 016	311			RET
10 017	315 000 020		TABX,	CAL FPFIX
10 022	056 124		TAB1,	LLI 124
10 024	176			LAM
10 025	056 043			LLI 043
10 027	226			SUM
10 030	056 177			LLI 177
10 032	046 026	**		LHI 026
10 034	066 377			LMI 377
10 036	372 217 031			JTS BACKSP
10 041	310			RTZ
10 042	117		TABC,	LCA
10 043	076 240			LAI 240
10 045	315 202 003		TABLOP,	CAL ECHO
10 050	015			DCC
10 051	302 045 010			JFZ TABLOP
10 054	311			RET
10 055	056 201		STOSYM,	LLI 201
10 057	046 027	**		LHI 027
10 061	176			LAM
10 062	247			NDA
10 063	312 100 010			JTZ STOSY1
10 066	066 000			LMI 000
10 070	056 204			LLI 204
10 072	156			LLM

10 073	046 057	††	LHI 057
10 075	303 255 022		JMP FSTORE
10 100	056 370		STOSY1, LLI 370
10 102	046 026	**	LHI 026
10 104	066 000		LMI 000
10 106	056 120		LLI 120
10 110	026 027	**	LDI 027
10 112	036 210		LEI 210
10 114	176		LAM
10 115	376 001		CPI 001
10 117	302 126 010		JFZ STOSY2
10 122	056 122		LLI 122
10 124	066 000		LMI 000
10 126	056 121		STOSY2, LLI 121
10 130	046 026	**	LHI 026
10 132	315 356 022		CAL SWITCH
10 135	176		LAM
10 136	054		INL
10 137	106		LBM
10 140	054		INL
10 141	315 356 022		CAL SWITCH
10 144	276		CPM
10 145	302 156 010		JFZ STOSY3
10 150	054		INL
10 151	170		LAB
10 152	276		CPM
10 153	312 227 010		JTZ STOSY5
10 156	315 256 006		STOSY3, CAL AD4DE
10 161	056 370		LLI 370
10 163	046 026	**	LHI 026
10 165	106		LBM
10 166	004		INB
10 167	160		LMB
10 170	056 077		LLI 077
10 172	046 027	**	LHI 027
10 174	170		LAB
10 175	276		CPM
10 176	302 126 010		JFZ STOSY2
10 201	056 077		LLI 077
10 203	046 027	**	LHI 027
10 205	106		LBM
10 206	004		INB
10 207	160		LMB
10 210	170		LAB
10 211	376 025		CPI 025
10 213	362 222 002		JFS BIGERR
10 216	056 121		LLI 121
10 220	046 026	**	LHI 026

10 222	006 002		LBI 002
10 224	315 013 021		CAL MOVEIT
10 227	315 356 022		STOSY5, CAL SWITCH
10 232	315 255 022		CAL FSTORE
10 235	303 255 002		JMP CLESYM
10 240	056 120		SAVESY, LLI 120
10 242	046 026	**	LHI 026
10 244	124		LDH
10 245	036 144		LEI 144
10 247	303 261 010		JMP MOVECP
10 252	056 144		RESTSY, LLI 144
10 254	046 026	**	LHI 026
10 256	124		LDH
10 257	036 120		LEI 120
10 261	106		MOVECP, LBM
10 262	004		INB
10 263	303 013 021		JMP MOVEIT
10 266	061 000 032		EXEC, LXS 000 032
10 271	315 330 031		CAL EXECSP
10 274	000		NOP
10 275	056 000		EXEC1, LLI 000
10 277	046 026	**	LHI 026
10 301	315 014 003		CAL STRIN
10 304	176		LAM
10 305	247		NDA
10 306	312 275 010		JTZ EXEC1
10 311	056 335		LLI 335
10 313	046 001	**	LHI 001
10 315	026 026	**	LDI 026
10 317	036 000		LEI 000
10 321	315 332 002		CAL STRCP
10 324	302 354 010		JFZ NOLIST
10 327	056 000		LLI 000
10 331	046 033	††	LHI 033
10 333	176		LIST, LAM
10 334	247		NDA
10 335	312 266 010		JTZ EXEC
10 340	315 121 003		CAL TEXTC
10 343	315 377 002		CAL ADV
10 346	315 141 003		CAL CRLF
10 351	303 333 010		JMP LIST
10 354	056 342		NOLIST, LLI 342
10 356	046 001	**	LHI 001

10 360	036 000		LEI 000
10 362	026 026	**	LDI 026
10 364	036 000		LEI 000
10 366	315 332 002		CAL STRCP
10 371	312 070 013		JTZ RUN
10 374	026 026	**	LDI 026
10 376	036 000		LEI 000
11 000	056 346		LLI 346
11 002	046 001	**	LHI 001
11 004	315 332 002		CAL STRCP
11 007	302 071 011		JFZ NOSCR
11 012	046 026	**	LHI 026
11 014	056 364		LLI 364
11 016	066 033	††	LMI 033
11 020	054		INL
11 021	066 000		LMI 000
11 023	056 077		LLI 077
11 025	046 027	**	LHI 027
11 027	066 001		LMI 001
11 031	056 075		LLI 075
11 033	066 000	@@	LMI 000
11 035	056 120	@@	LLI 120
11 037	066 000	@@	LMI 000
11 041	056 210		LLI 210
11 043	066 000		LMI 000
11 045	054		INL
11 046	066 000		LMI 000
11 050	046 033	††	LHI 033
11 052	056 000		LLI 000
11 054	066 000		LMI 000
11 056	046 057	@@	LHI 057
11 060	066 000	@@	SCRLOP,
11 062	054	@@	LMI 000
11 063	302 060 011	@@	INL
11 066	303 266 010		JFZ SCRLOP
			JMP EXEC
11 071	036 272		NOSCR,
11 073	026 001	**	LEI 272
11 075	046 026	**	LDI 001
11 077	056 000		LHI 026
11 101	315 332 002		LLI 000
11 104	312 ††† †††	††	CAL STRCP
11 107	056 277		JTZ SAVE
11 111	046 001	**	LLI 277
11 113	026 026	**	LHI 001
11 115	036 000		LDI 026
11 117	315 332 002		LEI 000
11 122	312 ††† †††	††	CAL STRCP
11 125	056 360		JTZ LOAD
11 127	046 026	**	LLI 360
			LHI 026

11 131	066 033	††	LMI 033
11 133	054		INL
11 134	066 000		LMI 000
11 136	315 000 002		CAL SYNTAX
11 141	056 203		LLI 203
11 143	046 026	**	LHI 026
11 145	176		LAM
11 146	247		NDA
11 147	362 161 011		JFS SYNTOK
11 152	076 323		SYNERR, LAI 323
11 154	016 331		LCI 331
11 156	303 226 002		JMP ERROR
11 161	056 340		SYNTOK, LLI 340
11 163	176		LAM
11 164	247		NDA
11 165	312 211 013		JTZ DIRECT
11 170	056 360		LLI 360
11 172	066 033	††	LMI 033
11 174	054		INL
11 175	066 000		LMI 000
11 177	056 201		GETAUX, LLI 201
11 201	046 026	**	LHI 026
11 203	066 001		LMI 001
11 205	056 350		LLI 350
11 207	066 000		LMI 000
11 211	056 201		GETAU0, LLI 201
11 213	315 123 012		CAL GETCHP
11 216	312 242 011		JTZ GETAU1
11 221	376 260		CPI 260
11 223	372 267 011		JTS GETAU2
11 226	376 272		CPI 272
11 230	362 267 011		JFS GETAU2
11 233	056 350		LLI 350
11 235	046 026	**	LHI 026
11 237	315 314 002		CAL CONCT1
11 242	056 201		GETAU1, LLI 201
11 244	046 026	**	LHI 026
11 246	106		LBM
11 247	004		INB
11 250	160		LMB
11 251	056 360		LLI 360
11 253	046 026	**	LHI 026
11 255	116		LCM
11 256	054		INL
11 257	156		LLM
11 260	141		LHC

11 261	176		LAM
11 262	005		DCB
11 263	270		CPB
11 264	302 211 011		JFZ GETAU0
11 267	056 360		GETAU2, LLI 360
11 271	046 026	**	LHI 026
11 273	126		LDM
11 274	054		INL
11 275	156		LLM
11 276	142		LHD
11 277	176		LAM
11 300	247		NDA
11 301	302 336 011		JFZ NOTEND
11 304	303 005 012		JMP NOSAME

Note open addresses.
This space available
for patching.

11 336	056 350		NOTEND, LLI 350
11 340	046 026	**	LHI 026
11 342	026 026	**	LDI 026
11 344	036 340		LEI 340
11 346	315 332 002		CAL STRCP
11 351	372 073 012		JTS CONTIN
11 354	302 005 012		JFZ NOSAME
11 357	056 360		LLI 360
11 361	046 026	**	LHI 026
11 363	116		LCM
11 364	054		INL
11 365	156		LLM
11 366	141		LHC
11 367	106		LBM
11 370	004		INB
11 371	315 144 012		CAL REMOVE
11 374	056 203		LLI 203
11 376	046 026	**	LHI 026
12 000	176		LAM
12 001	247		NDA
12 002	312 266 010		JTZ EXEC
12 005	056 360		NOSAME, LLI 360
12 007	046 026	**	LHI 026
12 011	126		LDM
12 012	054		INL
12 013	136		LEM
12 014	056 000		LLI 000
12 016	046 026	**	LHI 026
12 020	106		LBM
12 021	004		INB

12 022	315 205 012		CAL INSERT
12 025	056 360		LLI 360
12 027	046 026	**	LHI 026
12 031	126		LDM
12 032	054		INL
12 033	136		LEM
12 034	056 000		LLI 000
12 036	046 026	**	LHI 026
12 040	315 046 012		CAL MOVEC
12 043	303 275 010		JMP EXEC1
12 046	106		MOVEC, LBM
12 047	004		INB
12 050	176		MOVEPG, LAM
12 051	315 377 002		CAL ADV
12 054	315 356 022		CAL SWITCH
12 057	167		LMA
12 060	315 377 002		CAL ADV
12 063	315 356 022		CAL SWITCH
12 066	005		DCB
12 067	302 050 012		JFZ MOVEPG
12 072	311		RET
12 073	056 360		CONTIN, LLI 360
12 075	046 026	**	LHI 026
12 077	126		LDM
12 100	054		INL
12 101	136		LEM
12 102	142		LHD
12 103	153		LLE
12 104	106		LBM
12 105	004		INB
12 106	315 305 012		CAL ADBDE
12 111	056 360		LLI 360
12 113	046 026	**	LHI 026
12 115	162		LMD
12 116	054		INL
12 117	163		LME
12 120	303 177 011		JMP GETAUX
12 123	046 026	**	GETCHP, LHI 026
12 125	106		LBM
12 126	056 360		LLI 360
12 130	126		LDM
12 131	054		INL
12 132	136		LEM
12 133	315 305 012		CAL ADBDE
12 136	142		LHD
12 137	153		LLE
12 140	176		LAM

12 141	376 240		CPI 240
12 143	311		RET
12 144	315 174 003		REMOVE, CAL INDEXB
12 147	116		LCM
12 150	315 113 003		CAL SUBHL
12 153	161		LMC
12 154	171		LAC
12 155	247		NDA
12 156	312 167 012		JTZ REMOV1
12 161	315 377 002		CAL ADV
12 164	303 144 012		JMP REMOVE
12 167	056 364		REMOV1, LLI 364
12 171	046 026	**	LHI 026
12 173	126		LDM
12 174	054		INL
12 175	176		LAM
12 176	220		SUB
12 177	167		LMA
12 200	320		RFC
12 201	055		DCL
12 202	025		DCD
12 203	162		LMD
12 204	311		RET
12 205	056 364		INSERT, LLI 364
12 207	046 026	**	LHI 026
12 211	176		LAM
12 212	054		INL
12 213	156		LLM
12 214	147		LHA
12 215	315 174 003		CAL INDEXB
12 220	174		LAH
12 221	376 054	††	CPI 054
12 223	362 222 002		JFS BIGERR
12 226	315 113 003		CAL SUBHL
12 231	116		INSER1, LCM
12 232	315 174 003		CAL INDEXB
12 235	161		LMC
12 236	315 113 003		CAL SUBHL
12 241	315 277 012		CAL CPHLDE
12 244	312 255 012		JTZ INSER3
12 247	315 164 003		CAL DEC
12 252	303 231 012		JMP INSER1
12 255	056 000		INSER3, INCLIN, LLI 000
12 257	046 026	**	LHI 026
12 261	106		LBM
12 262	004		INB

12 263	056 364		LLI 364
12 265	126		LDM
12 266	054		INL
12 267	136		LEM
12 270	315 305 012		CAL ADBDE
12 273	163		LME
12 274	055		DCL
12 275	162		LMD
12 276	311		RET
12 277	174	CPHLDE,	LAH
12 300	272		CPD
12 301	300		RFZ
12 302	175		LAL
12 303	273		CPE
12 304	311		RET
12 305	173	ADBDE,	LAE
12 306	200		ADB
12 307	137		LEA
12 310	320		RFC
12 311	024		IND
12 312	311		RET
12 313	076 336	CTRLC,	LAI 336
12 315	016 303		LCI 303
12 317	303 226 002		JMP ERROR
12 322	056 340	FINERR,	LLI 340
12 324	046 026	**	LHI 026
12 326	176		LAM
12 327	247		NDA
12 330	312 351 012		JTZ FINER1
12 333	056 366		LLI 366
12 335	046 001	**	LHI 001
12 337	315 121 003		CAL TEXTC
12 342	056 340		LLI 340
12 344	046 026	**	LHI 026
12 346	315 121 003		CAL TEXTC
12 351	315 141 003	FINER1,	CAL CRLF
12 354	303 266 010		JMP EXEC
12 357	076 304	DVERR,	LAI 304
12 361	016 332		LCI 332
12 363	303 226 002		JMP ERROR
12 366	076 306	FIXERR,	LAI 306
12 370	016 330		LCI 330
12 372	303 226 002		JMP ERROR

12 375	076 311		NUMERR,	LAI 311
12 377	016 316			LCI 316
13 001	056 220			LLI 220
13 003	046 001	**		LHI 001
13 005	066 000			LMI 000
13 007	303 226 002			JMP ERROR
13 012	026 026	**	INSTR,	LDI 026
13 014	036 000			LEI 000
13 016	315 064 013		INSTR1,	CAL ADVDE
13 021	315 317 022			CAL SAVEHL
13 024	106			LBM
13 025	315 377 002			CAL ADV
13 030	315 370 002			CAL STRCPC
13 033	312 337 022			JTZ RESTHL
13 036	315 337 022			CAL RESTHL
13 041	056 000			LLI 000
13 043	046 026	**		LHI 026
13 045	176			LAM
13 046	273			CPE
13 047	312 061 013			JTZ INSTR2
13 052	315 337 022			CAL RESTHL
13 055	303 016 013			JMP INSTR1
13 060	166			HLT
13 061	036 000		INSTR2,	LEI 000
13 063	311			RET
13 064	034		ADVDE,	INE
13 065	300			RFZ
13 066	024			IND
13 067	311			RET
13 070	056 073		RUN,	LLI 073
13 072	046 027	**		LHI 027
13 074	066 000			LMI 000
13 076	056 205			LLI 205
13 100	066 000			LMI 000
13 102	056 360			LLI 360
13 104	046 026	**		LHI 026
13 106	066 033	††		LMI 033
13 110	054			INL
13 111	066 000			LMI 000
13 113	303 156 013			JMP SAMLIN
13 116	056 360		NXTLIN,	LLI 360
13 120	046 026	**		LHI 026
13 122	126			LDM
13 123	054			INL
13 124	136			LEM

13 125	142		LHD
13 126	153		LLE
13 127	106		LBM
13 130	004		INB
13 131	315 305 012		CAL ADBDE
13 134	056 360		LLI 360
13 136	046 026	**	LHI 026
13 140	162		LMD
13 141	054		INL
13 142	163		LME
13 143	056 340		LLI 340
13 145	046 026	**	LHI 026
13 147	176		LAM
13 150	247		NDA
13 151	312 266 010		JTZ EXEC
13 154	000		NOP
13 155	000		NOP
13 156	056 360		SAMLIN, LLI 360
13 160	046 026	**	LHI 026
13 162	116		LCM
13 163	054		INL
13 164	156		LLM
13 165	141		LHC
13 166	026 026	**	LDI 026
13 170	036 000		LEI 000
13 172	315 046 012		CAL MOVEC
13 175	056 000		LLI 000
13 177	046 026	**	LHI 026
13 201	176		LAM
13 202	247		NDA
13 203	312 266 010		JTZ EXEC
13 206	315 000 002		CAL SYNTAX
13 211	056 203		DIRECT, LLI 203
13 213	046 026	**	LHI 026
13 215	176		LAM
13 216	376 001		CPI 001
13 220	312 116 013		JTZ NXTLIN
13 223	376 002		CPI 002
13 225	312 027 016		JTZ IF
13 230	376 003		CPI 003
13 232	312 031 015		JTZ LET
13 235	376 004		CPI 004
13 237	312 174 015		JTZ GOTO
13 242	376 005		CPI 005
13 244	312 345 013		JTZ PRINT
13 247	376 006		CPI 006
13 251	312 365 016		JTZ INPUT
13 254	376 007		CPI 007
13 256	312 164 017		JTZ FOR

13 261	376 010	CPI 010
13 263	312 013 030	JTZ NEXT
13 266	376 011	CPI 011
13 270	312 236 016	JTZ GOSUB
13 273	376 012	CPI 012
13 275	312 304 016	JTZ RETURN
13 300	376 013	CPI 013
13 302	312 365 055	JTZ DIM
13 305	376 014	CPI 014
13 307	312 266 010	JTZ EXEC
13 312	376 015	CPI 015
13 314	312 013 015	JTZ LETO
13 317	376 016	CPI 016
13 321	302 152 011	JFZ SYNERR
13 324	315 153 055	CAL ARRAY1
13 327	056 206	LLI 206
13 331	046 026	LHI 026
13 333	106	LBM
13 334	056 202	LLI 202
13 336	160	LMB
13 337	315 240 010	CAL SAVESY
13 342	303 042 015	JMP LET1
13 345	056 202	PRINT, LLI 202
13 347	046 026	LHI 026
13 351	176	LAM
13 352	056 000	LLI 000
13 354	276	CPM
13 355	372 366 013	JTS PRINT1
13 360	315 141 003	CAL CRLF
13 363	303 116 013	JMP NXTLIN
13 366	315 255 002	PRINT1, CAL CLESYM
13 371	056 202	LLI 202
13 373	046 026	LHI 026
13 375	106	LBM
13 376	004	INB
13 377	056 203	LLI 203
14 001	160	LMB
14 002	056 203	PRINT2, LLI 203
14 004	315 240 002	CAL GETCHR
14 007	376 247	CPI 247
14 011	312 203 014	JTZ QUOTE
14 014	376 242	CPI 242
14 016	312 203 014	JTZ QUOTE
14 021	376 254	CPI 254
14 023	312 043 014	JTZ PRINT3
14 026	376 273	CPI 273
14 030	312 043 014	JTZ PRINT3
14 033	056 203	LLI 203

14 035	315 003 003	CAL LOOP
14 040	302 002 014	JFZ PRINT2
14 043	056 202	PRINT3, LLI 202
14 045	106	LBM
14 046	004	INB
14 047	056 276	LLI 276
14 051	160	LMB
14 052	056 203	LLI 203
14 054	106	LBM
14 055	005	DCB
14 056	056 277	LLI 277
14 060	160	LMB
14 061	056 367	LLI 367
14 063	176	LAM
14 064	247	NDA
14 065	312 075 014	JTZ PRINT4
14 070	066 000	LMI 000
14 072	303 125 014	JMP PRINT6
14 075	315 224 003	PRINT4, CAL EVAL
14 100	056 177	LLI 177
14 102	046 026	** LHI 026
14 104	176	LAM
14 105	247	NDA
14 106	056 110	LLI 110
14 110	046 001	** LHI 001
14 112	066 377	LMI 377
14 114	314 314 014	PRINT5, CTZ PFPOUT
14 117	056 177	LLI 177
14 121	046 026	** LHI 026
14 123	066 000	LMI 000
14 125	056 203	PRINT6, LLI 203
14 127	315 240 002	CAL GETCHR
14 132	376 254	CPI 254
14 134	314 357 014	CTZ PCOMMA
14 137	056 203	LLI 203
14 141	046 026	** LHI 026
14 143	106	LBM
14 144	056 202	LLI 202
14 146	160	LMB
14 147	056 000	LLI 000
14 151	170	LAB
14 152	276	CPM
14 153	372 366 013	JTS PRINT1
14 156	056 000	LLI 000
14 160	315 240 002	CAL GETCHR
14 163	376 254	CPI 254
14 165	312 116 013	JTZ NXTLIN

14 170	376 273	CPI 273
14 172	312 116 013	JTZ NXTLIN
14 175	315 141 003	CAL CRLF
14 200	303 116 013	JMP NXTLIN
14 203	056 367	QUOTE, LLI 367
14 205	167	LMA
14 206	315 255 002	CAL CLESYM
14 211	056 203	LLI 203
14 213	106	LBM
14 214	004	INB
14 215	056 204	LLI 204
14 217	160	LMB
14 220	056 204	QUOTE1, LLI 204
14 222	315 240 002	CAL GETCHR
14 225	056 367	LLI 367
14 227	276	CPM
14 230	312 263 014	JTZ QUOTE2
14 233	315 202 003	CAL ECHO
14 236	056 204	LLI 204
14 240	315 003 003	CAL LOOP
14 243	302 220 014	JFZ QUOTE1
14 246	076 311	QUOTER, LAI 311
14 250	016 321	LCI 321
14 252	056 367	LLI 367
14 254	046 026	** LHI 026
14 256	066 000	LMI 000
14 260	303 226 002	JMP ERROR
14 263	056 204	QUOTE2, LLI 204
14 265	106	LBM
14 266	056 202	LLI 202
14 270	160	LMB
14 271	170	LAB
14 272	056 000	LLI 000
14 274	276	CPM
14 275	302 366 013	JFZ PRINT1
14 300	315 141 003	CAL CRLF
14 303	056 367	LLI 367
14 305	046 026	** LHI 026
14 307	066 000	LMI 000
14 311	303 116 013	JMP NXTLIN
14 314	056 126	PFPOUT, LLI 126
14 316	046 001	** LHI 001
14 320	176	LAM
14 321	247	NDA
14 322	312 336 014	JTZ ZERO
14 325	054	INL

14 326	176	LAM
14 327	247	NDA
14 330	312 350 014	JTZ FRAC
14 333	303 165 024	JMP FPOUT
14 336	076 240	ZERO, LAI 240
14 340	315 202 003	CAL ECHO
14 343	076 260	LAI 260
14 345	303 202 003	JMP ECHO
14 350	056 110	FRAC, LLI 110
14 352	066 000	LMI 000
14 354	303 165 024	JMP FPOUT
14 357	056 000	PCOMMA, LLI 000
14 361	176	LAM
14 362	056 203	LLI 203
14 364	226	SUM
14 365	370	RTS
14 366	056 043	LLI 043
14 370	046 001	** LHI 001
14 372	176	LAM
14 373	346 360	NDI 360
14 375	306 020	ADI 020
14 377	226	SUM
15 000	117	LCA
15 001	076 240	LAI 240
15 003	315 202 003	PCOM1, CAL ECHO
15 006	015	DCC
15 007	302 003 015	JFZ PCOM1
15 012	311	RET
15 013	315 240 010	LETO, CAL SAVSYM
15 016	056 202	LLI 202
15 020	046 026	** LHI 026
15 022	106	LBM
15 023	056 203	LLI 203
15 025	160	LMB
15 026	303 141 015	JMP LET5
15 031	315 255 002	LET, CAL CLESYM
15 034	056 144	LLI 144
15 036	046 026	** LHI 026
15 040	066 000	LMI 000
15 042	056 202	LET1, LLI 202
15 044	046 026	** LHI 026
15 046	106	LBM
15 047	004	INB
15 050	056 203	LLI 203

15 052	160		LMB
15 053	056 203		LET2, LLI 203
15 055	315 240 002		CAL GETCHR
15 060	312 122 015		JTZ LET4
15 063	376 275		CPI 275
15 065	312 141 015		JTZ LET5
15 070	376 250	@@	CPI 250
15 072	302 113 015		JFZ LET3
15 075	315 145 055	@@	CAL ARRAY
15 100	056 206	@@	LLI 206
15 102	046 026	@@**	LHI 026
15 104	106	@@	LBM
15 105	056 203	@@	LLI 203
15 107	160	@@	LMB
15 110	303 122 015	@@	JMP LET4
15 113	056 144		LET3, LLI 144
15 115	046 026	**	LHI 026
15 117	315 314 002		CAL CONCT1
15 122	056 203		LET4, LLI 203
15 124	315 003 003		CAL LOOP
15 127	302 053 015		JFZ LET2
15 132	076 314		LETERR, LAI 314
15 134	016 305		LCI 305
15 136	303 226 002		JMP ERROR
15 141	056 203		LET5, LLI 203
15 143	046 026	**	LHI 026
15 145	106		LBM
15 146	004		INB
15 147	056 276		LLI 276
15 151	160		LMB
15 152	056 000		LLI 000
15 154	106		LBM
15 155	056 277		LLI 277
15 157	160		LMB
15 160	315 224 003		CAL EVAL
15 163	315 252 010		CAL RESTSY
15 166	315 055 010		CAL STOSYM
15 171	303 116 013		JMP NXTLIN
15 174	056 350		GOTO, LLI 350
15 176	046 026	**	LHI 026
15 200	066 000		LMI 000
15 202	056 202		LLI 202
15 204	106		LBM
15 205	004		INB
15 206	056 203		LLI 203

15 210	160		LMB
15 211	056 203	GOTO1,	LLI 203
15 213	315 240 002		CAL GETCHR
15 216	312 240 015		JTZ GOTO2
15 221	376 260		CPI 260
15 223	372 250 015		JTS GOTO3
15 226	376 272		CPI 272
15 230	362 250 015		JFS GOTO3
15 233	056 350		LLI 350
15 235	315 314 002		CAL CONCT1
15 240	056 203	GOTO2,	LLI 203
15 242	315 003 003		CAL LOOP
15 245	302 211 015		JFZ GOTO1
15 250	056 360	GOTO3,	LLI 360
15 252	046 026	**	LHI 026
15 254	066 033	††	LMI 033
15 256	054		INL
15 257	066 000		LMI 000
15 261	315 255 002	GOTO4,	CAL CLESYM
15 264	056 204		LLI 204
15 266	066 001		LMI 001
15 270	056 204	GOTO5,	LLI 204
15 272	315 123 012		CAL GETCHP
15 275	312 315 015		JTZ GOTO6
15 300	376 260		CPI 260
15 302	372 340 015		JTS GOTO7
15 305	376 272		CPI 272
15 307	362 340 015		JFS GOTO7
15 312	315 310 002		CAL CONCTS
15 315	056 204	GOTO6,	LLI 204
15 317	046 026	**	LHI 026
15 321	106		LBM
15 322	004		INB
15 323	160		LMB
15 324	056 360		LLI 360
15 326	116		LCM
15 327	054		INL
15 330	156		LLM
15 331	141		LHC
15 332	176		LAM
15 333	005		DCB
15 334	270		CPB
15 335	302 270 015		JFZ GOTO5
15 340	056 120	GOTO7,	LLI 120

15 342	046 026	**	LHI 026
15 344	026 026	**	LDI 026
15 346	036 350		LEI 350
15 350	315 332 002		CAL STRCP
15 353	312 156 013		JTZ SAMLIN
15 356	056 360		LLI 360
15 360	046 026	**	LHI 026
15 362	126		LDM
15 363	054		INL
15 364	136		LEM
15 365	142		LHD
15 366	153		LLE
15 367	106		LBM
15 370	004		INB
15 371	315 305 012		CAL ADBDE
15 374	056 360		LLI 360
15 376	046 026	**	LHI 026
16 000	162		LMD
16 001	054		INL
16 002	163		LME
16 003	056 364		LLI 364
16 005	172		LAD
16 006	276		CPM
16 007	302 261 015		JFZ GOTO4
16 012	054		INL
16 013	173		LAE
16 014	276		CPM
16 015	302 261 015		JFZ GOTO4
16 020	076 325		GOTOER, LAI 325
16 022	016 316		LCI 316
16 024	303 226 002		JMP ERROR
16 027	056 202		IF, LLI 202
16 031	046 026	**	LHI 026
16 033	106		LBM
16 034	004		INB
16 035	056 276		LLI 276
16 037	160		LMB
16 040	315 255 002		CAL CLESYM
16 043	056 320		LLI 320
16 045	046 001	**	LHI 001
16 047	315 012 013		CAL INSTR
16 052	173		LAE
16 053	247		NDA
16 054	302 102 016		JFZ IF1
16 057	056 013		LLI 013
16 061	046 027	**	LHI 027
16 063	315 012 013		CAL INSTR
16 066	173		LAE
16 067	247		NDA

16 070	302 102 016		JFZ IF1
16 073	076 311	IFERR,	LAI 311
16 075	016 306		LCI 306
16 077	303 226 002		JMP ERROR
16 102	056 277	IF1,	LLI 277
16 104	046 026		LHI 026
16 106	035		DCE
16 107	163		LME
16 110	315 224 003		CAL EVAL
16 113	056 126		LLI 126
16 115	046 001	**	LHI 001
16 117	176		LAM
16 120	247		NDA
16 121	312 116 013		JTZ NXTLIN
16 124	056 277		LLI 277
16 126	046 026	**	LHI 026
16 130	176		LAM
16 131	306 005		ADI 005
16 133	056 202		LLI 202
16 135	167		LMA
16 136	107		LBA
16 137	004		INB
16 140	056 204		LLI 204
16 142	160		LMB
16 143	056 204	IF2,	LLI 204
16 145	315 240 002		CAL GETCHR
16 150	302 166 016		JFZ IF3
16 153	056 204		LLI 204
16 155	315 003 003		CAL LOOP
16 160	302 143 016		JFZ IF2
16 163	303 073 016		JMP IFERR
16 166	376 260	IF3,	CPI 260
16 170	372 200 016		JTS IF4
16 173	376 272		CPI 272
16 175	372 174 015		JTS GOTO
16 200	056 000	IF4,	LLI 000
16 202	176		LAM
16 203	056 204		LLI 204
16 205	226		SUM
16 206	107		LBA
16 207	004		INB
16 210	116		LCM
16 211	056 000		LLI 000
16 213	160		LMB
16 214	151		LLC
16 215	026 026	**	LDI 026

16 217	036 001		LEI 001
16 221	315 013 021		CAL MOVEIT
16 224	056 202		LLI 202
16 226	066 001		LMI 001
16 230	315 067 002		CAL SYNTX4
16 233	303 211 013		JMP DIRECT
16 236	056 340		GOSUB, LLI 340
16 240	046 026	**	LHI 026
16 242	126		LDM
16 243	024		IND
16 244	025		DCD
16 245	312 255 016		JTZ GOSUB1
16 250	056 360		LLI 360
16 252	126		LDM
16 253	054		INL
16 254	136		LEM
16 255	056 073		GOSUB1, LLI 073
16 257	046 027	**	LHI 027
16 261	176		LAM
16 262	306 002		ADI 002
16 264	376 021		CPI 021
16 266	362 347 016		JFS GOSERR
16 271	167		LMA
16 272	056 076		LLI 076
16 274	205		ADL
16 275	157		LLA
16 276	162		LMD
16 277	054		INL
16 300	163		LME
16 301	303 174 015		JMP GOTO
16 304	056 073		RETURN, LLI 073
16 306	046 027	**	LHI 027
16 310	176		LAM
16 311	326 002		SUI 002
16 313	372 356 016		JTS RETERR
16 316	167		LMA
16 317	306 002		ADI 002
16 321	056 076		LLI 076
16 323	205		ADL
16 324	157		LLA
16 325	126		LDM
16 326	024		IND
16 327	025		DCD
16 330	312 266 010		JTZ EXEC
16 333	054		INL
16 334	136		LEM
16 335	056 360		LLI 360
16 337	046 026	**	LHI 026

16 341	162		LMD
16 342	054		INL
16 343	163		LME
16 344	303 116 013		JMP NXTLIN
16 347	076 307	GOSERR,	LAI 307
16 351	016 323		LCI 323
16 353	303 226 002		JMP ERROR
16 356	076 322	RETERR,	LAI 322
16 360	016 324		LCI 324
16 362	303 226 002		JMP ERROR
16 365	315 255 002	INPUT,	CAL CLESYM
16 370	056 202		LLI 202
16 372	106		LBM
16 373	004		INB
16 374	056 203		LLI 203
16 376	160		LMB
16 377	056 203	INPUT1,	LLI 203
17 001	315 240 002		CAL GETCHR
17 004	312 042 017		JTZ INPUT3
17 007	376 254		CPI 254
17 011	312 063 017		JTZ INPUT4
17 014	376 250		CPI 250
17 016	302 037 017		JFZ INPUT2
17 021	315 160 055	@@	CAL ARRAY2
17 024	056 206	@@	LLI 206
17 026	046 026	@@**	LHI 026
17 030	106	@@	LBM
17 031	056 203	@@	LLI 203
17 033	160	@@	LMB
17 034	303 042 017	@@	JMP INPUT3
17 037	315 310 002	INPUT2,	CAL CONCTS
17 042	056 203	INPUT3,	LLI 203
17 044	315 003 003		CAL LOOP
17 047	302 377 016		JFZ INPUT1
17 052	315 104 017		CAL INPUTX
17 055	315 055 010		CAL STOSYM
17 060	303 116 013		JMP NXTLIN
17 063	315 104 017	INPUT4,	CAL INPUTX
17 066	315 055 010		CAL STOSYM
17 071	046 026	**	LHI 026
17 073	056 203		LLI 203
17 075	106		LBM
17 076	056 202		LLI 202
17 100	160		LMB

17 101	303 365 016	JMP INPUT
17 104	056 120	INPUTX, LLI 120
17 106	176	LAM
17 107	205	ADL
17 110	157	LLA
17 111	176	LAM
17 112	376 244	CPI 244
17 114	302 140 017	JFZ INPUTN
17 117	056 120	LLI 120
17 121	106	LBM
17 122	005	DCB
17 123	160	LMB
17 124	315 157 017	CAL FP0
17 127	315 221 003	CAL CINPUT
17 132	056 124	LLI 124
17 134	167	LMA
17 135	303 064 020	JMP FPFLT
17 140	056 144	INPUTN, LLI 144
17 142	046 026	** LHI 026
17 144	076 277	LAI 277
17 146	315 202 003	CAL ECHO
17 151	315 014 003	CAL STRIN
17 154	303 044 023	JMP DINPUT
17 157	046 001	FP0, LHI 001
17 161	303 247 006	JMP CFALSE
17 164	056 144	FOR, LLI 144
17 166	046 026	** LHI 026
17 170	066 000	LMI 000
17 172	056 146	LLI 146
17 174	066 000	LMI 000
17 176	056 205	LLI 205
17 200	046 027	** LHI 027
17 202	106	LBM
17 203	004	INB
17 204	160	LMB
17 205	056 360	LLI 360
17 207	046 026	** LHI 026
17 211	126	LDM
17 212	054	INL
17 213	136	LEM
17 214	170	LAB
17 215	007	RLC
17 216	007	RLC
17 217	306 134	ADI 134
17 221	157	LLA
17 222	046 027	** LHI 027
17 224	162	LMD

17 225	054		INL
17 226	163		LME
17 227	056 325		LLI 325
17 231	046 001	**	LHI 001
17 233	315 012 013		CAL INSTR
17 236	173		LAE
17 237	247		NDA
17 240	302 252 017		JFZ FOR1
17 243	076 306	FORERR,	LAI 306
17 245	016 305		LCI 305
17 247	303 226 002		JMP ERROR
17 252	056 202	FOR1,	LLI 202
17 254	046 026	**	LHI 026
17 256	106		LBM
17 257	004		INB
17 260	056 204		LLI 204
17 262	160		LMB
17 263	056 203		LLI 203
17 265	163		LME
17 266	056 204	FOR2,	LLI 204
17 270	315 240 002		CAL GETCHR
17 273	312 310 017		JTZ FOR3
17 276	376 275		CPI 275
17 300	312 323 017		JTZ FOR4
17 303	056 144		LLI 144
17 305	315 314 002		CAL CONCT1
17 310	056 204	FOR3,	LLI 204
17 312	315 003 003		CAL LOOP
17 315	302 266 017		JFZ FOR2
17 320	303 243 017		JMP FORERR
17 323	056 204	FOR4,	LLI 204
17 325	106		LBM
17 326	004		INB
17 327	056 276		LLI 276
17 331	160		LMB
17 332	056 203		LLI 203
17 334	106		LBM
17 335	005		DCB
17 336	056 277		LLI 277
17 340	160		LMB
17 341	315 224 003		CAL EVAL
17 344	315 252 010		CAL RESTSY
17 347	056 144		LLI 144
17 351	046 026	**	LHI 026
17 353	176		LAM
17 354	376 001		CPI 001
17 356	302 246 031		JFZ FOR5
17 361	056 146		LLI 146

17 363	066 000	LMI 000
17 365	303 246 031	JMP FOR5

Note open addresses.
This space available
for patching.

20 000	056 126		FPFIX, LLI 126
20 002	046 001	**	LHI 001
20 004	176		LAM
20 005	056 100		LLI 100
20 007	167		LMA
20 010	247		NDA
20 011	374 202 020		CTS FPCOMP
20 014	056 127		LLI 127
20 016	076 027		LAI 027
20 020	106		LBM
20 021	004		INB
20 022	005		DCB
20 023	372 051 020		JTS FPZERO
20 026	220		SUB
20 027	372 366 012		JTS FIXERR
20 032	117		LCA
20 033	056 126		FPFIXL, LLI 126
20 035	006 003		LBI 003
20 037	315 211 022		CAL ROTATR
20 042	015		DCC
20 043	302 033 020		JFZ FPFLT
20 046	303 175 020		JMP RESIGN
20 051	056 126		FPZERO, LLI 126
20 053	257		XRA
20 054	167		LMA
20 055	055		DCL
20 056	167		LMA
20 057	055		DCL
20 060	167		LMA
20 061	055		DCL
20 062	167		LMA
20 063	311		RET
20 064	006 027		FPFLT, LBI 027
20 066	170		FPNORM, LAB
20 067	046 001	**	LHI 001
20 071	056 127		LLI 127
20 073	247		NDA
20 074	312 100 020		JTZ NOEXCO
20 077	160		LMB
20 100	055		NOEXCO, DCL
20 101	176		LAM
20 102	056 100		LLI 100

20 104	167		LMA	
20 105	247		NDA	
20 106	362 120 020		JFS ACZERT	
20 111	006 004		LBI 004	
20 113	056 123		LLI 123	
20 115	315 150 022		CAL COMPLM	
20 120	056 126	ACZERT,	LLI 126	
20 122	006 004		LBI 004	
20 124	176	LOOK0,	LAM	
20 125	247		NDA	
20 126	302 143 020		JFZ ACNONZ	
20 131	055		DCL	
20 132	005		DCB	
20 133	302 124 020		JFZ LOOKO	
20 136	056 127		LLI 127	
20 140	257		XRA	
20 141	167		LMA	
20 142	311		RET	
20 143	056 123	ACNONZ,	LLI 123	
20 145	006 004		LBI 004	
20 147	315 177 022		CAL ROTATL	
20 152	176		LAM	
20 153	247		NDA	
20 154	372 166 020		JTS ACCSET	
20 157	054		INL	
20 160	106		LBM	
20 161	005		DCB	
20 162	160		LMB	
20 163	303 143 020		JMP ACNONZ	
20 166	056 126	ACCSET,	LLI 126	
20 170	006 003		LBI 003	
20 172	315 211 022		CAL ROTATR	
20 175	056 100	RESIGN,	LLI 100	
20 177	176		LAM	
20 200	247		NDA	
20 201	360		RFS	
20 202	056 124	FPCOMP,	LLI 124	
20 204	006 003		LBI 003	
20 206	303 150 022		JMP COMPLM	
20 211	056 126		FPADD,	LLI 126
20 213	046 001	**		LHI 001
20 215	176		LAM	
20 216	247		NDA	
20 217	302 235 020		JFZ NONZAC	
20 222	056 124	MOVOP,	LLI 124	
20 224	124		LDH	
20 225	135		LEL	
20 226	056 134		LLI 134	
20 230	006 004		LBI 004	
20 232	303 013 021		JMP MOVEIT	

20 235	056 136	NONZAC,	LLI 136
20 237	176		LAM
20 240	247		NDA
20 241	310		RTZ
20 242	056 127	CKEQEX,	LLI 127
20 244	176		LAM
20 245	056 137		LLI 137
20 247	276		CPM
20 250	312 341 020		JTZ SHACOP
20 253	107		LBA
20 254	176		LAM
20 255	230		SBB
20 256	362 264 020		JFS SKPNEG
20 261	107		LBA
20 262	257		XRA
20 263	230		SBB
20 264	376 030	SKPNEG,	CPI 030
20 266	372 303 020		JTS LINEUP
20 271	176		LAM
20 272	056 127		LLI 127
20 274	226		SUM
20 275	370		RTS
20 276	056 124		LLI 124
20 300	303 222 020		JMP MOVOP
20 303	176	LINEUP,	LAM
20 304	056 127		LLI 127
20 306	226		SUM
20 307	372 327 020		JTS SHIFTO
20 312	117		LCA
20 313	056 127	MORACC,	LLI 127
20 315	315 374 020		CAL SHLOOP
20 320	015		DCC
20 321	302 313 020		JFZ MORACC
20 324	303 341 020		JMP SHACOP
20 327	117	SHIFTO,	LCA
20 330	056 137	MOROP,	LLI 137
20 332	315 374 020		CAL SHLOOP
20 335	014		INC
20 336	302 330 020		JFZ MOROP
20 341	056 123	SHACOP,	LLI 123
20 343	066 000		LMI 000
20 345	056 127		LLI 127
20 347	315 374 020		CAL SHLOOP
20 352	056 137		LLI 137
20 354	315 374 020		CAL SHLOOP
20 357	124		LDH
20 360	036 123		LEI 123
20 362	006 004		LBI 004
20 364	315 127 022		CAL ADDER
20 367	006 000		LBI 000
20 371	303 066 020		JMP FPNORM

20 374	106		SHLOOP,	LBM
20 375	004			INB
20 376	160			LMB
20 377	055			DCL
21 000	006 004			LBI 004
21 002	176		FSHIFT,	LAM
21 003	247			NDA
21 004	362 211 022			JFS ROTATR
21 007	027		BRING1,	RAL
21 010	303 212 022			JMP ROTR
21 013	176		MOVEIT,	LAM
21 014	054			INL
21 015	315 356 022			CAL SWITCH
21 020	167			LMA
21 021	054			INL
21 022	315 356 022			CAL SWITCH
21 025	005			DCB
21 026	310			RTZ
21 027	303 013 021			JMP MOVEIT
21 032	056 124		FSUB,	LLI 124
21 034	046 001	**		LHI 001
21 036	006 003			LBI 003
21 040	315 150 022			CAL COMPLM
21 043	303 211 020			JMP FPADD
21 046	315 166 021		FPMULT,	CAL CKSIGN
21 051	056 137		ADDEXP,	LLI 137
21 053	176			LAM
21 054	056 127			LLI 127
21 056	206			ADM
21 057	306 001			ADI 001
21 061	167			LMA
21 062	056 102		SETMCT,	LLI 102
21 064	066 027			LMI 027
21 066	056 126		MULTIP,	LLI 126
21 070	006 003			LBI 003
21 072	315 211 022			CAL ROTATR
21 075	334 270 021			CTC ADOPPP
21 100	056 146			LLI 146
21 102	006 006			LBI 006
21 104	315 211 022			CAL ROTATR
21 107	056 102			LLI 102
21 111	116			LCM
21 112	015			DCC
21 113	161			LMC
21 114	302 066 021			JFZ MULTIP
21 117	056 146			LLI 146
21 121	006 006			LBI 006
21 123	315 211 022			CAL ROTATR
21 126	056 143			LLI 143

21 130	176	LAM
21 131	027	RAL
21 132	247	NDA
21 133	374 302 021	CTS MROUND
21 136	056 123	LLI 123
21 140	135	LEL
21 141	124	LDH
21 142	056 143	LLI 143
21 144	006 004	LBI 004
21 146	315 013 021	EXMLDV, CAL MOVEIT
21 151	006 000	LBI 000
21 153	315 066 020	CAL FPNORM
21 156	056 101	LLI 101
21 160	176	LAM
21 161	247	NDA
21 162	300	RFZ
21 163	303 202 020	JMP FPCOMP
21 166	056 140	CKSIGN, LLI 140
21 170	046 001	LHI 001
21 172	006 010	LBI 010
21 174	257	XRA
21 175	167	CLRNX, LMA
21 176	054	INL
21 177	005	DCB
21 200	302 175 021	JFZ CLRNX
21 203	006 004	CLROPL, LBI 004
21 205	056 130	LLI 130
21 207	167	CLRNX1, LMA
21 210	054	INL
21 211	005	DCB
21 212	302 207 021	JFZ CLRNX1
21 215	056 101	LLI 101
21 217	066 001	LMI 001
21 221	056 126	LLI 126
21 223	176	LAM
21 224	247	NDA
21 225	372 251 021	JTS NEGFPA
21 230	056 136	OPSGNT, LLI 136
21 232	176	LAM
21 233	247	NDA
21 234	360	RFS
21 235	056 101	LLI 101
21 237	116	LCM
21 240	015	DCC
21 241	161	LMC
21 242	056 134	LLI 134
21 244	006 003	LBI 003
21 246	303 150 022	JMP COMPLM
21 251	056 101	NEGFPA, LLI 101

21 253	116		LCM
21 254	015		DCC
21 255	161		LMC
21 256	056 124		LLI 124
21 260	006 003		LBI 003
21 262	315 150 022		CAL COMPLM
21 265	303 230 021		JMP OPSGNT
21 270	036 141	ADOPPP,	LEI 141
21 272	124		LDH
21 273	056 131		LLI 131
21 275	006 006		LBI 006
21 277	303 127 022		JMP ADDER
21 302	006 003	MROUND,	LBI 003
21 304	076 100		LAI 100
21 306	206		ADM
21 307	167	CROUND,	LMA
21 310	054		INL
21 311	076 000		LAI 000
21 313	216		ACM
21 314	005		DCB
21 315	302 307 021		JFZ CROUND
21 320	167		LMA
21 321	311		RET
21 322	315 166 021	FPDIV,	CAL CKSIGN
21 325	056 126		LLI 126
21 327	176		LAM
21 330	247		NDA
21 331	312 357 012		JTZ DVERR
21 334	056 137	SUBEXP,	LLI 137
21 336	176		LAM
21 337	056 127		LLI 127
21 341	226		SUM
21 342	306 001		ADI 001
21 344	167		LMA
21 345	056 102	SETDCT,	LLI 102
21 347	066 027		LMI 027
21 351	315 101 022	DIVIDE,	CAL SETSUB
21 354	372 376 021		JTS NOGO
21 357	036 134		LEI 134
21 361	056 131		LLI 131
21 363	006 003		LBI 003
21 365	315 013 021		CAL MOVEIT
21 370	076 001		LAI 001
21 372	037		RAR
21 373	303 377 021		JMP QUOROT
21 376	257	NOGO,	XRA
21 377	056 144	QUOROT,	LLI 144
22 001	006 003		LBI 003
22 003	315 200 022		CAL ROTL
22 006	056 134		LLI 134

22 010	006 003	LBI 003
22 012	315 177 022	CAL ROTATL
22 015	056 102	LLI 102
22 017	116	LCM
22 020	015	DCC
22 021	161	LMC
22 022	302 351 021	JFZ DIVIDE
22 025	315 101 022	CAL SETSUB
22 030	372 070 022	JTS DVEXIT
22 033	056 144	LLI 144
22 035	176	LAM
22 036	306 001	ADI 001
22 040	167	LMA
22 041	076 000	LAI 000
22 043	054	INL
22 044	216	ACM
22 045	167	LMA
22 046	076 000	LAI 000
22 050	054	INL
22 051	216	ACM
22 052	167	LMA
22 053	362 070 022	JFS DVEXIT
22 056	006 003	LBI 003
22 060	315 211 022	CAL ROTATR
22 063	056 127	LLI 127
22 065	106	LBM
22 066	004	INB
22 067	160	LMB
22 070	056 144	DVEXIT,
22 072	036 124	LLI 144
22 074	006 003	LEI 124
22 076	303 146 021	LBI 003
22 101	036 131	JMP EXMLDV
22 103	124	SETSUB,
22 104	056 124	LEI 131
22 106	006 003	LDH
22 110	315 013 021	LLI 124
22 113	036 131	LBI 003
22 115	056 134	CAL MOVEIT
22 117	006 003	LEI 131
22 121	315 223 022	LLI 134
22 124	176	LBI 003
22 125	247	CAL SUBBER
22 126	311	LAM
22 127	247	NDA
22 130	176	ADDER,
22 131	315 356 022	ADDMOR,
22 134	216	NDA
22 135	167	LAM
22 136	005	CAL SWITCH
		ACM
		LMA
		DCB

22 137	310		RTZ
22 140	054		INL
22 141	315 356 022		CAL SWITCH
22 144	054		INL
22 145	303 130 022		JMP ADDMOR
22 150	176	COMPLM,	LAM
22 151	356 377		XRI 377
22 153	306 001		ADI 001
22 155	167	MORCOM,	LMA
22 156	037		RAR
22 157	127		LDA
22 160	005		DCB
22 161	310		RTZ
22 162	054		INL
22 163	176		LAM
22 164	356 377		XRI 377
22 166	137		LEA
22 167	172		LAD
22 170	027		RAL
22 171	076 000		LAI 000
22 173	213		ACE
22 174	303 155 022		JMP MORCOM
22 177	247	ROTATL,	NDA
22 200	176	ROTL,	LAM
22 201	027		RAL
22 202	167		LMA
22 203	005		DCB
22 204	310		RTZ
22 205	054		INL
22 206	303 200 022		JMP ROTL
22 211	247	ROTATR,	NDA
22 212	176	ROTR,	LAM
22 213	037		RAR
22 214	167		LMA
22 215	005		DCB
22 216	310		RTZ
22 217	055		DCL
22 220	303 212 022		JMP ROTR
22 223	247	SUBBER,	NDA
22 224	176	SUBTRA,	LAM
22 225	315 356 022		CAL SWITCH
22 230	236		SBM
22 231	167		LMA
22 232	005		DCB
22 233	310		RTZ
22 234	054		INL
22 235	315 356 022		CAL SWITCH

22 240	054		INL
22 241	303 224 022		JMP SUBTRA
22 244	026 001	**	FLOAD, LDI 001
22 246	036 124		LEI 124
22 250	006 004		LBI 004
22 252	303 013 021		JMP MOVEIT
22 255	135		FSTORE, LEL
22 256	124		LDH
22 257	056 124		LLI 124
22 261	046 001	**	LHI 001
22 263	303 272 022		JMP SETIT
22 266	026 001	**	OPLOAD, LDI 001
22 270	036 134		LEI 134
22 272	006 004		SETIT, LBI 004
22 274	303 013 021		JMP MOVEIT
22 277	315 317 022		FACXOP, CAL SAVEHL
22 302	056 124		LLI 124
22 304	046 001	**	LHI 001
22 306	315 266 022		CAL OPLOAD
22 311	315 337 022		CAL RESTHL
22 314	303 244 022		JMP FLOAD
22 317	174		SAVEHL, LAH
22 320	105		LBL
22 321	056 200		LLI 200
22 323	046 001	**	LHI 001
22 325	167		LMA
22 326	054		INL
22 327	160		LMB
22 330	054		INL
22 331	162		LMD
22 332	054		INL
22 333	163		LME
22 334	147		LHA
22 335	150		LLB
22 336	311		RET
22 337	056 200		RESTHL, LLI 200
22 341	046 001	**	LHI 001
22 343	176		LAM
22 344	054		INL
22 345	106		LBM
22 346	054		INL
22 347	126		LDM
22 350	054		INL
22 351	136		LEM
22 352	147		LHA

22 353	150		LLB
22 354	176		LAM
22 355	311		RET
22 356	114		SWITCH, LCH
22 357	142		LHD
22 360	121		LDC
22 361	115		LCL
22 362	153		LLE
22 363	131		LEC
22 364	311		RET
22 365	046 001	**	GETINP, LHI 001
22 367	056 220		LLI 220
22 371	116		LCM
22 372	014		INC
22 373	015		DCC
22 374	302 010 023		JFZ NOT0
22 377	153		LLE
23 000	142		LHD
23 001	116		LCM
23 002	014		INC
23 003	315 036 023		CAL INDEXC
23 006	066 000		LMI 000
23 010	056 220		NOT0, LLI 220
23 012	046 001	**	LHI 001
23 014	116		LCM
23 015	014		INC
23 016	161		LMC
23 017	153		LLE
23 020	142		LHD
23 021	315 036 023		CAL INDEXC
23 024	176		LAM
23 025	247		NDA
23 026	046 001	**	LHI 001
23 030	300		RFZ
23 031	056 220		LLI 220
23 033	066 000		LMI 000
23 035	311		RET
23 036	175		INDEXC, LAL
23 037	201		ADC
23 040	157		LLA
23 041	320		RFC
23 042	044		INH
23 043	311		RET
23 044	135		DINPUT, LEL
23 045	124		LDH
23 046	046 001	**	LHI 001

23 050	056 150		LLI 150
23 052	257		XRA
23 053	006 010		LBI 010
23 055	167	CLRNX2,	LMA
23 056	054		INL
23 057	005		DCB
23 060	302 055 023		JFZ CLRNX2
23 063	056 103		LLI 103
23 065	006 004		LBI 004
23 067	167	CLRNX3,	LMA
23 070	054		INL
23 071	005		DCB
23 072	302 067 023		JFZ CLRNX3
23 075	315 365 022		CAL GETINP
23 100	376 253		CPI 253
23 102	312 115 023		JTZ NINPUT
23 105	376 255		CPI 255
23 107	302 120 023		JFZ NOTPLM
23 112	056 103		LLI 103
23 114	167		LMA
23 115	315 365 022	NINPUT,	CAL GETINP
23 120	376 256	NOTPLM,	CPI 256
23 122	312 201 023		JTZ PERIOD
23 125	376 305		CPI 305
23 127	312 221 023		JTZ FNDEXP
23 132	376 240		CPI 240
23 134	312 115 023		JTZ NINPUT
23 137	247		NDA
23 140	312 311 023		JTZ ENDINP
23 143	376 260		CPI 260
23 145	372 375 012		JTS NUMERR
23 150	376 272		CPI 272
23 152	362 375 012		JFS NUMERR
23 155	056 156		LLI 156
23 157	117		LCA
23 160	076 370		LAI 370
23 162	246		NDM
23 163	302 115 023		JFZ NINPUT
23 166	056 105		LLI 105
23 170	106		LBM
23 171	004		INB
23 172	160		LMB
23 173	315 056 024		CAL DECBIN
23 176	303 115 023		JMP NINPUT
23 201	107	PERIOD,	LBA
23 202	056 106		LLI 106
23 204	176		LAM
23 205	247		NDA

23 206	302 375 012	JFZ NUMERR
23 211	056 105	LLI 105
23 213	167	LMA
23 214	054	INL
23 215	160	LMB
23 216	303 115 023	JMP NINPUT
23 221	315 365 022	FNDEXP, CAL GETINP
23 224	376 253	CPI 253
23 226	312 241 023	JTZ EXPINP
23 231	376 255	CPI 255
23 233	302 244 023	JFZ NOEXPS
23 236	056 104	LLI 104
23 240	167	LMA
23 241	315 365 022	EXPINP, CAL GETINP
23 244	247	NOEXPS, NDA
23 245	312 311 023	JTZ ENDINP
23 250	376 260	CPI 260
23 252	372 375 012	JTS NUMERR
23 255	376 272	CPI 272
23 257	362 375 012	JFS NUMERR
23 262	346 017	NDI 017
23 264	107	LBA
23 265	056 157	LLI 157
23 267	076 003	LAI 003
23 271	276	CPM
23 272	372 375 012	JTS NUMERR
23 275	116	LCM
23 276	176	LAM
23 277	247	NDA
23 300	027	RAL
23 301	027	RAL
23 302	201	ADC
23 303	027	RAL
23 304	200	ADB
23 305	167	LMA
23 306	303 241 023	JMP EXPINP
23 311	056 103	ENDINP, LLI 103
23 313	176	LAM
23 314	247	NDA
23 315	312 327 023	JTZ FININP
23 320	056 154	LLI 154
23 322	006 003	LBI 003
23 324	315 150 022	CAL COMPLM
23 327	056 153	FININP, LLI 153
23 331	257	XRA
23 332	167	LMA

23 333	124		LDH
23 334	036 123		LEI 123
23 336	006 004		LBI 004
23 340	315 013 021		CAL MOVEIT
23 343	315 064 020		CAL FPFLT
23 346	056 104		LLI 104
23 350	176		LAM
23 351	247		NDA
23 352	056 157		LLI 157
23 354	312 365 023		JTZ POSEXP
23 357	176		LAM
23 360	356 377		XRI 377
23 362	306 001		ADI 001
23 364	167		LMA
23 365	056 106	POSEXP,	LLI 106
23 367	176		LAM
23 370	247		NDA
23 371	312 000 024		JTZ EXPOK
23 374	056 105		LLI 105
23 376	257		XRA
23 377	226		SUM
24 000	056 157	EXPOK,	LLI 157
24 002	206		ADM
24 003	167		LMA
24 004	372 033 024		JTS MINEXP
24 007	310		RTZ
24 010	056 210	FPX10,	LLI 210
24 012	046 001		LHI 001
24 014	315 277 022		CAL FACXOP
24 017	315 046 021		CAL FPMULT
24 022	056 157		LLI 157
24 024	116		LCM
24 025	015		DCC
24 026	161		LMC
24 027	302 010 024		JFZ FPX10
24 032	311		RET
24 033	056 214	MINEXP, FPD10,	LLI 214
24 035	046 001		LHI 001
24 037	315 277 022		CAL FACXOP
24 042	315 046 021		CAL FPMULT
24 045	056 157		LLI 157
24 047	106		LBM
24 050	004		INB
24 051	160		LMB
24 052	302 033 024		JFZ FPD10
24 055	311		RET

24 056	315 317 022		DECBIN,	CAL SAVEHL
24 061	056 153			LLI 153
24 063	171			LAC
24 064	346 017			NDI 017
24 066	167			LMA
24 067	036 150			LEI 150
24 071	056 154			LLI 154
24 073	124			LDH
24 074	006 003			LBI 003
24 076	315 013 021			CAL MOVEIT
24 101	056 154			LLI 154
24 103	006 003			LBI 003
24 105	315 177 022			CAL ROTATL
24 110	056 154			LLI 154
24 112	006 003			LBI 003
24 114	315 177 022			CAL ROTATL
24 117	036 154			LEI 154
24 121	056 150			LLI 150
24 123	006 003			LBI 003
24 125	315 127 022			CAL ADDER
24 130	056 154			LLI 154
24 132	006 003			LBI 003
24 134	315 177 022			CAL ROTATL
24 137	056 152			LLI 152
24 141	257			XRA
24 142	167			LMA
24 143	055			DCL
24 144	167			LMA
24 145	056 153			LLI 153
24 147	176			LAM
24 150	056 150			LLI 150
24 152	167			LMA
24 153	036 154			LEI 154
24 155	006 003			LBI 003
24 157	315 127 022			CAL ADDER
24 162	303 337 022			JMP RESTHL
24 165	046 001	**	FPOUT,	LHI 001
24 167	056 157			LLI 157
24 171	066 000			LMI 000
24 173	056 126			LLI 126
24 175	176			LAM
24 176	247			NDA
24 177	372 207 024			JTS OUTNEG
24 202	076 240			LAI 240
24 204	303 220 024			JMP AHEAD1
24 207	056 124		OUTNEG,	LLI 124
24 211	006 003			LBI 003
24 213	315 150 022			CAL COMPLM
24 216	076 255			LAI 255

24 220	315 202 003	AHEAD1,	CAL ECHO
24 223	056 110		LLI 110
24 225	176		LAM
24 226	247		NDA
24 227	312 253 024		JTZ OUTFLT
24 232	056 127		LLI 127
24 234	076 027		LAI 027
24 236	106		LBM
24 237	004		INB
24 240	005		DCB
24 241	372 253 024		JTS OUTFLT
24 244	220		SUB
24 245	372 253 024		JTS OUTFLT
24 250	303 271 024		JMP OUTFIX
24 253	056 110	OUTFLT,	LLI 110
24 255	066 000		LMI 000
24 257	076 260		LAI 260
24 261	315 202 003		CAL ECHO
24 264	076 256		LAI 256
24 266	315 202 003		CAL ECHO
24 271	056 127	OUTFIX,	LLI 127
24 273	076 377		LAI 377
24 275	206		ADM
24 276	167		LMA
24 277	362 336 024	DECEXT,	JFS DECEXD
24 302	076 004		LAI 004
24 304	206		ADM
24 305	362 360 024		JFS DECOUT
24 310	056 210		LLI 210
24 312	046 001	**	LHI 001
24 314	315 277 022		CAL FACXOP
24 317	315 046 021		CAL FPMULT
24 322	056 157		LLI 157
24 324	116		LCM
24 325	015		DCC
24 326	161		LMC
24 327	056 127	DECREP,	LLI 127
24 331	176		LAM
24 332	247		NDA
24 333	303 277 024		JMP DECEXT
24 336	056 214	DECEXD,	LLI 214
24 340	046 001	**	LHI 001
24 342	315 277 022		CAL FACXOP
24 345	315 046 021		CAL FPMULT
24 350	056 157		LLI 157
24 352	106		LBM

24 353	004	INB
24 354	160	LMB
24 355	303 327 024	JMP DECREP
24 360	036 164	DECOUT, LEI 164
24 362	124	LDH
24 363	056 124	LLI 124
24 365	006 003	LBI 003
24 367	315 013 021	CAL MOVEIT
24 372	056 167	LLI 167
24 374	066 000	LMI 000
24 376	056 164	LLI 164
25 000	006 003	LBI 003
25 002	315 177 022	CAL ROTATL
25 005	315 223 025	CAL OUTX10
25 010	056 127	COMPEN, LLI 127
25 012	106	LBM
25 013	004	INB
25 014	160	LMB
25 015	312 032 025	JTZ OUTDIG
25 020	056 167	LLI 167
25 022	006 004	LBI 004
25 024	315 211 022	CAL ROTATR
25 027	303 010 025	JMP COMPEN
25 032	056 107	OUTDIG, LLI 107
25 034	066 007	LMI 007
25 036	056 167	LLI 167
25 040	176	LAM
25 041	247	NDA
25 042	312 165 025	JTZ ZERODG
25 045	056 167	OUTDGS, LLI 167
25 047	176	LAM
25 050	247	NDA
25 051	302 105 025	JFZ OUTDGX
25 054	056 110	LLI 110
25 056	176	LAM
25 057	247	NDA
25 060	312 104 025	JTZ OUTZER
25 063	056 157	LLI 157
25 065	116	LCM
25 066	015	DCC
25 067	014	INC
25 070	362 104 025	JFS OUTZER
25 073	056 166	LLI 166
25 075	176	LAM
25 076	346 340	NDI 340
25 100	302 104 025	JFZ OUTZER
25 103	311	RET

25 104	257	OUTZER, XRA
25 105	306 260	OUTDGX, ADI 260
25 107	315 202 003	CAL ECHO
25 112	056 110	DECRDG, LLI 110
25 114	176	LAM
25 115	247	NDA
25 116	302 137 025	JFZ CKDECP
25 121	056 107	LLI 107
25 123	116	LCM
25 124	015	DCC
25 125	161	LMC
25 126	312 300 025	JTZ EXPOUT
25 131	315 223 025	PUSHIT, CAL OUTX10
25 134	303 045 025	JMP OUTDGS
25 137	056 157	CKDECP, LLI 157
25 141	116	LCM
25 142	015	DCC
25 143	161	LMC
25 144	302 154 025	JFZ NODECP
25 147	076 256	LAI 256
25 151	315 202 003	CAL ECHO
25 154	056 107	NODECP, LLI 107
25 156	116	LCM
25 157	015	DCC
25 160	161	LMC
25 161	310	RTZ
25 162	303 131 025	JMP PUSHIT
25 165	056 157	ZERODG, LLI 157
25 167	116	LCM
25 170	015	DCC
25 171	161	LMC
25 172	056 166	LLI 166
25 174	176	LAM
25 175	247	NDA
25 176	302 112 025	JFZ DECRDG
25 201	055	DCL
25 202	176	LAM
25 203	247	NDA
25 204	302 112 025	JFZ DECRDG
25 207	055	DCL
25 210	176	LAM
25 211	247	NDA
25 212	302 112 025	JFZ DECRDG
25 215	056 157	LLI 157
25 217	167	LMA
25 220	303 112 025	JMP DECRDG

25 223	056 167	OUTX10,	LLI 167
25 225	066 000		LMI 000
25 227	056 164		LLI 164
25 231	124		LDH
25 232	036 160		LEI 160
25 234	006 004		LBI 004
25 236	315 013 021		CAL MOVEIT
25 241	056 164		LLI 164
25 243	006 004		LBI 004
25 245	315 177 022		CAL ROTATL
25 250	056 164		LLI 164
25 252	006 004		LBI 004
25 254	315 177 022		CAL ROTATL
25 257	056 160		LLI 160
25 261	036 164		LEI 164
25 263	006 004		LBI 004
25 265	315 127 022		CAL ADDER
25 270	056 164		LLI 164
25 272	006 004		LBI 004
25 274	315 177 022		CAL ROTATL
25 277	311		RET
25 300	056 157	EXPOUT,	LLI 157
25 302	176		LAM
25 303	247		NDA
25 304	310		RTZ
25 305	076 305		LAI 305
25 307	315 202 003		CAL ECHO
25 312	176		LAM
25 313	247		NDA
25 314	372 324 025		JTS EXOUTN
25 317	076 253		LAI 253
25 321	303 333 025		JMP AHEAD2
25 324	356 377	EXOUTN,	XRI 377
25 326	306 001		ADI 001
25 330	167		LMA
25 331	076 255		LAI 255
25 333	315 202 003	AHEAD2,	CAL ECHO
25 336	006 000		LBI 000
25 340	176		LAM
25 341	326 012	SUB12,	SUI 012
25 343	372 353 025		JTS TOMUCH
25 346	167		LMA
25 347	004		INB
25 350	303 341 025		JMP SUB12
25 353	076 260	TOMUCH,	LAI 260
25 355	200		ADB

25 356	315 202 003
25 361	176
25 362	306 260
25 364	315 202 003
25 367	311

CAL ECHO
LAM
ADI 260
CAL ECHO
RET

Note open addresses.
This space available
for patching.

NOTE: Pages 26 and 27 in memory are used for temporary data registers, pointers, counters and look-up tables. The following data should be placed on those pages. An entry marked XXX indicates the initial contents of the location are irrelevant to the program's operation.

26 000	000	(cc) for INPUT LINE BUFF
26 001	XXX	These locations used as the INPUT LINE BUFFER storage area
.	.	These locations used as the SYMBOL BUFFER storage area
26 117	XXX	These locations used as the AUXILIARY SYMBOL BUFFER storage area
26 120	000	These locations used as the TEMP SCAN storage register TAB FLAG
.	.	EVAL CURRENT temp. reg.
26 143	000	SYNTAX LINE NUMBER
26 144	000	SCAN temporary register STATEMENT TOKEN
.	.	Temporary working register
26 175	000	Temporary working register ARRAY pointer
26 176	000	ARRAY pointer
26 177	000	OPERATOR STACK pointer
26 200	000	These locations used as the OPERATOR STACK storage area
26 201	000	These locations used as the FUN/ARRAY STACK pointer
26 202	000	These locations used as the FUNCTION/ARRAY STACK storage area
26 203	000	
26 204	000	
26 205	000	
26 206	000	
26 207	000	
26 210	000	
26 211	XXX	
.	.	
26 277	XXX	
26 280	000	
26 281	XXX	
.	.	
26 237	XXX	

Heirarchy table (for out of stack ops).
Used by PARSER routine.

26 240	000	EOS
26 241	003	Plus sign
26 242	003	Minus sign
26 243	004	Multiplication sign
26 244	004	Division sign
26 245	005	Exponentiation sign
26 246	006	Left parenthesis
26 247	001	Right parenthesis
26 250	002	Not assigned
26 251	002	Less than sign
26 252	002	Equal sign
26 253	002	Greater than sign
26 254	002	Less than or equal combo
26 255	002	Equal to or greater than
26 256	002	Less than or greater than

Heirarchy table (for into stack ops).
Used by PARSER routine.

26 257	000	EOS
26 260	003	Plus sign
26 261	003	Minus sign
26 262	004	Multiplication sign
26 263	004	Division sign
26 264	005	Exponentiation sign
26 265	001	Left parenthesis
26 266	001	Right parenthesis
26 267	002	Not assigned
26 270	002	Less than sign
26 271	002	Equal sign
26 272	002	Greater than sign
26 273	002	Less than or equal combo
26 274	002	Equal to or greater than
26 275	002	Less than or greater than
26 276	000	EVAL (start) pointer
26 277	000	EVAL FINISH pointer

FUNCTION NAMES TABLE

26 300	003	(cc) for INT
26 301	311	I
26 302	316	N
26 303	324	T
26 304	003	(cc) for SGN
26 305	323	S
26 306	307	G
26 307	316	N

26 310	003	(cc) for ABS
26 311	301	A
26 312	302	B
26 313	323	S
26 314	003	(cc) for SQR
26 315	323	S
26 316	321	Q
26 317	322	R
26 320	003	(cc) for TAB
26 321	324	T
26 322	301	A
26 323	302	B
26 324	003	(cc) for RND
26 325	322	R
26 326	316	N
26 327	304	D
26 330	003	(cc) for CHR
26 331	303	C
26 332	310	H
26 333	322	R
26 334	003	(cc) for UDF
26 335	325	U
26 336	304	D
26 337	306	F
26 340	000	These locations used as the LINE NUMBER BUFFER storage area
.	.	.
26 347	000	These locations used as the AUX LINE NUMBER BUFFER
26 350	000	storage area
.	.	.
26 357	000	USER PGM LINE pointer (pg)
26 360	000	USER PGM LINE pntr (low)
26 361	000	AUX PGM LINE pointer (pg)
26 362	000	AUX PGM LINE pntr (low)
26 363	000	END of USER PGM BFR (pg)
26 364	000	END of USER PGM BFR pntr
26 365	000	Parenthesis counter
26 366	000	QUOTE Indicator
26 367	000	Table counter
26 370	000	Not assigned
26 371	XXX	
.	.	
26 377	XXX	Not assigned

End of page 26.

STATEMENT KEYWORD TABLE

27 000	003	(cc) for REM
27 001	322	R
27 002	305	E
27 003	315	M
27 004	002	(cc) for IF
27 005	311	I
27 006	306	F
27 007	003	(cc) for LET
27 010	314	L
27 011	305	E
27 012	324	T
27 013	004	(cc) for GOTO
27 014	307	G
27 015	317	O
27 016	324	T
27 017	317	O
27 020	005	(cc) for PRINT
27 021	320	P
27 022	322	R
27 023	311	I
27 024	316	N
27 025	324	T
27 026	005	(cc) for INPUT
27 027	311	I
27 030	316	N
27 031	320	P
27 032	325	U
27 033	324	T
27 034	003	(cc) for FOR
27 035	306	F
27 036	317	O
27 037	322	R
27 040	004	(cc) for NEXT
27 041	316	N
27 042	305	E
27 043	330	X
27 044	324	T
27 045	005	(cc) for GOSUB
27 046	307	G
27 047	317	O
27 050	323	S
27 051	325	U
27 052	302	B
27 053	006	(cc) for RETURN
27 054	322	R
27 055	305	E
27 056	324	T
27 057	325	U
27 060	322	R

27 061	316	N
27 062	003	(cc) for DIM
27 063	304	D
27 064	311	I
27 065	315	M
27 066	003	(cc) for END
27 067	305	E
27 070	316	N
27 071	304	D
27 072	000	End of Table
27 073	000	GOSUB STACK pointer
27 074	XXX	Not assigned
27 075	000	Number of arrays counter
27 076	000	ARRAY pointer
27 077	000	VARIABLES counter
27 100	000	These locations used as the GOSUB STACK
.	.	storage
.	.	area
27 117	000	These locations used as the ARRAY VARIABLES
27 120	000	TABLE
.	.	storage area
.	.	These locations used as the FOR/NEXT STACK
27 137	000	storage
27 140	000	area
.	.	
27 177	000	
27 200	000	FOR/NEXT STACK pointer
27 201	000	ARRAY/VARIABLE flag
27 202	000	STOSYM counter
27 203	000	FUN/ARRAY STACK pointer
27 204	000	ARRAY VALUES pointer
27 205	XXX	Not assigned
.	.	
27 207	XXX	Not assigned
27 210	000	These locations
27 211	XXX	used as the VARIABLES SYMBOL
.	.	TABLE
27 377	XXX	storage area

End of page 27.

Note open addresses
at start of page 30.
These locations avail-
able for patching.

30 013	056 144		NEXT,	LLI 144
30 015	046 026	**		LHI 026
30 017	066 000			LMI 000
30 021	056 202			LLI 202
30 023	106			LBM
30 024	004			INB
30 025	056 201			LLI 201
30 027	160			LMB
30 030	056 201		NEXT1,	LLI 201
30 032	315 240 002			CAL GETCHR
30 035	312 045 030			JTZ NEXT2
30 040	056 144			LLI 144
30 042	315 314 002			CAL CONCT1
30 045	056 201		NEXT2,	LLI 201
30 047	315 003 003			CAL LOOP
30 052	302 030 030			JFZ NEXT1
30 055	056 144			LLI 144
30 057	176			LAM
30 060	376 001			CPI 001
30 062	302 071 030			JFZ NEXT3
30 065	056 146			LLI 146
30 067	066 000			LMI 000
30 071	056 205		NEXT3,	LLI 205
30 073	046 027	**		LHI 027
30 075	176			LAM
30 076	007			RLC
30 077	007			RLC
30 100	306 136			ADI 136
30 102	046 027	**		LHI 027
30 104	157			LLA
30 105	026 026	**		LDI 026
30 107	036 145			LEI 145
30 111	006 002			LBI 002
30 113	315 370 002			CAL STR CPC
30 116	312 130 030			JTZ NEXT4
30 121	076 306		FORNXT,	LAI 306
30 123	016 316			LCI 316
30 125	303 226 002			JMP ERROR
30 130	056 360		NEXT4,	LLI 360
30 132	046 026	**		LHI 026
30 134	126			LDM
30 135	054			INL
30 136	136			LEM
30 137	054			INL
30 140	162			LMD
30 141	054			INL
30 142	163			LME
30 143	056 205			LLI 205
30 145	046 027	**		LHI 027
30 147	176			LAM
30 150	007			RLC

30 151	007	RLC
30 152	306 134	ADI 134
30 154	157	LLA
30 155	126	LDM
30 156	054	INL
30 157	136	LEM
30 160	056 360	LLI 360
30 162	046 026	LHI 026
30 164	162	LMD
30 165	054	INL
30 166	163	LME
30 167	142	LHD
30 170	153	LLE
30 171	026 026	LDI 026
30 173	036 000	LEI 000
30 175	315 046 012	CAL MOVEC
30 200	056 325	LLI 325
30 202	046 001	LHI 001
30 204	315 012 013	CAL INSTR
30 207	173	LAE
30 210	247	NDA
30 211	312 121 030	JTZ FORNXT
30 214	306 002	ADI 002
30 216	056 276	LLI 276
30 220	046 026	LHI 026
30 222	167	LMA
30 223	056 330	LLI 330
30 225	046 001	LHI 001
30 227	315 012 013	CAL INSTR
30 232	173	LAE
30 233	247	NDA
30 234	302 300 030	JFZ NEXT5
30 237	056 004	LLI 004
30 241	046 001	LHI 001
30 243	315 244 022	CAL FLOAD
30 246	056 304	LLI 304
30 250	315 255 022	CAL FSTORE
30 253	056 000	LLI 000
30 255	046 026	LHI 026
30 257	106	LBM
30 260	056 277	LLI 277
30 262	160	LMB
30 263	315 224 003	CAL EVAL
30 266	056 310	LLI 310
30 270	046 001	LHI 001
30 272	315 255 022	CAL FSTORE
30 275	303 351 030	JMP NEXT6
30 300	035	NEXT5, DCE
30 301	056 277	LLI 277
30 303	046 026	LHI 026

30 305	163		LME
30 306	315 224 003		CAL EVAL
30 311	056 310		LLI 310
30 313	046 001	**	LHI 001
30 315	315 255 022		CAL FSTORE
30 320	056 277		LLI 277
30 322	046 026	**	LHI 026
30 324	176		LAM
30 325	306 005		ADI 005
30 327	055		DCL
30 330	167		LMA
30 331	056 000		LLI 000
30 333	106		LBM
30 334	056 277		LLI 277
30 336	160		LMB
30 337	315 224 003		CAL EVAL
30 342	056 304		LLI 304
30 344	046 001	**	LHI 001
30 346	315 255 022		CAL FSTORE
30 351	056 144		NEXT6, LLI 144
30 353	046 026	**	LHI 026
30 355	066 000		LMI 000
30 357	056 034		LLI 034
30 361	046 027	**	LHI 027
30 363	315 012 013		CAL INSTR
30 366	173		LAE
30 367	247		NDA
30 370	056 202		LLI 202
30 372	046 026	**	LHI 026
30 374	167		LMA
30 375	312 121 030		JTZ FORNXT
31 000	306 003		ADI 003
31 002	056 203		LLI 203
31 004	167		LMA
31 005	056 203		NEXT7, LLI 203
31 007	315 240 002		CAL GETCHR
31 012	312 027 031		JTZ NEXT8
31 015	376 275		CPI 275
31 017	312 042 031		JTZ NEXT9
31 022	056 144		LLI 144
31 024	315 314 002		CAL CONCT1
31 027	056 203		NEXT8, LLI 203
31 031	315 003 003		CAL LOOP
31 034	302 005 031		JFZ NEXT7
31 037	303 121 030		JMP FORNXT
31 042	056 202		NEXT9, LLI 202
31 044	046 026	**	LHI 026

31 046	176	LAM
31 047	306 003	ADI 003
31 051	056 276	LLI 276
31 053	167	LMA
31 054	056 203	LLI 203
31 056	106	LBM
31 057	005	DCB
31 060	056 277	LLI 277
31 062	160	LMB
31 063	315 224 003	CAL EVAL
31 066	056 304	LLI 304
31 070	046 001	** LHI 001
31 072	315 277 022	CAL FACXOP
31 075	315 211 020	CAL FPADD
31 100	056 314	LLI 314
31 102	046 001	** LHI 001
31 104	315 255 022	CAL FSTORE
31 107	056 310	LLI 310
31 111	315 277 022	CAL FACXOP
31 114	315 032 021	CAL FPSUB
31 117	056 306	LLI 306
31 121	176	LAM
31 122	247	NDA
31 123	056 126	LLI 126
31 125	176	LAM
31 126	312 121 030	JTZ FORNXT
31 131	372 170 031	JTS NEXT11
31 134	247	NDA
31 135	372 177 031	JTS NEXT12
31 140	312 177 031	JTZ NEXT12
31 143	056 363	NEXT10, LLI 363
31 145	046 026	** LHI 026
31 147	136	LEM
31 150	055	DCL
31 151	126	LDM
31 152	055	DCL
31 153	163	LME
31 154	055	DCL
31 155	162	LMD
31 156	056 205	LLI 205
31 160	046 027	** LHI 027
31 162	106	LBM
31 163	005	DCB
31 164	160	LMB
31 165	303 116 013	JMP NXTLIN
31 170	247	NEXT11, NDA
31 171	302 177 031	JFZ NEXT12
31 174	303 143 031	JMP NEXT10

31 177	056 314		NEXT12,	LLI 314
31 201	046 001	**		LHI 001
31 203	315 244 022			CAL FLOAD
31 206	315 252 010			CAL RESTSY
31 211	315 055 010			CAL STOSYM
31 214	303 116 013			JMP NXTLIN
31 217	076 215		BACKSP,	LAI 215
31 221	315 202 003			CAL ECHO
31 224	315 202 003			CAL ECHO
31 227	056 043			LLI 043
31 231	046 001	**		LHI 001
31 233	066 001			LMI 001
31 235	056 124			LLI 124
31 237	176			LAM
31 240	247			NDA
31 241	370			RTS
31 242	310			RTZ
31 243	303 022 010			JMP TAB1
31 246	056 205		FOR5,	LLI 205
31 250	046 027	**		LHI 027
31 252	176			LAM
31 253	007			RLC
31 254	007			RLC
31 255	306 136			ADI 136
31 257	137			LEA
31 260	124			LDH
31 261	056 145			LLI 145
31 263	046 026	**		LHI 026
31 265	006 002			LBI 002
31 267	315 013 021			CAL MOVEIT
31 272	315 055 010			CAL STOSYM
31 275	303 116 013			JMP NXTLIN
31 300	056 176		PARSEP,	LLI 176
31 302	066 000			LMI 000
31 304	315 324 004			CAL PARSER
31 307	056 227			LLI 227
31 311	046 001	**		LHI 001
31 313	176			LAM
31 314	376 230			CPI 230
31 316	310			RTZ
31 317	303 152 011			JMP SYNERR

Note open addresses.

This space available
for patching.

31 330	041 352 001	**	EXECSP,	LXH 352 001
31 333	315 121 003			CAL TEXTC
31 336	311			RET

32 000	056 014	SQRX,	LLI 014
32 002	046 001		LHI 001
32 004	315 255 022		CAL FSTORE
32 007	056 126		LLI 126
32 011	176		LAM
32 012	247		NDA
32 013	372 217 032		JTS SQRERR
32 016	312 247 006		JTZ CFALSE
32 021	056 017		LLI 017
32 023	176		LAM
32 024	247		NDA
32 025	372 041 032		JTS NEGEXP
32 030	037		RAR
32 031	107		LBA
32 032	076 000		LAI 000
32 034	027		RAL
32 035	167		LMA
32 036	303 062 032		JMP SQREXP
32 041	107	NEGEXP,	LBA
32 042	257		XRA
32 043	220		SUB
32 044	247		NDA
32 045	037		RAR
32 046	107		LBA
32 047	076 000		LAI 000
32 051	217		ACA
32 052	167		LMA
32 053	312 057 032		JTZ NOREMD
32 056	004		INB
32 057	257	NOREMD,	XRA
32 060	220		SUB
32 061	107		LBA
32 062	056 013	SQREXP,	LLI 013
32 064	160		LMB
32 065	056 004		LLI 004
32 067	036 034		LEI 034
32 071	124		LDH
32 072	006 004		LBI 004
32 074	315 013 021		CAL MOVEIT
32 077	315 247 006		CAL CFALSE
32 102	056 044		LLI 044
32 104	315 255 022		CAL FSTORE
32 107	056 034	SQRLOP,	LLI 034
32 111	315 244 022		CAL FLOAD
32 114	056 014		LLI 014
32 116	315 266 022		CAL OPLOAD
32 121	315 322 021		CAL FPDIV

32 124	056 034		LLI 034
32 126	315 266 022		CAL OPLOAD
32 131	315 211 020		CAL FPADD
32 134	056 127		LLI 127
32 136	106		LBM
32 137	005		DCB
32 140	160		LMB
32 141	056 034		LLI 034
32 143	315 255 022		CAL FSTORE
32 146	056 044		LLI 044
32 150	315 266 022		CAL OPLOAD
32 153	315 032 021		CAL FPSUB
32 156	056 127		LLI 127
32 160	176		LAM
32 161	376 367		CPI 367
32 163	372 203 032		JTS SQRCNV
32 166	056 034		LLI 034
32 170	124		LDH
32 171	036 044		LEI 044
32 173	006 004		LBI 004
32 175	315 013 021		CAL MOVEIT
32 200	303 107 032		JMP SQRLOP
32 203	056 013	SQRCNV,	LLI 013
32 205	176		LAM
32 206	056 037		LLI 037
32 210	206		ADM
32 211	167		LMA
32 212	056 034		LLI 034
32 214	303 244 022		JMP FLOAD
32 217	076 323	SQRERR,	LAI 323
32 221	016 321		LCI 321
32 223	303 226 002		JMP ERROR

Note open addresses.
This space available
for patching.

32 240	056 064	RNDX,	LLI 064
32 242	046 001	**	LHI 001
32 244	315 244 022		CAL FLOAD
32 247	056 050		LLI 050
32 251	315 266 022		CAL OPLOAD
32 254	315 046 021		CAL FPMULT
32 257	056 060		LLI 060
32 261	315 266 022		CAL OPLOAD
32 264	315 211 020		CAL FPADD
32 267	056 064		LLI 064
32 271	315 255 022		CAL FSTORE
32 274	056 127		LLI 127

32 276	167	LMA
32 277	326 020	SUI 020
32 301	167	LMA
32 302	315 000 020	CAL FPFIX
32 305	056 123	LLI 123
32 307	066 000	LMI 000
32 311	056 127	LLI 127
32 313	066 000	LMI 000
32 315	315 064 020	CAL FPFLT
32 320	056 127	LLI 127
32 322	176	LAM
32 323	306 020	ADI 020
32 325	167	LMA
32 326	056 064	LLI 064
32 330	315 266 022	CAL OPLOAD
32 333	315 032 021	CAL FPSUB
32 336	056 064	LLI 064
32 340	315 255 022	CAL FSTORE
32 343	056 127	LLI 127
32 345	176	LAM
32 346	326 020	SUI 020
32 350	167	LMA
32 351	311	RET

Note open addresses
to end of page 32.

Pages 33 to remainder
of memory (or start of
optional ARRAY
handling routines) used as
USER PROGRAM BUFFER.

Optional ARRAY routines
assembled for operation in
the upper three pages of a
12 K system are listed here.

55 000	056 126	PRIGH1,	LLI 126
55 002	046 001	**	LHI 001
55 004	176		LAM
55 005	247		NDA
55 006	372 136 055		JTS OUTRNG
55 011	315 000 020		CAL FPFIX
55 014	056 124		LLI 124
55 016	176		LAM
55 017	326 001		SUI 001
55 021	007		RLC
55 022	007		RLC
55 023	117		LCA
55 024	056 203		LLI 203

55 026	046 027	**	LHI 027
55 030	176		LAM
55 031	356 377		XRI 377
55 033	007		RLC
55 034	007		RLC
55 035	306 120		ADI 120
55 037	046 027	**	LHI 027
55 041	157		LLA
55 042	054		INL
55 043	054		INL
55 044	176		LAM
55 045	201		ADC
55 046	157		LLA
55 047	046 057	††	LHI 057
55 051	303 244 022		JMP FLOAD
55 054	056 202		FUNAR2, LLI 202
55 056	046 027	**	LHI 027
55 060	106		LBM
55 061	004		INB
55 062	160		LMB
55 063	016 002		LCI 002
55 065	056 114		LLI 114
55 067	046 027	**	LHI 027
55 071	315 230 007		CAL TABADR
55 074	026 026	**	LDI 026
55 076	036 120		LEI 120
55 100	315 332 002		CAL STRCP
55 103	312 124 055		JTZ FUNAR3
55 106	056 202		LLI 202
55 110	046 027	**	LHI 027
55 112	176		LAM
55 113	056 075		LLI 075
55 115	276		CPM
55 116	302 054 055		JFZ FUNAR2
55 121	303 172 007		JMP FAERR
55 124	056 202		FUNAR3, LLI 202
55 126	046 027	**	LHI 027
55 130	257		XRA
55 131	236		SBM
55 132	167		LMA
55 133	303 207 007		JMP FUNAR4
55 136	076 317		OUTRNG, LAI 317
55 140	016 322		LCI 322
55 142	303 226 002		JMP ERROR
55 145	315 252 010		ARRAY, CAL RESTSY
55 150	303 160 055		JMP ARRAY2

55 153	056 202		ARRAY1, LLI 202
55 155	303 162 055		JMP ARRAY3
55 160	056 203		ARRAY2, LLI 203
55 162	046 026	**	ARRAY3, LHI 026
55 164	106		LBM
55 165	004		INB
55 166	056 276		LLI 276
55 170	160		LMB
55 171	056 206		LLI 206
55 173	160		LMB
55 174	056 206		ARRAY4, LLI 206
55 176	315 240 002		CAL GETCHR
55 201	376 251		CPI 251
55 203	312 225 055		JTZ ARRAY5
55 206	056 206		LLI 206
55 210	315 003 003		CAL LOOP
55 213	302 174 055		JFZ ARRAY4
55 216	076 301		LAI 301
55 220	016 306		LCI 306
55 222	303 226 002		JMP ERROR
55 225	056 206		ARRAY5, LLI 206
55 227	106		LBM
55 230	005		DCB
55 231	056 277		LLI 277
55 233	160		LMB
55 234	056 207		LLI 207
55 236	066 000		LMI 000
55 240	056 207		ARRAY6, LLI 207
55 242	046 026	**	LHI 026
55 244	106		LBM
55 245	004		INB
55 246	160		LMB
55 247	016 002		LCI 002
55 251	056 114		LLI 114
55 253	046 027	**	LHI 027
55 255	315 230 007		CAL TABADR
55 260	036 120		LEI 120
55 262	026 026	**	LDI 026
55 264	315 332 002		CAL STRCP
55 267	312 312 055		JTZ ARRAY7
55 272	056 207		LLI 207
55 274	046 026	**	LHI 026
55 276	176		LAM
55 277	056 075		LLI 075
55 301	046 027	**	LHI 027
55 303	276		CPM

55 304	302 240 055	JFZ ARRAY6
55 307	303 172 007	JMP FAERR
55 312	315 224 003	ARRAY7, CAL EVAL
55 315	315 000 020	CAL FPFIX
55 320	056 207	LLI 207
55 322	046 026	LHI 026
55 324	106	LBM
55 325	016 002	LCI 002
55 327	056 114	LLI 114
55 331	046 027	LHI 027
55 333	315 230 007	CAL TABADR
55 336	054	INL
55 337	054	INL
55 340	116	LCM
55 341	056 124	LLI 124
55 343	046 001	LHI 001
55 345	176	LAM
55 346	326 001	SUI 001
55 350	007	RLC
55 351	007	RLC
55 352	201	ADC
55 353	056 204	LLI 204
55 355	046 027	LHI 027
55 357	167	LMA
55 360	056 201	LLI 201
55 362	066 377	LMI 377
55 364	311	RET
55 365	315 255 002	DIM, CAL CLESYM
55 370	056 202	LLI 202
55 372	106	LBM
55 373	004	INB
55 374	056 203	LLI 203
55 376	160	LMB
55 377	056 203	DIM1, LLI 203
56 001	315 240 002	CAL GETCHR
56 004	312 017 056	JTZ DIM2
56 007	376 250	CPI 250
56 011	312 032 056	JTZ DIM3
56 014	315 310 002	CAL CONCTS
56 017	056 203	DIM2, LLI 203
56 021	315 003 003	CAL LOOP
56 024	302 377 055	JFZ DIM1
56 027	303 337 056	JMP DIMERR
56 032	056 206	DIM3, LLI 206
56 034	066 000	LMI 000

56 036	056 206		DIM4, LLI 206
56 040	046 026	**	LHI 026
56 042	176		LAM
56 043	007		RLC
56 044	007		RLC
56 045	306 114		ADI 114
56 047	046 027	**	LHI 027
56 051	157		LLA
56 052	036 120		LEI 120
56 054	026 026	**	LDI 026
56 056	315 332 002		CAL STRCP
56 061	312 301 056		JTZ DIM9
56 064	056 206		LLI 206
56 066	046 026	**	LHI 026
56 070	106		LBM
56 071	004		INB
56 072	160		LMB
56 073	056 075		LLI 075
56 075	046 027	**	LHI 027
56 077	176		LAM
56 100	005		DCB
56 101	270		CPB
56 102	302 036 056		JFZ DIM4
56 105	056 075		LLI 075
56 107	046 027	**	LHI 027
56 111	106		LBM
56 112	004		INB
56 113	160		LMB
56 114	056 076		LLI 076
56 116	160		LMB
56 117	056 206		LLI 206
56 121	046 026	**	LHI 026
56 123	160		LMB
56 124	176		LAM
56 125	007		RLC
56 126	007		RLC
56 127	306 114		ADI 114
56 131	137		LEA
56 132	026 027	**	LDI 027
56 134	056 120		LLI 120
56 136	046 026	**	LHI 026
56 140	315 046 012		CAL MOVEC
56 143	315 255 002		CAL CLESYM
56 146	056 203		LLI 203
56 150	046 026	**	LHI 026
56 152	106		LBM
56 153	004		INB
56 154	056 204		LLI 204
56 156	160		LMB
56 157	056 204		DIM5, LLI 204

56 161	315 240 002	CAL GETCHR
56 164	312 211 056	JTZ DIM6
56 167	376 251	CPI 251
56 171	312 224 056	JTZ DIM7
56 174	376 260	CPI 260
56 176	372 337 056	JTS DIMERR
56 201	376 272	CPI 272
56 203	362 337 056	JFS DIMERR
56 206	315 310 002	CAL CONCTS
56 211	056 204	DIM6, LLI 204
56 213	315 003 003	CAL LOOP
56 216	302 157 056	JFZ DIM5
56 221	308 337 056	JMP DIMERR
56 224	056 120	DIM7, LLI 120
56 226	046 026	LHI 026
56 230	315 044 023	CAL DINPUT
56 233	315 000 020	CAL FPFIX
56 236	056 124	LLI 124
56 240	176	LAM
56 241	007	RLC
56 242	007	RLC
56 243	117	LCA
56 244	056 076	LLI 076
56 246	046 027	LHI 027
56 250	176	LAM
56 251	326 001	SUI 001
56 253	007	RLC
56 254	007	RLC
56 255	306 122	ADI 122
56 257	157	LLA
56 260	046 027	LHI 027
56 262	106	LBM
56 263	306 004	ADI 004
56 265	157	LLA
56 266	170	LAB
56 267	201	ADC
56 270	167	LMA
56 271	056 204	DIM8, LLI 204
56 273	046 026	LHI 026
56 275	106	LBM
56 276	056 203	LLI 203
56 300	160	LMB
56 301	056 203	DIM9, LLI 203
56 303	315 240 002	CAL GETCHR
56 306	376 254	CPI 254
56 310	312 326 056	JTZ DIM10
56 313	056 203	LLI 203

56 315	315 003 003	CAL LOOP
56 320	302 301 056	JFZ DIM9
56 323	303 116 013	JMP NXTLIN
56 326	056 203	DIM10, LLI 203
56 330	106	LBM
56 331	056 202	LLI 202
56 333	160	LMB
56 334	303 365 055	JMP DIM
56 337	076 304	DIMERR, LAI 304
56 341	016 305	LCI 305
56 343	303 226 002	JMP ERROR

Note open addresses
to end of page 56.

Page 57 reserved
for use by the
ARRAY VALUES TABLE.

OPERATING SCELBAL

This chapter will present detailed information on the use of SCELBAL as a higher level language. Examples of the usage of the various types of commands, statements, and functions will be presented.

It is assumed in the following discussion that the reader has loaded an appropriate version of the program (either for an 8008 or 8080 system) along with the appropriate user provided I/O routines for whatever I/O devices will be used in the user's system. (Information on this subject is presented in a chapter titled "I/O Routines.") For the sake of discussion it will be assumed that an ASCII encoded keyboard is being utilized as the operator's input device and some sort of printing device is being used as the display mechanism.

STARTING SCELBAL

The SCELBAL program as presented in the assembled object code listings in this publication has a starting address of 10 266. Some users may wish to place a vector to this starting location in one of the RESTART locations available in 8008 or 8080 systems.

When the program is first started, the message:

READY

will be displayed on the output device to notify the operator that the program is in the EXECUTIVE COMMAND mode.

The SCRatch Command

There are five EXECUTIVE COMMANDS. All commands and other entries by the operator are terminated by entering a carriage return. Perhaps the first command the user should utilize when the program is first started is the SCRatch command. This com-

mand is issued by typing in the mnemonic:

SCR

following by a carriage return.

The program will acknowledge receipt of the SCRatch command by displaying the message READY. The SCRatch command effectively clears the USER PROGRAM BUFFER and VARIABLES LOOK-UP table. It should thus be used whenever the operator desires to start entering a new higher level language program into the USER PROGRAM BUFFER.

CALCULATOR Mode of Operation

SCELBAL is able to operate in two basic modes. The first type of mode will be referred to as the "calculator" mode. This mode is available at any time that the program is not actually performing operations in the second mode which is the stored program mode.

The "calculator" mode will be used to introduce some of the uses of the SCELBAL statement directives. The calculator mode is automatically assumed by the program if a statement is entered without being preceded by a line number.

The PRINT Statement

Perhaps the first type of statement to consider is the PRINT statement since this directive must be used whenever an operator desires to obtain some information from the program!

The next several paragraphs will discuss the use of the PRINT statement when the user does not precede the statement by a line number. The program will then be operating in the calculator mode and will immed-

iately execute the statement directive when it is terminated by a carriage return.

The use of the PRINT statement by itself will simply result in the issuance of a carriage return and line feed combination! This fact may be of little use when SCELBAL is being used in the calculator mode, but it is valuable when it is used in a program as it may simply be used to provide formatting spaces between lines of information outputted by a program!

In its more typical application, however, the PRINT statement may be followed by a variety of terms. These terms may either be interpreted as representing mathematical values (represented as numbers, variables or expressions), or text strings. To signify that terms following a PRINT statement are to be interpreted as a text string, they must be enclosed by single or double quotation marks. For example, the statements:

PRINT "HELLO"

or:

PRINT 'HELLO'

would result in the program displaying the text message:

HELLO

when the statement line was terminated by the operator striking the carriage return key. After displaying the HELLO text message, a carriage return and line feed combination would also be issued. In fact, a carriage return and line feed combination will always be issued at the conclusion of the execution of a PRINT statement unless the statement line is terminated by a comma (,) or semicolon (;). The comma and semicolon signs are special indicators when used in a PRINT statement line. Both signs may be used to separate terms in a line. However, the comma sign, while separating terms, will also provide a special feature. It will cause the display device to space over to the next "tabbing" position in the line being displayed by the program!

In the version of SCELBAL presented these "tabbing" positions are set at every sixteenth column in a line. (However, the tabbing positions may be modified. See the source listing for the PRINT statement.) The semicolon does not provide the tabbing capability. It is used in place of the comma sign when the programmer desires the output of the next term to begin on the next position in the line.

Several examples of the use of the comma and semicolon signs should be helpful to the reader at this point. The statement:

PRINT "HELLO";"HELLO";

would result in the output device displaying:

HELLOHELLO

That is, the two words would be run together. Additionally, since the statement line also ends with a semicolon, the display mechanism would not issue a carriage return and line feed combination after the second word. The display unit would be positioned to start typing at the next character position in the same line.

The statement line:

PRINT "HELLO","HELLO",

would result in the output device displaying:

HELLO HELLO

The second word would start at the sixteenth column position in the line. Since a comma was also used to end the line, no carriage return and line feed combination would be issued and the display device would be positioned to start typing (the next time a PRINT statement was encountered) at the thirty-second column in the line.

It will be mentioned that a text string may consist of letters, numbers, words, and punctuation marks; including the comma and semicolon signs! Whatever is enclosed within quotation marks on a PRINT statement line

will be considered part of the text message. Thus, one may have entire sentences displayed. The statement:

```
PRINT 'HELLO! I AM A COMPUTER.');
```

Will result in then sentence:

```
HELLO! I AM A COMPUTER.
```

being displayed.

When terms on a PRINT statement are not enclosed by single or double quotation marks they are assumed to represent mathematical quantities. Mathematical quantities may be expressed in the form of a number, the name of a variable, or a combination of these two forms coupled by mathematical operators which would be considered an expression.

The statement:

```
PRINT 123456;
```

will result in the display of the number:

```
123456
```

(Some readers might note that in the case of pure simple numbers, one would get the same result when using the PRINT statement when the number was enclosed in quotation marks!)

A statement such as:

```
PRINT A;
```

would result in the current value of the variable named A to be displayed. If the operator entered the above statement immediately after using the SCRatch command, the system would display:

```
0
```

as the variable A would have had no previous value assigned to it.

Perhaps the most common application of the PRINT statement when used in the cal-

culator mode, is to use it to obtain the value of a mathematical expression. For example, typing in:

```
PRINT (412*3.14159/16)*14
```

would result in the program displaying the result of performing the calculations contained in the expression. The number:

```
1132.544
```

would thus be displayed back to the operator. Of course, one does not have to use pure numbers in a mathematical expression that is to be evaluated. If the values for variable names have previously been defined (by the LET statement which will be discussed later) so that, for instance:

```
A = 412  
B = 3.14159  
C = 16  
D = 14
```

and the PRINT statement:

```
PRINT (A*B/C)*D
```

was entered, the result:

```
1132.544
```

would again be displayed back to the operator. Naturally, one may also mix variable names and numeric values in an expression. However, when in the calculator mode, if variables are being used, one must ensure that they are first defined before attempting to use them in an expression. Otherwise, their values will be zero when they are encountered in the expression.

The use of the PRINT statement when it is used as part of a program (in which case the statement is preceeded by a line number) is essentially the same as described. However, in the stored program mode, one is more likely to make use of the capability of having both text messages and mathematical values displayed using a single PRINT statement. An

example of this capability is illustrated here:

10 PRINT 'THE ANSWER IS: ' ;(A*B/C)*D

would result in the program displaying:

THE ANSWER IS: 1132.544

(Assuming the variable values were the same as mentioned earlier in the discussion when the statement was executed!)

The reader should note in the statement line above that a space character was inserted after the colon at the end of the text string just before the quotation character marking the end of the text string. This was done so that there would be normal spacing between the text string and the answer when it was displayed. The prospective high level language programmer should keep the tip in mind when mixing text strings and mathematical values as in the above example.

The PRINT statement is truly a "work-horse" directive in SCELBAL. It controls all the outputting of data to the operator. The above discussion covers the primary forms of its use. However, later the reader will see how several special functions (TAB and CHR) may be used within the statement to provide even more output capability and flexibility.

Remember, when you want some output from SCELBAL, tell it to PRINT.

PRECEDENCE of Mathematical Operators

The PRINT statement just discussed, and many other statements in SCELBAL may cause the program to evaluate mathematical expressions in order to obtain a numerical value. In order to perform such calculations in a consistent manner, it is necessary to establish a system of "operator precedence," and rules for evaluating an expression. This system must be learned by the high level programmer because it is the system that has been "fixed" in the computer.

The first rule to learn is that SCELBAL evaluates all expressions by proceeding to "read" expressions on a left to right basis.

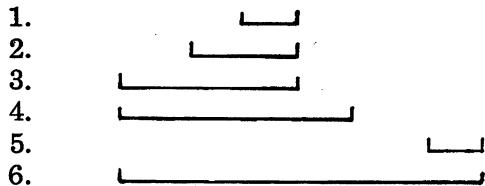
As a mathematical expression is read from left to right, mathematical terms (numbers and variable names representing numbers) are joined by operator signs. (These are the parenthesis, exponentiate (" \uparrow "), multiply ("*"), divide ("/"), add ("+") and subtract ("−") signs.) The operator signs are assigned PRECEDENCE values which are used to determine when to perform an operation. The highest operator precedence is assigned to the exponentiate sign. Next, having equal precedence are the multiply and divide signs. Lower in the precedence order are the add and subtract signs (having equal precedence with each other). Terms within parenthesis are always evaluated before proceeding further in a line. (Thus, the left hand parenthesis sign serves as a start of group marker, the right hand one as an end of group marker.)

As an expression is evaluated, each mathematical term (number or variable value) is saved on an ARITHMETIC STACK. Each time an operator is encountered, a test is made to determine if that operator is less than or equal to any previously unprocessed operator that has been encountered. If not, that is, if the operator is higher in precedence than the preceding one encountered in the line, it is saved on an OPERATOR STACK. If, however, the precedence of the operator is less than or equal to the previous one, the operation dictated by the previous operator is executed upon the two previous mathematical terms contained in the arithmetic stack.

This method of stacking the lower precedence operators results in the higher precedence operations being performed first!

This may be seen more clearly, perhaps, by following the evaluation of an expression which contains a variety of terms with different precedences. Such an expression is presented on the next page.

(X + 2 * Y^N - 1) + Z * N



When the above expression is scanned by SCELBAL the left hand parenthesis will be the first operator detected. A left hand parenthesis is always placed on the operator stack as though it had the highest possible precedence. However, once on the stack, its precedence is changed to be lower than all other operators. This precedence switching "trick" results in all the terms within a pair of parenthesis being evaluated before remaining terms on a line are calculated as will become apparent shortly.

The first term to be encountered in the example expression is the name of a variable called X. This variable is followed by the operator "+" for addition. The "+" operator has a higher precedence than the left hand parenthesis on the operator stack. (Remember, once the left hand parenthesis is on the operator stack, it has the lowest operator sign precedence!) Thus, the "+" sign will be placed on the top of the operator stack. The value of X will be placed on the arithmetic stack.

Next, the program will find the number 2 followed by the "*" multiplication sign. The multiplication sign has higher precedence than the "+" sign on the top of the operator stack so it becomes the top entry on that stack. The number 2 is added to the top of the arithmetic stack.

Continuing to scan the line the program will find the variable name Y and the "^N" exponentiate operator. The exponentiate operator has higher precedence than the "*" operator so it is placed on the operator stack. The value for Y is placed on the arithmetic stack.

Next, the variable name N and the minus sign operator "-" will be scanned. This is the

first point in the line that will result in actual mathematical operations being performed! This is because the "-" operator has a lower precedence than the exponentiate sign processed earlier in the line. Since it does have a lower precedence, the exponentiate operator must be executed. It will operate on the value for the variable N and Y (stored on the top of the arithmetic stack). The result of that operation will be placed on the top of the arithmetic stack. The exponentiate operator is removed from the top of the operator stack. Now, the top of that stack will contain the "*" operator. The "-" has lower precedence than the "*" operator too, so now the multiplication operation can be performed. It will be performed between the previously calculated quantity Y^N and the number 2. Those two quantities on the arithmetic stack are replaced by the result of the multiplication operation. The "*" operator is removed from the top of the operator stack. Now the "+" sign will be on the top of that stack. The current "-" sign is equal in precedence to the "+" sign so once again the operation on the stack is performed! The quantity 2 times Y to the N power will be added to the value of X. Those two entries in the arithmetic stack are replaced by the current total. At this point, only the left hand parenthesis sign, having a lower precedence than the "-" sign is left on the operator stack. Thus, the "-" sign is placed on top of the operator stack.

As the program continues to scan the expression it will next encounter the number 1 and the right hand parenthesis operator. The ")" operator has the lowest possible operator sign precedence. Therefore, all operations on the operator stack must be performed until the initiating left hand parenthesis is located! In this case, the "-" operator is on top of the operator stack. So, the quantity one will be subtracted from the quantity X plus 2 times Y raised to the power N. The result is placed on the top of the arithmetic stack replacing the previous entry. The "-" sign is removed from the operator stack leaving just the initial "(" left hand parenthesis sign. This effectively cancels with the right hand parenthesis leaving the operator stack cleared.

At this point the arithmetic stack contains the value of the expression contained in the parenthesis. The operator stack is empty. The program will continue to scan the line and pick up the “+” operator that follows the right hand parenthesis. Since the operator stack is empty, the “+” sign will be placed on the top of the stack. Next, the program will find the variable name Z and the operator “*” for multiplication. The multiplication sign has higher precedence than the “+” sign which is on the operator stack so the “*” sign is placed on the stack with the value for the variable Z going on top of the arithmetic stack.

Finally, the program will encounter the second occurrence of the variable name N at the end of the expression. When the end of an expression is reached, all terms in the line are processed according to the operators contained in the operator stack. The top of the operator stack contains the “*” sign. Thus, the values for the variables Z and N will be multiplied. The result will replace those entries on the top of the arithmetic stack. The “*” sign is removed from the operator stack leaving just the “+” operator. The quantity Z * N will then be added to the quantity in parenthesis which will yield the final result for the entire expression contained on the line!

The lines under the expression on the previous page illustrate the order in which actual operations would be performed when the expression was evaluated by SCELBAL.

Readers who desire a more detailed explanation of the process involved in evaluating mathematical expressions should refer to the appropriate chapters which present and explain the machine language routines which perform the mathematical evaluations.

The user of SCELBAL must realize that the precedence given to mathematical operators is important when writing the mathematical expressions that one desires to have a program solve. If one desires to have an expression such as the quantity N plus 2 multiplied by

the quantity M minus 3, it must be written in the form:

$$(N + 2) * (M - 3)$$

and not:

$$N + 2 * M - 3$$

For, the latter format would result in the expression being evaluated as the quantity N plus the quantity 2 times M minus the quantity 3!

Remember, the higher precedence operators are executed first! A good rule of thumb, when in doubt about the precedence rules, is to group terms using parenthesis. Thus, to raise 2 to the N minus one power, write the expression as:

$$2 \uparrow (N - 1)$$

not:

$$2 \uparrow N - 1$$

The second format will result in one being subtracted from the quantity 2 raised to the power N.

What other rules must the programmer know about writing mathematical expressions when using SCELBAL? This: Terms (names of variables, or numbers) must be separated by operators, and, as a general rule, operators must be separated by terms. EXCEPT, when the operator is a parenthesis! A parenthesis must always be followed by an operator other than an opposite parenthesis.

Thus, the following formats are valid:

$$\begin{aligned} &A + B \\ &(A + B) * (C + D) \end{aligned}$$

and the following are not valid:

$$\begin{aligned} &A + * B \\ &(A + B) (C + D) \\ &(A + B) N \end{aligned}$$

Special mention must be made of the case when the programmer desires to use the minus sign as a unary operator. That is, when it is used to specify a minus number. The mathematical routines in SCELBAL perform the unary minus operation by subtracting the value that follows the minus sign from zero. Thus, if one enters the expression:

A/-B

The program will attempt to perform the operation as:

A/0 - B

This particular example case would result in a program error message being generated for an attempted divide by zero operation!

The proper way to handle expressions containing the unary minus operator is to enclose the term and the unary minus sign in parenthesis. The above example expression would be properly executed if it was written as:

A/(-B)

because the program would execute it as:

A/(0 - B)

(Provided that B is not zero in this case!)

NUMBERS

Numerical values used in mathematical expressions for SCELBAL programs may be entered in two formats. Decimal fixed point notation and decimal floating point notation.

With either notation, the programmer is limited to six to seven significant decimal digits. SCELBAL will accept six significant digits at all times, and seven significant digits if the number does not exceed 5242879. This limitation on the size of seven digit numbers is related to the manner in which the mathematical input buffer is limited to prevent its

overflow. If one attempts to enter numbers larger than this only the first six digits (if the number is larger than 524287) or seven digits (if the number is between 524287 and the number 5242879) will be used. Perhaps the easiest rule of thumb for the novice programmer to remember is to simply limit inputs to six significant digits.

The number of significant digits one can enter of course provides the range in which numbers may be entered using fixed point notation. The limitations of fixed point notation in terms of the magnitude that they may express may be extended by the use of floating point notation. Floating point notation allows the programmer to specify an exponent portion indicating the power to which a number may be raised. Using floating point notation, one may enter numbers having magnitudes from minus the 38th to plus the 38th power of ten!

Some examples of numbers that may be entered using fixed and floating point notation are shown below.

1.234567
1234.567
999999
.0000123
105.68E+12
0.045E-9

All numbers entered in a SCELBAL program will be converted to binary floating point notation and manipulated in that format during calculations. Calculations are performed by the floating point mathematical routines contained in the program with all calculations maintained to twenty-three binary bits of precision (for the mantissa portion) and seven binary bits for the exponent. These values limit the precision and range of numbers that the program can successfully handle. Several factors are important from the programmers view point.

One important factor for the programmer to keep in mind is that if calculations exceed the allowable range of the floating point regis-

ters and cause the binary exponent to overflow, that the results will be erroneous. From the programmer's viewpoint, using decimal numbers, this means that the programmer must ensure that a program will not attempt to perform calculations where the decimal exponent value would exceed plus or minus the thirty-eighth power! Thus, performing a calculation such as:

1.0E+24 * 2.0E+20

which would theoretically yield a result of:

0.2E+45

would cause the floating point binary exponent register in SCELBAL to overflow and results displayed to the operator would be meaningless.

It is easy enough for a programmer to remember the allowable range of numbers for SCELBAL when performing routine calculations. However, one must be alert to cases where the possibility of exceeding the allowable range is hidden in a program. This case is more likely to occur in a program where one starts raising numbers to a power. For instance, if one has a program with a calculation such as:

$N \uparrow (X)$

and proceeds to iterate X, a point will be reached where the allowable range of magnitude as discussed above is exceeded. The user has been cautioned!

Another parameter that the programmer will want to keep in mind relates to the accuracy with which calculations can be maintained in a program. Since the floating point binary registers in which numbers are held are limited to twenty-three binary bits, fractional results from operations such as multiplication and division are rounded off to leave the 23 most significant binary bits. This operation may introduce a small error, particularly when the results of operations involve a non-ending binary series. In a chain of operations

such errors can accumulate. These small errors will often affect the least significant decimal digit displayed to the operator. The novice programmer who is not used to digital calculations may be initially surprised to find that a directive such as:

PRINT 999999

will result in the display showing:

999999.5

Or, a directive such as:

PRINT 500 * 500

will result in the answer:

250000.1

The first example above might be particularly surprising to an operator who surmises that the program cannot even display back the same value entered! The result one obtains in the first answer is affected by the fact that the program in performing the directive, actually converts the decimal number to floating point binary notation, and then performs the reverse procedure. The conversion process involves multiplying the binary number (representing 999999) by the value 0.1 (decimal). The value 0.1 in binary notation is a non-ending series that must be rounded to twenty-three binary bits. The rounding process during the conversion results in the error factor shown for the example.

The actual amount of error that can accumulate in a calculation depends on the actual numbers involved, the extent of chaining of calculations involving non-ending series, and so forth. It is not the purpose of this presentation to go into a discussion of the factors relating to the precision and accuracy of calculations performed on a digital machine. The main point being made here is that such deviations are normal.

(Actually, the reason the deviation in the above examples shows up is because the pro-

gram permits the display of seven digits, even though the entry was only six. Users who find the display of the above types of small errors disconcerting may consider revising the appropriate section of the floating point output routine to limit the display to six significant digits!)

Numbers outputted by SCELBAL are automatically displayed in fixed point format if they are in the range:

1.0 to 8388608

Numbers outside this range are automatically displayed in floating point format which appears as shown below.

0.8388609E+07

VARIABLE NAMES

SCELBAL allows the operator to create mnemonic names to represent variable values. All names must begin with a letter of the alphabet and regular variable names may consist of one or two characters. The second character of a regular name may be a number if desired. Some typical regular variable names might be:

A
AA
A1

Examples of illegal variable names would be:

1A
AAA

Up to twenty regular variable names may be used in a program.

The terminology "regular variables" refers to variables not associated with the optional DIMension or ARRAY handling capabilities of SCELBAL. Names of variables associated with an array will be called ARRAY VARIABLES. An array variable name may only consist of one letter of the alphabet and must

always be followed by a subscript enclosed in parenthesis. An example of an array variable name would be:

A(1)

Up to four array names may be assigned in a program (independent of the number of regular variable names assigned). Additionally, since an array variable name is identified by the presence of a subscript, the same letter may be assigned to an array variable and a regular variable in a program.

The LET Statement

Now that the prospective SCELBAL programmer has been introduced to some of the fundamental aspects of the language - enough so that one may sit at the keyboard and try the various capabilities of SCELBAL as they are explained, it is time to proceed to introduce and explain the use of the various remaining types of STATEMENTS that the program can interpret.

The LET statement is used to define the value of a variable name. This statement actually has two forms. The express form, and the implied form. The express form is implemented by entering the statement keyword LET followed by the name of the variable that is to be defined. The variable name is then followed by the equal ("=") sign. The statement line is concluded by expressing, in mathematical terms, just what the value for the variable name will be when the directive is executed. This may be signified by giving an actual numerical value, by specifying another variable name, or by a mathematical expression. Some typical LET statements are illustrated here:

LET X = 100
LET X = Y
LET X = 100 * Y

When a LET statement is executed, the value of the variable indicated on the left hand side of the equal sign will be made equal

to whatever quantity is specified to the right of the equal sign.

Because the LET statement is used so frequently in programs, it is the one statement type in SCELBAL that can be interpreted without actually giving the LET keyword at the start of a statement line. Thus, when no keyword is found at the start of a statement line, the program assumes that an IMPLIED LET statement is being processed. The LET statements given as examples previously could have been directed by simply stating:

```
X = 100  
X = Y  
X = 100 * Y
```

The IF Statement

The IF statement allows the programmer to have the program make a logical decision based on the value of an expression at the time the statement is encountered.

The IF statement has two basic formats:

```
IF X = Y GOTO LL
```

or,

```
IF X = Y THEN [NEW STATEMENT]
```

That is, a test may be made to see if the value of an expression has reached a certain point (or is within a selected range), and, if so, the program may be directed to jump to a specific statement line number in the program being executed. (This is indicated by the format that has the GOTO directive.) Or, by using the THEN directive, one may have the program proceed to execute a different statement. (That is, execute the statement that immediately follows the THEN directive on the same line.) If the conditional test made in the IF statement should fail (i.e., in the two examples just given the value of X was not equal to Y), then the program does not perform the GOTO or THEN directive and instead proceeds to the

next statement line in the program.

Now, remember this: The test specified in the IF statement does not have to be restricted to just testing for simple equality! Any of the following test conditions may be specified in place of the equal sign:

```
>  
<  
<=  
=>  
<>
```

Remember too, that both sides of the conditional sign(s) may contain mathematical expressions. They need not be just simple variable names as used in the format examples.

Some typical examples of the IF statement in use are shown next.

```
IF X <> 50 THEN GOSUB 120
```

(If X is not equal to 50 then perform the subroutine that starts at line number 120 in the program. Else, continue with the next line in the program.)

```
IF X = (A*B*C) GOTO 90
```

(If X is equal to the value of another mathematical expression, go directly to program line number 90. Else, continue with the next line in the program.)

```
IF X + 5 > Y - 10 THEN LET X = 1
```

(If the quantity (X + 5) is greater than the quantity (Y - 10) then reset X back to 1 by executing the LET statement on the same line. If the condition is not met, then the LET statement on the same line is not executed.)

The IF statement is a powerful statement that has many applications in higher level programming. One particularly effective application for this type of statement is to use it to create an effective conditional CALL instruction as shown in one of the examples above.

The GOTO Statement

The GOTO statement directive is simply used to direct the program to jump to a specified line number in a program. Its format is:

GOTO LL

where LL stands for any line number assigned in a program.

The GOTO statement is typically used to direct a program around a portion of a program (that might, for instance, contain a subroutine). It is also frequently used to direct a program back to a particular starting point in a program requiring multiple execution of the same series of instructions.

The GOSUB Statement

The GOSUB statement is similar to the GOTO statement just presented. It will cause the program to jump to a specified line number. However, before doing so, it will effectively save the value of the next line number in the program. (That is, the line number of the line that follows the line on which the GOSUB statement is found.) The line number it saves is placed on the top of a software last-in first-out stack. This process will enable the program to return to the line number following the GOSUB statement when the SUBROUTINE it is directed to has been executed. The GOSUB statement should only be used to cause a jump to another section in a program when that section has been organized as a subroutine (as will be explained in the next statement type to be discussed).

SCELBAL as presented in this publication has enough stack memory allocated to allow the program to nest up to eight subroutines at one time.

The format for the GOSUB statement is exactly the same as the GOTO statement. The statement keyword is given, followed by the line number to which the program is to jump to in order to start the execution of the de-

sired subroutine.

The RETURN Statement

The RETURN statement line is used to indicate the end of a group of statements that form a subroutine. When a RETURN statement is encountered, the program will return to the program line number found at the top of the last-in first-out GOSUB software stack. It will then remove that line number from the stack.

The INPUT Statement

The INPUT statement is used when a programmer wants to have a program stop and accept data from an operator. The format for the INPUT statement is:

INPUT A

where A is the name of a variable used in the program. Inputting of data for more than one variable may be specified using a single INPUT directive by separating the names of variables by a comma:

INPUT A, B, C, D, E,

When the INPUT statement is encountered during the operation of a program, a question mark ("?") will be displayed and the program will wait for the operator to enter the value of the variable. When the operator has completed the input operation (signified by entering a carriage-return) the program continues operation.

The INPUT statement may be used to have an operator enter a value for a regular variable as well as an array variable (if the optional DIM capabilities are included in the program being operated). Additionally, the INPUT statement is able to perform a special function related to the inputting of alphabetical characters which will be explained in a later section. (See the section on the CHR function further on in this chapter.)

The FOR/NEXT Statements

The FOR and NEXT statements allow the programmer to form iteration loops in a program with ease.

Essentially, the FOR statement is used to specify a range of values over which a parameter is to be varied in specific increments. Statements following the initial FOR directive may then be used to perform whatever calculations are desired as the specified variable is varied. The program statement lines that are a part of the program loop are delimited from other lines in a program by use of the NEXT statement.

Suppose, for instance, that a programmer wanted to solve a simple formula when a particular variable value was varied in unit increments from 1 to 10. The following program loop using the FOR/NEXT statements could be used:

```
100 FOR X = 1 TO 10  
110 LET Z = X*X + 2X + 5  
120 PRINT X,Z  
130 NEXT X
```

Note that the FOR statement in line number 100 specifies the name of the variable that is to be incremented (X), and the range over which it is to be varied (1 TO 10). Also note, that when not otherwise indicated, the increment or STEP size by which the variable value will be changed each time the FOR/NEXT loop is traversed, will be ONE. That is, the IMPLIED STEP size in a FOR statement is the value 1.0!

Lines number 110 and 120 contain directives to evaluate and display the results of a calculation involving the variable that will be varied by the FOR/NEXT loop.

Line 130 contains the NEXT statement that concludes the FOR/NEXT loop. Note that the NEXT statement must be followed by the name of the variable that is incremented and referred to in the initiating FOR statement!

The format of the FOR directive may be altered to allow the programmer to change the STEP size from the IMPLIED value of 1.0 to any desired value. This is accomplished by adding the STEP directive to the FOR statement line. Thus, if one desired to modify the example program just illustrated so that it evaluated the formula in line number 110 for every odd value of X in the specified range, one would simply make line number 100 appear as:

```
100 FOR X = 1 TO 10 STEP 2
```

The reader may take note of the fact that the range specified in the FOR statement may cover both positive and negative numbers. Furthermore, the STEP size may be made a negative number so that the value of a parameter is decremented over a designated range!

FOR/NEXT loops, like subroutines (using GOSUB statements), may be nested one inside another up to a maximum of eight levels in the version of SCELBAL presented. (This nesting of FOR/NEXT loops is independent of subroutine nesting.) However, the order in which nesting occurs is important. The nesting rule is: Last-in, first-out. For instance, the following order of nesting is valid:

```
200 FOR X = 1 TO 10
```

```
250 FOR Z = 1 TO 5
```

```
290 NEXT Z  
300 NEXT X
```

The nesting order below would be invalid:

```
200 FOR X = 1 TO 10
```

```
250 FOR Z = 1 TO 5
```

```
290 NEXT X  
300 NEXT Z
```

The reader should study the two examples and make sure the difference between the two types of nesting is understood. Stated in different terminology, the rule says that a FOR/NEXT loop inside a FOR/NEXT loop must be COMPLETED before the outer (first) loop is referred to by its delimiting NEXT statement.

The REMarks Statement

The REMarks statement is used to inform the interpreter that the information on the line is not connected with program execution. The REM directive should be used whenever the programmer wants to make notes that may be of interest to programmers. The information on lines containing the REM statement thus serves to document a program but has no other capability as far as program operation is concerned. During program execution the interpreter will ignore the contents of a line prefaced (after the line number) by the REM keyword.

The END Statement

The END statement may be used to signify the end of a high level program. When the interpreter encounters an END statement it will return control to the EXECutive portion of the program. (Control will also return to the EXECutive when the interpreter reaches the last line in the USER PROGRAM BUFFER. However, there are many cases in higher level programming, such as when subroutines are used, where the last line in the program may not be the point where program operation is to be halted!)

The Optional DIM Statement

If the system owner has elected to operate the version of SCELBAL that includes the optional capability of defining and manipulating single dimension arrays, then the DIM statement must be used to reserve space in the ARRAY VALUES TABLE for the variable values that will be associated with an array

variable name.

The DIMension statement is simply used to specify how many locations are to be reserved for variable values associated with a particular array variable name.

The basic format of the DIMension statement is shown below.

DIM A(††)

where A may be any array variable name (remember, array variable names may only consist of one letter), and "††" represents an integer value in the range of 1 to 64 indicating the number of elements in the array.

Now, when the optional array handling capability is installed in SCELBAL as discussed in this publication, a special page in memory is set aside for holding the values of array elements. This page can hold the values for up to 64 elements. These 64 elements may all be referenced by one array name, or, they may be DISTRIBUTED amongst up to four array variable names. Thus, one may assign 32 elements to two array variable names (or split it 63 to 1). Or, assign 16 elements to four array variable names (or split it 56, 4, 2 and 2 if desired). It makes no difference as long as the maximum value of four array variable names, and a total of 64 elements distributed amongst all the variable names is not exceeded!

Since up to four array variable names may be assigned and DIMensioned in a program, the programmer may specify the number of elements in several arrays using a single DIM statement. This is done by separating the array defining terms in a DIM statement line by a comma as illustrated here:

DIM A(††),B(††),C(††),D(††)

Once an array variable name has been defined and space for elements reserved for it by using the DIM statement, one may refer to in-

dividual elements of the array by using the array name followed by the element number enclosed in parenthesis. An element number may be expressed as an integer digit or digits, a variable name (the variable value should represent an integer number), or a mathematical expression as long as the expression does not contain parenthesis. Thus, the format for specifying a particular element in an array might appear as:

A(5)
A(X)
A(X + 5)

Notice that while an element of an array may be referenced using a variable name, the actual process of defining how many elements are to be assigned to an array using the DIM statement must be accomplished using an actual integer number and not a variable name or mathematical expression!

Note too, that all arrays in SCELBAL are single dimension. (However, it is possible to perform calculations involving two dimensional matrices using the single dimension array capability of the language.)

FUNCTIONS

The power of SCELBAL provided by the high level statement types just discussed is further enhanced by the availability of seven special functions that may be used in various types of statements. Additionally, SCELBAL has been provided with capability to recognize an additional function name. When this special function name is recognized by the program, it will direct program operation to an address specified by the system manager. That address would indicate the starting point of a user defined function that the reader may create using machine language programming methods.

The various types of functions provided in SCELBAL are discussed next.

The INTeger Function

The string of characters INT immediately followed by a parenthesis containing a number (or expression) indicates the program is to calculate the INTeger value of the number or expression. The INTeger value is defined here as the greatest integer number less than or equal to the number specified. For example, the directives remember, functions must always be part of a valid statement line:

INT(1.00001)
or
INT(1.5)
or
INT(1.99999)

would all result in the answer:

1.0

being displayed as the number 1 is the largest INTeger number that can be contained in any of the numbers expressed as the argument portion of the directives illustrated.

Remember, when dealing with negative numbers, that if the order of numbers is viewed on a scale that goes from left to right such as:

-5 -4 -3 -2 -1 0 +1 +2 +3 +4 +5

that the number minus four (-4) is greater than minus five (-5), thus the directive:

INT(-1.999)
or
INT(-1.001)

will result in the answer:

-2.0

being displayed in accordance with the definition given above for the INTeger value of a number!

The SiGN Function

The mnemonic SGN signifies a function that will check the sign (positive or negative) of a number, variable or expression and return a simple value of +1.0 if the sign of the value is positive. It will return -1.0 if the sign of the value is negative. Zero will be returned if the value is zero.

The value to be tested (typically a variable or expression) must be enclosed in a pair of parenthesis immediately following the SGN mnemonic as illustrated in the following examples:

SGN(R1)

or

SGN(X*2 + 4*X - 16)

The ABSolute Function

The ABSolute function simply returns the magnitude of the number, variable or expression that is enclosed in parenthesis immediately following the mnemonic, without regard to the sign of the value. Thus, the directives:

ABS(+8423)

and

ABS(-8423)

would result in the value:

8423

being returned when the function was executed.

The SQuare Root Function

This function simply returns the square root of the value that follows the SQR mnemonic. As with the other functions, the value or argument portion of the function must be enclosed in parenthesis. The argument of the function may be a number, a variable, or an expression. However, it must be greater than

or equal to zero. Attempting to obtain the square root of a value less than zero will result in an error message being displayed. (A good way to avoid such error messages, suitable in a good many applications, is to take the SQuare Root of an ABSolute value!) Remember, the value from which the square root will be extracted must be enclosed in parenthesis immediately following the function mnemonic as in the example:

SQR(49)

The result returned for the example would, of course, be the number 7.

The RaNDom Number Function

The following directive:

RND(0)

will result in a semi-psuedo-random number being generated in the range from zero to one. The random number obtained may be further manipulated to place it in a range suitable to the user. For instance, if the user desired to generate an integer value in the range 0 to 9 using the random number function, the expression:

INT(RND(0)*10)

could be used.

NOTE: All functions defined for the high level language SCELBAL require that the function mnemonic be followed by an argument enclosed in parenthesis. Following the RND mnemonic by "(0)" serves merely to satisfy this requirement and has no other significance.

The RaNDom function provided in the language has many applications in programs involving games and in simple statistical analysis. However, the version provided in this publication should not be considered as completely unbiased nor used in applications requiring strict scientific randomness.

The CHaRacter Function

The mnemonic CHR followed by a number, variable, or expression enclosed in parenthesis, when the value is within the range of those used in the ASCII code set, may be used to display the alphanumeric character that corresponds to the value. Thus, for instance, the directive:

CHR(193)

contained within a PRINT statement line, or the directive:

CHR(X)

in such a statement line, when X was equal in value to 193, would result in the character:

A

being displayed on the system's output device.

NOTE that the CHR function is intended only for use within a PRINT statement line!

A list of the decimal values that correspond to a subset of ASCII characters that SCELBAL is designed to operate with may be obtained by running a sample illustrative program provided later in this chapter.

The CHR function, as just described, allows the programmer to present numerical values as alphanumeric characters. There is a reverse function available in SCELBAL that allows the programmer to have alphanumeric characters which are being inputted using an INPUT statement converted to decimal numeric values corresponding to their ASCII code! The reverse function is specified by following a variable name in an INPUT statement by a dollar ("\$") sign. Thus, the following directive in an INPUT statement:

X\$

would indicate that the variable value assigned to X would be the decimal ASCII value for whatever character was entered by the operator when the directive was executed. Thus, if the operator entered the letter:

A

when the INPUT statement was executed and the program paused for the operator's response, then the value:

193

would be assigned to the variable name X as that is the decimal representation for the ASCII code that represents the letter A!

When the dollar sign follows the name of a variable in an INPUT statement, meaning that the special conversion function is to be performed upon whatever character is entered, the program will not print a question mark ("?") as it does for a regular variable entry. Instead, the program will simply wait for the operator to enter a character. Furthermore, once a character has been entered, the program will automatically continue operation. It is not necessary to enter a carriage return following the alphanumeric entry as is the case when one desires to terminate a purely numeric entry. This operation, the reader will discover, makes it possible to develop programs whereby the operator may respond with alphanumeric strings as will be illustrated in one of the sample programs in this chapter.

The TAB Function

The TAB function is also restricted to use only within a PRINT statement. The purpose of this function is quite simple. It permits the programmer to direct that the output device move over (tab) to a specified column number. The column number to which the display device is to move is simply the number that is enclosed in parenthesis immediately following the TAB mnemonic. For in-

stance, if a PRINT statement line contained the directive:

TAB(40)

then the display device would tab over to the fortieth position in the line it was currently on.

There are several powerful features that the programmer will want to remember regarding the TAB function. First, the argument of the function may be specified as a variable value or expression involving variable values. Second, the TAB function can effectively simulate backspacing in the event the column specified has already been passed by the display device. These features make the TAB function valuable for displaying data using graphic techniques. A sample program in this chapter will illustrate the use of the TAB function for such purposes.

The User DeFined Function

The use of the mnemonic UDF followed by an argument enclosed in parenthesis will cause the program to go to an address specified by the system programmer. That address should be the starting location for a user defined function which has been implemented on the system using machine language programming techniques. If the user does not elect to provide such a function, the use of the mnemonic UDF should be avoided by the high level language programmer. (Users who desire to implement a user defined function should refer to the appropriate chapter which presents the source listing for the FUNCTION subroutines.)

MORE EXECUTIVE COMMANDS

At the beginning of this chapter, the reader was introduced to the executive SCRatch command which is used to clear out the user program buffer and effectively initialize SCELBAL in preparation for creating a new stored program.

A program may be built up and stored in the user program buffer by simply preceding statement lines with a line number. Remember, if a statement does not have a line number, it will be immediately executed. Line numbers may be any whole number from 1 to 999999.

Lines preceded by a line number are placed in the area in memory designated as the user program buffer area according to the value of their line number. If a line number is less than any previous line numbers stored in the program buffer, then the line will be placed as the first entry in the buffer. If it is greater than any already present in the buffer, the line will be appended as the last entry in the buffer. If it is between line numbers already in the storage area, then the line will be inserted in the proper position within the buffer. If the same line number is used again, and the line contains a statement keyword, then the new line will replace the previous line having that number in the buffer.

To remove a line from the user program buffer, simply type the line number by itself! The program will acknowledge the effective delete command by responding with the message:

READY

At any time that the operator is entering information when SCELBAL is in the EXECutive mode, a typographical error may be deleted by depressing the RUBOUT key. Each time the RUBOUT key is depressed, a backslash character will be displayed and the last character entered will be effectively erased. Striking the rubout key several times will effectively erase several characters. Thus, the entry:

100 LET X = 12345\\

(with the backslash ("\\") signs indicating the repeated use of the rubout key), would result in the program accepting the statement:

100 LET X = 12

as the digits 3, 4 and 5 would have been effectively deleted by the three rubout characters.

The LIST Command

Whenever the operator desires to review the contents of the user program buffer (when the program is in the executive command mode) the word LIST followed by a carriage return should be entered. The LIST command will cause all the lines in the user program buffer to be listed for review purposes.

The RUN Command

When it is desired to execute a program that has been created and stored in the user program buffer, the executive command RUN must be issued.

When the RUN command is recognized SCELBAL will proceed to the first line in the user program buffer and commence interpreting the program. SCELBAL will remain in the stored program operating mode until one of the following occurs:

1. An END statement is encountered.
2. SCELBAL runs out of program lines while executing a program (such as may occur if a programmer fails to terminate a program with an END statement).
3. A program error condition is detected.

When any of the above conditions occur, the interpreter ceases operation and control is returned to the executive control routine.

The SAVE Command

The executive SAVE command is used to transfer the contents of the user program buffer to a bulk storage device such as a magnetic tape system. Thus, once high level language programs have been created they may be

permanently saved for quick and easy loading back into the computer.

The system operator should check with the person who implements SCELBAL on the individual system in regards to the details of I/O operations when the SAVE command is utilized. This is because the SAVE command simply directs the program to go to a user provided I/O handling routine to perform the necessary transfer operations with the bulk storage device.

If the system does not have a bulk storage device available then the SAVE command should not be issued by the operator.

If the system does not have a bulk storage device available then the LOAD command should not be issued by the operator.

The LOAD Command

The LOAD command is used to transfer a higher level program, previously stored on a bulk storage device using the SAVE command, back into the user program buffer so that it may be executed. Once again, the system operator should check with the person who implements SCELBAL on the system in regards to the details of I/O operations when the LOAD command is utilized. This is because the LOAD command simply directs the program to a user provided I/O routine.

ERROR MESSAGES

SCELBAL has been provided with the capability to detect many types of syntax error conditions as well as various types of operating error conditions. When such error conditions are detected, program execution will be halted and an error message will be displayed. If an error is detected when the program is in the executive mode, such as when lines are being entered into the user program buffer, or SCELBAL is being used in the calculator mode, then a simple two letter error code will be issued. When an error is de-

tected while a stored program is being executed, the two letter error code will be followed by a message indicating the line number that was being interpreted when the error was detected.

A list of the error codes used to indicate the various types of errors that SCELBAL can detect, arranged in alphabetical order is presented below. The condition(s) associated with each type of error code is also listed.

ERROR CODE	ERROR CONDITION
AF	Array Format error. An array element is missing a right hand parenthesis.
BG	BiG error. An input is too big for a buffer. Used to indicate when user program buffer is filled (line causing overflow of the user program buffer will not be accepted). Also issued if too many characters placed on a line, or variables table filled.
DE	Dimension Error. A DIMension statement line is invalid.
DZ	Divide by Zero error. A calculation involving division by zero was encountered.
FE	For Error condition. The “=” sign is missing in a FOR statement.
FN	For/Next error. Invalid FOR statement or improper nesting of FOR/NEXT statements.
FX	FiX error. An attempt was made to integerize a number that cannot be displayed as fixed point.
GS	GoSub error. More than eight levels of subroutine nesting attempted in a program.
IF	IF error. An IF statement does not contain a GOTO or THEN directive.
IN	Illegal Number. A number string is invalid, such as: 1..23X45.
IQ	Imbalanced Quotes. The type of quotation mark used to commence a text string in a PRINT statement is different from the one used to terminate the string. For example: PRINT “HI’.
I(Imbalanced Parenthesis error condition.
LE	Let Error condition. A LET statement does not contain an equal (“=”) sign.
OR	Out of Range. The number indicated for an array element is not in the range allowed for the array variables storage table.

ERROR CODE	ERROR CONDITION
RT	ReTurn error. A RETURN statement occurred when a subroutine had not been called (by a GOSUB statement).
SQ	SQUare root error. A calculation involving taking the square root of a negative number was encountered.
SY	SYntax error. Issued for the use of incorrect keywords or invalid commands.
UN	UNdefined line number such as using a GOTO or GOSUB statement keyword and not following it with a line number, or referencing a line number that does not exist in a program.

Use of CONTROL/'C'

At times an operator may desire to terminate the operation of a program without having to wait for an END statement to be encountered. If the program includes any INPUT statements, such a program may be terminated at any time that the program is expecting to receive an input from an operator. This is accomplished by the operator simultaneously depressing the CONTROL

key and the key for the letter C on the keyboard input device. When this occurs, the program will cease performing the operations dictated by the high level program and go back to the EXEcutive mode. As it does this it will display the message:

↑ C AT LINE LL

where LL stands for the line number being processed when the program was aborted.

ILLUSTRATIVE SCELBAL PROGRAMS

The remainder of this chapter will be devoted to presenting a series of high level programs written in SCELBAL language. As the example programs are presented, brief discussions will highlight points of interest to the prospective SCELBAL programmer.

The first such sample program illustrates the use of the PRINT, INPUT, LET, GOSUB, RETURN and GOTO statements while demonstrating how a small higher language subroutine may be used in place of "extended functions" in a language. The program is one that will calculate the SINE of an angle entered in degrees (when in the range: greater than zero, on up to 90 degrees).

The reader may note that the PRINT state-

ment in line 05 is terminated by a semicolon sign so that a carriage return and line feed combination will NOT be issued after the text message is displayed. Line 20 in the program illustrates the use of a subroutine which starts at line 50 and is terminated by the RETURN statement at line 60. (While it was not necessary to establish a subroutine for this example, and in fact was wasteful of program storage space to do so, the subroutine was presented to illustrate the technique as well as provide the reader with a useful function. The instructions contained in lines 50, 55 and 60 calculate the sine of an angle when the angle is expressed in degrees (the variable value D) using a Taylor series expansion formula. The subroutine should be of value to many readers!)

LIST

```
05 PRINT "ENTER NUMBER OF DEGREES:";  
10 INPUT D  
20 GOSUB 50  
25 PRINT "THE SINE OF";D;" DEGREES IS EQUAL TO:";SN  
30 PRINT  
35 GOTO 05  
50 LET X = D/57.296  
55 LET SN = X - ((X↑3)/(2*3)) + ((X↑5)/(2*3*4*5)) - ((X↑7)/(2*3*4*5*6*7))  
60 RETURN
```

RUN

```
ENTER NUMBER OF DEGREES:30  
THE SINE OF 30.0 DEGREES IS EQUAL TO: 0.4999980
```

```
ENTER NUMBER OF DEGREES:60  
THE SINE OF 60.0 DEGREES IS EQUAL TO: 0.8660190
```

```
ENTER NUMBER OF DEGREES:45  
THE SINE OF 45.0 DEGREES IS EQUAL TO: 0.7071040
```

```
ENTER NUMBER OF DEGREES?:C AT LINE 10
```

The illustration above shows the example program being listed after the executive LIST command was issued. Next, several examples of the program's operation are shown (with program execution being initiated by the operator entering the executive RUN command). The reader should take note of how the PRINT statements used quotation marks and semicolons to obtain the desired formatting of the messages that appear when the program is executed. (Operator inputs during program operation are underlined in the above and following examples.)

The values shown are the actual values that SCELBAL produces. Note, for instance that the answer given for the sine of 30 degrees is quite close to the theoretical value (0.5). The margin of error is attributable to the precision obtainable when using 23 bin-

ary bits in calculations, the fact that the number of degrees per radian (line 50) was approximated in the above formula, and that only four terms were used in the expansion formula. Most users should find the degree of accuracy quite suitable for routine calculations.

The last line in the above example illustrates the use of the "CONTROL/C" combination by the operator to terminate the program. Remember, this special directive can be issued whenever a program expects an input from an operator. (Note that the program forms an endless loop and will simply keep asking the operator for new data as long as it is running.)

Prefer to obtain the cosine of a number rather than the sine? Just change line 55 in the above example to read;

```
55 LET CS = 1 - ((X↑2)/(2)) + ((X↑4)/(2*3*4)) - ((X↑6)/(2*3*4*5*6))
```

Changing line 55 to the formula just presented will change lines 50 through 60 to a subroutine for calculating the cosine of an angle in the range zero to ninety degrees. (If one wants to use the same type of program to obtain cosine values, just change line 25 so that the last variable is CS instead of SN!)

The next program to be presented will illustrate the use of another type of state-

ment and the use of a function directive. The statement type to be illustrated is the IF statement. The function demonstrated is the INTeger function.

The program, shown below along with several examples of output from the program when it is in operation, may be used to obtain the lowest common factor between two integer numbers.

LIST

```
10 INPUT A,B
40 X1 = A
50 IF A>B GOTO 80
60 X1 = B
70 B = A
80 X2 = B
90 X3 = X2
100 X2 = INT(X2*(X1/X2 - INT(X1/X2)))
110 X1 = X3
120 IF X2 <>0 GOTO 90
130 PRINT 'THE GCF IS';X1
140 GOTO 10
```

RUN

```
?20
?40
THE GCF IS 20.0
?112
?1143
THE GCF IS 1.0
?32
?64
THE GCF IS 32.0
```

First, the reader might take note of the use of the IMPLIED LET statements in lines numbered: 40, 60, 70, 80, 90, 100 and 110. The IMPLIED LET statements are simply LET statements without the LET keyword actually having to be stated. They simply save the programmer a little less work when entering programs.

Lines 50 and 120 illustrate the use of the

IF statement. In line 50 a single condition (IF A is greater than B) is specified. If the condition is satisfied when the program is executed, then the GOTO directive at the end of the line is followed. The GOTO 80 directive, when executed, causes the program to effectively skip program lines 60 and 70. If the condition is not met, then the above program continues directly on to execute statement lines 60 and 70. The IF statement in line 120

illustrates a double condition specification. That is, IF the value of X2 is less than OR greater than zero, then the GOTO 90 directive is followed. The satisfaction of either of those conditions results in the program effectively looping back to line 90. Otherwise, the program continues on to line 130.

(NOTE: Line 120 in the program being discussed could have been stated as:

120 IF X2 = 0 GOTO 130

provided that another line had been inserted between line 120 and line 130 such as:

125 GOTO 90

When dealing with pure integer values, as is the case in this example with the INTeger function being used, such a test is perfectly sound programming practice. However, in other situations, the use of "less than" or "greater than" tests are generally preferable. This is because the "exactly equal" test may not occur in many situations (even though the programmer may know that theoretically they do occur) due to the small inaccuracies that are often introduced into binary calculations that involve non-ending series of digits that must be limited (rounded) to a finite number of bits. The situation is analogous to dividing the decimal number one by three (yielding .333333.....), then multiplying by three yielding .99999....., when theoretically the result would be the original value of one! A machine performing the calculation and then testing for the theoretical one condition would not find the theoretical result. Thus, the SCELBAL programmer will be wise to limit the use of the exactly equal test in IF statements to calculations involving simple integer quantities!)

Line number 100 in the program illustrates the use of the INTeger function. In fact, the reader may observe that it is permissible to specify a function within a function as in the example statement line.

Finally, line 10 in the program illustrates

how more than one input may be requested in an INPUT statement line by separating the names of variable values to be inputted by a comma sign.

The results of the program being operated is illustrated following the program listing.

Earlier in this chapter a discussion of the use of the CHR (character) function was presented. It was pointed out that this function could be used when it was desired to output alphanumeric characters using their decimal ASCII encoded values. A reverse capability, that of converting alphanumeric characters received as inputs into their decimal ASCII encoded equivalents was also mentioned. This capability is implemented by following variable names specified in INPUT statements by a dollar ("\$") sign.

The following program serves to demonstrate the use of the CHR function. While doing so, it will generate a list of the decimal and octal values of the ASCII code for a subset of commonly used alphanumeric characters. This list may then be used for reference purposes by SCELBAL programmers. Additionally, the program illustrates the practical application of the FOR and NEXT statements to form a program loop.

Line number 100 in the program (presented on the next page) is used to establish the start of a FOR/NEXT loop. The loop is set up by initializing a variable named N to a decimal value of 160. Since no specific STEP size is indicated in the FOR statement, the program assumes an IMPLIED STEP value of one. The FOR line also indicates that the FOR/NEXT loop is to be terminated after the value of N reaches 228.

Line number 160 contains the NEXT N statement which marks the end of the FOR/NEXT loop connected with the variable N. When this statement is reached, the program will loop back to the original FOR statement.

Line number 170, the next statement line

after the NEXT N statement, is the line to which program operation will be transferred once the value of N exceeds the TO value of 223 indicated in the FOR statement line. The line contains an END statement to indicate to the interpreter that execution of the high level program may be halted and control passed back to the executive. Had line number 160 been the last line in the program, the interpreter would still have ceased operation

and returned control to the executive. However, the use of the END statement might be considered better programming practice in such a situation.

The program is presented below. The reader may enter and RUN the program to obtain a list of the data it generates for reference purposes. The output from the program will not be duplicated here.

```
10 PRINT
20 PRINT
30 PRINT 'TABLE OF ASCII CHARACTERS'
40 PRINT
50 PRINT ' CHAR OCTAL DECIMAL'
100 FOR N = 160 TO 223
110 Q1 = INT(N/64)
120 Q2 = INT((N - 64*Q1)/8)
130 Q3 = INT(N - 64*Q1 - Q2)
140 PRINT '      ';CHR(N);      ',CHR(176 + Q1);CHR(176 + Q2);CHR(176 + Q3);
150 PRINT '      ',N
160 NEXT N
170 END
```

Just about anyone who has a computer system likes to have a game program that will run on the machine. If not for the system owner to play with, at least such a program may be used to amuse those that might not have the deep appreciation for the machine that most readers of this publication undoubtably possess. It would be impolite to say the least, if this publication did not contain some such program written in SCELBAL.

The program on the next page is a game program, which, while it may be used for amusement purposes, will ostensibly be presented to demonstrate the use of the RND (random number generating) function and a few other SCELBAL programming points which will be mentioned in the following discussion.

The only new directive used (in the sense

of not having been used in previous sample programs) is the RND function contained in line number 90. Note that this is a case where the random number (generated in the range from zero to one) is immediately multiplied to put it in another range (by multiplying by seven). Note too, that the RND mnemonic must be followed by a pair of parenthesis (enclosing the zero) to identify it to the interpreter as a function! Additionally, the reader may observe that the RND function is contained as part of an expression for another function (the INT directive), which, as pointed out previously, is perfectly valid when using SCELBAL.

The reader has already been introduced to the practical applications of the other types of statements and directives contained in the game program. However, since a few new "twists" are utilized, the following tech-

niques will be pointed out.

Lines 15, 16, 22, 24 and several other IF statements illustrate the use of entire mathematical expressions as test values (instead of just a simple variable name).

Line 20 (and later line 26) establishes a subroutine at line 80 which in turn calls

another subroutine at line 90. This is an example of the use of nested subroutines in a program. Remember, this nesting process can be carried up to eight levels if required.

What does the game play? Dice. Be careful! People have reported difficulty in getting computers to pay off after players have had winning streaks!

```
05 LET S = 0
10 PRINT
11 PRINT 'BET';
12 INPUT A
13 PRINT
14 IF A<1 GOTO 50
15 IF (A - 1000)>0 GOTO 50
16 IF (A - INT(A))<>0 GOTO 50
20 GOSUB 80
21 LET X = R
22 IF (R - 7)*(R - 11) = 0 GOTO 60
24 IF (R - 2)*(R - 3) = 3 GOTO 70
26 GOSUB 80
30 IF (R - 7) = 0 GOTO 70
32 IF (X - R) = 0 GOTO 60
40 GOTO 26
50 PRINT 'ILLEGAL BET!'
52 GOTO 10
60 PRINT 'YOU WIN! '
62 LET S = S + A
64 GOTO 74
70 PRINT 'YOU LOSE. ';
72 LET S = S - A
74 PRINT ' YOUR Winnings ARE: ' ;S
76 GOTO 10
80 GOSUB 90
81 PRINT ' X';
82 LET D1 = R
84 GOSUB 90
85 PRINT
86 LET R = R + D1
88 RETURN
90 LET R = (INT(RND(0)*7))
92 IF R>6 GOTO 90
94 IF R<1 GOTO 90
96 PRINT R;
98 RETURN
```

The next example program to be presented was chosen primarily to illustrate the use of the TAB function in a PRINT statement. The program uses the TAB function to plot the points on a circle. The line containing the TAB directives in the program is number 60.

Several other points of interest in the program include the use of the comma sign in

line 21 to implement a standard tabbing operation (causes the display device to space over to the next column number that is a multiple of sixteen), the use of a FOR/NEXT loop with the variable value ranging from a negative value to a positive value (line 30) and the use of the SQR function in line 40.

The program followed by a sample of its output is shown below.

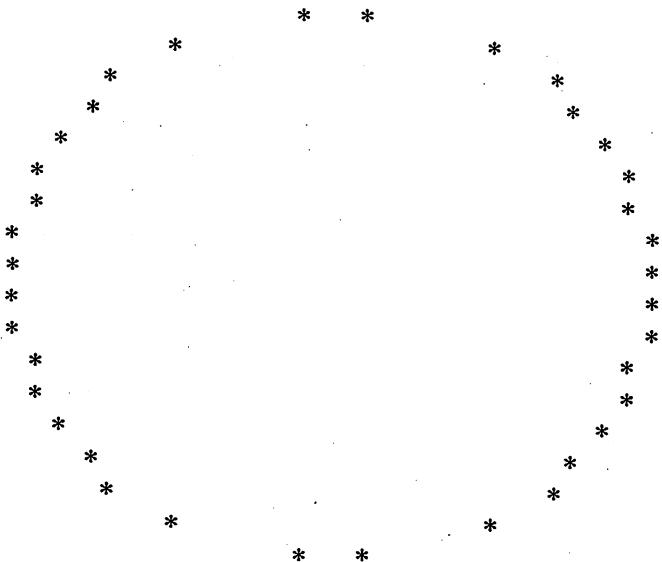
```
10 PRINT 'RADIUS';
15 INPUT R
16 K = 1.6
20 R2 = R ↑ 2
21 PRINT 'AREA =' ; 3.14159 * R2, 'CIRCUM =' ; 3.14159 * 2 * R
22 PRINT
23 PRINT
25 K1 = K * R
30 FOR X = -R TO R + 0.1
40 Y = K * SQR(R2 - X ↑ 2)
50 PRINT TAB(2.5 + K1 - Y);";TAB(5.5 + K1 + Y);"
60 NEXT X
70 END
```

RUN

RADIUS?10

AREA = 314.1592

CIRCUM = 62.83185



The example programs presented to this point in the chapter can all be executed in a version of SCELBAL implemented in a minimum configuration (8 K of memory without the optional array handling routines).

The remaining example programs in this chapter utilize the optional array handling capabilities of SCELBAL. The next two programs could actually be run in an 8 K version of SCELBAL that had the DIMension and array handling routines installed. (In this configuration, the user program buffer would only have 512 bytes available for program storage. As mentioned in an earlier chapter this implementation is not recommended because of the small storage space it leaves for user programs). The final program in this chapter would require more than a minimum system configuration. (The configuration assumed for the assembled listings of the program with array handling routines presented in this book would be more than sufficient.) It requires that the user program buffer have about 1000 bytes available when the array handling routines (and array values table storage area) are installed. Thus, about a 9 K system would be required, as a minimum, to execute the final example program. However, even if one does not have the capability at present to try the program (or any desire to use it, for that matter), one may desire to examine the listing. That is because the program will illustrate how single dimension array handling capability can be utilized to solve problems typically processed using two dimensional array techniques!

The first sample program involving an array is presented on the next page. It is a program that will calculate the mean and standard deviation values after receiving a number of inputs. The important feature of this program is that it shows how the array feature may be used to effectively increase the number of variable values that may be stored and manipulated by a program. Instead of having to use a new variable name for each value, one may simply assign the value to a position (element) in an array that has one name, with elements in the named array being identified

by a subscript (number).

Noting the following items in the listing of the example program should prove valuable for the novice programmer who is not familiar with the use of arrays.

Line 10 is the all important DIMensioning statement. The DIMension statement must be given in a SCELBAL program before any attempt is made to reference an array element. The DIMension statement in the example creates an array having the name A and provides for up to 64 elements to be assigned to this array name. (Remember, that is the maximum number of elements that may be assigned amongst all arrays in a SCELBAL program.)

Line number 70 in the program illustrates how the element of an array may be referenced. Note particularly that here it is permissible to use a variable name as a subscript. (It is not permissible to use a variable name when setting up the size of an array using the DIMension statement!)

Lines 80 and 130 illustrate the subscripted variable A(J) being used as part of a mathematical expression just as a regular variable value may be used. In these cases, the value for A(J) will be the value currently existing for the J'th element of the array named A. (The subscript number J, indicating which element in the array is being referenced, is determined by the FOR statement in line 50 or 120. The FOR/NEXT loops, the reader may observe, will step the number for J from a value of 1 to N, where N is the number of values to be entered by the operator.)

The reader may observe that the program uses an array in which to store values as they are inputted by the operator until all the data has been inputted. Then, all the data stored in the array is processed to obtain the desired information. An example of the program being used to calculate the mean score and standard deviation for a group of hypothetical test scores is illustrated following the program listing.

```

10 DIM A(64)
20 PRINT 'NR OF SCORES';
30 INPUT N
40 S = 0.0
50 FOR J = 1.0 TO N
60 PRINT 'SCORE NR.';J;
70 INPUT A(J)
80 S = S + A(J)
90 NEXT J
100 M = S/N
110 D = 0.0
120 FOR J = 1.0 TO N
130 D = D + (M - A(J))2
140 NEXT J
150 PRINT 'M =' ; M
160 PRINT 'SD =' ; SQR(D/N)
170 END

```

RUN

NR OF SCORES?20
 SCORE NR. 1.0?100
 SCORE NR. 2.0?76
 SCORE NR. 3.0?32
 SCORE NR. 4.0?89
 SCORE NR. 5.0?72
 SCORE NR. 6.0?33
 SCORE NR. 7.0?75
 SCORE NR. 8.0?76
 SCORE NR. 9.0?84
 SCORE NR. 10.0?83
 SCORE NR. 11.0?16
 SCORE NR. 12.0?95
 SCORE NR. 13.0?91
 SCORE NR. 14.0?55
 SCORE NR. 15.0?55
 SCORE NR. 16.0?78
 SCORE NR. 17.0?70
 SCORE NR. 18.0?68
 SCORE NR. 19.0?64
 SCORE NR. 20.0?88
 M = 70.0
 SD = 21.67948

The next program is a program to demonstrate how the CHR function may be used with arrays to handle the processing of very simple text strings. The technique to be illustrated can be quite useful if one wants to

have a program perform an operation such as reading in a name and later displaying it back to the operator. (Note that this capability is quite different from displaying a message previously stored by the programmer!)

Line number 01 in the program contains the DIMension statement, which in this case assigns all 64 array element storage locations to the array named L.

Line 03 contains an INPUT statement that specifies an array element in which the value of the variable to be inputted is to be stored. Following the variable name by a dollar sign ("\$") means that the character that is entered by the operator will be converted to its decimal ASCII representation.

The instructions in lines 03 through 07 form a program loop that operates to accept characters and store their decimal ASCII values in elements of the array. The process will continue until 64 characters have been received or the operator enters a carriage return on the input device. (The test in line

04 will identify the inputting of a carriage return!)

Once a character string has been inputted the balance of the program will cause the string to be outputted. Line 12 using the CHR function will output the characters by converting the decimal ASCII values stored in the array to alphanumeric characters.

The example is an extremely simple case, but it demonstrates the capability. Naturally, one may manipulate several smaller text strings within a program by assigning several array variable names and splitting the available storage locations up among the several arrays.

The program listing and a sample of its operation is provided below.

```
500 01 DIM L(64)
    02 LET X = 1
    03 INPUT L(X)$
    04 IF L(X) = 141 GOTO 10
    05 LET X = X + 1
    06 IF X > 64 GOTO 10
    07 GOTO 03
    10 PRINT RGTN
    11 LET X = 1
    12 PRINT CHR(L(X));
    13 IF L(X) = 141 GOTO 20
    14 LET X = X + 1
    15 GOTO 12
    20 PRINT
    30 END RGTN
```

RUN

ABCDEFGHIJKLMNPQRSTUVWXYZ 1234567890
ABCDEFGHIJKLMNPQRSTUVWXYZ 1234567890

The final SCELBAL example program to be illustrated is a program that will solve simultaneous equations. The program is presented primarily to demonstrate how the

optional single dimension array handling capability may be utilized to solve problems commonly written using two dimensional arrays. The trick, of course, is to manipulate

```

10 DIM A(64)
20 PRINT
40 PRINT 'NO OF EQUATIONS';
50 INPUT N
55 PRINT
60 N2 = N + 1
80 FOR R = 1 TO N
90 FOR C = 1 TO N + 1
100 INPUT A(N2*R - N2 + C)
110 NEXT C
115 PRINT
120 NEXT R
130 PRINT
140 FOR I = 1 TO N
141 FOR J = 1 TO N2
142 PRINT A(N2*I - N2 + J),
143 NEXT J
144 PRINT
145 NEXT I
150 PRINT
160 FOR P = 1 TO N
170 IF A(N2*P - N2 + P) = 0 GOTO 470
190 M = 1/A(N2*P - N2 + P)
200 FOR C = 1 TO N2
210 A(N2*P - N2 + C) = A(N2*P - N2 + C)*M
220 NEXT C
240 FOR R = 1 TO N
250 IF P = R GOTO 380
260 FOR C = N + 1 TO 1 STEP - 1
270 A(N2*R - N2 + C) = A(N2*R - N2 + C) - A(N2*P - N2 + C)*A(N2*R - N2 + P)
280 NEXT C
290 FOR I = 1 TO N
300 FOR J = 1 TO N2
310 PRINT A(N2*I - N2 + J),
320 NEXT J
330 PRINT
340 NEXT I
360 PRINT
380 NEXT R
390 NEXT P
410 FOR R = 1 TO N
420 PRINT A(N2*R - N2 + N + 1)
430 NEXT R
440 END
470 IF P = N GOTO 620
480 X = P
490 FOR R = X + 1 TO N
495 R1 = R
500 IF A(N2*R - N2 + X) <> 0 THEN R = N
520 NEXT R
522 IF A(N2*R1 - N2 + X) = 0 GOTO 620

```

```

530 FOR C = 1 TO N + 1
540 A(N2*N + C) = A(N2*X - N2 + C)
550 A(N2*X - N2 + C) = A(N2*R1 - N2 + C)
560 A(N2*R1 - N2 + C) = A(N2*N + C)
570 NEXT C
580 GOTO 190
620 PRINT 'SINGULAR MATRIX'
630 END

```

the data in the one dimensional array in a manner that simulates having a two dimensional array storage area. This is easy to do with SCELBAL because elements in an array may be identified by using a mathematical expression containing one or more regular variables (as long as no additional parenthesis are required in the expression other than the pair that identify the expression as the subscript of an array name).

Line 100 in the program illustrates how elements in a single dimension array can be mathematically assigned to sections within the array. If those sections were then viewed as being side-by-side, one would effectively obtain a two dimensional array. The formula in the subscript for the array named A in the program, given in line 100, is:

$$N2*R - N2 + C$$

By examining lines 60, 80 and 90 in the program, the reader may observe that the regular variable R referred to in the expression will be incremented from a value of 1 to a value of N. (N represents the number of variables/equations to be solved. Taking a two dimensional view, this number would represent the number of entries along the Y axis of a two dimensional matrix.) The variable C will advance from a value of 1 to a value of N + 1. (This represents the number of entries along the X axis if a two dimensional view is considered.) The FOR/NEXT loops established in lines 80 and 90 will cause the value of C to be incremented through its range for each value of R. If one takes for instance, a value of 3 for N and solves the formula for all the possible values as R and C are advanced through their ranges one would obtain a range of values that could be arranged in a two dimen-

sional table as illustrated here:

	C = 1	C = 2	C = 3	C = 4
R = 1	(1)	(2)	(3)	(4)
R = 2	(5)	(6)	(7)	(8)
R = 3	(9)	(10)	(11)	(12)

This table illustrates how a formula for the subscript (element) of a single dimension array may be implemented to effectively create a two dimensional array pattern. This is the technique used in the program for solving simultaneous equations.

The program presented can handle equations with up to 7 unknowns. (An equation with 7 unknowns requires 56 (7 times 8) entries in a matrix. Remember, there are only 64 array elements available. Thus, an equation with 8 unknowns, which would require the storage of 72 (8 times 9) elements, would not fit in the available array storage area.)

Another point of interest in the program is the use of nested FOR/NEXT statements at several points. Note how the inner-most FOR statement is terminated by its corresponding NEXT statement before an earlier FOR statement may be closed. (See, for example, lines 80 and 90, then 110 and 120.)

Finally, notice in line number 260 the use of the STEP directive in the FOR statement line, and how the step directive may be used to decrement a value over a range going from high to low just as easily as one may use it to increment a value over a range.

RUN

NO OF EQUATIONS? 3

?1

?1

?1

?6

?3

?-2

?1

?2

?10

?6

?-3

?13

1.0	1.0	1.0	6.0
3.0	-2.0	1.0	2.0
10.0	6.0	-3.0	13.0

1.0	1.0	1.0	6.0
0	-5.0	-2.0	-16.0
10.0	6.0	-3.0	13.0

1.0	1.0	1.0	6.0
0	-5.0	-2.0	-16.0
0	-4.0	-13.0	-47.0

1.0	0	0.5999999	2.8
0	1.0	0.3999999	3.199999
0	-4.0	-13.0	-47.0

1.0	0	0.5999999	2.8
0	1.0	0.3999999	3.199999
0	0	-11.4	-34.2

1.0	0	0	1.0
0	1.0	0.3999999	3.199999
0	0	1.0	3.0

1.0	0	0	1.0
0	1.0	0	1.999999
0	0	1.0	3.0

1.0
1.999999
3.0

The preceding page shows a sample of the output from the program when it is given the task of solving for the unknowns in the three equations:

$$\begin{aligned} X + Y + Z &= 6 \\ 3X - 2Y + Z &= 2 \\ 10X + 6Y - 3Z &= 13 \end{aligned}$$

Readers should now have a pretty good grasp of how to use SCELBAL. At this point the process of creating programs will be left to the individual user. Between the examples, explanations, and error messages table presented in this chapter, the reader who gets down to practical experience should have little difficulty in learning how to enjoy SCELBAL.



SUGGESTIONS FOR PROGRAM TINKERERS

The whole purpose of presenting the program SCELBAL in the form of this publication was so that readers could acquire the knowledge that would give them the freedom to modify and adapt the program to meet their individual requirements. It is fully expected that many readers will want to take advantage of this aspect. The purpose of this final chapter is to provide some assistance and suggestions to those readers who contemplate modifying the program.

Perhaps one of the first aspects of the program that a user might have need to alter is the storage area assigned to the user program buffer. In the assembled version of SCELBAL presented (for a 12 K system with the optional array capability installed) the buffer is assigned memory locations starting at address page 33 location 000 and extending up to page 54 location 377. The most common alteration to this buffer size will undoubtedly be simply to reduce or extend the upper limit depending on the amount of memory available in the user's system and whether the optional array routines are installed in the upper portion (3 pages) of available memory. Changing the upper limit of the user program buffer only requires changing one location. In the assembled listing provided the address of this location is at page 12 location 222. This is the address of the second byte in a CPI instruction in the INSERT routine that checks to see if adding a line to the user program buffer will cause it to overflow. This byte should contain the page value of the highest page in memory that is to be allocated to the buffer! Thus, if a user only has an 8 K system, this location should contain a value of 037 (page 37), assuming that the optional array routines were not included. If they were included, one would need to reduce this value to 034. If one had, say, a 10 K system; and intended to install the array handling routines on pages 45 and 46 (reserving page 47 for the array values table), then the limit value in the CPI instruction at the address indicated would simply be 044. (Remember, the buffer will

use the locations on the page specified as the upper limit, the limit specifies that the buffer is not to extend beyond that page!)

Since changing the upper limit of the user program instruction only requires altering one byte in the entire SCELBAL program, one may see that it is easiest to make additions to the program by placing routines in the area originally assigned to be the highest address region of the user program buffer and then simply lower the buffer size by changing the indicated location. Thus, for instance, a user who might not be able to place the I/O routines required by the program on page 00 as suggested in the chapter on I/O operations, might place them on page 37 (in an 8 K system without arrays) and change the buffer limit value to page 36. Or, a user that wanted to append a lengthy machine language routine that was executed as part of a user defined function, would probably find it easiest to place the new routine in the highest locations available for the buffer and then lower the buffer limit value as required.

Of course, there may be instances when the user desires to change the lower boundaries of the user program buffer area. Doing so, however, requires altering considerably more than a single location in the program. Altering this limit requires changing the data in the following addresses (as they appear in the assembled listing provided):

10 332
11 017
11 051
11 132
11 173
13 107
15 255

All of the above locations would have to be changed from their original values of 033 to whatever value represented the page number at which the programmer desired the user program buffer to start.

Another type of alteration that some readers may wish to implement actually is related to the user provided I/O routines. This has to do with processing and displaying relatively short lines such as may be required by some CRT and TV display systems. Such systems often are limited to 32 or 40 characters to a line. While many users might not be concerned with having such short lines, and would be content with simply writing all SCELBAL programs in forms that did not exceed this limitation, some users might be hampered by such a limitation. (While all statement types and commands can easily be handled in such a short line length, the line length of some types of statements will be a function of the complexity of the mathematical expressions contained in the line. A short line length can thus affect the manner in which one writes mathematical formulas.)

When inputting lines to SCELBAL, the user with such a display system can handle the situation without any modification to SCELBAL itself. This may be done by having the user provided input subroutine simply screen for a special character such as a line feed. When the routine encountered the special character it could simply issue a "new line" directive to the display device (assumed to be echoing the input) and discard the character so that it was not processed by the main program. Thus, whenever inputting information the operator would simply enter the special character on the keyboard so that the display would go to a new line, yet only irrelevant characters would go into the line input buffer used by the program. (Remember, however, that the line input buffer used in SCELBAL is limited to holding 72 characters at a time!)

On the outputting side, one could make a minor modification to the ECHO subroutine in SCELBAL (using patching techniques) along the following line. Examine the output character counter. When it reaches the value equal to the maximum length of a line issue a "new line" directive to the display device. Then, reset the output character counter. This capability may be inserted just before

the end of the ECHO subroutine. (The ECHO subroutine starts on page 03 location 202 in the assembled listing. The source listing for the subroutine is presented in the chapter that discusses the SYNTAX routine.) This procedure would take care of the displaying of lengthy statements or cases where the programmer failed to properly format PRINT statements. (One would, of course, plan on formatting PRINT statements to suit the display device being utilized.)

The program in this publication was developed and presented in a fashion that would lend itself to easy modification by the reader. Indeed, with the organizational and conceptual information that has been presented, along with the multitude of routines, serious students of this manual are in a position to customize the higher level language to their individual desires. Some thoughts on such customizing will be presented in the next few pages.

One area in which a user might desire additional capability, for example, could be in the number of user defined functions that the user could add to the program. The FUNCTION LOOK-UP table only provides for the mnemonic UDF. How could one easily add the capability to perform several different user defined routines?

One way this could be accomplished would be to let the argument associated with the mnemonic specify a particular subfunction! For instance, the terms:

UDF(1)
UDF(2)
UDF(3)
UDF(4)

could represent four different types of functions. To determine what type of function was to be performed, the programmer would simply arrange the first part of the user provided UDF subroutine so that it checked the value of the argument (which would be residing in the FPACC) and then directed

the program to the appropriate subfunction!

It is important to note that while some readers might automatically relate functions with the performance of mathematical operations, such a narrow interpretation is not necessary. One can have a user defined function perform practically any useful type of operation such as control an external device. One interesting and useful idea for such a user defined function is to have it control a tape unit. Thus, one could read in the next section of a multiple-segment program if the user program buffer was too small to hold all the needed directives for a large program. (Be careful, though, when organizing the high level program, not to overlay when inside a nested statement type such as a FOR/NEXT loop or a GOSUB directive!)

Going on to another area of customizing, consider the mnemonics for statement names, and indeed, the specific tasks that the various statement types perform. The user who doesn't like the statement keywords as presented, can change the statement keyword table quite readily. If one takes care not to exceed the space allotted to the table, and keeps the same order (so that the token value structure is not altered as discussed in the chapter dealing with the SYNTAX routines) among the various types, one can simply rename the offensive mnemonics with no further alterations to the program! Thus, if a user prefers to use the mnemonic SET instead of the mnemonic LET, a simple change to the name in the STATEMENT KEYWORD table is all it takes.

The next step is to alter the operation of a statement type, or substitute a different type of statement. Perhaps a particular user finds that the FOR/NEXT statement types are of no particular benefit to the user's applications. Presto! Change two entries in the keyword table, to say, do THIS and do THAT as the mnemonics for two new statement types. Then, remove the subroutines relating to the FOR/NEXT statements and substitute routines that perform THIS and THAT.

Suppose a user likes all the statement types presented but could use a few more? Well, the names assigned to the various statement types are almost all longer than necessary. By compacting those keyword names (thereby opening up room in the keyword table) to just one or two characters, assigning some new token values, and adding the appropriate tests for the new token values in the DIRECT routine (refer to the chapter that presents the source listings for the statement type routines), one can enhance the program by adding new statement types. The actual routines to perform the new statements may be placed in areas in memory formerly used as the user program buffer by appropriately limiting the size of the buffer as mentioned at the beginning of this chapter.

The reader with a little imagination will soon find all kinds of possibilities for enhancing the described package. With all the various utility routines available within the program, one will find that many kinds of capabilities that a user might desire can be added to the program with relative ease. Suppose, for instance, that one is interested in manipulating text strings and would like to implement some string function capabilities in SCELBAL. A little review of some of the kinds of subroutines already present in the program described will show that there are a number of routines available that may be combined to rapidly build up some string handling functions. Just to name a few, consider the following:

The MOVEIT subroutine can transfer strings of characters from one area in memory to another.

The CONCTA and associated subroutines can concatenate (append) characters from one buffer to another.

The STRCP subroutine can determine if character strings in buffers are the same length.

The STRCPL and associated routines can

determine whether character strings match one another.

(NOTE. The locations of the routines mentioned within the assembled program provided, as well as the chapter and page number in which the source listing was presented, may be found in the Appendix.)

These types of routines, coupled with appropriate user provided linking instructions, etc., can very quickly be capitalized upon by the adventurous and ambitious programmer to add string handling type functions to the language if desired.

The amount of creative additions of the above nature, given the base that one has to start within the SCELBAL package, is virtually unlimited.

There are a few other aspects about the package that will be interest to those that desire to tinker with the program. The machine language programmer with even a modest amount of experience will find the program quite easy to modify using patching or compressing techniques. This is because several design guidelines followed during program development for this publication have side effects that are useful in this regards.

For example, perhaps the most significant decision made regarding the package's development had to do with whether or not to utilize locations on page zero in memory. Doing so would have meant the program could have been organized and compressed to reside in about 1.5 K bytes less memory from this one factor alone! Why wasn't it done? Experience indicated that many small system owners dedicated all or part of page zero in their systems to monitor functions or similar permanent or semi-permanent programs. Requiring the use of page zero would have meant these users would have to re-assemble SCELBAL for use on their systems. Furthermore, the features that would have been so useful to capitalize on, had page zero been used, would have made such re-assembly a somewhat difficult task.

For instance, the RST (Restart) locations on page zero could have been used to hold commonly used instructional sequences (particularly in the 8008 version) such as:

LBM
INB
LMB

or

LAM
NDA

Just being able to replace those two and three byte instructions by one byte RST instructions, would have enabled some two to three pages of memory to be saved! But, woe to the poor user who had to re-assemble the program. That would require finding all the RST instructions, replacing them with the multi-byte sequences, and greatly expanding the size of the program. One is far more likely to be upset about seeing a program expand than to discover that with a little effort the program can be made to contract!

Secondly, using page zero for most of the pointers and counters, say, would have meant a good many LHI XXX type instructions (to set the page portion of the memory pointer) could have been reduced to a one byte LHX instruction (because quite often a CPU register will contain the value zero) or eliminated altogether because of less frequent changes to the pointer page. Again, woe to the user who might have been forced to re-assemble because page zero was not available. All such single byte (or worse, non-existent) set ups would have had to be located and the multi-byte LHI XXX inserted!

As the package has been presented, if it is necessary to relocate the program, the re-assembly process can be made quite straightforward. If the pages containing pointers, counters and buffers must be altered, then only the page value byte in the LHI XXX instructions need be altered. All such locations have been marked in the book by the double asterisk "##" indicator. Program size, the relative locations or subroutines, etc., would all remain fixed. In many instances

involving relocation, only one data page, or a few might have to be relocated, so the number of LHI XXX instructions that would be altered would be even less. (Re-assembly might also affect the values in locations marked by "††.")

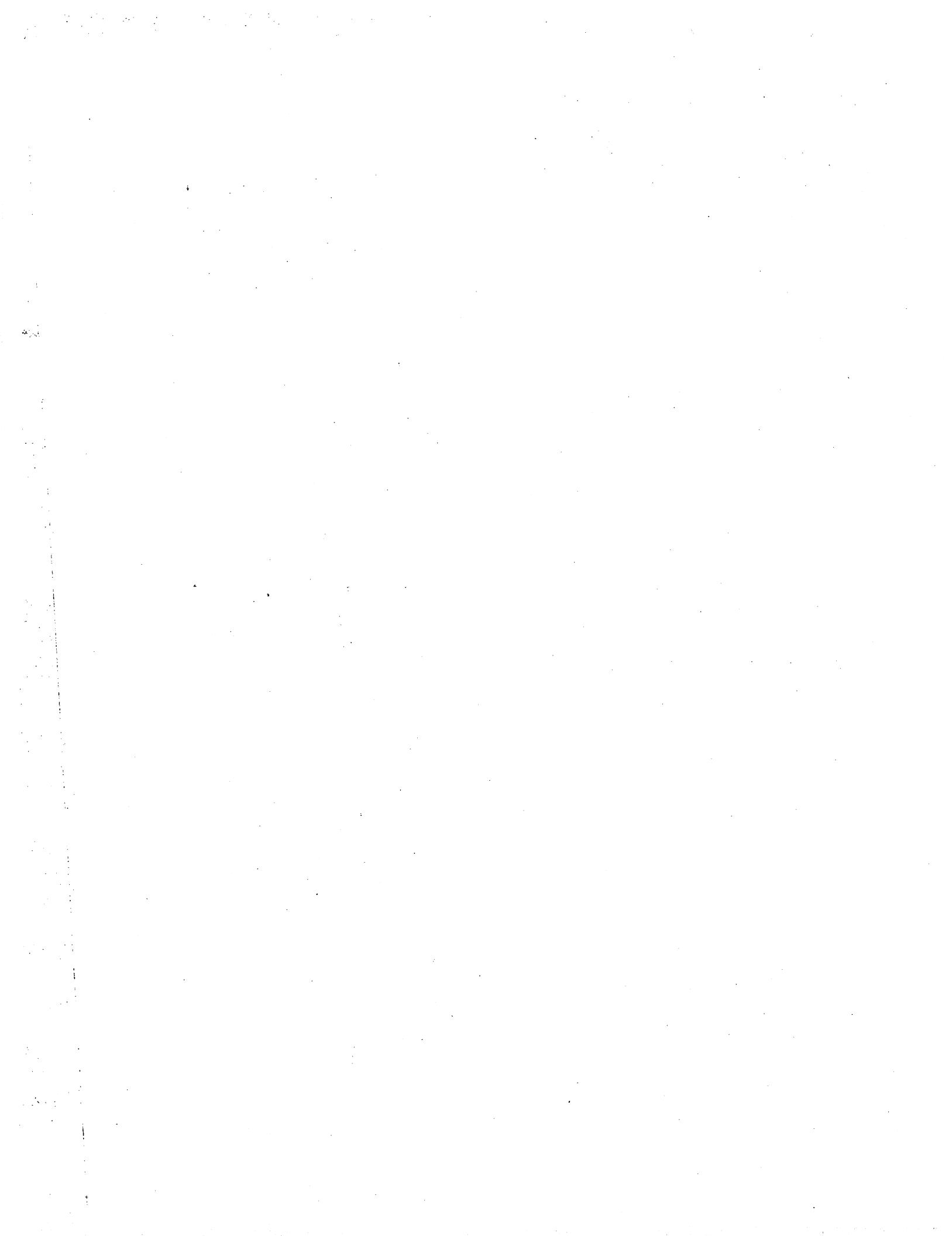
These guidelines provide additional benefits for the user. Those with systems that do have page zero available will find they have a package with the potential for being considerably reduced in size if they wish to re-assemble the package to take advantage of the possibilities.

Those not interested in that type of project, but that should find they desire to make minor changes or patches to various portions of the program, will find that frequently it is possible to compress even one lengthy routine by quite a few bytes. This may be done in many cases by replacing a few of the frequently used instructional sequences with one byte RST instructions and placing the instructions in the sequence on page zero at the appropriate restart address. Room can thus readily be made to accomodate some extra in-

structions in the routine one desires to alter.

The use of page zero could save up to 1.5 K bytes of memory in the 8008 version of the program. It hardly need be mentioned that the amount of compression that can be obtained in the 8080 version is considerably more. This is because some of the frequently used small subroutines and instructional sequences used in the program actually have shorter (in terms of the number of bytes required) equivalent commands available in the 8080 instruction set. A good machine language programmer who wants to take the time and effort, should have little difficulty getting an 8080 version of SCELBAL (say, without array capability, in order to allow for a decent sized user program buffer) in a 4 K system.

But, such undertakings are not at all necessary to enjoy SCELBAL. Using the program as it has been presented may keep many readers occupied for years. But, should any start to get bored, it is always nice to know that the freedom to make changes is right in this book!



APPENDIX A - SCELBAL LABELS

The following is a list of the names used as labels to identify routines and subroutines in SCELBAL. The list is arranged in alphabetical order. The first column shows the name, the second column shows the address of the label in the assembled version of the program provided in the book, and the last column indicates the chapter and page within the chapter where the label appears in the source listing.

ABSX	07 346	9-8	CLRNX3	23 067	10-19
ACCSET	20 166	10-4	CLRPL	21 203	10-10
ACNONZ	20 143	10-4	COMPEN	25 010	10-25
ACZERT	20 120	10-4	COMPLM	22 150	10-14
ADBDE	12 305	4-11	CONCTA	02 264	5-7
ADDER	22 127	10-14	CONCTE	02 327	5-7
ADDEXP	21 051	10-8	CONCTN	02 276	5-7
ADDMOR	22 130	10-14	CONCTS	02 310	5-7
ADOPPP	21 270	10-10	CONCT1	02 314	5-7
ADV	02 377	5-8	CONTIN	12 073	4-9
ADVDE	13 064	4-12	CPHLDE	12 277	4-11
AD4DE	06 256	8-14	CRLF	03 141	5-10
AHEAD1	24 220	10-24	CROUND	21 307	10-11
AHEAD2	25 333	10-28	CTRLC	12 313	4-11
ARRAY	55 145	9-12	CTRUE	06 242	8-14
ARRAY1	55 153	9-12	DEC	03 164	5-10
ARRAY2	55 160	9-12	DECBIN	24 056	10-22
ARRAY3	55 162	9-12	DECEXD	24 336	10-25
ARRAY4	55 174	9-14	DECEXT	24 277	10-25
ARRAY5	55 225	9-14	DECNO	03 172	5-10
ARRAY6	55 240	9-14	DECOUT	24 360	10-25
ARRAY7	55 312	9-14	DECRDG	25 112	10-26
BACKSP	31 217	9-9	DECREP	24 327	10-25
BIGERR	02 222	5-6	DIM	55 365	6-42
BRING1	21 007	10-6	DIM1	55 377	6-42
CFALSE	06 247	8-14	DIM2	56 017	6-42
CHRX	07 377	9-9	DIM3	56 032	6-42
CINPUT	03 221	5-10	DIM4	56 036	6-42
CKDECP	25 137	10-26	DIM5	56 157	6-46
CKEQEX	20 242	10-5	DIM6	56 211	6-46
CKSIGN	21 166	10-9	DIM7	56 224	6-46
CLESYM	02 255	5-7	DIM8	56 271	6-47
CLRNX	21 175	10-10	DIM9	56 301	6-47
CLRNX1	21 207	10-10	DIM10	56 320	6-47
CLRNX2	23 055	10-19	DIMERR	56 387	6-47
			DINPUT	23 046	10-19
			DIRECT	13 211	6-4
			DIVIDE	21 351	10-12
			DVERR	12 357	4-11
			DVEXIT	22 070	10-13
			DVLOOP	06 362	8-15
			ECHO	03 202	5-10
			ENDINP	23 311	10-21
			EQ	06 136	8-14
			ERROR	02 226	5-6
			EVAL	03 224	7-2

EXEC	10 266	4-2	GETAU0	11 211	4-7
EXEC1	10 275	4-4	GETAU1	11 242	4-7
EXECSP	31 330	13-2	GETAU2	11 267	4-8
EXMLDV	21 146	10-9	GETAUX	11 177	4-7
EXOUTN	25 324	10-28	GETCHP	12 123	4-9
EXPINP	23 241	10-20	GETCHR	02 240	5-6
EXPOK	24 000	10-22	GETINP	22 365	10-18
EXPOUT	25 300	10-28	GOSERR	16 347	6-25
			GOSUB	16 236	6-21
			GOSUB1	16 255	6-21
FACXOP	22 277	10-16	GOTO	15 174	6-14
FAERR	07 172	9-3	GOTO1	15 211	6-14
FINERR	12 322	4-11	GOTO2	15 240	6-14
FINER1	12 351	4-11	GOTO3	15 250	6-16
FININP	23 327	10-21	GOTO4	15 261	6-16
FIXERR	12 366	4-11	GOTO5	15 270	6-16
FLOAD	22 244	10-16	GOTO6	15 315	6-16
FNDEXP	23 221	10-20	GOTO7	15 340	6-16
FOR	17 164	6-29	GOTOER	16 020	6-17
FOR1	17 246	6-29	GT	06 153	8-14
FOR2	17 262	6-31			
FOR3	17 304	6-31	IF	16 027	6-19
FOR4	17 317	6-31	IF1	16 102	6-20
FOR5	31 246	6-31	IF2	16 143	6-20
FORERR	17 237	6-29	IF3	16 166	6-20
FORNXT	30 121	6-36	IF4	16 200	6-20
FP0	17 157	6-28	IFERR	16 073	6-20
FPADD	20 211	10-5	INCLIN	12 255	4-10
FPCOMP	20 202	10-5	INDEXB	03 174	5-10
FPD10	24 033	10-22	INDEXC	23 036	10-19
FPDIV	21 322	10-11	INPUT	16 365	6-25
FPFIX	20 000	10-3	INPUT1	16 377	6-27
FPFIXL	20 033	10-3	INPUT2	17 037	6-27
FPFLT	20 064	10-4	INPUT3	17 042	6-27
FPMULT	21 046	10-8	INPUT4	17 063	6-27
FPNORM	20 066	10-4	INPUTN	17 140	6-28
FPONE	06 242	8-14	INPUTX	17 104	6-27
FPOPER	05 364	8-13	INSERT	12 205	4-10
FPOUT	24 165	10-24	INSER1	12 231	4-10
FPX10	24 010	10-22	INSER3	12 255	4-10
FPZERO	20 051	10-3	INSTR	13 012	4-12
FRAC	14 350	6-11	INSTR1	13 016	4-12
FSHIFT	21 002	10-6	INSTR2	13 061	4-12
FSTORE	22 255	10-16	INT1	07 327	9-8
FSUB	21 032	10-7	INT2	07 341	9-8
FUNAR1	07 115	9-3	INTEXP	06 263	8-15
FUNAR2	55 054	9-4	INTX	07 243	9-8
FUNAR3	55 124	9-4			
FUNAR4	07 207	9-3	LE	06 173	8-14
FUNARR	07 100	9-3	LET	15 031	6-13
GE	06 213	8-14	LETO	15 013	6-13

LET1	15 042	6-13	NOLIST	10 354	4-5
LET2	15 053	6-13	NONZAC	20 235	10-5
LET3	15 113	6-13	NOREMD	32 057	9-10
LET4	15 222	6-13	NOSAME	12 005	4-8
LET5	15 141	6-13	NOSCR	11 071	4-6
LETERR	15 132	6-13	NOT0	23 010	10-18
LINEUP	20 303	10-6	NOTDEL	03 045	5-9
LIST	10 333	4-5	NOTEND	11 336	4-8
LOOK0	20 124	10-4	NOTPLM	23 120	10-20
LOOKU1	05 061	8-10	NUMERR	12 375	4-11
LOOKU2	05 111	8-10	NXTLIN	13 116	6-3
LOOKU4	05 201	8-11	OPLOAD	22 266	10-16
LOOKUP	05 033	8-10	OPSGNT	21 230	10-10
LOOP	03 003	5-8	OUTDGS	25 045	10-26
LT	06 121	8-13	OUTDGX	25 105	10-26
MINEXP	24 033	10-22	OUTDIG	25 032	10-26
MORACC	20 313	10-6	OUTFIX	24 271	10-24
MORCOM	22 155	10-14	OUTFLT	24 253	10-24
MOROP	20 330	10-6	OUTNEG	24 207	10-24
MOVEC	12 046	4-9	OUTRNG	55 136	9-7
MOVECP	10 261	6-51	OUTX10	25 223	10-27
MOVEIT	21 013	10-7	OUTZER	25 104	10-26
MOVEPG	12 050	4-9	PARNER	06 104	8-13
MOVOP	20 222	10-5	PARNUM	04 356	8-5
MROUND	21 302	10-11	PARSE	05 231	8-11
MULOOP	06 341	8-15	PARSE1	05 307	8-12
MULTIP	21 066	10-9	PARSE2	05 332	8-12
NE	06 230	8-14	PARSEP	31 300	7-8
NEGEXP	32 041	9-10	PARSER	04 324	8-5
NEGFPFA	21 251	10-10	PCOM1	15 003	6-11
NEXT	30 013	6-35	PCOMMA	14 357	6-11
NEXT1	30 030	6-35	PERIOD	23 201	10-20
NEXT2	30 045	6-36	PFPOUT	14 314	6-10
NEXT3	30 071	6-36	POSEXP	23 365	10-21
NEXT4	30 130	6-36	PRIGHT	07 003	9-5
NEXT5	30 300	6-37	PRIGH1	55 000	9-7
NEXT6	30 351	6-38	PRINT	13 345	6-5
NEXT7	31 005	6-38	PRINT1	13 366	6-5
NEXT8	31 027	6-38	PRINT2	14 002	6-5
NEXT9	31 042	6-38	PRINT3	14 043	6-8
NEXT10	31 143	6-39	PRINT4	14 075	6-8
NEXT11	31 170	6-39	PRINT5	14 114	6-8
NEXT12	31 177	6-39	PRINT6	14 125	6-8
NINPUT	23 115	10-19	PUSHIT	25 131	10-26
NODECP	25 154	10-27	QUOROT	21 377	10-13
NOEXCO	20 100	10-4	QUOTE	14 203	6-10
NOEXPO	05 005	8-10	QUOTE1	14 220	6-10
NOEXPS	23 244	10-20	QUOTE2	14 263	6-10
NOGO	21 376	10-13			

QUOTER	14 246	6-10	SQRERR	32 217	9-11
REMOVE	12 144	4-9	SQREXP	32 062	9-10
REMOV1	12 167	4-10	SQRLOP	32 107	9-10
RESIGN	20 175	10-4	SQRX	32 000	9-10
RESTHL	22 337	10-17	STOSY1	10 100	6-49
RESTSY	10 252	6-50	STOSY2	10 126	6-50
RETERR	16 356	6-25	STOSY3	10 156	6-50
RETURN	16 304	6-23	STOSY5	10 227	6-50
RNDX	32 240	9-11	STOSYM	10 055	6-49
ROTATL	22 177	10-15	STRCP	02 332	5-7
ROTATR	22 211	10-15	STRCPA	02 370	5-8
ROTL	22 200	10-15	STRCPB	02 356	5-8
ROTR	22 212	10-15	STRCPL	02 344	5-7
RUN	13 070	6-3	STRIN	03 014	5-8
			STRIN1	03 016	5-8
			STRINF	03 102	5-9
SAMLIN	13 156	6-3	SUB12	25 341	10-28
SAVEHL	22 317	10-16	SUBBER	22 223	10-15
SAVESY	10 240	6-50	SUBEXP	21 334	10-11
SCAN1	03 254	7-5	SUBHL	03 113	5-9
SCAN2	03 300	7-5	SUBTRA	22 224	10-15
SCAN3	03 345	7-5	SWITCH	22 356	10-17
SCAN4	03 357	7-5	SYNERR	11 152	4-6
SCAN5	03 373	7-5	SYNTAX	02 000	5-5
SCAN6	04 007	7-5	SYNTOK	11 161	4-6
SCAN7	04 038	7-6	SYNTX1	02 015	5-5
SCAN8	04 064	7-6	SYNTX2	02 044	5-5
SCAN9	04 100	7-6	SYNTX3	02 061	5-5
SCAN10	04 301	7-7	SYNTX4	02 067	5-5
SCAN11	04 143	7-6	SYNTX5	02 124	5-6
SCAN12	04 206	7-7	SYNTX6	02 171	5-6
SCAN13	04 251	7-7	SYNTX7	02 210	5-6
SCAN14	04 260	7-7	SYNTX8	02 215	5-6
SCAN15	04 267	7-7	SYNTXL	02 137	5-6
SCAN16	04 276	7-7			
SCANFN	03 351	7-5	TAB1	10 022	9-9
SCRLOP	11 060	4-6	TABAD1	07 231	9-4
SETDCT	21 345	10-12	TABADR	07 230	9-4
SETIT	22 272	10-16	TABC	10 042	9-9
SETMCT	21 062	10-8	TABLOP	10 045	9-9
SETSUB	22 101	10-13	TABX	10 017	9-9
SGNX	07 360	9-9	TEXTC	03 121	5-9
SHACQP	20 341	10-6	TEXTCL	03 125	5-9
SHIFTO	20 327	10-6	TOMUCH	25 353	10-28
SHLOOP	20 374	10-6			
SKPNNEG	20 264	10-5	ZERO	14 336	6-11
SQRCNV	32 203	9-11	ZERODG	25 165	10-27

NOTES

In the assembled listings the contents of address locations 01 272 through 01 303 as presented on page 4 in chapters 12 and 13 should be changed to contain the following data.

01 272	004	(cc) for SAVE
01 273	323	S
01 274	301	A
01 275	326	V
01 276	305	E
01 277	004	(cc) for LOAD
01 300	314	L
01 301	317	O
01 302	301	A
01 303	304	D

These locations were incorrectly identified in the listing as being part of the Arithmetic Stack and a temporary storage location for the FPACC. Failure to make the above corrections in the listings will result in the program being unable to correctly respond to a SAVE or LOAD executive command directive.

PATCH NR. 1

The following patch is recommended to correct a condition in the floating point addition subroutine. If the patch is not installed, small mathematical errors may be introduced into calculations (in the order of 10 to the minus seventh power) under certain conditions. These errors are caused by residue left in the FPOP (Floating Point OPerand) extension byte. The patch clears the FPOP extension byte when it is used by the portion of the FPADD subroutine labeled SHACOP.

To correct the source listing make a note in chapter ten on page 6 that two instructions:

LLI 133
LMI 000

should be inserted between the second and third lines of the subroutine labeled SHACOP.

listings of the program provided, it is recommended that the first two instructions of the subroutine labeled SHACOP be changed to read:

SHACOP, CAL PATCH1
LAA

PATCH1 may be inserted starting at location 000 on page 30 in the assembled versions. Making this change to direct the program to the patch would result in the 8008 listing shown on page 48 of chapter

12 in the vicinity of the label SHACOP to appear as:

20 341 106 000 030 SHACOP, CAL PATCH1
20 344 300 LAA

Similarly, the 8080 version listing on page 48 of chapter 13 would appear as:

20 341 315 000 030 SHACOP, CAL PATCH1
20 344 177 LAA

The actual PATCH1 subroutine would need to contain the two instructions replaced by the calling sequence as well as the two instructions being added. The unused bytes starting at location 000 on page 30 (in the program listing) as shown for the 8008 version on page 68 of chapter 12 would appear as:

30 000 000 100

30 004 066 133	LLI 133
30 006 076 000	LMI 000
30 010 007	RET

While the 8080 version (page 68 of chapter 13) would appear as:

30 000 056 123	PATCH1, LLI 123
30 002 066 000	LMI 000
30 004 056 133	LLI 133
30 006 066 000	LMI 000
30 010 311	RET

NOTES

PATCH NR. 2 - Revised

The implementation of PATCH NR. 1 will cause a problem to occur in the square root function subroutine if the following PATCH NR. 2 is not also installed. This is because the square root routine initially only tested for convergence by examining the size of the exponent involved. The increased accuracy that results when PATCH NR. 1 is implemented can cause certain values to converge to zero as the square root is calculated. PATCH NR. 2 inserts an additional test for the zero condition in that subroutine. Failure to implement this patch when the first patch has been implemented can result in the square root function subroutine "hanging up" in an endless loop when an attempt is made to take the square root of a number such as 1.0 or 4.0! The following patch corrects for this possibility.

To correct the source listing make a note in chapter nine on page 11 that the following instructions:

DCL
LAM
NDA
JTZ SQRCNV

should be inserted between the 16th and 17th lines from the top of that page between the instructions:

JTS SQRCNV
and
LLI 034

To implement the correction for the assembled listings of the program provided, it is recommended that a patch be made by changing the JTS SQRCNV instruction which starts at address 32 163 to the instruction JMP PATCH2 and adding a label SQR1 to the LLI 034 instruction which is at location 32 166. The actual patching instructions may be placed starting at address 32 364 and would consist of the

sequence:

PATCH2, JTS SQRCNV
DCL
LAM
NDA
JTZ SQRCNV
JMP SQR1

Implementing the patch in this recommended fashion would result in the 8008 version (chapter 12 page 75) being altered at the following addresses to appear as:

32 163 104 364 032 JMP PATCH2
32 166 066 034 SQR1, LLI 034

at the patch to appear as:

32 364 160 203 032 PATCH2, JTS SQRCNV
32 367 061 DCL
32 370 307 LAM
32 371 240 NDA
32 372 150 203 032 JTZ SQRCNV
32 375 104 166 032 JMP SQR1

Similarly, for the 8080 version (chapter 13 on page 75) the patch would result in the following locations being changed to:

32 163 303 364 032 JMP PATCH2
32 166 056 034 SQR1, LLI 034

and the patch locations to appear as:

32 364 372 203 032 PATCH2, JTS SQRCNV
32 367 055 DCL
32 370 176 LAM
32 371 247 NDA
32 372 312 203 032 JTZ SQRCNV
32 375 303 166 032 JMP SQR1

NOTES

PATCH NR. 2 - Revised

The implementation of PATCH NR. 1 will cause a problem to occur in the square root function subroutine if the following PATCH NR. 2 is not also installed. This is because the square root routine initially only tested for convergence by examining the size of the exponent involved. The increased accuracy that results when PATCH NR. 1 is implemented can cause certain values to converge to zero as the square root is calculated. PATCH NR. 2 inserts an additional test for the zero condition in that subroutine. Failure to implement this patch when the first patch has been implemented can result in the square root function subroutine "hanging up" in an endless loop when an attempt is made to take the square root of a number such as 1.0 or 4.0! The following patch corrects for this possibility.

To correct the source listing make a note in chapter nine on page 11 that the following instructions:

DCL
LAM
NDA
JTZ SQRCNV

should be inserted between the 16th and 17th lines from the top of that page between the instructions:

JTS SQRCNV
and
LLI 034

To implement the correction for the assembled listings of the program provided, it is recommended that a patch be made by changing the JTS SQRCNV instruction which starts at address 32 163 to the instruction JMP PATCH2 and adding a label SQR1 to the LLI 034 instruction which is at location 32 166. The actual patching instructions may be placed starting at address 32 364 and would consist of the

sequence:

PATCH2, JTS SQRCNV
DCL
LAM
NDA
JTZ SQRCNV
JMP SQR1

Implementing the patch in this recommended fashion would result in the 8008 version (chapter 12 page 75) being altered at the following addresses to appear as:

32 163 104 364 032 JMP PATCH2
32 166 066 034 SQR1, LLI 034

at the patch to appear as:

32 364 160 203 032 PATCH2, JTS SQRCNV
32 367 061 DCL
32 370 307 LAM
32 371 240 NDA
32 372 150 203 032 JTZ SQRCNV
32 375 104 166 032 JMP SQR1

Similarly, for the 8080 version (chapter 13 on page 75) the patch would result in the following locations being changed to:

32 163 303 364 032 JMP PATCH2
32 166 056 034 SQR1, LLI 034

and the patch locations to appear as:

32 364 372 203 032 PATCH2, JTS SQRCNV
32 367 055 DCL
32 370 176 LAM
32 371 247 NDA
32 372 312 203 032 JTZ SQRCNV
32 375 303 166 032 JMP SQR1

NOTES

In the source listing, on page 13 of chapter 10, change the first three instructions in the routine labeled DVEXIT to appear as follows:

DVEXIT, LLI 143
LEI 123
LBI 004

Change the corresponding section of the assembled listing for the 8008 version on page 52 of chapter 12 to read:

22 070 066 143
22 072 046 123
22 074 016 004

DVEXIT, LLI 143
LEI 123
LBI 004

And the 8080 version on page 52 of chapter 13 to read:

22 070 056 143
22 072 036 123
22 074 006 004

DVEXIT, LLI 143
LEI 123
LBI 004

This revision will correct a residue problem which can cause incorrect results to occur when a number with a negative exponent is divided into the value zero.

NOTES

CHANGE OF ADDRESS NOTICE

NAME:

COMPANY:
(if applicable)

NEW ADDRESS

STREET:

CITY: STATE: ZIP:

FORMER ADDRESS

STREET:

CITY: STATE: ZIP: