$14.95

IMPLEMENTING BASICs:

◆How BASICs Work◆

◆William Payne & Patricia Payne◆

IMPLEMENTING BASICs
Payne & Payne

REWARD BOOKS

Implementing BASICs

How BASICs Work

Implementing BASICs

How BASICs Work

William Payne
and
Patricia Payne

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

Library of Congress Cataloging in Publication Data

Payne, William H.
Implementing BASICs.

Includes index.
1. Basic (Computer program language)
I. Payne, Patricia, 1940- . II. Title.
QA76.73.B3P259 001.64'24 81-23451
ISBN 0-8359-3045-9 AACR2
ISBN 0-8359-3044-0 (pbk.)

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

To four Ph.D.'s well-educated in the computing discipline

H. Blair Burner
Richard J. Hanson
Theodore G. Lewis
John S. Sobolewski

Contents

Preface

Most schools do not teach techniques for implementing high-level computer languages for microcomputers directly from microcode or machine language. These techniques have been developed in the commercial world. BASIC, because of its simplicity and arbitrary set of verbs, is the most frequently and directly implemented high-level language.

Applications programming that uses these BASICs, which are built using new language design and implementation techniques, is considerably different from applications programming that uses languages based on older principles. Writing applications programming using verbs such as HEXPACK, SOUND, COLOR, MATSEARCH, LOAD, MATSORT, DATASAVE, PAINT, and so on offers exciting new challenges. Manipulating high-level language stacks, and writing either microcode or machine language subprograms in the high-level language, give the application programmer the power to make computers easily perform tasks that were considered difficult by previous standards.

Many microprocessors have rich instruction sets. Those with many addressing modes and other hardware features present programmers with the problem of how to develop quality software at a reasonable cost. Understanding the techniques used by commercial microcomputer systems programmers to simplify the complex is valuable for all computer programmers.

The purpose of this book is to help an individual working with the BASIC language to achieve better software system designs and more efficient programming techniques. This book achieves its purpose by exploring principles and techniques used in microcomputer high-level language design and implementation.

Acknowledgements

Some of the more significant events which led to the writing of this book occurred over a 15-year period in both academic and commercial settings. In the mid-1960s, I participated as a committee member on Terry Hamm's masters thesis on interpreters at Washington State University in Pullman, Washington. Blair Burner, who was on the computer science faculty at W.S.U., directed Terry's thesis work. Two of my graduate students at the time, Dave Anderson and Ted Lewis, took Blair's compiler course, and both gave me detailed accounts of the course content. Shortly thereafter, I observed how Blair and some of Blair's students brought up a state-oriented timesharing system on an Interdata minicomputer, and was struck by the amount of work it involved.

In 1972, Jim Robertson, a professor of computer science at the University of Illinois, gave me a copy of Bruce DeLugish's Ph.D. thesis. DeLugish's thesis developed continued product algorithms for evaluation of elementary functions. The following year I taught a graduate course at W.S.U. on microcomputer algorithms.

In 1974, I directed two W.S.U. graduate student's masters projects on language development. Stephen Choi implemented a mobile language system, and Dave Gruhn wrote a BASIC interpreter system he called BASIC MINUS.

In the spring of 1976, I taught a graduate seminar on language development for microcomputers. Charles Moore demonstrated FORTH to me in November of 1976.

In 1976, Patty and I also began working commercially on software using Wang 2200 computers. Jon Estep and Mike Korach had introduced us to turnkey applications systems, and we all became expert in problems of software development. These problems include distribution; maintenance—for instance, fixing software bugs or isolating hardware failures; user training; and the high costs involved.

Dale Swenson, Larry Stein, Stan Wagner, and Pat Corey helped firm up our ideas on how software-engineered systems must be built. We

used P. Polgar's software standards manual for Boeing 757/767 airplanes to clarify our thoughts.

John Brackett, the president of Softech Microsystems in San Diego, is in the process of trying to show that it may be possible to earn a living selling language and operating systems for microcomputers.

Application systems programmers Wally Stricklin, Bonnie Meyer, Monti Mecham, Celina DeQuadros, Ron Bayley, and John Robertson all showed us how productive programmers can be when equipped with Wang BASIC-2 language systems.

In 1976, Joe Sidowski suggested that I write an academic article on high-level language development techniques for microcomputers. After about four years and several attempts, we decided to get serious about this project, abandon writing an article for some obscure academic journal, and write this book.

William Payne

1

Language Commands, Statements, and Their Variables

The purpose of Chapter 1 is to explain how commands, statements, and their variables are stored and processed in microcomputer memory.

In the case of a BASIC Language System, the microcomputer memory is partitioned into distinct functional regions. Each memory region is controlled by the Language System microcomputer programs. These programs place statements, commands, and variables in particular regions of the microcomputer's memory.

Within the confines of this first chapter, we will discuss three regions of microcomputer memory. They are: the program text area, the variable table, and the systems tables. Statements, commands, and their variables present information in the variable table and program text area of microcomputer memory. The systems tables of microcomputer memory contain information concerning the variable table and the program text area.

Microcomputer memory layout is diagrammed from an applications standpoint in Figure 1. Language statements, language commands, and variables determine how the program text area and the variable table are filled.

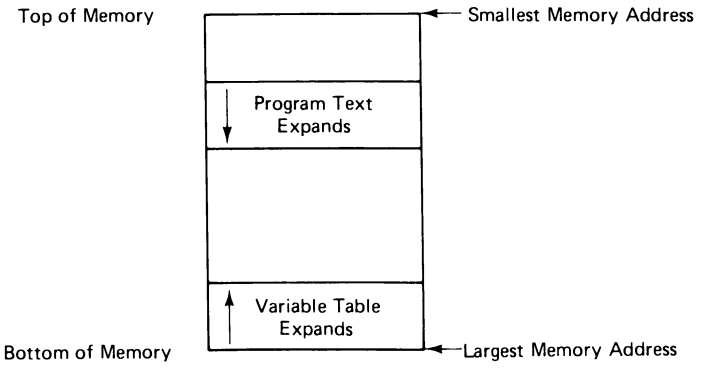


Figure 1: Rough diagram of microcomputer memory layout. Program text expands upward in memory as statements are added to the computer program. Memory storage area for the variables expands from high order memory address to lower addresses as variables are defined in the computer program.

LANGUAGE COMMANDS

Language commands are sometimes referred to as *immediate mode commands* or, simply, *commands*. Commands are entered into the microcomputer from a keyboard. When the carriage return (CR) is keyed, the command is immediately executed by the microcomputer's BASIC Language System.

In our first example, entry of the command:

PRINT A (CR)

causes the Language System to search the variable table for the variable A. If the variable is not found, then the Language System adds it to the variable table and sets it at an initial value. The initial value for a number in most BASIC systems is 0; the initial value for a character string is all blanks. When the value of A has been set, its number is printed on the device selected for PRINT. Although this print device is usually some type of cathode ray tube (CRT), it could well be some other instrument, such as a printer.

If a command references a variable, the Language System will enter it in the variable table, provided it is not there already. Command text is executed immediately and never entered into the program text region. Hence, there is no reason to save it there.

Some commands do not reference any variables. An example is:

```
SELECT PRINT 215(132)
```

This particular command selects the address of a print device and also the maximum number of print spaces that can be allocated before a CR and line feed are automatically generated. When such a command is executed, the contents of the variable table are not affected.

Suppose the Language System was initialized prior to issuing the PRINT A command. Remember that Language System initialization occurs at the time when the computer is powered up or when the Language System is loaded from a permanent storage device. Now suppose the two commands:

```
PRINT A   (CR)
B=2       (CR)
```

were both issued. The Language System searches the variable table for variable A; when A is not found, it is placed in the variable table and is given the initial value of 0. When the command B=2 is accepted by the Language System, the system then searches the variable table. Because B is not found either, B, like A, is added to the table. B is then initialized, or set to 0. Scan of the command B=2 is continued and eventually the value of B is set to 2.

Figure 2 is a diagram of the contents of microcomputer memory following the entry of these two commands. The important points to note are: 1) Nothing is stored in the program text region of memory, and 2) each of the variable names and its value is added to the variable memory region, beginning at the bottom and expanding backwards through the memory.

If variables other than A and B are referenced in commands, then they and their values are added to the variable table behind, rather than ahead of, those which preceded them.

Commands are useful for two reasons. The first is because the Language System can resemble a calculator. An example of this is the command:

```
PRINT 3*2+4   (CR)
```

This command would cause 10 to be printed on the device selected for print output. Commands are also useful for printing or changing variables during program debugging.

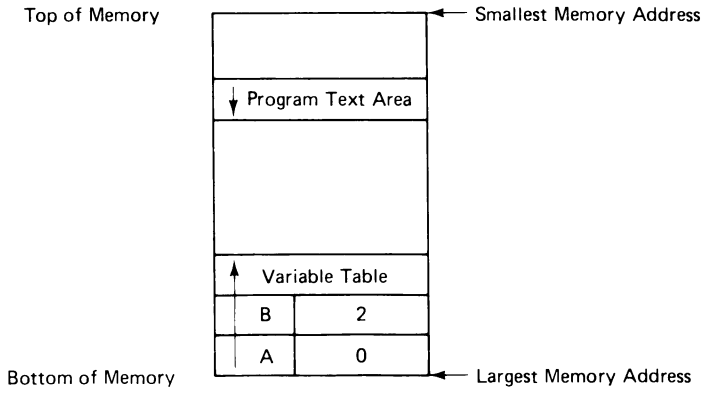


Figure 2: Rough diagram of the microcomputer memory after the PRINT A and B=2 have been processed by the Language System in the variable table.

LANGUAGE STATEMENTS

Language statements (or just statements) resemble commands, but with one important difference: statements must be preceded by a label. In BASIC the label is a number, but in other languages the label may be an alphanumeric character string.

Figure 3 shows the formats of commands as they compare with statements. One difference between the two is that in some highly interactive languages, commands are distinguished from statements by the presence or absence of a label. The language FORTH is different because the entire statement must be enclosed in a : ; sequence.

Unlike commands, statements are not executed immediately, but are placed first in the program text region of memory. To accommodate additional statements the size of the region is increased in a forward direction.

Another difference between commands and statements concerns variables. They are not placed in the variable table at the same time statements are being entered into the program text region, but rather at a later time, called *program resolution.*

Suppose the Language System is initialized such that the program text region and variable table are empty. If the statements:

```
10 X = 3        (CR)
30 PRINT Q,X    (CR)
20 Z,Q = X*8    (CR)
```

are entered into the Language System, then statement 10 is placed in the program text region first, with statement 30 directly following. BASIC program statements are ordered by increasing line number, so, when statement 20 is entered, the Language System moves statement 30 to allow room for 20.

A rough diagram of the microcomputer memory holding this program is shown in Figure 4.

Example Number	Command	Statement
1	A=1	10 A=1
2	B$="SANDIA"	20 B$="SANDIA"
3	1 ' A !	: TEN 1 ' A ! ;

Figure 3: Comparisons of commands and statements in several languages. BASIC-like commands and statements are compared in examples 1 and 2: 10 and 20 are the labels. A FORTH command is compared with the statement in example 3. TEN is a label (called a "word" in FORTH). FORTH requires that a statement must be enclosed in a : ; sequence of delimiters. All commands and statements are terminated by a carriage return (CR). Language Systems often acknowledge a CR with a line feed.

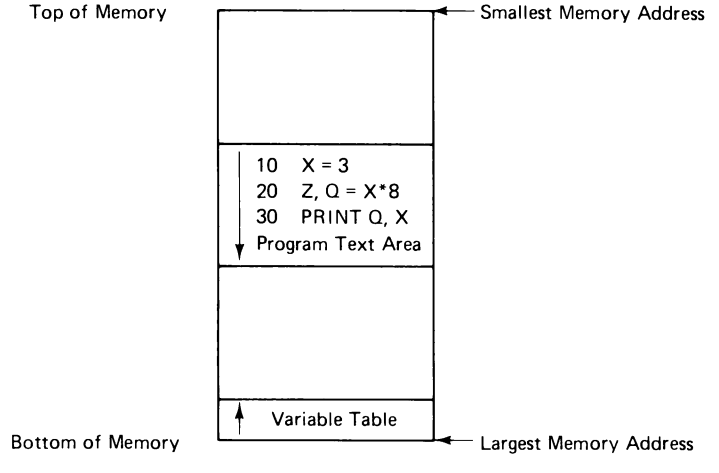


Figure 4: Rough diagram of computer memory after a three line program has been entered. BASIC statements must be preceded by a line number. The Language System orders lines by their number and also performs any necessary text insertion. The variable table is not constructed when statements are entered, but at a later time, called *resolution.*

The program shown in Figure 4 can be run by entering the RUN command into the Language System. When RUN is entered, the Language System performs a series of operations, called *program resolution,* on the statements contained in the program text area of memory. One of the functions of program resolution is to scan the program statements for variable names and then place the names and initial values of the variables in the variable table. Another function of resolution is to mark the program text executable if no obvious errors have been found in the statements by the Language System. This process occurs "before" the program is actually run. The RUN command has more functions associated with it other than merely causing the program to run; it also performs the series of operations that together constitute program resolution.

Figure 5 provides a rough diagram of the memory that would appear following resolution, but before running the program shown in Figure 4.

When the Language System enters the run phase of the RUN, program execution begins. Statement 10 causes the Language System to search the variable table for the variable named X. When X is found, its value is changed to 3. Statement 20 causes the Language System to search the variable table for variables Z, Q, and X. The values of variables Z and Q are then replaced by 24, which is the product of X*8. Statement

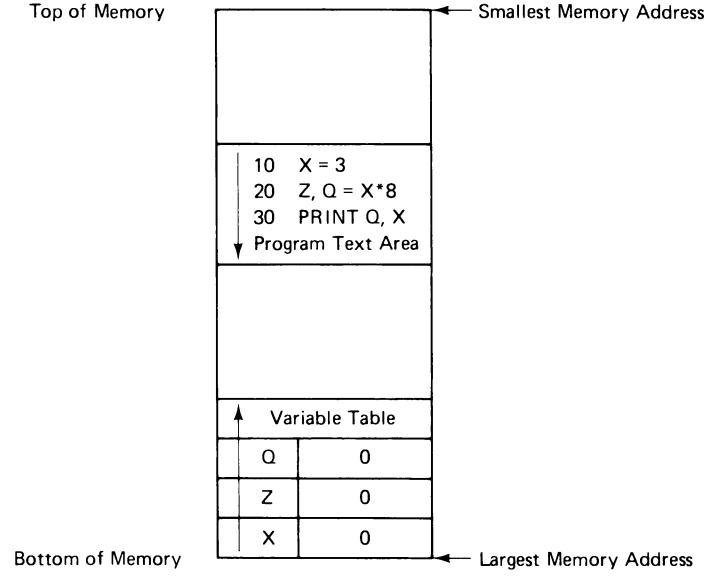


Figure 5: Rough memory diagram of the program seen in Figure 4 after both the RUN command has been entered, and the resolution of the program statements has occurred, but before the program has been executed.

30 will result in Q and X being searched for in the variable table. When the values of these variables are found, they are sent to the PRINT device.

Figure 6 is a rough memory diagram of the program seen in Figure 5 when execution is complete.

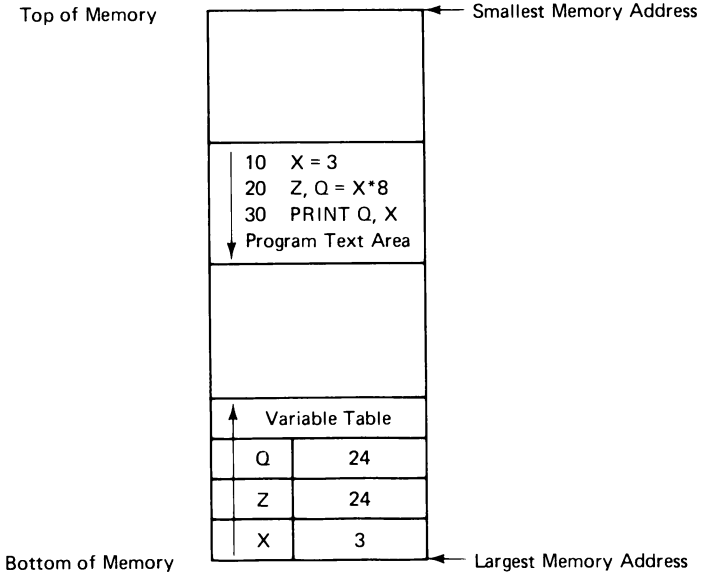


Figure 6: Rough memory diagram of the program seen in Figure 5 when program execution is complete. Program execution was initiated by entering the RUN command.

INTERSPERSION OF COMMANDS AND STATEMENTS

Commands and statements may be interspersed. To better explain how this interspersion is carried out, we here give several examples. Suppose that the Language System has been initialized and that this series of commands and statements is entered:

```
R=2              (CR)
10 P,Q=T+R       (CR)
T=3              (CR)
PRINT Q,T,R      (CR)
20 PRINT S,P     (CR)
```

R = 2 is a command—so when it is entered, the Language System scans the variable table to try to find R. R is not found, so it is entered into the variable table and its value is set at 2. 10 P,Q = T + R is a statement, so it is placed in the program text area. T = 3 is a command and is executed immediately. The name T is searched for in the variable table. Since T is not found, its name is entered into the table and its value is set at 3. PRINT Q,T,R is a command so it is executed immediately. The Language System searches for Q. Since Q is not found, it is entered into the variable table, and its value is initialized at 0. Because both T and R are found in the variable table, the values of Q, T, and R (which are 0, 3, and 2 respectively) are sent to the device selected for PRINT. 20 PRINT S,P is a statement, so it is added to the statements in the program text area of memory.

Figure 7 is a rough diagram of the microcomputer memory after all of the above statements have been entered through the keyboard.

Variables S and P are not included in the variable table because program resolution, initiated by the RUN command, has not yet been performed.

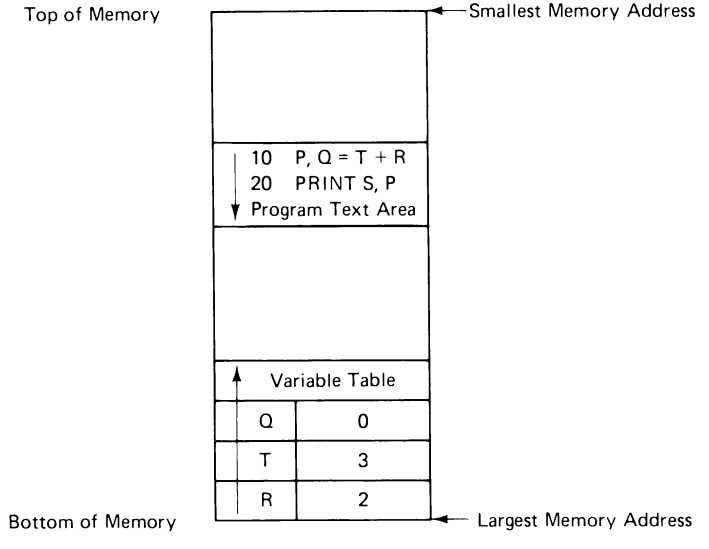


Figure 7: Rough diagram of computer memory after entry of three commands and two program statements. Variables Q, T, and R are included in the variable table because these variables were referenced in commands. Statements 10 and 20 are in memory, but variables S and P are not entered in the variable table as program resolution has not yet occurred.

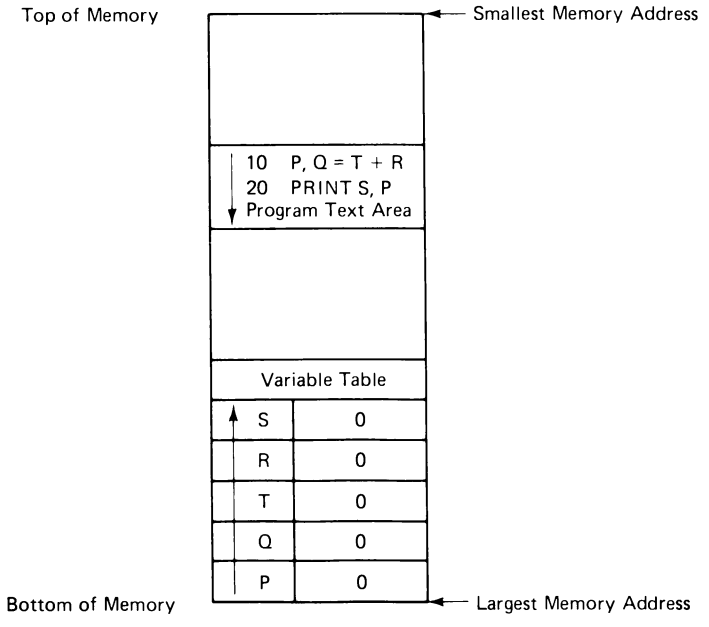


Figure 8: Rough diagram of the resultant memory after the program text seen in Figure 7 is resolved. Program resolution was initiated by entry of the RUN command.

Entry of the RUN command into the Language System causes three actions to occur:

1. the variable table is cleared of all variables;
2. program resolution is performed;
3. the program is executed.

When RUN is entered from the console input device, usually a keyboard, all the variables are cleared from the variable table.

If the Language System detects no obvious errors in the program text, the program is marked "executable" and the Language System will begin to orchestrate execution of the program text. Figure 8 is a rough diagram of computer memory following completion of the resolution phase of RUN for the program text seen in Figure 7.

During program resolution, the program text is scanned and analyzed. Proceeding from first to last, each and every statement is scanned and analyzed from left to right. During program resolution the variables are identified, placed in the variable table, and then initialized.

For statement 10 P,Q = T + R, the Language System has decided that P is a variable. P is searched for in the variable table. No P was found, so it is added to the variable table, and its value initialized at 0. Q is identified as the next variable. No Q is found, so it, like P, is entered into the variable table and its value initialized at zero. T and R are also identified as variables, placed in the variable table and their values initialized at zero.

Statement 20 is processed next. Variable P is found in the variable table, so no action is taken. Variable S is not found in the variable table, so it is entered and its value set at 0.

When the program is run, the variable table does not change, since zeros are added to zeros. Only two zeros, the values of S and P, are printed.

Suppose the command:

Z,R = 5

is now entered and executed by the Language System. Figure 9 presents

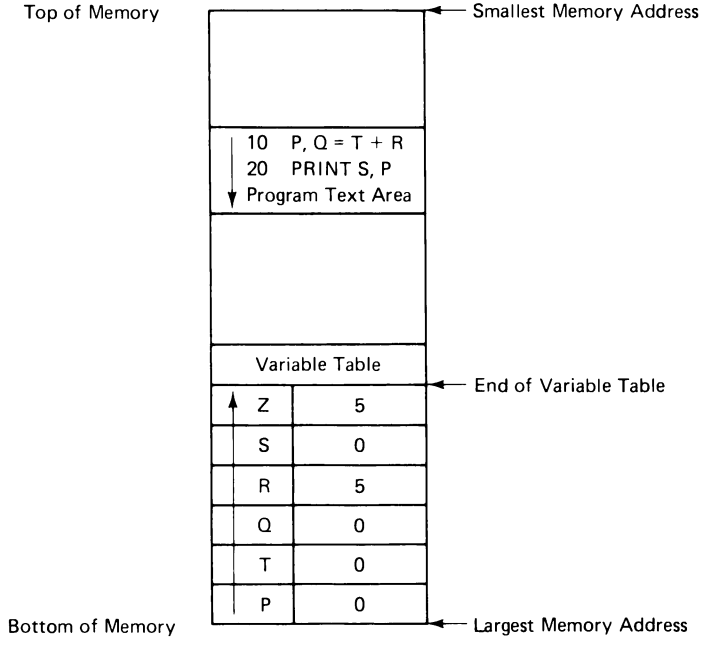


Figure 9: Rough memory diagram of the result after the program text seen in Figure 8 has been executed and the command Z,R = 5 has been entered and executed.

the variable table after this command has been executed. The variable table is searched for variable Z. It is not found, so it is entered into the table. The variable table is searched for variable R. It is found. Both Z and R are then set to the value 5.

Suppose that by entering the RUN command, the program text seen in Figure 9 is run once again. As a consequence of the program resolution phase, all variables are removed from the variable table, reentered again, and reinitialized.

Suppose the command Z,R = 5 is issued again, but that by issuing the command GOTO 10 followed by a CONTINUE command, the program is rerun. The program text region of memory has not been changed, so re-resolution of the program does not have to be made. A rough diagram of the program text region of memory and the variable table is shown in Figure 10.

The variable table was not reconstructed and reinitialized when this method of rerunning the program was used. If the program text region of memory was changed in any manner, then the Language System would

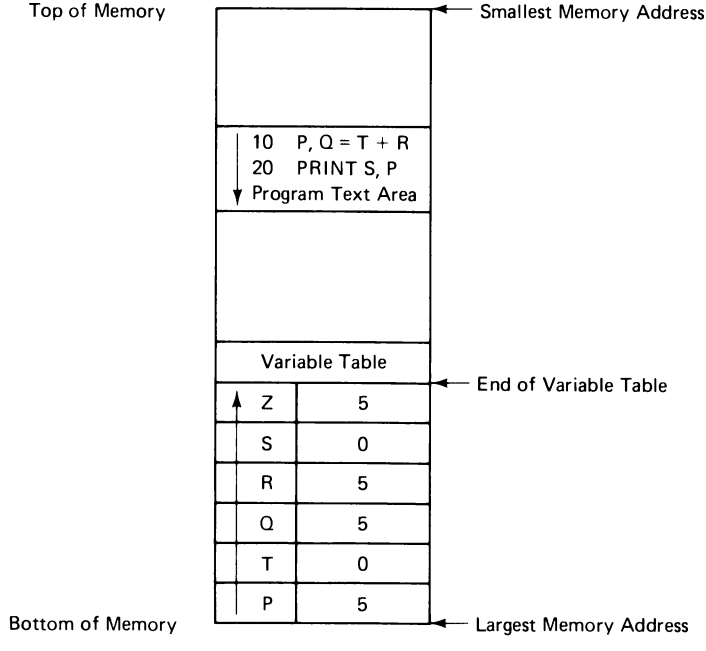


Figure 10: Rough memory diagram of the result after the program text seen in Figure 9 has been run for a second time, after Z,R = 5 was executed, and initiating execution with a GOTO 10 and CONTINUE commands.

require a resolution of the program via entry of a RUN command.

Initiation of program execution by a GOTO-CONTINUE command sequence is a valuable procedure when debugging programs.

INTRODUCTION TO SYSTEMS TABLES

Four numbers are necessary to locate the variable table and the program text area. These numbers are:

1. Pointer to the start of the program text area.
2. Pointer to the end of the program text area.
3. Pointer to the end of the variable table.
4. Address of the beginning of the variable table.

All four of these numbers are kept in the systems tables of microcomputer memory by the Language System. Three of these numbers are somewhat arbitrary. The word "somewhat" is necessary here since the pointer to the end of the program text area must be numerically smaller than the pointer to the end of the variable table.

The address of the beginning of the variable table is the largest existing address of memory. Memory can usually be added to computers. A microcomputer might be able to address 1,000,000,000,000,000,000,000 bytes of memory, but may only have 1,000 bytes of memory plugged into its boards.

One can write a program to calculate the largest address of existing memory. Suppose a microcomputer which can address 16 memory locations is used as the host computer. Microcomputer memory contains a certain number of memory locations; each memory location contains 1 byte or 8 bits. The existing memory contains less than 16 locations. The problem is to write a computer program for the host microcomputer allowing it to discover the address of the largest existing memory location. The program first attempts to load the contents of memory location 9 into one of its registers. If this operation fails, the microcomputer returns an addressing error, and the conclusion is that less than 9 memory locations exist. Suppose the operation failed. The next step is to address location 5. Suppose this operation was successful. The conclusion is that the memory contains somewhere between 5 and 8 locations. The next step is to attempt to address location 7. Suppose this is successful. This leads to the conclusion that there are either 7 or 8 memory locations. The last step is to attempt to address location 8. Suppose this fails. The conclusion is that the microcomputer system has 7 memory locations. The number of memory access attempts needed to locate the maximum address of existing memory is roughly the base 2 logarithm of the maximum address size of the microcomputer.

The address of the beginning of the variable table should always be calculated by the Language System whenever the microcomputer system is "powered up" because more memory might have been added since the last power up.

For many reasons, the pointer to the start of the program text area is almost never the lowest address of computer memory. Some computers reserve low order addresses for hardware functions. Also, the Language System uses some memory in front of the program text area for its *systems tables*.

Figure 11 contains a more detailed yet still rough diagram of computer memory; it shows the addition of the systems tables region of memory.

The systems tables region of microcomputer memory, unlike the variable table and program text regions, is of fixed size.

SUMMARY

Microcomputer memory, when activated by the BASIC Language System, is divided into distinct regions. So far, the variable table, the program text area, and the systems tables have been identified as parts of the BASIC Language System. The address of the beginning of the variable table is the largest existing address of memory.

The Language System processes commands in one way and state-

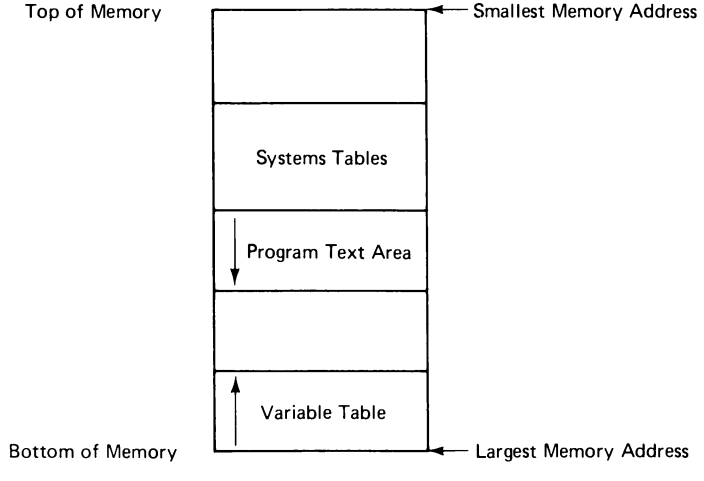


Figure 11: Rough diagram of microcomputer memory showing inclusion of the systems tables region of memory. The systems tables region of memory is of fixed size. Arrows show the directions the program text area and variable table expand.

ments in another. Command variables are first initialized, and then given values in the variable table. Commands are executed immediately; statements are not. The latter are placed first in the program text area and are not executed immediately. The variable table is constructed not when statements are entered but at a later time, called *resolution*. Program resolution occurs after the RUN command has been entered from the input device.

Some of the functions of program resolution are to remove command defined variables from the variable table, scan the program statements for variable names and place discovered names and initial values of the variables in the variable table. If no obvious errors have been found in the statements during program resolution, then the program is marked executable. All of this occurs before the program is run. Program execution is initiated by the RUN command.

The systems tables of the Language System contain a pointer to the start of the program text area, a pointer to the end of the program text area, a pointer to the end of the variable table, and the address of the beginning of the variable table. The systems tables region, unlike the variable table and program text regions, is of fixed size.

2

Microcomputer Data Structures

In this chapter we explain the four types of data structures, or variables, that are used by microcomputers, and the information needed to define them.

Four different basic types of variables are needed for microcomputers:

1. Numeric scalar variables:
   Examples: A, A0, A1, . . . Z7, Z8, Z9
2. Numeric array variables:
   Examples: A(), A0(), A1(), . . . Z7(), Z8(), Z9()
3. Alphanumeric scalar character string variables:
   Examples: A$, A0$, A1$, . . . Z7$, Z8$, Z9$
4. Alphanumeric character string array variables:
   Examples: A$(), A0$(), A1$(), . . . Z7$(), Z8$(), Z9$()

Numeric scalar and numeric array variables have numeric values. For example:

```
B = 2
```

B is a numeric scalar variable, and 2 is the numeric value of B. The exact format of numeric values will vary depending both on the language and the microcomputer implementation. All numeric values will, however,

have a length measured in bytes associated with them. The length of a numeric value is the number of bytes of memory needed to store the numeric value in memory. Integer numeric value lengths are often 2 or 4 bytes. Real numeric value lengths are often 4 or 8 bytes. Throughout this book we have chosen to assign the length of 8 bytes for numeric values. (Numeric values are initialized in memory to zeros.)

The numeric value representation for a variable depends on the application, there is no one "best" numeric value representation. Functional requirements for these values must be met by implementing BASIC.

Alphanumeric scalar character strings and alphanumeric character string arrays consist of a contiguous sequence of bytes. Associated with each is the name of the variable and its maximum length. If no maximum length is specified, a default value of 16 bytes is often assigned. Alphanumeric scalar character string and alphanumeric character string array values are initialized to all blanks.

Numeric array variables and string arrays can be of either one or two dimensions. Several examples of valid array references are: A(10), B6(8,4), Z8$(100,200), R$(16555), ... Also, numeric array values are initialized to all zeros while string array values are initialized to all blanks.

Readers may initially be distressed by the apparent lack of data types for the newer languages but no alarm is necessary. The reason is that many different data types, such as logical, binary, and packed decimal, are all subsumed under the string data type.

Explicit lengths for string variables are usually specified by declaration. An example of specifying the maximum length of 80 for a string variable named B$ is:

```
DIM B$80
```

where DIM means dimension. An example of specifying the maximum length of each element of a double dimensioned string array named R$() as 32, where the row × column maximum dimensions are 2 × 3, is:

```
DIM R$ (2,3)32
```

Information required to define attributes of any variable is:

1. ***Name:*** The name of the variable. Some examples are A, A0, A1, . . . Z7, Z8, Z9.
2. ***Type:*** The variable type is numeric scalar, alphanumeric scalar character string, single dimensioned numeric array, double dimensioned numeric array, single dimensioned alphanumeric character string array, double dimensioned alphanumeric character string ar-

ray . . . (The dots in the last sentence were included to indicate that other variable types could be added if required.)

Variable type specifications can be conveniently represented in a single byte by a binary number. Since definition of more than 256 different data types is unlikely, a single byte will often suffice to contain the variable type attribute value. These variable type attribute values are arbitrarily selected.

To facilitate example construction in later portions of this book, values and definitions of variable types will be defined now.

Hexadecimal Attribute Value	Variable Type
00	Numeric scalar
01	Alphanumeric scalar character string
02	Single dimensioned numeric array
03	Single dimensioned alphanumeric character string array
04	Double dimensioned numeric array
05	Double dimensioned alphanumeric character string array

3. ***Length:*** The length attribute is the maximum length (measured in bytes) required for storage of the variable in computer memory. Lengths of both numeric and string variables should be stored in computer memory.
4. ***Dimension:*** The maximum value of a single dimensioned array, or the values for the maximum dimensions of a double dimensioned array variable, must be stored in computer memory.

Double dimensioned arrays are usually specified by row × column coordinates. A table of the coordinate indexes, I, J, of an array with maximum row × column dimensions of M × N is shown in Figure 12.

Elements in a double dimensioned array are usually stored in a linear reverse order by row in computer memory. The order is 1,1 1,2 . . . 2,1 2,2 . . . . . M,1 M,2 . . . M,N. A specific example of the storing order of elements of a 2 × 3 maximum dimensioned array is given in Figure 13.

A single offset (an offset is a pointer) from the beginning of the array's storage can be calculated from knowledge of the coordinates of the array elements. This offset pointer is used to help retrieve or store array elements and is given by the *array mapping function*. For an array with row × column coordinates of I and J and with maximum row ×

column dimensions of M × N, the array mapping function is:

$$K = \underset{\text{Offset}}{\text{LENGTH}}*(N*(I-1) + J-1)$$

An example of calculation of K for the various values of I and J is given in Figure 13.

Indexes of double dimensioned arrays have the restriction that 1 <= I <= M and 1 <= J <= N. Indexes should be checked before

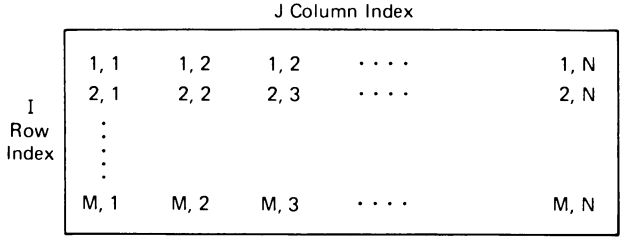


Figure 12: Row and column double dimensioned array coordinate specification conventions. The maximum dimensions of the array are M × N (rows × columns).

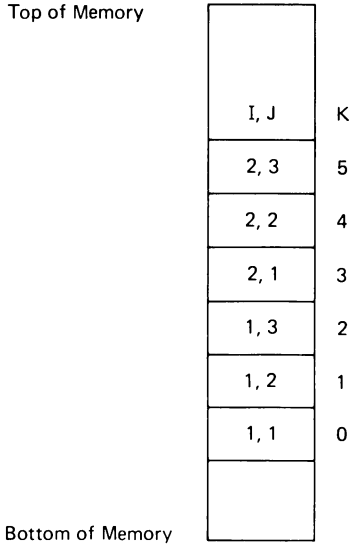


Figure 13: Example of element storage of a 2 × 3 array. The value of the array mapping function K = 1*(3*(I − 1) + J − 1) is given for each value of I and J.

array elements are accessed to insure they are within the proper limits. Indexes should be truncated to integers before use.

The array mapping function for a single dimensioned array is

$$K = LENGTH*(J-1) \text{ where } 1 <= J <= N.$$

The variable type concatenated with the attribute value defines unique variables. Thus a double dimensioned array could be distinguished from a single dimensioned array with the same name. In practice, allowing double dimensioned and single dimensioned arrays within the same program is not allowed. The reasons for this are:

A. Matrix statements such as

```
MAT A=0
```

where A is a matrix are usually allowed in the languages. Double dimensioned and single dimensioned variables could not be distinguished in such statements;

B. Alphanumeric literal string matrix statements such as

```
B$() = "THIS IS A VERY LONG STRING WHICH NEEDS TO BE
       STORED IN AN ARRAY"
```

could not be used because there could be both a single and double dimensioned B$().

SUMMARY

The four basic types of data structures for microcomputers are: numeric scalar, alphanumeric scalar character string, numeric array, and alphanumeric character string array. In memory, numeric values are initialized to zeros, and alphanumeric values are initialized to all blanks.

The attributes of any variable are: name, type, length, and dimension. The array mapping function is used to compute a single offset pointer from the knowledge of the array elements coordinates and is used to help retrieve or store array elements.

3

Variable Table Structure

This chapter demonstrates the procedure by which variables are stored in the variable table. Variable attribute information is stored with the value of variables in the variable table. Some of this information can be modified with statements, such as MATREDIM.

The variable table begins at the bottom of memory and expands toward the top. The limits of the variable table region in memory are defined by two numbers:

1. The address of the beginning of the variable table. This address is the highest location existing in the microcomputer memory.
2. The address of the end of the variable table. This number decreases in value as variables are added to the variable table.

Information about variables is placed in the variable table in one of the two following formats:

1. For numeric scalar or alphanumeric scalar character string variables:
   E. Data
   D. Length
   C. Type
   B. Name of variable
   A. Pointer to the next variable in the variable table

2. For alphanumeric character string or numeric array variables:

   G. Data
   F. Maximum column dimension } or Maximum dimension for
   E. Maximum row dimension } single dimensioned variable
   D. Length
   C. Type
   B. Name of variable
   A. Pointer to the next variable in the variable table

The pointer to the next variable in the variable table has two or four bytes of memory allocated for its storage. This number is usually binary. One byte of storage is commonly allocated for storage of the type attribute value.

The length is usually allocated one byte of storage. The length is usually stored as a binary number. A length of zero is commonly used for a special purpose, so normal variables can have lengths between 1 and 255. Numeric values have been assigned a length of 8 bytes. Alphanumeric values have been assigned a default value of 16 bytes if no length is specified.

The maximum row dimension and the maximum column dimension for double dimensioned arrays are usually 255. The maximum dimension for a single dimensioned array is often 655535 = 256*256-1. In either case, two bytes of storage will be sufficient to contain dimension information.

These simplified diagrams of six variable tables make clear how information can be stored. △ denotes a blank.

1. *Example:* Z=3

Bytes	Value	Comments
6–13	3	Eight byte number
5	8	LENGTH
4	00	TYPE
2–3	Z△	NAME
0–1	Pointer	Pointer explained later

2. *Example:* B9$="SANDIA"

Bytes	Value	Comments
6–21	SANDIA△△△△△△△△△△	Variable value
5	16	LENGTH
4	01	TYPE
2–3	B9	NAME
0–1	Pointer	Pointer explained later

3. *Example:* DIM Y2(4)

Bytes	Value	Comments
32–39	0	Initial value Y(4)
24–31	0	Initial value Y(3)
16–23	0	Initial value Y(2)
8–15	0	Initial value Y(1)
6–7	4	Maximum array dimension
5	8	LENGTH
4	02	TYPE
2–3	Y2	NAME
0–1	Pointer	Pointer explained later

4. *Example:* DIM T0$(3)4

Bytes	Value	Comments
16–19	△△△△	Initial value T0$(3)
12–15	△△△△	Initial value T0$(2)
8–11	△△△△	Initial value T0$(1)
6–7	3	Maximum array dimension
5	4	LENGTH
4	03	TYPE
2–3	T0	NAME
0–1	Pointer	Pointer explained later

5. *Example:* DIM F4(2,2)

Bytes	Value	Comments
32–39	0	Initial value F4(2,2)
24–31	0	Initial value F4(2,1)
16–23	0	Initial value F4(1,2)
8–15	0	Initial value F4(1,1)
7	2	Maximum column dimension
6	2	Maximum row dimension
5	8	LENGTH
4	04	TYPE
2–3	F4	NAME
0–1	Pointer	Pointer explained later

6. *Example:* DIM Z6$(3,2)3

Bytes	Value	Comments
23–25	△△△	Initial value Z6$(3,2)
20–22	△△△	Initial value Z6$(3,1)
17–19	△△△	Initial value Z6$(2,2)
14–16	△△△	Initial value Z6$(2,1)
11–13	△△△	Initial value Z6$(1,2)

6. *Example:* DIM Z6$(3,2)3—*Continued*

Bytes	Value	Comments
8–10	△△△	Initial value Z6$(1,1)
7	2	Maximum column dimension
6	3	Maximum row dimension
5	3	LENGTH
4	05	TYPE
2–3	Z6	NAME
0–1	Pointer	Pointer explained later

Suppose the address of the beginning of the variable table is 1000. This value would be calculated by a Language System program. When the Language System is initialized:

Variable table beginning address = Variable table end address.

A rough diagram of microcomputer memory after entry of the two commands

```
PRINT A
B=2
```

was given in Figure 2. A more detailed diagram of the variable table for these two commands is given in Figure 14.

A function of the pointers is to point to the microcomputer memory address where the next variable definition begins or the last memory is located.

Although the variable table expands backwards through memory, the variable search sequence proceeds in a forward direction. The last variable entered in the variable table is the first variable found, while the first variable entered in the variable table is the last variable found.

A detailed analysis of the variable table for a more complex series of commands will further clarify how the Language System constructs the variable table. The series of commands:

```
 DIM P2$(2,3)4,A4(4)
:FOR I7=1 TO 2
:A4(I7),H6(I7)=100
:P2$(I7,I7)="ABCD"
:NEXT I7 (CR)
```

can be entered as a single string of characters through the console input device. Carriage return (CR) is keyed only after the last character, a 7, is entered. The Language System will process the entered commands in the command mode.

The Language System will first identify P2$(2,3)4. The Language System searches the variable table for P2$() and does not find it. As a result, the P2$() variable information and initialized data are placed at the beginning of the variable table. The Language System next identifies A4(4). The variable table is searched for A4() and it is not found. The A4() descriptive information and the array initialized to zero are added to the variable table. The Language System identifies I7, in the command FOR I7 = 1 TO 2, as a variable. The variable table is searched for I7 and it is not found. I7 descriptor information and value initialized to zero are added to the variable table. The Language System then searches for both A4() and I7 and finds these variables in the variable table; thus no action is taken.

The Language System now searches for H6() in the variable table and does not find it. H6() is now entered into the variable table and a default value of 10 is assigned for the single dimension. The data elements of H6() are also initialized to zeros.

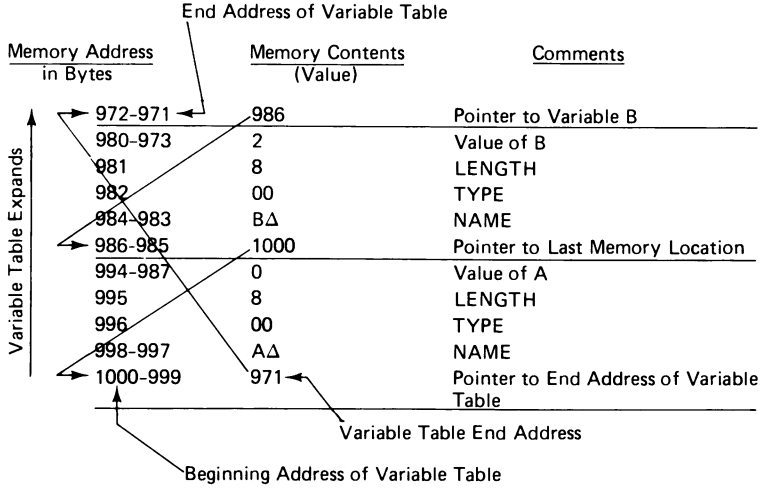


Figure 14: Diagram of the variable table for the two commands PRINT A and B = 2. The address of the beginning of the variable table is 1000 and the end of memory address is 971. 971-972 is the beginning of the storage area if another variable is added to the variable table. The last existing two bytes of memory contain the variable table end address.

If a double dimensioned array element is accessed for an undimensioned array, the default dimension values of 10 × 10 are commonly assigned row × column dimension values.

The Language System then searches for the variables I7, P2$(), and in the last command, NEXT I7. All variables are found in the variable table; thus no action is taken by the system.

Figure 15 shows a diagram of microcomputer memory after the command program has been run.

A six byte overhead of memory storage requirements is associated with either a numeric scalar variable or an alphanumeric scalar character string variable. An eight byte overhead of memory storage requirements is associated with an array variable.

For a numeric scalar or an alphanumeric scalar character string variable, the number of bytes of memory required to contain the variable overhead information and the variable data is given by:

```
6 + LENGTH
```

bytes.

For single dimensioned variables, the memory requirements for storage of both the variable overhead and variable data is given by:

```
8 + LENGTH*DIMENSION
```

where DIMENSION is the value of the single dimension.

For a double dimensioned variable, the memory requirements for storage of both the overhead information and variable data is given by:

```
8 + LENGTH*ROW*COLUMN
```

where ROW is the value of the row dimension, and COLUMN is the value of the column dimension.

The data area length of any variable can be computed from a knowledge of the address value of the pointer to the variable, the pointer value itself, and the overhead, depending on the variable type.

For example, the length of the H6() data area seen in Figure 15 is:

	Comments
916	Value of pointer to H6()
828	Value of memory address of pointer to H6()
88	
8	Overhead for an array
80	Length of data area of H6()

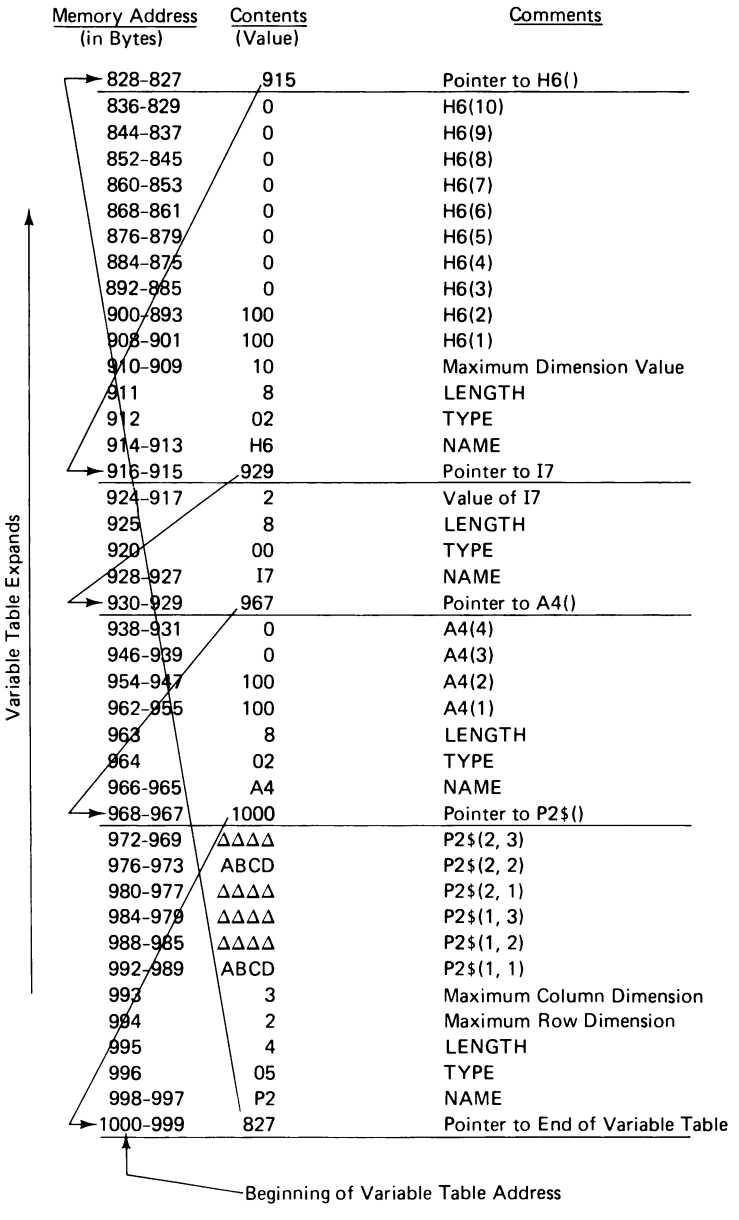


Figure 15: Diagram of the variable table *after execution* of a command sequence. Variable H6() was assigned a dimension default value of 10.

The number of bytes of storage required to contain variable P2$(), which is also seen in Figure 15, can be computed from:

	Comments
1000	Value of pointer to P2$()
968	Value of address of pointer to P2$()
32	
8	Overhead for an array
24	Length of data area of P2$()

This value can be checked by multiplying LENGTH*ROW*COLUMN for P2$() which is 4* 2*3 = 24

The ability of the Language System to calculate the length in bytes of available storage for an array variable is very important.

Newer languages often include the verb:

```
MATREDIM
```

which allows redimensioning of both single and double dimensioned arrays. Execution of the verb causes the Language System to search for the variable in the variable table. The dimension value (or values) are replaced in the variable overhead area by the redimensioned value (or values) provided that

```
LENGTH*NEW_DIMENSION <= Originally dimensioned data space
```

where NEW_DIMENSION is the redimension value for a single dimensioned array, and

```
LENGTH*NEW_ROW*NEW_COLUMN <= Originally dimensioned data
                             space
```

where NEW_ROW and NEW_COLUMN are the redimensioned values for the row and columns.

For example, the variable P2$() seen in Figure 13 could be redimensioned by the statement

```
MATREDIM P2$(2,2)4
```

to a 2 × 2 array since 4*2*2 = 16. Redimensioning P2$() to

```
MATREDIM P2$(1,6)4
```

is also valid since 4*1*6 = 24, but redimensioning P2$() by

```
MATREDIM P2$(4,2)4
```

would be signaled as an error, because 4*4*2 > 24.

A dimension statement can be placed anywhere in a program provided the statement precedes the first array element reference. This restriction is required because referencing an array element would cause an automatic array definition in the variable table and thus a later dimension statement would contradict the previous definition.

SUMMARY

The variable table begins at the bottom of memory and expands toward the top of memory. The highest memory address is at the beginning of the variable table. Both numeric scalar and alphanumeric scalar character string variables are placed in the variable table in the following format:

↑ E. Data
| D. Length
| C. Type
| B. Name of variable
| A. Pointer to the next variable in the variable table

Both alphanumeric character string and numeric array variables are placed in the variable table in the following format:

↑ G. Data
| F. Maximum column dimension } or Maximum dimension for
| E. Maximum row dimension } single dimensioned variable
| D. Length
| C. Type
| B. Name of variable
| A. Pointer to the next variable in the variable table

Two to four bytes of memory are usually allocated for storing the pointer to the next variable in the variable table.

One byte of storage is usually allocated for storing the type attribute.

The length is usually allocated one byte of storage. For the purposes of this book the values of the numeric variables have been assigned a

length of 8 bytes. If no dimension is specified, the length of the values of the alphanumeric variables have a default value of 16 bytes. Storage for variable dimension information is two bytes.

Variable overhead information is used in calculating the length of the data area of a variable for storage. A six byte overhead is associated with either a numeric scalar variable or an alphanumeric scalar character string variable. An eight byte overhead is associated with an array variable.

The verb MATREDIM allows the redimensioning of both single and double dimensioned arrays. The dimensioned value or values are replaced by the redimensioned values or values provided

LENGTH*NEW_DIMENSION <= ORIGINALLY DIMENSIONED DATA SPACE.

4

Common Variables

Common variables, and their placement in the variable table, are presented in Chapter 4. The variable table includes both common and non-common variables. Techniques can be used to make common variables, non-common, and to make non-common variables, common. Program resolution removes all non-common variables from the variable table, but leaves the common variables in the table.

The variable table memory region is defined by two numbers:

1. The address of the beginning of the variable table. This number is the greatest existing memory address.
2. The address of the end of the variable table. This number is the address of the pointer to the last variable stored in the variable table area.

Large applications software systems usually consist of a number of small software modules. During system runs, these modules are overlaid in the program test area of memory.

One method of passing information between system modules is to leave data in a common variable subregion of the variable table. When software module overlay occurs, the variable table is cleared of all non-common variables; but common variables are left in the variable table.

The technique used to implement common variables is to subdivide the variable table into non-common and common variable regions. The common variable region occupies a contiguous block of memory at the beginning of the variable table. Figure 16 displays a rough diagram of the variable table with the common variable feature included.

A third address referencing variable table boundaries is needed to

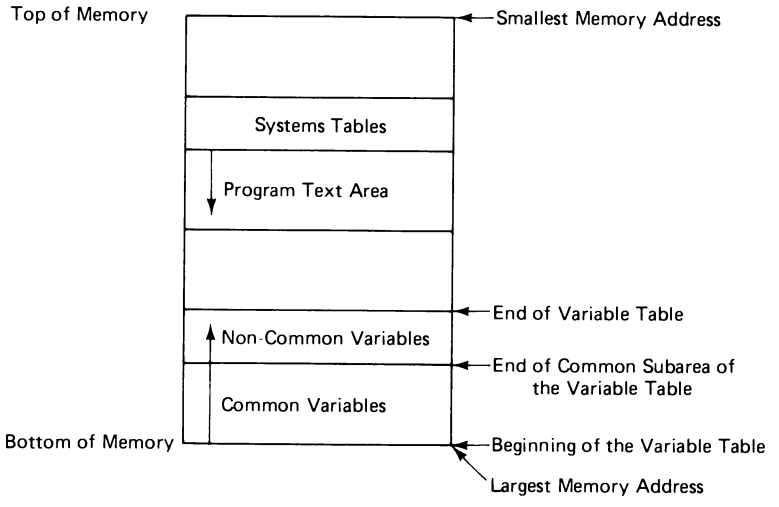


Figure 16: A rough diagram of microcomputer memory showing inclusion of a common variable subarea of the variable table. A third address, in addition to the beginning and end of the variable table, is required to define the common area.

give the address of the pointer to the last variable entered in the common variable subregion of the variable table. This address is kept in the systems tables.

Declaration of common variables is made with a COM statement. A detailed variable table diagram for the statements

```
10 COM A0,P9$5,C2(4),F7$(2,2)4
20 DIM R0,T0,Q2$(2,2)3
```

after this program is run is seen in Figure 17.

The address of the pointer to the last variable entered in the common area is 911.

If the program text area of memory were either cleared or partially cleared by a software module overlay, the end address of the variable table would be set equal to the end of common subarea address. This is the method used to clear the non-common variables from memory.

A language statement, such as:

```
COMCLEAR
```

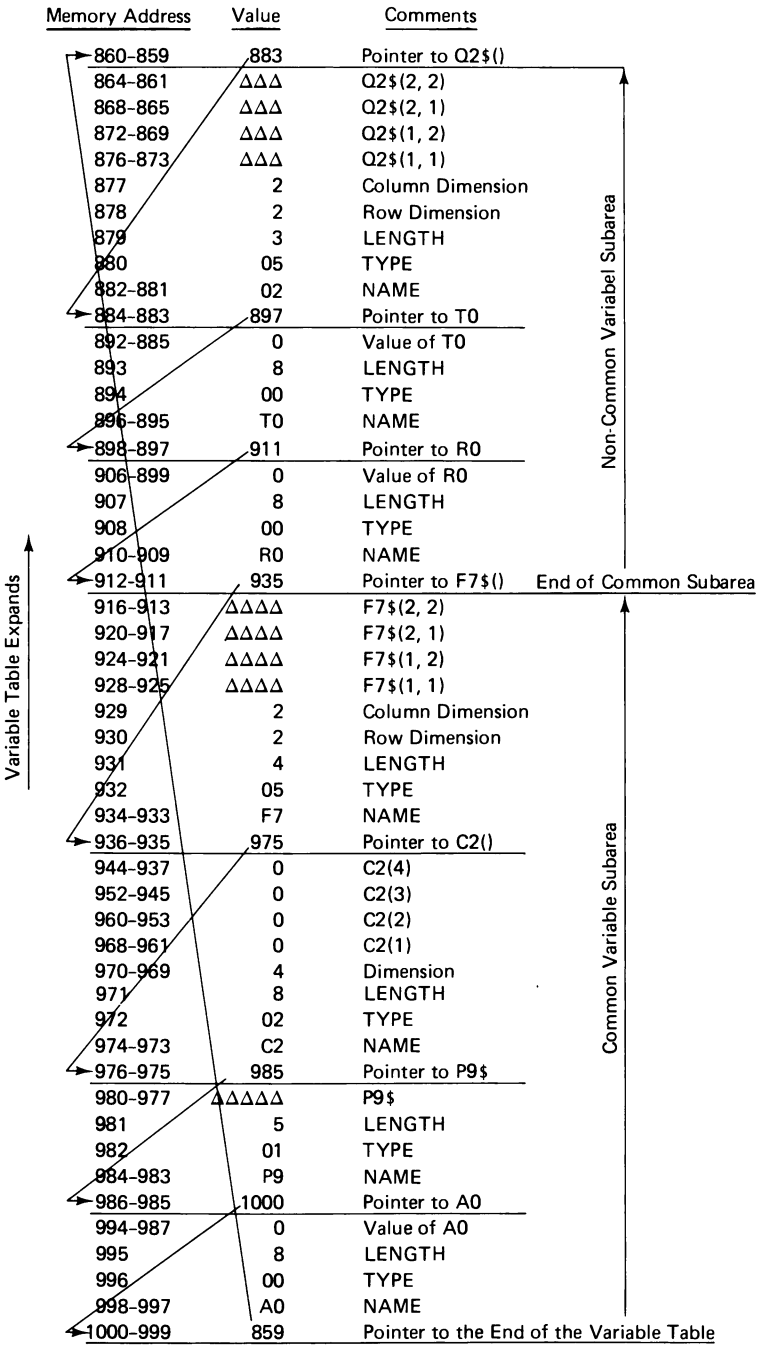


Figure 17: Layout of a variable table with both common and non-common variables included. F7$() is a common variable while R0 is a non-common variable. The end of common subarea address is 911.

should be included in the language system to allow the applications software engineer to move the common subarea address. Non-common variables could then be made common, or common variables could then be made non-common.

For example, the command:

```
COMCLEAR F7$()
```

would change the end of common subarea address to 935 for the variable table layout given in Figure 17. All variables below, and including F7$() (F7$(), R0, T0, Q2$()), would be made non-common variables. On the other hand, the command:

```
COMCLEAR T0
```

would change the common subarea address to 897. Thus R0, F7$(), C2(), P9$(), and A0 would be made common variables. Q2$() and T0 would be non-common variables.

Language System BASICs often allow commands, such as:

```
CLEARN
```

which cause all noncommon variables to be cleared from memory. A statement such as:

```
CLEARV
```

would clear all variables from the variable table, but leave the program text region unaltered.

A statement such as:

```
CLEAR
```

would clear both the variable table and program text regions of memory.

A statement such as:

```
CLEARP 10, 1000
```

would cause all program text between lines 10 to 1000 to be removed from the program text region of memory.

Program resolution does not cause common variables to be cleared from the variable table. Only non-common variables are cleared during program resolution.

SUMMARY

The variable table of memory consists of common and non-common variables. The common variable region occupies a contiguous block of memory at the beginning of the variable table. Non-common variables reside at the end of the common subarea of the variable table.

Declaration of common variables is made with a COM statement.

Non-common variables can be cleared from memory by clearing the program text area of memory, or by using an overlay to partially clear the program text area. Program resolution also causes all non-common variables to be removed from the variable table but leaves the common variables in the variable table. Non-common variables can be made common, or common variables can be made non-common, with the language statement COMCLEAR.

5

Lexical Analysis, Text Atomization, and Syntax Analysis

The specific functions of lexical analysis, text atomization, and syntax analysis are covered in Chapter 5. These functions are Language System programs which operate on both statements and commands in the buffer area. The buffer area is located between the program text and the variable table regions of microcomputer memory. Software rules which apply to all software code blocks, modules, and systems are presented in this chapter. Examples of verb and value stacks are used in this chapter to demonstrate syntax analysis.

For every line of text which is entered into the Language System, the console input device is scanned by software routines, or programs, which do the following:

1. Attempt to identify verbs, variable names, arrays, numeric constants and literals. This is called lexical analysis.
2. Compress commonly used multicharacter verbs into a one or two byte encoding. This is called text atomization.
3. Check commands/statements for correct grammar. This is called syntax analysis.

Most program text is stored in computer memory as seven bit ASCII characters. There are 128 different ASCII characters defined. The definition of these characters is seen in Figure 18.

The high order bit of an ASCII eight bit character is often used as a parity bit redundancy measure of the seven bit character for data communications purposes.

If the high order bit is not used for a parity redundancy measure, then 128 additional symbols can be added to the ASCII character set.

ASCII Code*

High order 4 bits \ Low order 4 bits	0_0	0_1	1_0	1_1	2_0	2_1	2_2	2_3	3_0	3_1	3_2	3_3	4_0	4_1	4_2	4_3
hex digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000 0	NUL 0	SOH 1	STX 2	ETX 3	EOT 4	ENQ 5	ACK 6	BEL 7	BS 8	HT 9	LF 10	VT 11	FF 12	CR 13	SO 14	SI 15
0001 1	DLE 16	DC1 17	DC2 18	DC3 19	DC4 20	NAK 21	SYN 22	ETB 23	CAN 24	EM 25	SUB 26	ESC 27	FS 28	GS 29	RS 30	US 31
0010 2	Space 32	! 33	" 34	# 35	$ 36	% 37	& 38	(apos) ' 39	(40	) 41	* 42	+ 43	(comma) , 44	(dash) - 45	(period) . 46	/ 47
0011 3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
0100 4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
0101 5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	↑ 94	(underline) _ 95
0110 6	grave accent ` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
0111 7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	¦ 124	} 125	~ 126	DEL 127

***Numbers in the lower right corner of each box represent the decimal equivalent of the binary and the hexadecimal code for the character shown in the box, e.g., A = $(41)_{16}$ = $(01000001)_2$ = $(65)_{10}$.**

LEGEND FOR ASCII CONTROL CHARACTERS			
NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell (audible or attention signal)	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation (punched card skip)	EM	End of Medium
		SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tabulation	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator
SO	Shift Out	US	Unit Separator
SI	Shift In	DEL	Delete

Figure 18: ASCII code table showing characters and their definitions.

These additional digits begin with 80 hexadecimal (HEX(80)), and extend through FF hexadecimal (HEX(FF)). These additional one byte symbols can be conveniently used for program text compression and delimiter purposes in Language Systems.

ASCII symbols, associated with hexadecimal digits HEX(00) through HEX(1F), are often used for control characters. Symbols, such as SOH, STX, ETX, ENQ, ACK, DLE, NAK, SYN, ETB, CAN, ESC, and DEL are often used as computer network communications control characters.

A Language System's display characters range from HEX(20), a blank, to HEX(7E), a ~. Not all of these characters may be enterable from a keyboard, but all are usually printable.

The command:

B	$	=	H	E	X	(	2	A	B	F	)

would be processed by the Language System and entered into the microcomputer memory as:

B	$	=	D2	2	A	B	F	)	0D

The HEX(D2) is a Language System text atom representing the ASCII character string HEX(. All commands/statements are terminated by a carriage return (CR), a HEX(0D).

The left parentheses are "atomized out" of many functions. Some examples are: SIN(, COS(, TAN(, STR(, POS(, BIN(, VAL(, NUM(, LEN(. . . Appendix A contains a list of text atoms used for the powerful Wang BASIC-2 Language System.

Line numbers can be delimited by a special symbol. For example the statement:

1	0	5	6	A	6	=	3

would be stored in microcomputer memory as:

FF	10	56	A	6	=	3	0D

The HEX(FF) delimits a line number which is stored in packed binary coded decimal (BCD).

Atomization of commonly used verbs and functions serves several purposes:

1. Memory requirements for storage of program text are reduced.
2. The value of the text atom can be used to facilitate a branch to the subroutine which implements the verb or function associated with the atom.
3. Microcomputer memory can be searched simply for line numbers, verbs, or functions. The reason is that verbs or function atoms lie between HEX(80) and HEX(FE), while line number delimiters must be preceded by HEX(FF). This is a particularly useful design consideration when the Language System is implemented on a microcoded microcomputer.

Display characters in the range HEX(20), a space, to HEX(7E), the ~, can be grouped into several classes:

1. Operators: Some examples are;

Symbol	Definition
+	Addition of numerics
−	Unary minus (example: Y = − A) or numeric
−	subtraction
*	Multiplication of numerics
/	Division of numerics
↑	Exponentiation
&	Concatenation of two character strings
<	Less than—for either strings or numerics
=	Equals—for either strings or numerics
>	Greater than—for either strings or numerics
The operators of <, =, and > can be concatenated to give	
<>	Not equal to—for either strings or numerics
<=	Less than, or equal to—for either strings or numerics
>=	Greater than, or equal to—for either strings or numerics
The left and right parentheses used in algebraic notation act very much like operators in evaluation or numeric expressions.	
(	Left parenthesis
)	Right parenthesis
,	Variable and separators of other symbols
;	Symbol separators

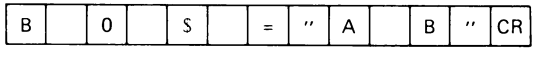
2. Delimiters: Some examples are;

Symbol	Definition
:	Separates statements on the same line
"	Delimits strings of characters
%	Delimits picture formats for PRINTUSING verbs

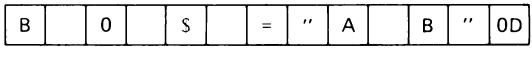
3. Functions, verbs, and data:

Symbols	Definition
$ 0 1 2 3 4 5 6 7 8 9 0 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z	These symbols may be concatenated together to form names of variables, verbs, functions, and data.
a b c d e f g h i j l m n o p q r s t u v w x y z	Lower case alphabetic characters are usually used only for alphanumeric literal string data.

Blanks not enclosed in quotation marks are ignored but not usually removed during lexical analysis. As an example,

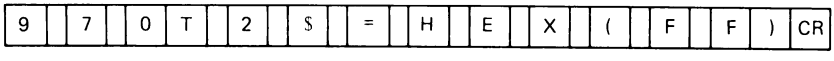


would be stored in microcomputer memory as

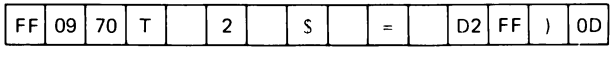


The variable B0$ would be recognized, even though it contained embedded blanks. The blanks would not be removed.

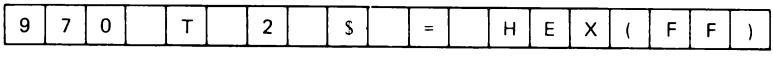
However, the entered statement



would be stored in memory as:



If this statement were recalled from memory for display on the console output device, statement:



would be displayed. The reason is that the statement number would have to be reconstructed from its three byte (FF 09 70) delimiter and packed

binary coded decimal representation, and the HEX(would also have to be inserted in display text for the atom HEX(D2).

In summary, blanks are not normally removed from input statements/commands unless those blanks are embedded within verbs, line numbers, or some function arguments.

The statements/commands entered from the keyboard and displayed on the console output device are first placed in the buffer area of memory.

Each line entered into the Language System is:

1. Atomized if possible during lexical analysis.
2. Subjected to syntax analysis.

The purpose of syntax analysis is to check each input line for any errors detectable from just an analysis of the input character string. An example is:

```
A = B$
    ↑ ERR
```

for the reason that a numeric variable cannot be set equal to a string variable. A second example is:

```
Y = X*/Z
      ↑ ERR
```

for the reason that the syntax of the language does not permit the operator / to immediately follow the * operator.

The time the Language System is given to perform lexical and syntax analysis extends from:

1. the time the Language System user keys carriage return (CR) until
2. the input line is accepted by the Language System and marked as apparently error free, or
3. the Language System returns the character string to the console output device flagging the *first* error found in the line with ↑ ERR.

The lexical and syntax analysis programs work in conjunction with each other. Thus, only the first of several errors on a line will be flagged. As soon as the first error is corrected, then the second error will be flagged. This interactive attribute of the Language System makes it unnecessary that the syntax analyzer catch all errors in a line simultaneously. Thus, the design of the syntax analyzer is more simple than those which must try to catch all errors on a single line or even in an entire program.

A practical question is: How much time can be allowed to elapse between the instant the language system user keys CR and the instant the system responds by either flagging an error or by indicating a successful line entry? Even in the worst case, perhaps a little less than a second delay would be judged acceptable by most Language System users. Please note that the longer the acceptable time, the more inefficient (simpler and more reliable) the lexical and syntax software that can be tolerated.

Designing engineering grade software for Language Systems requires a disciplined approach. Software standards must be established. Rather than enumerating software standards policy rules all at one time, these rules will be given as they are needed.

Three fundamental software rules that apply to all software code blocks, modules, and systems are:

1. Each software code block and module must perform a clearly defined, and simple intended function.
2. Each software code block and module should minimize the possibility of performing any unintended function, whether harmful or not.
3. Each software code block and module must provide adequate warning in event of failure.

Software code blocks which perform Language System functions such as syntax and lexical analysis resemble parts of a machine more than they do computer programs. Data in these code blocks is always kept separate from computer code. Extensive tables in a code block's data area describe what the computer code accomplished during and after its execution. This type of system implementation is often called *table driven* for the reason that tables, instead of a computer code, can be examined to determine what action a code block took.

Appendix A gives several examples of verbs and their respective atomizations. Quite a few of these verbs can be represented by a single byte atom. This number is, of course, less than 128, since we have only hexadecimal 80 to FF available for atomization symbols. To give a couple of examples: HEX (can be atomized by the single byte D2 and HEXOF(by F6. Compound verbs, however, require two byte atomizations. The compound verbs:

```
HEXPRINT
HEXPACK
HEXUNPACK
```

are atomized as follows: HEX by E5, PRINT by A0, PACK by E2, and

UNPACK by E6. Examples of simple and compound verbs are given in Figure 19.

Simple Verb	Hexadecimal Atom
HEX(	D2
HEXOF(	F6
PACK	E2
UNPACK	E6
PRINT	A0
PRINTUSING	A7
HEX	E5
MAT	A8
ARC	CB
SIN(	D0
COS(	D1

Compound Verb	Hexadecimal Atoms
HEXPACK	E5E2
HEXUNPACK	E5E6
HEXPRINT	E5A0
MATPRINT	A8A0
ARCSIN(	CBD0
ARCCOS(	CBD1

Figure 19: Example atomization of simple and compound verbs. Verbs HEX and ARC are only used with other verbs. MAT, which stands for matrix operations, can be used alone; for example, MAT A = B.

Constructing the software code blocks necessary to locate valid verbs in the input buffer requires that all of the possible verbs and their atomizations be stored in a portion of the systems area of memory. Figure 20 presents a diagram of the verb atomization table using the verbs and their atomizations given in Appendix A.

The verb atomization table contains three essential types of information:

1. The verb name.
2. The verb atomization hexadecimal value.
3. The length in characters of the verb name.

To calculate the length in characters of the verb name, begin the search at the start of the verb name to locate the next byte whose value is greater than hexadecimal 7F.

Starting the search in index position 7, the beginning of CLEAR

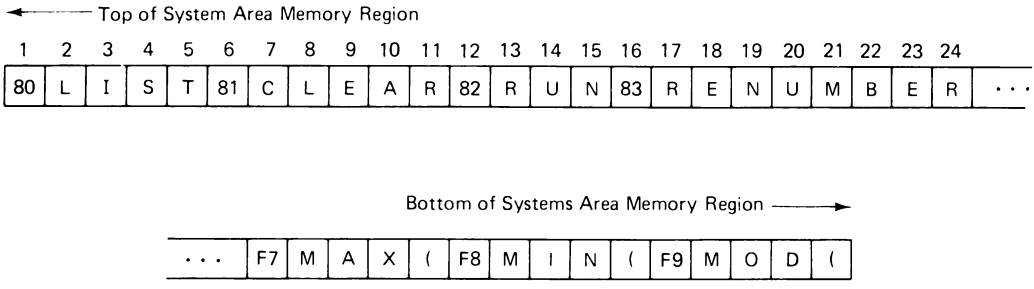
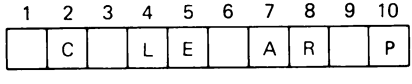


Figure 20: Diagram of the layout of memory for storage of the verb atomization table, using the sample verbs and atoms seen in Appendix A. The verb atomization table resides in a portion of the systems area of memory. All atoms have values in the range hexadecimal 80 to FF. All program command/statement characters have values in the range hexadecimal 20 to 7F.

(seen in Figure 20) for the next byte whose value is greater than 7F gives a final search index value of 12; this points to the hexadecimal 82. The length of CLEAR is 12-7=5 characters.

Valid verbs are identified by comparing increasingly longer contiguous strings of characters taken from the buffer with valid verb names contained in the verb atomization table. All blanks in the buffer are ignored. Several examples should make the verb identification process clear.

If the input buffer contains:



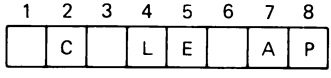
and the scan for a verb begins at index position 2 (which contains "C"), then the following steps must be taken to determine if a valid verb can be found.

1. The table seen in Figure 20 is searched repeatedly for the following character strings:

Character string	Result
a) C	Found
b) CL	Found
c) CLE	Found
d) CLEA	Found
e) CLEAR	Found
f) CLEARP	Not found

2. A search for a byte value greater than 7F is begun at index value 7 in the verb atomization table. Index value 7 was the beginning index where the last "Found" character string was found. A hexadecimal 82 is found at index position 12. Thus the length of the verb is 12-7 = 5 characters.
3. The verb candidate, CLEAR, and the verb, CLEAR, both contain five characters. The conclusion is that the verb candidate is valid and its hexadecimal atom is 81.

If the buffer contained



then the steps for the search for a valid verb would be:

1. The verb atomization table seen in Figure 20 is repeatedly searched for the character strings:

Character string	Result
a) C	Found
b) CL	Found
c) CLE	Found
d) CLEA	Found
e) CLEAP	Not found

2. A search for a byte value greater than hexadecimal 7F is begun at index position 7 of the verb/function atomization table. Index position 7 was the position where the last successfully found string, CLEA, was found. A hexadecimal 82 is found at index position 12. Thus the length of the verb is 12-7 = 5 characters. The length of CLEA is 4 which is not equal to 5. The conclusion is that no valid verb was found.

Appendix A contains nearly 128 verb atoms. Since new verbs will likely be required as additions to any language, a technique must exist to add new verbs to any evolving language.

Hexadecimal FA through FE are labelled "reserved" in Appendix A. Suppose a new verb, SPEAK, is to be added to the language represented in the verb atomization table of Appendix A. A possible solution to the problem of extending the verb atomization table is concatenate a "reserved" hexadecimal FA with a previously used atom to create a new two byte atomization. As an example, the verb atomization table seen in Figure 20 could be modified to read

80	FA	S	P	E	A	K	80	L	I	S	T	81	C	L	E	A	R	...

A single byte atom of a verb was stored one byte before the beginning of the ASCII verb name. If this atom byte is found to be a hexadecimal FA, then the language system would access the memory location two bytes before the beginning of the ASCII verb name to complete the two byte verb atomization. This technique allows for blocks of 128 verbs to be added with each "reserved" character or multiple "reserved" characters.

The intended function of VERB ATOMIZER software code blocks is to search a portion of buffer memory area for a valid verb. If a valid verb or function is found, then the atomization of the verb must be given. The VERB ATOMIZER software must return an index which points to the character in the buffer which caused its scan to be stopped. An outcome status must also be returned.

A data area for VERB ATOMIZER software is seen in Figure 21.

VERB ATOMIZER software is able to access the addresses of both the start and the end of the buffer; both are located in the systems area.

Byte Position	Null Value	Definition and Comments
1-2	HEX(0000)	Binary pointer in the range of 1 to 65535 to where the scan of the buffer is to begin.
3-18	ALL(" ")	Accepted ASCII verb substrings plus the character which stopped the scan. The verb ALL (means that bytes 3-18 are all set to the value within the parentheses which is a blank.
19	HEX(00)	Binary count of accepted ASCII characters.
20-22	HEX(000000)	Atomization of verb.
23	HEX(00)	Binary length of atomization.
24-25	HEX(0000)	Binary pointer in the range of 1 to 65535 to the character in the buffer which stopped the scan.
26	HEX(00)	Status of search; 00 Null 01 Buffer limit overrun 02 Begin scan pointer is zero 03 No verb found FF Valid verb found

Figure 21: Data area for the software code blocks VERB ATOMIZER. VERB ATOMIZER software attempts to locate valid verbs in the buffer.

Suppose the buffer contains the command:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	L		I	S		T	S	D		3	,	2	0	CR

and the verb atomizer software is invoked with the pointer, which marks where the scan is to begin, set at 2; this is also the location of the "L". A table representing VERB ATOMIZER's data area after it has executed is seen in Figure 22.

Byte Position	Value	Comments
1-2	HEX(0002)	Start of scan index
3-18	LISTS	Accepted verb substring ASCII characters plus the character which stopped the scan.
19	HEX(04)	Length of accepted ASCII characters
20-22	HEX(800000)	Atomization of LIST
23	HEX(01)	Length of atomization
24-25	HEX(0008)	Index value of 8 which points to the "S" in the buffer. "S" was the character which stopped the scan.
26	HEX(FF)	Verb was successfully found

Figure 22: Example of VERB ATOMIZER's data area after the command L IS TSD 3,20 was processed by VERB ATOMIZER with the scan beginning with the letter "L".

Atomization of verbs and packing line numbers causes a reduction in the number of characters needed to define a line of text in the buffer. The input line can almost be reconstructed from the atomized text. The exception is reconstruction of embedded blanks within line numbers or verbs. Suppose the statement/command line:

1			9		7		B		2		$		=		"		Z	"	:	L	I	S	T	CR

were entered into the buffer. The lexical analyzer would transform the information in the buffer to read:

FF	01	97		B		2		$		=		"		Z	"	:	80	0D

Some newer computer console input devices have a RECALL key which causes a line in the buffer to be converted from atomized form to ASCII representation. The RECALL key is essential because the lexical

and syntax analyzers only identify one error at a time. If an error is found, then the programmer can depress the RECALL key to bring the line in error back on the screen for editing.

If the RECALL key were depressed for the line above, then:

1	9	7		B		2		$		=		"		Z	"	:	L	I	S	T

would appear on the screen. The RECALL routine would not be able to reinsert the embedded blanks in the line number or in the verb LIST. The RECALL routine searches the verb table for hexadecimal 80 to find the ASCII text string LIST. All atomized text must be reconverted to ASCII since all characters in the line must be subject to editing.

For a summary example, the statements:

```
10 INPUT A: 20 IF A=0 then 10: ELSE PRINT A: GOTO 10
```

would be atomized in the buffer by the lexical analyzer to

FF	00	10	99	A	:	9F	A	=	0	B1	FF	00	10	:	F2	A0	A	9C	FF	00	10	0D

using the atomization table seen in Appendix A.

The lexical analyzer works in *conjunction* with the syntax analyzer. A line in the buffer is not completely analyzed by the lexical analyzer before being subjected to the syntax analyzer. Both the lexical analyzer and the syntax analyzer may find errors. If all lexical analysis were performed prior to syntax analysis, the possibility arises that the lexical analyzer might find an error at the end of a line. When this error is corrected, then the syntax analyzer might find an error at the beginning of the line. This situation is undesirable as all discoverable errors should be detected and corrected as the scan of the buffer proceeds, character by character, from left to right.

SYNTAX ANALYZER

The syntax analyzer performs a "mock execution" of the line. The syntax analyzer references the program text region of memory to check for valid line number references.

The syntax analyzer does not reference the variable table since the variable table is not constructed or augmented until run time program resolution.

The syntax analyzer examines:

1. data types (numeric or string variables)
2. verbs
3. delimiters

to assure that they appear in an order consistent with the definition of the language. If combinations of verbs, data types, and delimiters are found which violate rules of the language, then an error is flagged so that a programmer can correct the fault.

Two arrays, called stacks, are used to aid in syntax analysis:

1. The syntax analyzer's verb stack.
2. The syntax analyzer's value stack.

A rough diagram of the syntax analyzer's value and verb stacks is seen in Figure 23.

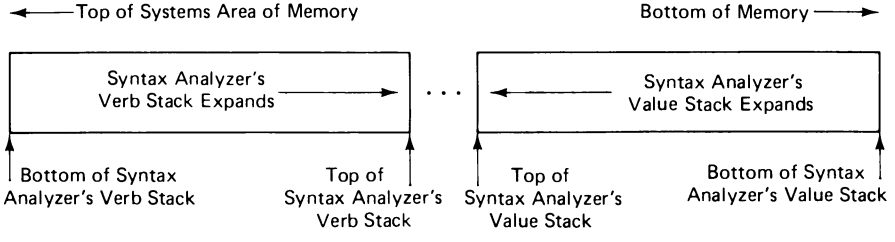


Figure 23: The syntax analyzer's verb stack is located in the systems area of memory. The syntax analyzer's value stack is located above the variable table, but below the program text region of computer memory. The verb stack expands toward the bottom of memory and shrinks toward the top. The value stack expands toward the top of memory and shrinks toward the bottom. One element of the value stack occupies one byte. One element of the verb stack usually occupies two bytes.

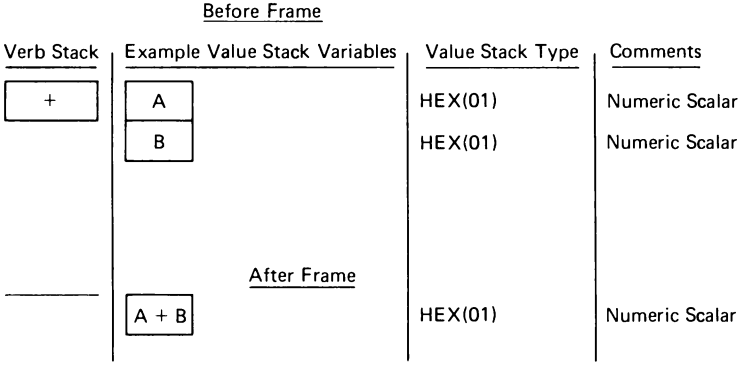
The value and verb stacks expand and contract as the lines are syntactically analyzed. Each element in the value stack need only be one byte in length. Each element in the value stack identifies: a numeric scalar variable, numeric array, an alphanumeric scalar character string, or an alphanumeric character string array. A convenient hexadecimal coding for each of these elements is given in Figure 24.

The elements contained in the verb stack are either verbs or a left parenthesis ("(") delimiter. Many verbs exist, so it is advisable to allow each element on the verb stack to occupy two bytes.

Stack Element Value	Definition
HEX(00)	Null
HEX(01)	Numeric scalar
HEX(02)	Numeric array
HEX(03)	Alphanumeric scalar character string
HEX(04)	Alphanumeric character string array
HEX(05)	True or false logical
HEX(06)	Line number

Figure 24: Example of a hexadecimal coding for the type of element which can appear on the syntax analyzer's value stack. These value types are used by the syntax analyzer to aid it in verifying the correctness of the structure of a statement/command.

Associated with each verb are two value *stack frames:* the *before stack frame* and the *after stack frame*. The stack frame concept is best explained by example. The verb "+" which causes addition of two numbers, has before-after stack frames:



The value stack of the before frame for the verb "+" contains two numeric elements. When the verb "+" is executed, the value stack after frame will contain a single numeric. It might be said that the verb "+" uses two and leaves one when referring to the value stack before and after frames.

The example value stack variables diagram should not be interpreted literally; the diagram is presented to help the reader conceptualize what is taking place. Conceptually, the values of variables A and B may be thought of as being placed on the value stack. The verb "+" causes these

two values to be replaced by their sum. When this is done the "+" is removed from the verb stack.

The importance of only identifying the variable type placed on the value stack is that of being able to catch such errors as:

Buffer command	Error
A+B$ ↑ ERR	A string variable cannot be added to a numeric variable.
B$+A ↑ ERR	Addition of a string variable to a numeric variable is prohibited.

The conceptual diagram of the value stack for the before and after frames is much easier to understand than a diagram containing only value stack variable types. For this reason, conceptual diagrams will be used in many explanations.

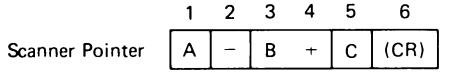
Figure 25 presents a diagram of value stack before and after frames for common arithmetic verbs.

Verb	Value Stack Before Frame	Value Stack After Frame	Definition
+	A B	A + B	Addition
Unary −	A	−A	Unary Minus
−	A B	A − B	Subtraction
*	A B	A * B	Multiplication
/	A B	A/B	Division
↑	A B	A ↑ B	Exponentiation

Figure 25: Conceptual diagram of value stack before and after frames for common mathematical verbs. The variable types for A and B must be numeric scalar variables. The result is always a number. All these verbs except unary minus "use two and leave one." Unary minus "uses one and leaves one."

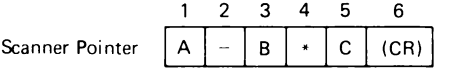
Mock evaluation of arithmetic expressions requires that the syntax analysis scanner look one verb or delimiter ahead. The decision of whether an "after" frame can replace a "before" frame on the value stack depends on the value of the next symbol.

For example, compare syntax analysis steps for the buffer command:

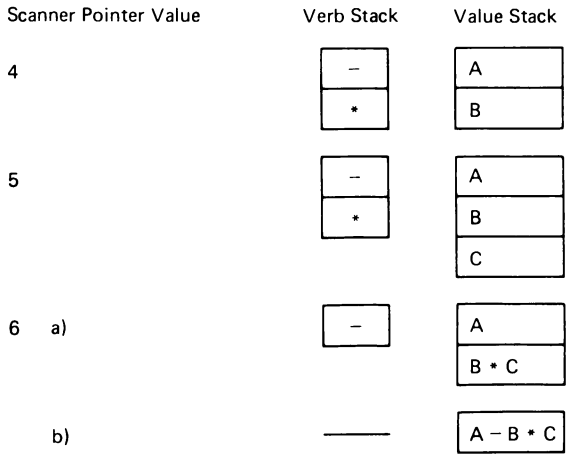


Scanner Pointer	Verb Stack	Value Stack
1	———	A
2	–	A
3	–	A B
4	+	A – B
5	+	A – B C
6	———	A – B + C

where (CR) stands for carriage return.



Scanner Pointer Value	Verb Stack	Value Stack
1	———	A
2	–	A
3	–	A B



In the latter example, the minus sign had to be stacked in the verb stack until the multiply was done.

The syntax analyzer has to make one of two decisions when examining the next symbol:

1) *Create* after frames for the verb and value stacks, or
2) *Stack* the symbol.

This decision is based on the values of:

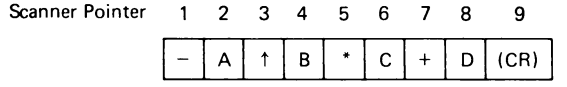
1) The next symbol.
2) The last symbol on the verb stack.

A precedence table for the arithmetic verbs is shown in Figure 26.

Next Verb \ Last Symbol on Verb Stack	Unary –	+	–	*	/	↑
Unary –	Stack	Stack	Stack	Stack	Stack	Stack
+	Create	Create	Create	Create	Create	Create
–	Create	Create	Create	Create	Create	Create
*	Create	Stack	Stack	Create	Create	Create
/	Create	Stack	Stack	Create	Create	Create
↑	Stack	Stack	Stack	Stack	Stack	Stack

Figure 26: A precedence table for arithmetic operators. "Create" signifies creating an after frame from a before frame on the value and verb stacks when encountering the "Next Verb" given the "Last Symbol on Verb Stack". "Stack" means that the "Next Symbol" must be placed on the verb stack and the syntax analysis scan resumed.

The arithmetic expression



is conceptually analyzed in the following steps:

Scanner Pointer Value	Verb Stack	Value Stack	Comments
1	–u		–u is Unary Minus
2	–u	A	
3	–u ↑	A	
4 a)	–u ↑	A B	
b)	–u	A ↑ B	
c)		–A ↑ B	
5	*	–A ↑ B	
6 a)	*	–A ↑ B C	
b)		–A ↑ B * C	
7	+	–A ↑ B * C	
8 a)	+	–A ↑ B * C D	
b)		–A ↑ B * C + D	

No "Last Symbol on Verb Stack" exists for the scanner pointer value of 1, so the unary minus, denoted –u, is placed on the verb stack.

The syntax analyzer determines that the next symbol in the buffer is a variable, so it is placed on the value stack. The scanner pointer is set to 3. The next verb is ↑. Column 1 of Figure 26 shows that when the last verb was a unary minus and the next verb is a ↑, the unary minus cannot be immediately executed and must be stacked. The ↑ is placed on the verb stack. The syntax analyzer then identifies the variable, B, and places it on the syntax value stack. When the syntax analyzer identifies the *, the table in Figure 26 is consulted. The last symbol in the verb stack is ↑, the next verb is *, so the rule "create" that produces after stack frames from before stack frames is followed. This creation continues as long as the verb-stack can be processed. Examples of this successive reduction of the verb-stack are seen in steps 4 a, b, and c; 6 a and b; and 8 a and b. No examples of verbs which stop creation of after stack frames from before stack frames have been given yet.

The * is placed on the verb stack. C is identified as a variable and placed on the syntax value stack. + is then identified as the next verb. The last symbol on the verb-stack was a *. The precedence table in Figure 26 is consulted. The rule is "create", so the steps seen in 6 a and b are performed. The + is placed on the verb stack. The syntax analyzer identifies the D and places it on the syntax analyzer's value stack. The last symbol encountered is a carriage return (CR), a hexadecimal 08. CR is a verb and triggers a "create". The results are seen in steps 8 a and b.

Examination of Figure 26 reveals that such expressions as: ---A, A + − B, A/---A, . . . are permitted. No theoretical problems follow unrestricted use of unary minus. Most syntax analyzers, however, permit use of unary minus only in such expressions as A = − B, A = B ↑ − C, or A = B*(− C).

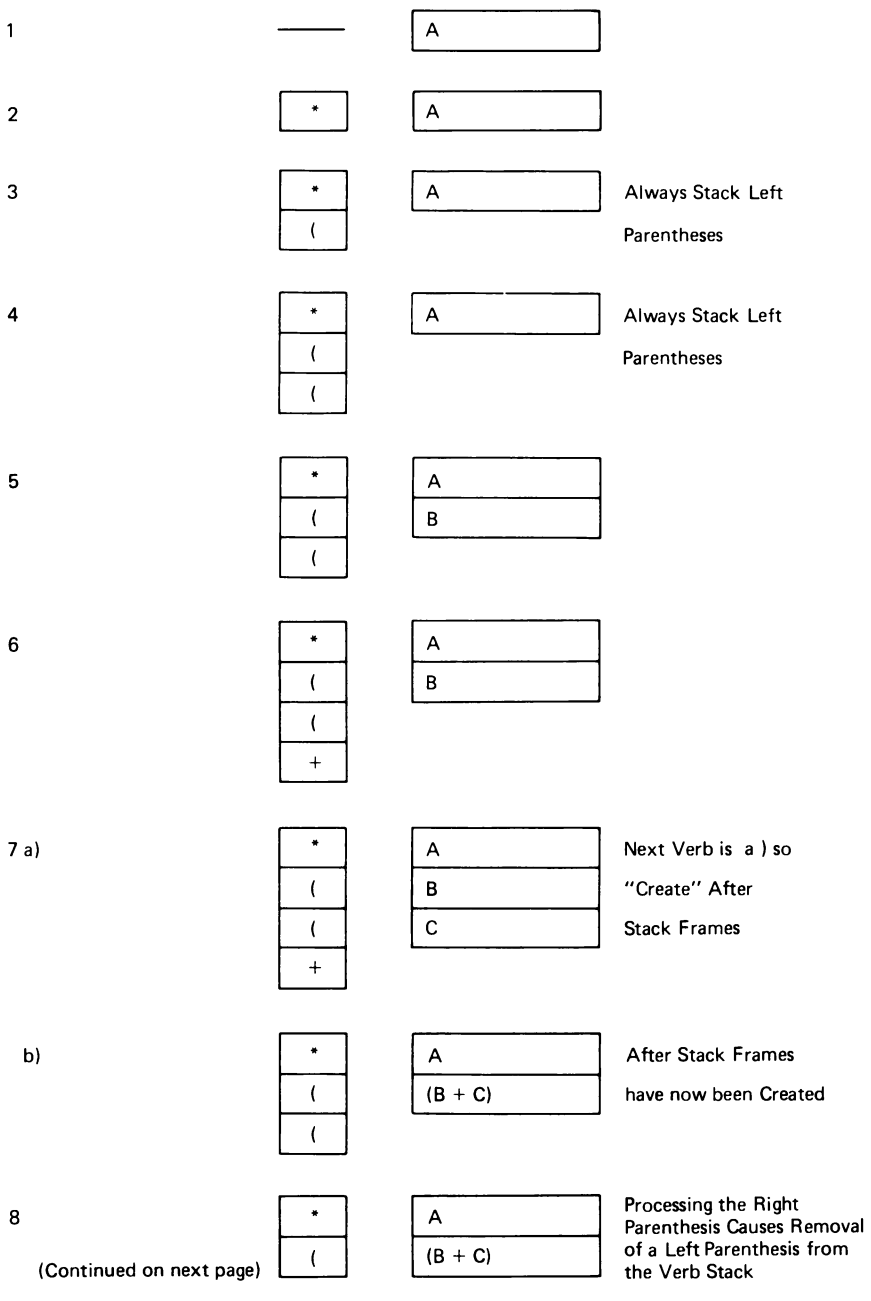
This rough conceptual explanation of the verb and syntax value stacks were designed to give the reader an introduction to the "two stack" method of checking for the correctness of the grammar of BASIC. Explanation of how parentheses are processed will be given by this same rough conceptual method.

Left parentheses are processed in a manner similar to arithmetic verbs. Right parentheses are processed similar to a carriage return (CR).

The expression:

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A	*	(	(	B	+	C	)	/	D	−	E	)	(CR)

is processed by the following steps



(Continued on next page)

Scanner Pointer Value	Verb Stack	Value Stack	Comments
9	* (/	A (B + C)	
10 a)	* (/	A (B + C) D	The Next Verb is a – so a "Create" is Performed. See / Followed by – Precedence in Figure 26
b)	* (	A (B + C)/D	
11	* (–	A (B + C)/D	
12 a)	* (–	A (B + C)/D E	The Next Verb is a) and Causes a "Create" to be Performed
b)	* (	A (B + C)/D – E	
13 a)	*	A (B + C)/D – E	Processing the) Causes a (to be Removed from the Verb Stack. The Next Verb is a (CR) so a "Create" is Performed
b)		A* ((B + C)/D – E)	

Left parentheses, as seen in scanner pointer steps 8 and 13, serve as verb stack markers. These markers bind the number of verbs which can be processed when a right parenthesis is found in the buffer.

The equal sign (=) is also a verb. The equal sign has two before-after stack frame configurations:

Example Assignment	Verb Stack Before	Verb Stack After	Value Stack Before	Value Stack After
CONFIGURATION 1				
A = B	=	–	A B	–

Example Assignment	Verb Stack Before	Verb Stack After	Value Stack Before	Value Stack After
CONFIGURATION 2				
A, B = C	=	=	A B C	A C

In Configuration 1, the value stack only contains two elements. The equal sign is removed from the verb stack, and the two elements are removed from the value stack.

In Configuration 2 the value stack contains more than two elements. The equal sign is left on the verb stack, and the top two elements of the value stack are replaced by the top element of the value stack. This feature allows using multiple variable names on the left hand side of an assignment statement.

As an example of = sign processing the command

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12
	Y	=	–	A	↑	(	–	B	–	C	)	(CR)

Scanner Pointer	Verb Stack	Value Stack	Comments
1	—	Y	
2	=	Y	
3	= –u	Y	–u is the Unary Minus Symbol
4	= –u	Y A	
5	= –u ↑	Y A	
6	= –u ↑ (	Y A	

Scanner Pointer	Verb Stack	Value Stack	Comments
7	= −u ↑ (−u	Y A	
8 a)	= −u ↑ (−u	Y A B	The Next Verb is − so the Rule is "Create"
b)	= −u ↑ (	Y A −B	The After Stack Frames have been Created
9	= −u ↑ (−	Y A −B	
10 a)	= −u ↑ −	Y A −B C	The Next Verb is a) so the Rule is "Create" After Stack Frames from before Stack Frames
b)	= −u ↑ (	Y A (−B −C)	
11 a)	= −u ↑	Y A (−B −C)	Processing a Right Parenthesis Causes Left Parenthesis to be Removed from the Verb Stack. The Next Verb is (CR) so the Rule is "Create"

Scanner Pointer	Verb Stack	Value Stack	Comments
11 b)	=	Y	
	−u	A ↑ (−B −C)	
c)	=	Y	
		−A ↑ (−B −C)	
d)	—	—	Both the Verb and Value Stacks are Cleared when an = Sign with only Two Elements on the Value Stack is Processed

Syntax analyzers often operate on the principle of expectation or nonexpectation. The syntax analyzer either expects or does not expect to see one of several types of characters at each scan pointer value. If the syntax analyzer's expectations are not fulfilled, then the syntax analyzer will signal an error by pointing to the character in the buffer which caused the scan to stop. Sent to the console output device are: the line of text in the buffer, a pointer (↑) pointing to the character on which the scan stopped, and an error message.

An example of the expectation-nonexpectation principle can be given that will show what happens when the scanner pointer value is 3. The previous verb, at scanner pointer value 2, was an =. The syntax analyzer expects a(n):

a) alphabetic character
b) left parenthesis
c) unary minus
d) digit
e) quote (for enclosing a character string)

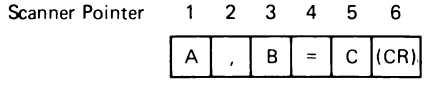
but does not expect a:

a) +
b) *
c) /
d) ↑
e) '
f) !
g) %
h) &
•
•

The syntax analyzer is usually written so as to check the "expectation" or "nonexpectation" list containing the fewest number of alternatives. In this example, the "expectation" list would be selected.

An expanded precedence table is shown in Figure 27. The verbs, carriage return (CR) and), are never placed on the verb stack. These verbs only cause a "create" producing after stack frames from before stack frames. Examples of this "create" action of these verbs are seen in scanner pointer steps 8, 10, and 11.

The command



has syntax analysis steps

Scanner Pointer Value	Verb Stack	Value Stack	Comments
1	—	A	
2	—	A	is a Delimiter Used to Separate Symbols
3	—	A B	
4	=	A B	
5	=	A B C	
6 a)	=	A C	Configuration 2 =
b)	—	—	Configuration 1 =. Both "Creates" were Initiated by the Next Verb (CR)

The importance of this example is that it shows the = sign is not removed from the verb stack until only two elements remain on the value stack. The multiple configuration before-after stack frame conventions for the = sign allows multiple assignments of a variable.

Syntax analysis is performed on a command/statement only basis;

Next Verb in Buffer \ Last Verb on Stack	Unary -	+	−	*	/	↑	(	=
Unary -	Stack	Stack	Stack	Stack	Stack	Stack	Stack	Stack
+	Create	Create	Create	Create	Create	Create	Stack	Stack
*	Create	Stack	Stack	Create	Create	Create	Stack	Stack
/	Create	Stack	Stack	Create	Create	Create	Stack	Stack
↑	Stack	Stack	Stack	Stack	Stack	Stack	Stack	Stack
(	Stack	Stack	Stack	Stack	Stack	Stack	Stack	Stack
)	Create	Create	Create	Create	Create	Create	Create	ERR
(CR)	Create	Create	Create	Create	Create	Create	ERR	Create

Figure 27: Precedence table for arithmetic verbs, (,), =, and (CR). (CR) and) are never stacked. Expressions such as A = (B (CR) or beginning with A = B) are flagged in error (ERR). Variables or constants are sometimes required to be between verbs (for example A + B − C), but sometimes these verbs can be adjacent (for example A = (B) or A = − B).

it does not bridge commands/statements. The command program

FOR A = 1 TO 10: PRINT A

is syntax correct even though the program contains an error of a missing NEXT A.

The rough conceptual method will continue to be used to explain how the lexical and syntax analyzers work to analyze a more complicated command.

The lexical and syntax analysis steps required for analysis of the command:

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	F	O	R	A	0	=	2	*	C	O	S	(	B	)	T	O	C	↑	−	D	S	T	E	P	3	*	E	(CR)

are

Scanner Pointer Value	Verb Stack	Value Stack	Comments
1	———	———	Not a Statement Because "F" is Not a Digit.
2	———	———	Not a Variable Name Because "O" is Not a Digit
3	———	———	FOR is Found in the Verb Table. FORA is Not Found in the Verb Table. FOR is a Length 3 and Thus is a Valid Verb. Hexadecimal 9E is the Atomization for FOR Given in Appendix A.

A code block in the lexical analyzer now atomizes FOR in the buffer to

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	9E	A	0	=	2	*	C	O	S	(	B	)	T	O	C	↑	–	D	S	T	E	P	3	*	E	(CR)		

The command in the buffer is underoing a compression resulting from text atomization. The contents of the entire buffer should be left shifted the appropriate amount during atomization and filled out with blanks.

A verb was located and atomized. The command was compressed in the buffer. With the scanner pointer reset to 1 the syntax analyzer is called. The syntax analyzer can identify the text atom of FOR, a hexadecimal 9E. The syntax analyzer will be looking for stack frames in the format required by the FOR-TO-STEP command/statement. This form is:

Verb Stack	Value Stack
FOR	Number of Start Index
TO	End of Index Number
STEP	Step Increment/Decrement Number

The analysis resumes

Scanner Pointer Value	Verb Stack	Value Stack	Comments
1	FOR	———	

To determine what the next sequence of symbols represents the lexical analyzer is called with the scanner pointer set to 2. These meaningful sequences of symbols, such as verbs, variables, constants, literals, . . ., are often called *tokens*.

Scanner Pointer Value	Verb Stack	Value Stack	Comments
2	FOR	———	May be a Variable
3	FOR	A0	0 is Found so A0 is a Variable

The lexical analyzer reports that it has found a variable. The syntax analyzer ensures that the variable is placed on the value stack. The scanner pointer is advanced to 4. The "=" verb is identified, and then placed on

the verb stack.

Scanner Pointer Value	Verb Stack	Value Stack	Comments
4	FOR =	A0	

The syntax analyzer expected the "=" so it passes approval by advancing the scanner pointer. The syntax analyzer now expects to encounter a number, a variable, or an expression that will eventually produce a number.

Scanner Pointer Value	Verb Stack	Value Stack	Comments
5	FOR =	A0 2	Next Symbol "*" Stops Scan for Number
6	FOR = *	A0 2	
7-10	FOR = *	A0 2	The Lexical Analyzer Identifies COS(as a Valid Verb

COS(is atomized to a hexadecimal C3 which is found in Appendix A. The buffer becomes

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	9E	A	0	=	2	*	C3	B	)	T	O	C	↑	-	D	S	T	E	P	3	*	E	(CR)

The scanner pointer is reset to 7 and syntax and lexical analysis resumes.

Scanner Pointer Value	Verb Stack	Value Stack	Comments
7	FOR = * COS(	A0 2	

Scanner Pointer Value	Verb Stack	Value Stack	Comments
8 a)	FOR = * COS(	A0 2 B	The Variable B is Recognized and Placed on the Value Stack. The Next Symbol ")" Causes a "Create" to Occur.
b)	FOR = *	A0 2 COS(B)	

When the scanner pointer is 9, the lexical analyzer must know the next meaningful sequence of symbols (token).

The "TO" is recognized as a verb with hexadecimal atom B2. The buffer is compressed to

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	9E	A	0	=	2	*	C3	B	)	B2	C	↑	–	D	S	T	E	P	3	*	E	(CR)

When the next symbol is a "TO", the "create" produces an after stack frame from a before stack frame. This is a suboperation within a FOR-TO-STEP.

Before Verb Stack	Before Value Stack	After Verb Stack	After Value Stack
FOR =	Number 1 Number 2	FOR	Number 2

The purpose of this "create" is to place the start of the loop index value on the value stack, while eliminating the index name and "=" verb from the verb stack. The "=" verb in the FOR-TO-STEP context is handled in an entirely different manner than was the "=" in an assignment. The "=" in a FOR-TO-STEP can always be distinguished from an "=" in an assignment (example of an assignment: A=B) because a FOR precedes it on the verb stack. The "TO" triggers the following steps since it is the next token when the scanner pointer is 9.

Scanner Pointer Value	Verb Stack	Value Stack	Comments
9 a)	FOR =	A0 2*COS(B)	
b)	FOR	2*COS(B)	The FOR = is Reframed. This was Caused by Encountering the "TO" as the Next Token

The syntax analyzer resumes its scan at pointer value 10. The syntax analyzer examines the verb stack, and sees the last verb as "FOR". This is the condition it expects. The syntax analyzer examines the value stack and sees one number on it. The syntax analyzer also expected to see this condition.

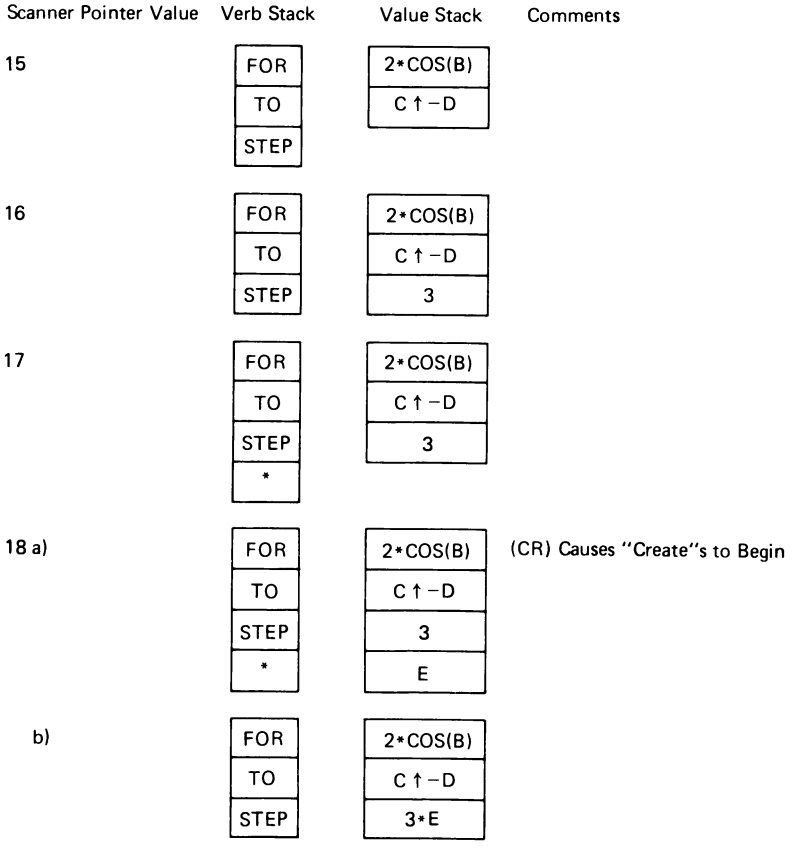
Scanner Pointer Value	Verb Stack	Value Stack	Comments
10	FOR TO	2*COS(B)	
11	FOR TO	2*COS(B) C	Next Verb is ↑ so the Rule is "Stack"
12	FOR TO ↑	2*COS(B) C	
13	FOR TO ↑ −u	2*COS(B) C	The Symbol for Unary − is −u and it is Stacked
14 a)	FOR TO ↑ −u	2*COS(B) C D	
b)	FOR TO ↑	2*COS(B) C −D	The Next Token, the Verb STEP, Causes "Create"s to Occur.
c)	FOR TO	2*COS(B) C↑−D	

"STEP" is recognized as a verb with the text atom hexadecimal B0. The buffer is compressed to

Scanner Pointer

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
9E	A	0	=	2	*	C3	B	)	B2	C	↑	−	D	B0	3	*	E	(CR)

The syntax analyzer sees that the previous verb on the stack is "TO" and also that there are two numbers on the value stack. Thus "STEP" is a legitimate verb. The scanner pointer value is reset to 15 and syntax analysis resumed.

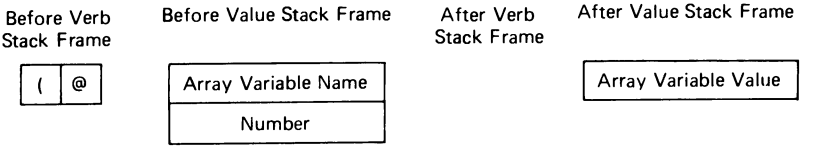


The syntax analyzer checks the value stack to make sure that only three elements are on it. Since this is the case, all the expectations the analyzer had for a FOR-TO-STEP compound verb have been fulfilled. In addition, all of the expectations it had pertaining to the arithmetic expressions within the FOR-TO-STEP were also satisfied.

The value stack frames used for syntax analysis are simpler than those required for program execution. More information has to be placed on the value stack for the FOR-TO-STEP stack frame for program execution than the three numbers seen in the above example.

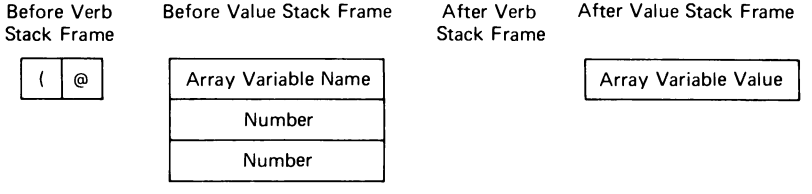
Single dimensioned arrays have the verb and the value before and

after stack frames.



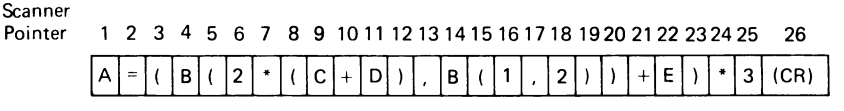
where [(| @] is a symbol placed on the verb stack showing that an array was encountered. The two byte symbol [(| @] could be called the verb "Array". This verb forces the value of an array element to be placed on the value stack when a terminating ")" is found.

Double dimensioned arrays have the verb and the value before and after stack frames.



The "array" verb is generated by the syntax analyzer and placed on the verb stack when an array reference is discovered.

The lexical and syntax analysis steps for the command

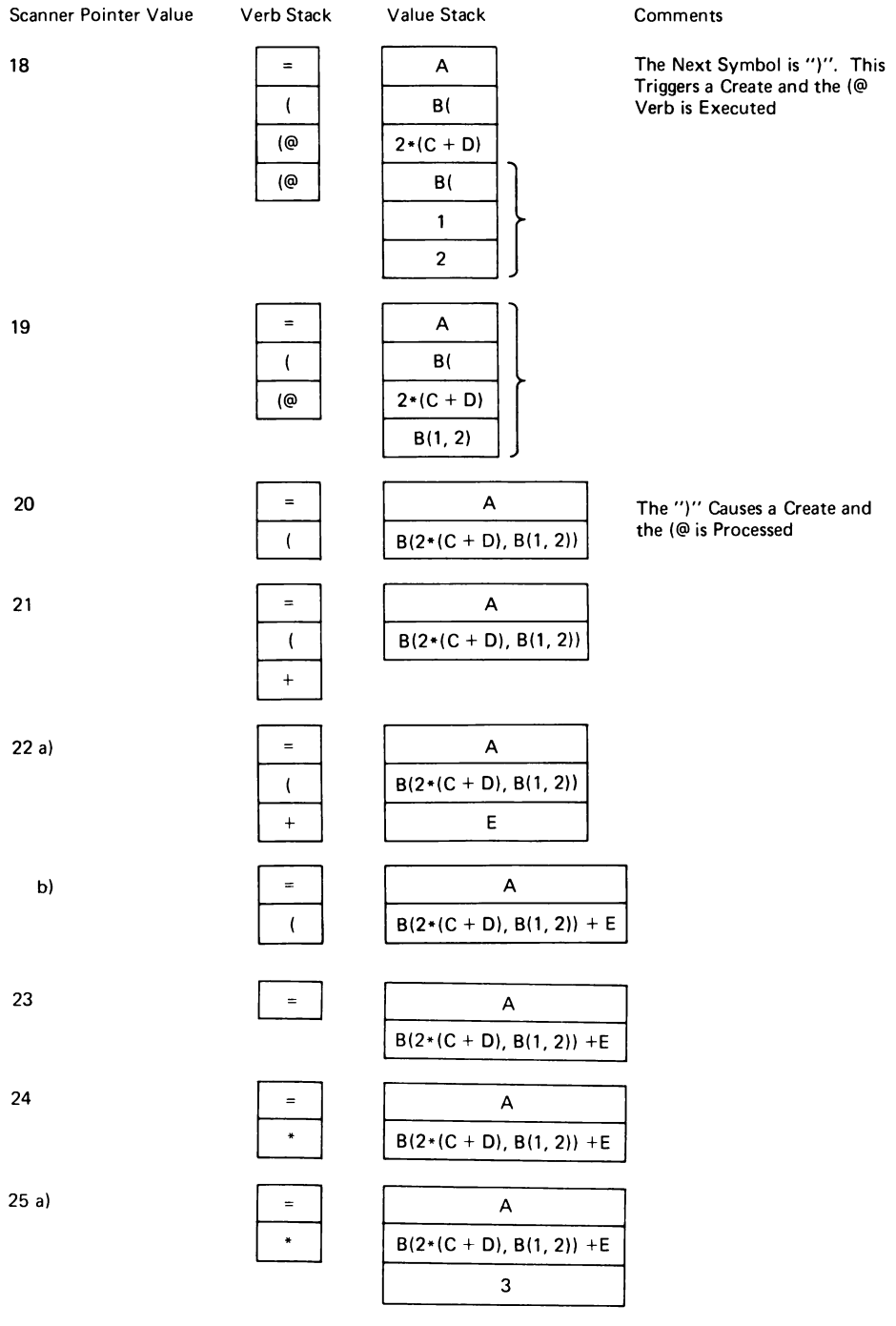


are

Scanner Pointer Value	Verb Stack	Value Stack	Comments
1		A	
2	=	A	
3	= (	A	
4-5	= ((@	A B(	The Array Variable Name Token B(is Recongnized and Placed on the Value Stack. (@ is Placed on the Verb Stack

Scanner Pointer Value	Verb Stack	Value Stack	Comments
6	= ((@	A B(2	
7	= ((@ *	A B(2	
8	= ((@ * (	A B(2	
9	= ((@ * (	A B(2 C	
10	= ((@ * (+	A B(2 C	
11 a)	= ((@ * (+	A B(2 C D	The Next Symbol is ")" which Causes a "Create" Sequence to Commence

Scanner Pointer Value	Verb Stack	Value Stack	Comments
11 b)	=	A	
	(	B(	
	(@	2	
	*	C + D	
	(		
12	=	A	Processing ")" Causes "(" to be Removed from the Verb Stack
	(	B(	
	(@	2	
	*	C + D	
13	=	A	The Delimiter "." Acts in a Manner Similar to (CR) in "Create"s to Begin. The "(" on the Verb Stack Stops the Sequence of "Create"s
	(	B(	
	(@	2*(C + D)	
14-15	=	A	The Array Variable Name Token B(is Recognized and Placed on the Value Stack
	(	B(	
	(@	2*(C + D)	
	(@	B(	
16	=	A	
	(	B(	
	(@	2*(C + D)	
	(@	B(	
		1	
17	=	A	"." is a Delimiter Separating the Two Index Numbers of B(
	(	B(	
	(@	2*(C + D)	
	(@	B(	
		1	


Scanner Pointer Value
Verb Stack
Value Stack
Comments
18
=
(
(@
(@
A
B(
2*(C + D)
B(
1
2
The Next Symbol is ")". This Triggers a Create and the (@ Verb is Executed
19
=
(
(@
A
B(
2*(C + D)
B(1, 2)
20
=
(
A
B(2*(C + D), B(1, 2))
The ")" Causes a Create and the (@ is Processed
21
=
(
+
A
B(2*(C + D), B(1, 2))
22 a)
=
(
+
A
B(2*(C + D), B(1, 2))
E
b)
=
(
A
B(2*(C + D), B(1, 2)) + E
23
=
A
B(2*(C + D), B(1, 2)) +E
24
=
*
A
B(2*(C + D), B(1, 2)) +E
25 a)
=
*
A
B(2*(C + D), B(1, 2)) +E
3

Scanner Pointer Value	Verb Stack	Value Stack	Comments
b)	=	A (B(2*(C + D), B(1, 2)) +E)*3	
c)	———	———	

Notice that in this example an array reference causes the syntax analyzer to generate an "Array" verb, (@, and places this verb on the verb stack.

The syntax analyzer is unable to catch array dimensioning errors such as:

$$A = B(1)*B(2,3)$$

where B( ) is illegally single and double dimensioned.

An example of a statement which involves true/false logic is:

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	1	0	I	F	A	=	B	O	R	C	<	D	T	H	E	N	1	2	(CR)

has lexical and syntax analysis steps:

Scanner Pointer Value	Comments
1-2	A Line Number is Identified by the Lexical Analyzer. The Line Number is Converted to Packed and Marked by the Hexadecimal Atom FF

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	FF	00	10	I	F	A	=	B	O	R	C	<	D	T	H	E	N	1	2	(CR)

The scanner pointer is reset to 4, and the lexical analyzer identifies the verb "IF". The hexadecimal atom for "IF" is 9F as shown in Appendix A. The buffer line is compressed to:

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	FF	00	10	9F	A	=	B	O	R	C	<	D	T	H	E	N	1	2	(CR)

Scanner Pointer Value	Verb Stack	Value Stack	Comments
4	IF	———	
5	IF	A	
6	IF =	A	
7 a)	IF =	A B	
b)	IF	A = B	The Token "OR" is Treated as a Delimiter which Causes a "Create" for the Verb Sequence IF =. The Element A = B has Values TRUE or FALSE

The before and after stack frames for comparisons are:

Definition	Verb Before Stack Frame	Verb After Stack Frame	Value Before Stack Frame	Value After Stack Frame
	IF	IF	Numeric or Character String	Logical
Equal	=			
	or			
Less Than or Equal	<=		Numeric or Character String	
	or			
Greater than or Equal	<=			
	or			
Greater Than	>			
	or			
Less Than	<			
	or			
Not Equal	<>			

The top two elements in the before value stack frame have to be of the same types. A = B or A$ = B$ are valid statements, but the syntax analyzer would catch the invalid comparison:

```
IF A = B$ THEN 10
     ↑ ERR
```

The "IF" verb on the verb stack can be followed by any of the comparisons. Any of the comparisons can also be preceded by the logical verbs or AND, OR, NOT, and XOR (for exclusive OR), provided these verbs are preceded by an "IF".

The lexical analyzer identifies "OR" as a valid verb with hexadecimal text atom 8B (see Appendix A). The buffer is compressed to

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	FF	00	10	9F	A	=	B	8B	C	<	D	T	H	E	N	1	2	(CR)

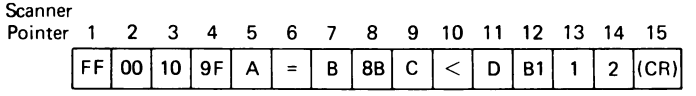
and the scanner pointer value is set to 8.

Scanner Pointer Value	Verb Stack	Value Stack	Comments
8	IF OR	A = B	
9	IF OR	A = B C	
10	IF OR <	A = B C	
11 a)	IF OR <	A = B C D	

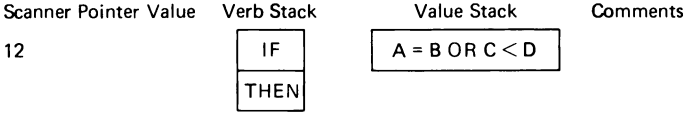
The lexical analyzer is called to determine what the next token in the buffer is. The lexical analyzer discovers the "THEN" which in turn triggers a series of "create"s back to the "IF" verb.

Scanner Pointer Value	Verb Stack	Value Stack	Comments
11 b)	IF OR	A = B C < D	The Top Two Elements on the Value Stack are of Logical Type
c)	IF	A = B OR C < D	The Element on the Value Stack is of Logical Type

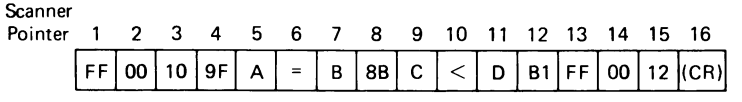
The "THEN" has hexadecimal text atom B1 so the buffer is compressed to



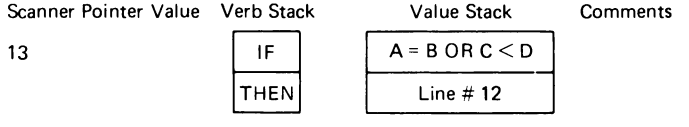
and the scanner pointer is set to 12.



The lexical analyzer is called to identify the next token. A line number, which is expected by the syntax analyzer, is found. The line number is packed and the buffer changed to read

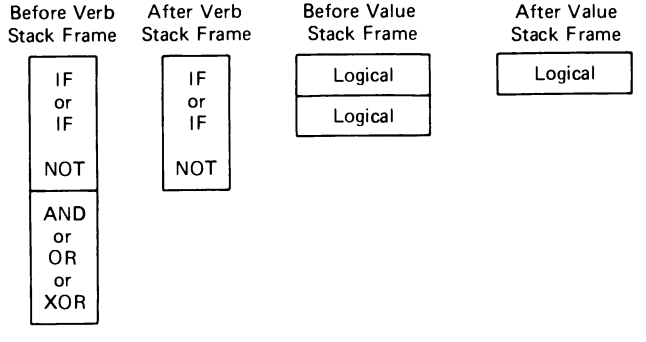


where the hexadecimal FF indicates the beginning of a line number.

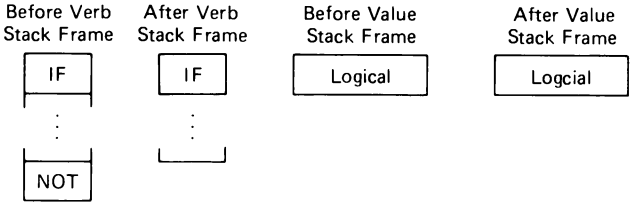


The syntax analyzer has been satisfied; the verb and value stack frames are in a legitimate format.

Logical verbs "AND", "OR", and "XOR", when preceded by the verb "IF" on the verb stack, have the before and after stack frames



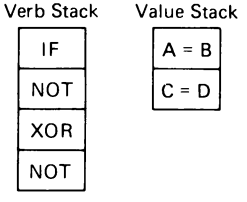
The verb "NOT" has the before and after stack frames



The syntax analyzer must be able to verify the correctness of structures such as:

IF NOT A = B XOR NOT C = D THEN 20

which would be stacked when the scanner pointer was pointing at D



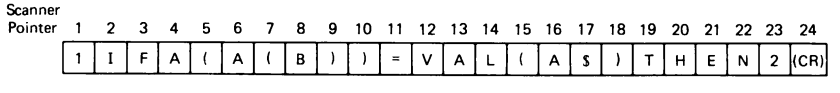
The importance of the "IF" preceding these logical verbs is that byte-wise logical statement/commands are allowed in BASICs. An example is:

IF A$ = B$ AND NOT C$ XOR D$ THEN ... .

Parentheses within the logical structure of an "IF" can be implemented using rules similar to those used for evaluation of arithmetic expressions. Including this capability in the language may be of doubtful value when cost/benefit is weighed. As a result of cost/benefit analysis few BASICs allow parentheses within logical expressions.

Rough conceptual explanations of how the syntax analyzer works was given to increase the reader's insight into the mock execution method.

An example of step-by-step syntax analysis using the variable types defined in Figure 24 of the statement



using both the conceptual method and actual method of lexical and syntax analysis is:

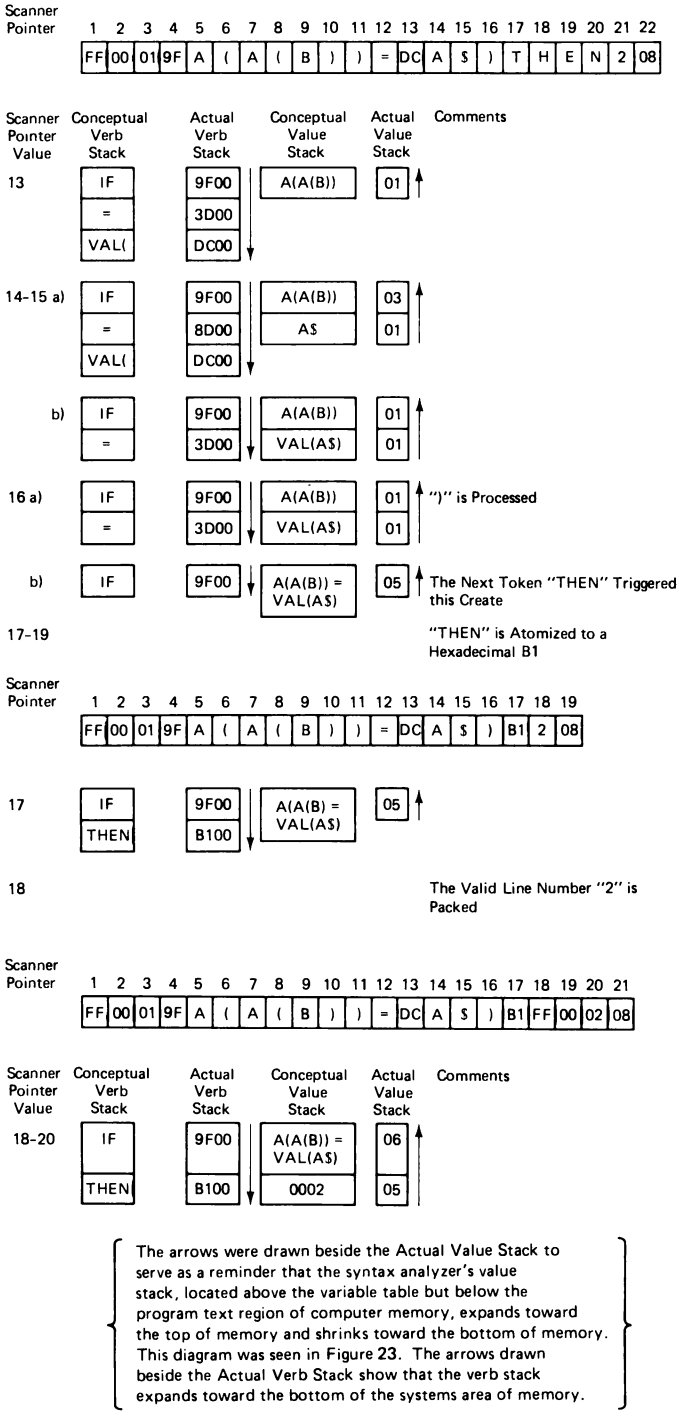
Scanner Pointer Value	Conceptual Verb Stack	Actual Verb Stack	Conceptual Value Stack	Actual Value Stack	Comments
1	—	—	—	—	Lexical Analyzer Identifies "1" as a Line Number and has if Packed in the Buffer

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
	FF	00	01	I	F	A	(	A	(	B	)	)	=	V	A	L	(	A	$	)	T	H	E	N	2	(CR)

Scanner Pointer Value	Conceptual Verb Stack	Actual Verb Stack	Conceptual Value Stack	Actual Value Stack	Comments
4-5	—	—	—	—	The Verb "IF" is Identified by the Lexical Analyzer then is Atomized. Hexadecimal 08 is a (CR)

Scanner Pointer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	FF	00	01	9F	A	(	A	(	B	)	)	=	V	A	L	(	A	$	)	T	H	E	N	2	08

4	IF	9F00	—	—	
5-6	IF	9F00 (@	A(	02	Hexadecimal 02 Represents a Numeric Array. (@ is the "Array" Verb Generated by the Syntax Analyzer
7-8	IF	9F00 (@ (@	A(A(	02 02	
9 a)	IF	9F00 (@ (@	A(A(B	01 02 02	Next Token is ")". There is a Valid Stack Frame Configuration for a Numeric Array. "Create" an After Stack Frame on the Value Stack
b)	IF	9F00 (@	A(A(B)	01 02	
10 a)	IF	9F00 (@	A(A(B)	01 02	Process ")"
b)	IF	9F00	A(A(B))	01	")" Next Token Triggers a "Create". There is a Valid Numeric Stack Frame on the Value Stack
11	IF	9F00	A(A(B))	01	Process ")"
12	IF =	9F00 3D00	A(A(B))	01	An "=" has Hexadecimal Value 3D
13-15					The Verb VAL(is Identified by the Lexical Analyzer. VAL(is Atomized to a Hexadecimal DC



The arrows were drawn beside the Actual Value Stack to serve as a reminder that the syntax analyzer's value stack, located above the variable table but below the program text region of computer memory, expands toward the top of memory and shrinks toward the bottom of memory. This diagram was seen in Figure 23. The arrows drawn beside the Actual Verb Stack show that the verb stack expands toward the bottom of the systems area of memory.

The lexical and syntax analysis program module is a collection of code blocks. Each code block is a short subroutine which should contain no more than about 10–20 machine language statements. No subroutine arguments are passed to these blocks. The reason for this is because each code block obtains any information it needs from a knowledge of values such as: the top and bottom of the verb stack addresses, the top and bottom addresses of the syntax analyzer's value stack, the type and value of the next token, . . . High level syntax and lexical analysis code blocks primarily consist of subroutine calls to these low level code blocks. Each code block performs its intended functions, minimizes the possibility of performing any unintended functions, and returns a status indicating the action it took. High level code blocks call the low level code blocks, analyze the returned status, and, depending on the value of the status, call other low level code blocks.

The next step in constructing the lexical and syntax analysis module is to write a detailed design document specifying the function of each code block required to aid in the analysis of each verb defined in the language functional requirements document. This task is not, however, a function within the scope of this design document.

SUMMARY

The Language System with its many programs scans the applications programmer's software program. Three programs and their functions are:

1. Lexical analysis, which attempts to identify verbs, variable names, arrays, numeric constants, and literals.
2. Text atomization, which compresses multicharacter verbs into a one or two byte encoding.
3. Syntax analysis, which checks commands/statements for correct grammar.

The lexical analysis, text atomization, and the syntax analysis programs operate in the buffer area of memory. The buffer area's approximate location is between the end of the program text region and the end of the variable table.

Three fundamental software rules which apply to all software code blocks, modules, and systems are:

1. Each software code block and module performs a clearly defined intended function.
2. Each software code block and module minimizes the possibility of performing either adverse or benign unintended functions.

3. Each software code block and module must provide adequate warning in event of failure.

The verb atomization table is stored in a portion of the systems area of memory. The verb atomization table contains three essential types of information:

1. The verb name.
2. The verb atomization hexadecimal value.
3. The length in characters of the verb name.

Two arrays, called stacks, are used to aid in syntax analysis:

1. The syntax analyzer's verb stack.
2. The syntax analyzer's value stack.

The syntax analyzer's verb stack is located in the systems area of memory. The syntax analyzer's value stack is located above the variable table but below the program text region of computer memory. Associated with each verb are two value stack frames: the before stack frame and the after stack frame.

6

Program Resolution

An analysis of program resolution is the topic of Chapter 6. Program resolution occurs after the RUN command has been entered from the input device. When the RUN command is entered, the variable table is cleared of all non-common variables. During program resolution, statement variables are identified, placed in the variable table, and initialized. Program resolution has other intended functions which are discussed in this chapter.

The intended functions of program resolution are:

1. Allocation of memory space in the variable table for all referenced variables in a command sequence or program text.
2. Verification that both double and single dimensioned array references with the same name do not occur in the same command sequence or program text. An example of this error which can be identified at program resolution time is the command sequence

```
DIM A(2,2): A(1)=1 (CR).
            ↑ ERR
```

   The array A( ) cannot be referenced as both double and single dimensioned.
3. Verification that line number labels or line number references do not appear in command sequences. An example of a line number label illegally appearing in a command sequence is:

```
PRINT A: 10 B=1 (CR).
         ↑ ERR
```

   An example of a line number reference appearing illegally in a com-

mand sequence is:

```
PRINT A: IF B=1 THEN 10 (CR).
                    ↑ ERR
```

4. Verification of line number references in programs against existing line numbers for validity. For example, the BASIC program

```
10 INPUT A
20 IF A<=0 THEN 40
30 PRINT "A IS GREATER THAN 0"
40 GO TO 5
         ↑ ERR
```

contains existing line numbers 10, 20, 30, and 40. THEN 40 is a valid line number reference since line number 40 exists. The line number reference GO TO 5 is an invalid reference since no line number 5 exists.

5. Repetition of lexical and syntax analysis. Program text with identifiable errors in lexical content and syntax are stored in the program text area of memory. Programs containing such errors can be stored on permanent storage device such as magnetic disk or tape for the presumed intention of later correcting the program text in error. An example of an attempt to run a program which contains a syntax error is:

```
10 A0==1    (CR)
      ↑ ERR
RUN         (CR)
10 A0==1
      ↑ ERR
```

The statement at line number 10 was entered and the second "=" sign was flagged as a syntax error. The RUN command was entered in an attempt to execute the program text at line number 10. When syntax and lexical analysis was redone during program text resolution, line number 10 was displayed on the console output device and the syntax error was identified.

6. Possible construction of auxiliary tables which are used to speed program execution. Some BASICs use a technique whereby the

program text region of memory is partitioned into, say, 16 segments. BASICs usually search the program text region of memory to determine where execution should resume as a result of a transfer of control to a line number reference. A "coarse" search of an auxiliary table which contains some label line numbers and their associated pointers indicating where these line numbers reside is made before searching the program text region segment of memory containing the sought line number. The "coarse" search of the auxiliary table eliminates the necessity of searching all of the program text area of memory to find the line number. For the program text statement:

```
GO TO 99
```

the auxiliary table:

Label Line Number	Pointer to Program Text memory area
10	0
30	54
80	160
120	245

would be searched locating line number 99 in the segment of Program Text memory beginning at relative position 160. The search for line number 99 would be started at 160 rather than 0 which would be required if no auxiliary table were used.

If a program or command sequence successfully passes program resolution, it is first marked *executable*, and then execution begins. If an error is found during program resolution, the program is marked *not executable;* program resolution is aborted and control is returned to the *entry phase*. *Entry phase* is defined as accepting input from the console input device.

Once a program has been successfully resolved, there is no need to resolve it again each time it is run, provided that the program text remains unchanged. If program text is modified in any way, then the status of program resolution is set by the Language System to *not executable*.

Suppose the statements:

```
10 PRINT "TEN"
20 PRINT "TWENTY"
```

are entered into the Language System. The command:

```
GO TO 20 (CR)
        ↑ ERR
```

would be marked in error with the reason of "unresolved program". Now if the command

```
RUN (CR)
```

were entered,

```
TEN
TWENTY
```

would be printed on the console output device. Now if the command:

```
GOTO 20 (CR)
```

were entered, it would be accepted without error. The reason is that the program has been marked *executable* at the time of the previous run. When the HALT/STEP key or its equivalent (the HALT/STEP key either stops execution of the program or causes one statement at a time to be executed each time the key is depressed) is depressed,

```
TWENTY
```

will appear on the console output device.

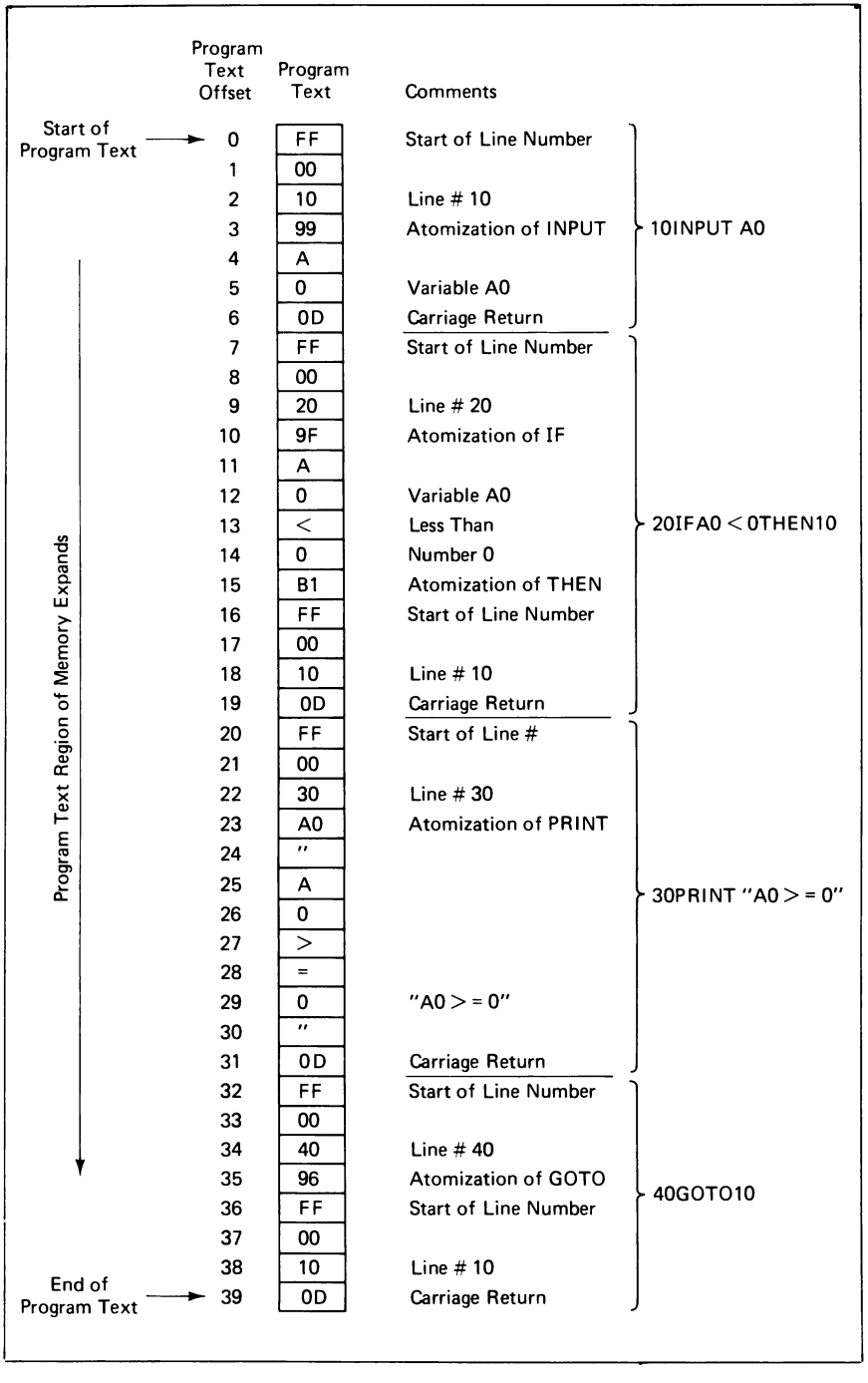
These examples illustrate that once a program has been successfully resolved it is unnecessary to do so again unless the program text has been modified.

Techniques used for lexical and syntax analysis have been covered in Chapter 5. Techniques used for inspection and additions to variables in the Variable Table have been explained in Chapters 3 and 4.

When the Variable Table is queried for the presence of a variable previously located in a program statement or command, the type of the variable is checked against the type of an already existing variable. This check is used to discover (the error of) whether an array variable is both double and single dimensioned.

Suppose that the BASIC program

```
10 INPUT A0
20 IF A0<0THEN10
30 PRINT"A0>=0"
40 GOTO10
```


Program Text Offset
Program Text
Comments
Start of Program Text
End of Program Text
Program Text Region of Memory Expands
0 FF Start of Line Number
1 00
2 10 Line # 10
3 99 Atomization of INPUT
4 A
5 0 Variable A0
6 0D Carriage Return
10INPUT A0
7 FF Start of Line Number
8 00
9 20 Line # 20
10 9F Atomization of IF
11 A
12 0 Variable A0
13 < Less Than
14 0 Number 0
15 B1 Atomization of THEN
16 FF Start of Line Number
17 00
18 10 Line # 10
19 0D Carriage Return
20IFA0 < 0THEN10
20 FF Start of Line #
21 00
22 30 Line # 30
23 A0 Atomization of PRINT
24 "
25 A
26 0
27 >
28 =
29 0 "A0 > = 0"
30 "
31 0D Carriage Return
30PRINT "A0 > = 0"
32 FF Start of Line Number
33 00
34 40 Line # 40
35 96 Atomization of GOTO
36 FF Start of Line Number
37 00
38 10 Line # 10
39 0D Carriage Return
40GOTO10

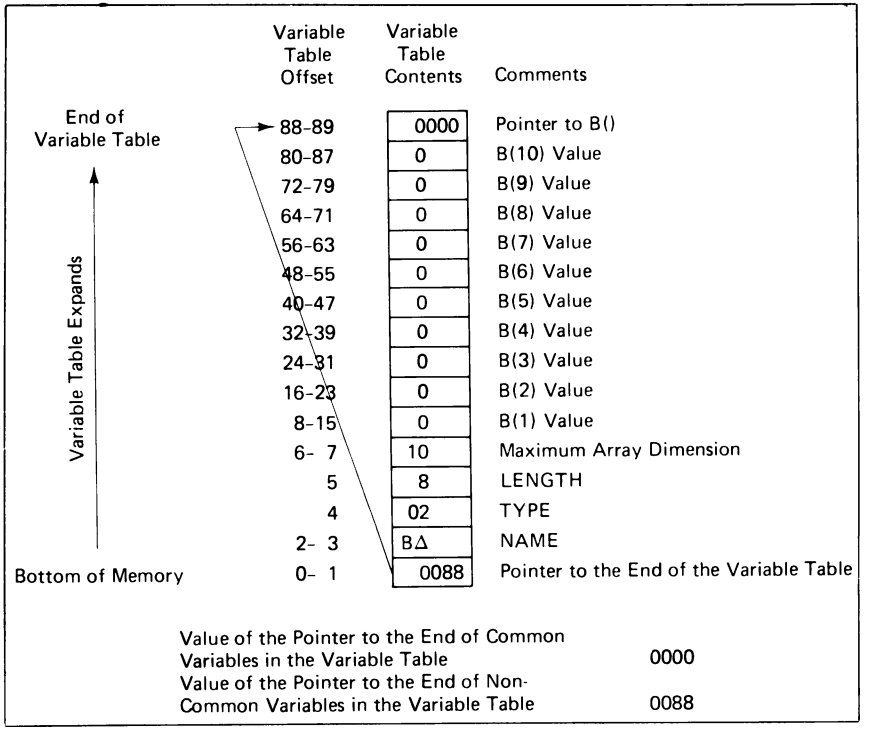


Figure 28: Diagram of computer memory prior to program resolution, but after a four statement program has been entered without lexical or syntax errors. The commands DIM B(10): PRINT B(2) have also been executed. The verb atomization values are taken from the table seen in Appendix A.

has been entered without error into the program text region of memory. Suppose the commands

DIM B(10): PRINT B(2)

have also been executed. The program text region of memory and variable table for this sequence of events are seen in Figure 28.

The steps for resolution of this program begin when the command RUN is entered at the console input device.

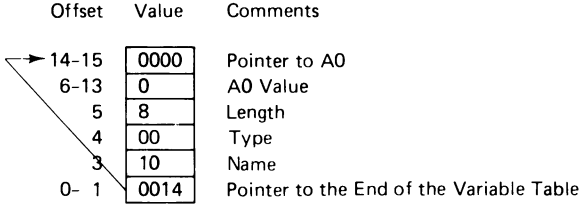
The first step is to set the pointer, which indicates the end of the variable table, equal to the value of the pointer, which indicates the end of the non-common defined variables. This step effectively eliminates all non-common variables from the variable table. In Figure 28, variable table offset locations 0-1 are set to zero.

It must be emphasized that just the non-common variables are removed from the variable table at resolution time. Variable values are only initialized when a variable is first entered into the variable table. If all variables were cleared from the variable table at the beginning of resolution, then information could not be passed between BASIC program overlays because each new overlayed program must be resolved prior to its execution.

When lexical and syntax analysis are begun, the program text pointer is initially set to 0. The lexical and syntax analyzers must be written so that they will accept both atomized and plain text since either form is acceptable. All program text that was entered without syntax or lexical errors will be expressed in atomized form in the program text region of memory.

As the program text pointer progresses from 0 to 2, as in the example given in Figure 28, a valid line number is discovered. A valid verb atomization is discovered when the program text pointer is set to 3.

A0 is identified as a numeric scalar variable. A0 is assigned TYPE = 00 and LENGTH = 8. The variable table is searched for A0. The search begins at the end of the variable table. Since the variable table contains no entries, A0 is not found. A0 is entered into the variable table, and its value is initialized to 0. At this point the diagram of the variable table is



The program text pointer is advanced to 15 and the Carriage Return found. The statement labelled 10 has been successfully processed.

The syntax for line number 20 is correct and the program text pointer is advanced to 10. Atomization of IF is HEX(9F) and the syntax is correct. A0 is identified as a variable. The variable table is searched and A0 is found, so no action is taken. Verb and value stacks for syntax analysis are constructed for the remainder of the statement. When both line number 10 is placed on the syntax analyzer's value stack, and THEN, a HEX(B1), is on the syntax analyzer's verb stack, the resolution phase software recognizes that a valid reference to line number 10 has been made. Program resolution software then initiates a sequential search, beginning at the Start of Program Text and ending either at the End of Program Text or at the point line number 10, during which a HEX(FF0010) is found in

the program text. If a statement label corresponding to the line number reference is not found, then an error is signaled. This is an *unresolved line number reference* error. Syntax analysis of the statement 30PRINT"A0>=0" reveals no errors.

At one point during the syntax analysis of the statement 40GOTO10, the verb stack of the syntax analysis will contain the atomization of GOTO, a HEX(96), and the syntax analyzer's value stack will contain the line number 10, a HEX(FF0010). This situation identifies a line number reference. A search for a statement label HEX(FF0010) is begun at the Start of Program Text. Some care must be taken to insure that a statement label line number satisfies only the line number reference and not any other. Valid line number labels occur only at the beginning of the program, or must be preceded by a Carriage Return, a HEX(0D).

In the program seen in Figure 28, there are three line number 10s and a HEX (FF0010). Of these line numbers, only the first will satisfy a line number reference.

No errors have been found in the program in Figure 28, so it is marked *executable*, and program execution is begun. This program does not need to be resolved again until the program text region of memory has been modified.

Applications programmers occasionally resort to keying RUN, keying Carriage Return, then immediately keying HALT/STEP in an attempt to halt the BASIC program after resolution, but before the first instruction of the program has been executed. Accomplishing this allows the programmer to "single step" the BASIC program, a possible aid in debugging the program.

Program resolution can be an involved process, resembling source code compilation. Auxiliary tables, which will enhance program execution speed, may be constructed at resolution time.

SUMMARY

Program resolution occurs after the RUN command has been entered from the input device. The intended functions of program resolution are:

1. Allocation of memory space in the variable table for all referenced variables in a command sequence or program text.
2. Verification that all double and single dimensioned array references with the same name do not occur in the same command sequence or program text.
3. Verification that line labels or line number references do not appear in command sequences.

4. Verification of line number references in programs against existing line numbers for validity.
5. Repetition of lexical and syntax analysis when errors in program text are stored in the program text area of memory.
6. Possible construction of auxiliary tables which are used to speed program execution.

Once the program or command sequences succeeds in passing program resolution, each is then marked executable, and the execution is begun. If an error is found during program resolution, the program is marked not executable; program resolution is aborted and control is returned to the entry phase.

7

Program Text Coordinates

The three program text coordinates are assigned to each character in a BASIC program statement. Presented in this chapter is a discussion of: the use of the colon; the execution of LIST in conjunction with S,D, and SD; and the entering of a RUN command when accompanied by a line number and a statement number within a line number.

Many BASICs allow more than one statement on a line. Statements within a line are separated by ":", colon. For example, the BASIC program:

```
10 PRINT "10,1": PRINT "10,2": PRINT "10,3" (CR)
20 PRINT "20,1": PRINT "20,2"               (CR)
```

is entered in two lines.

The next step is the execution of LIST or LIST S (S stands for "scroll"). Scrolling means that a full screen of program text is displayed each time the Carriage Return is keyed following entry of LIST S. Doing this would cause:

```
10 PRINT "10,1": PRINT "10,2": PRINT "10,3"
20 PRINT "20,1": PRINT "20,2"
```

to be displayed on the console output device.

Execution of the BASIC command LIST D (D stands for decompressed) or LIST SD (SD stands for (S) scroll and (D) decompress) would

Program Text Offset	Program Text	Program Text Coordinates	Modified Program Text Coordinates	Comments
0	FF	10, 1, 1	0, 0, 0	Line # 10
1	00	10, 1, 2	0, 0, 1	
2	10	10, 1, 3	0, 0, 2	
3	A0	10, 1, 4	0, 0, 3	Atomization of PRINT
4	"	10, 1, 5	0, 0, 4	
5	1	10, 1, 6	0, 0, 5	
6	0	10, 1. 7	0, 0, 6	
7	,	10, 1, 8	0, 0, 7	
8	1	10, 1, 9	0, 0, 8	"10, 1"
9	"	10, 1, 10	0, 0, 9	
10	:	10, 2, 1	0, 1, 0	Statement Separator
11	A0	10, 2, 2	0, 1, 1	Atomization of PRINT
12	"	10, 2, 3	0, 1, 2	
13	1	10, 2, 4	0, 1, 3	
14	0	10, 2, 5	0, 1, 4	
15	,	10, 2, 6	0, 1, 5	
16	2	10, 2, 7	0, 1, 6	"10, 2"
17	"	10, 2, 8	0, 1, 7	
18	:	10, 3, 1	0, 2, 0	Statement Separator
19	A0	10, 3, 2	0, 2, 1	Atomization of PRINT
20	"	10, 3, 3	0, 2, 2	
21	1	10, 3, 4	0, 2, 3	
22	0	10, 3, 5	0, 2, 4	
23	,	10, 3, 6	0, 2, 5	
24	3	10, 3, 7	0, 2, 6	"10,3"
25	"	10, 3, 8	0, 2, 7	
26	0D	10, 3, 9	0, 2, 8	Carriage Return
27	FF	20, 1, 1	1, 0, 0	Line # 20
28	00	20, 1, 2	1, 0, 1	
29	20	20, 1, 3	1, 0, 2	
30	A0	20, 1, 4	1, 0, 3	Atomization of PRINT
31	"	20, 1, 5	1, 0, 4	
32	2	20, 1, 6	1, 0, 5	
33	0	20, 1, 7	1, 0, 6	
34	,	20, 1, 8	1. 0. 7	
35	1	20, 1, 9	1, 0, 8	"20, 1"
36	"	20, 1, 10	1, 0, 9	
37	:	20, 2, 1	1, 1, 0	Statement Separator
38	A0	20, 2, 2	1, 1, 1	Atomization of PRINT
39	"	20, 2, 3	1, 1, 2	
40	2	20, 2, 4	1, 1, 3	
41	0	20, 2, 5	1, 1, 4	
42	,	20, 2, 6	1, 1, 5	
43	2	20, 2, 7	1, 1, 6	"20, 2"
44	"	20, 2, 8	1, 1, 7	
45	0D	20, 2, 9	1, 1, 8	Carriage Return

Figure 29: Example of assignment of coordinates to a BASIC program.

cause:

```
10 PRINT "10,1"
   :PRINT "10,2"
   :PRINT "10,3"
20 PRINT "20,1"
   :PRINT "20,2"
```

to be displayed on the console output device.

Execution of the RUN command would cause the program to be resolved and

```
10,1
10,2
10,3
20,1
20,2
```

to be printed on the console output device.

A RUN command can be designed to have two numerical arguments:

RUN Line Number, Statement Number within Line Number.

Execution of RUN 20 would cause

```
20,1
20,2
```

to be printed on the console output device.

Execution of RUN 10,3 would cause

```
10,3
20,1
20,2
```

to be printed on the console output device.

A diagram of this program as it resides in the program region of memory is shown in Figure 29.

Each character in a BASIC program can be identified by these three coordinates:

(Line Number, Statement Number with Line Number, Character Position within Statement)

An example of this coordinate system is likewise shown in Figure 29.

BASIC line number labels are in numerical order. Line number labels need not be consecutive numbers, but a one-to-one correspondence between line numbers and non-negative integers can be established. A modified program text coordinate system based on the establishment of a one-to-one correspondence between the original coordinate and the non-negative integers is also shown in Figure 29.

BASIC syntax could allow two part line references. Consequently, statements such as:

```
GOTO 10,2
```

would allow transfer of control to a statement in a line which follows the first statement on the line.

SUMMARY

The colon ":" can separate statements within a line.

LIST or LIST S and then a carriage return produces a full screen of text.

LIST D or LIST SD produces a full screen of text which has been decompressed with each line on the screen containing program text located between two colons.

The RUN command, when accompanied by a certain line number, a comma, and a statement number within that line number, produces the results of that line number, that statement number (within the line number) and any other remaining results of the program text.

Each character in a BASIC program statement can be identified by three coordinates:

1. line number,
2. statement within the line number, and
3. character position within the statement.

BASIC line number labels are in numerical order. Line number labels need not be consecutive numbers, but a one-to-one correspondence between line numbers and the non-negative integers can be established.

8

Interpreted Program Execution

The five states of a Language System are explained in Chapter 8. Also examined are processing numeric and character string data during program execution; and, working with the verbs GOSUB/RETURN, FOR-TO-STEP/NEXT, RETURNCLEAR, and READ/RESTORE/DATA.

A Language System is comprised of five states:

1. Language System initialization
2. Entry phase
3. Resolution phase
4. Language System self-test
5. Interpreted execution phase

Language System initialization occurs when the computer is powered up. The Language System is either loaded into random-access memory (RAM) (from a permanent storage device), or it may be contained in read-only memory (ROM). Many tasks, one of which is to discover the bottom of memory address, are completed during Language System initialization.

The entry phase is the state during which the Language System is either awaiting or accepting command or statement input from the console input device.

The resolution phase is the state of the Language System during which program text is being resolved. This phase is initiated by the command RUN.

As the Language System self-test proceeds, the Language System

enters the state during which the computer senses an abnormal condition, such as a momentary power loss or memory failure. If, in the course of checking itself, the Language System detects an error in its computer code or tables, then it will request (on the console output device) that it be reloaded or, if possible, it will leave a message pointing to a failed part for a hardware engineer.

The interpreted execution phase is entered directly after successful program resolution. Program resolution is initiated by executing a RUN command.

Interpreted execution means a BASIC program is being executed by a Language System program called an *interpreter.*

The interpreter scans BASIC program text and invokes appropriate Language System modules that cause the program to be executed. Like the syntax analyzer, the interpreter uses two stacks: verb and value. The interpreter invokes some of the syntax analyzer's code blocks. Tokens must be identified and placed on either the verb or value stacks. The syntax analyzer has code blocks that perform this function.

A diagram of computer memory showing the verb and value stacks is shown in Figure 30.

The memory area reserved for storing both the verb stack and syntax analyzer's verb stack is of fixed size. The syntax analyzer's verb stack is used only during the Entry and Resolution phases. In neither phase does the verb stack change in size. Only during the Execution phase does the verb stack grow and shrink.

As program text is added, the BASIC program text region of memory expands toward the bottom of memory, and shrinks toward the top of memory if program statements are removed.

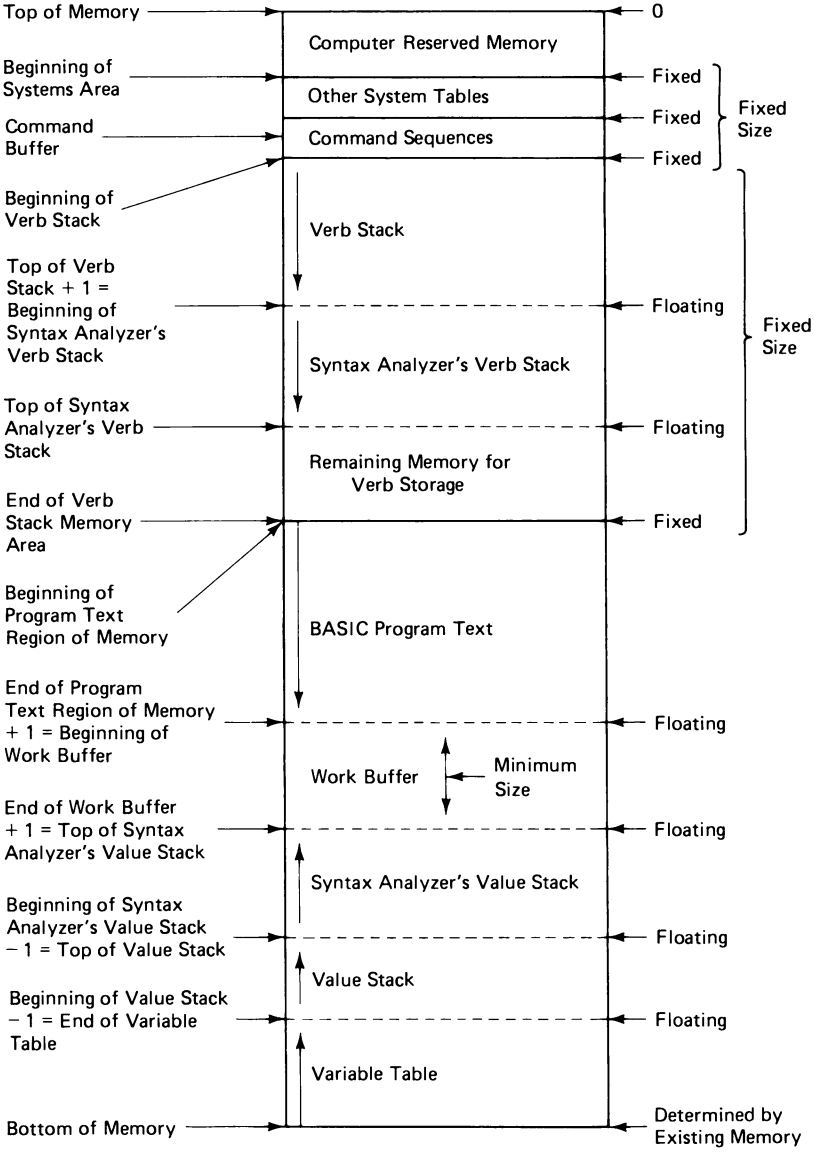
Size of the variable table is determined at the time of program resolution. During the Execution phase the value stack is used; that phase must follow the Resolution phase. The value stack may contain some

---

Figure 30: Diagram of Language System computer memory. A fixed size region of memory is allocated for storage of immediate mode command sequences. Two hundred and fifty-six characters of storage is a common value used for this memory area. The memory area allocated for storage of the verb stack and syntax analyzer's verb stack is of fixed size. The syntax analyzer's verb stack "floats" on top of the verb stack, but cannot extend beyond the end of the verb stack memory. The BASIC program text region of memory begins at a fixed memory location and expands toward the bottom of memory. The variable table begins at the bottom of memory and expands toward the top of memory. The value stack floats on top of the variable table. The syntax analyzer's value stack floats on top of the value stack. The work buffer is allocated what memory space remains. The work buffer is allocated a minimum number of memory locations. An error is displayed on the console output device if an entry is made which would cause the work buffer to contain less than a minimum number of memory locations. ——→

common variables at the time of resolution. These common variables are retained at program resolution.

The unused computer memory located between the end of the program text region of memory and the top of the syntax analyzer's value stack is used as a Work Buffer. The Entry phase uses the Work Buffer



to store commands or statements which are entered from the console input device.

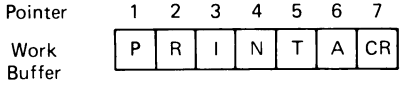
A minimum size is assigned for the Work Buffer. Although the size of the Work Buffer cannot be less than that assigned, it can be greater. The reason for this is that it would be possible to enter a program and yet not have sufficient room left in the Work Buffer to type the command RUN, or other commands used to save the program on a permanent storage device. (The Buffer described in Chapter 5 is the Work Buffer.)

Command sequence programs often cannot be executed in the Work Buffer. A separate memory region, located in a Systems Area of memory called the Command Buffer, must be allocated to contain some command sequences.

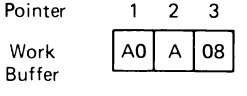
The command

PRINT A

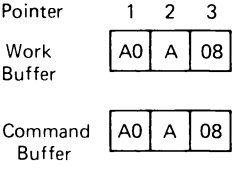
as entered into the Work Buffer is



and is atomized to



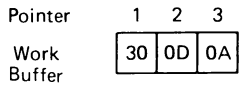
where HEX(A0) is the atom for PRINT and HEX(08) is Carriage Return. This line is sent to the Command Buffer. The contents of the Work Buffer and Command Buffer are:



The interpreter begins execution of the command in the Command Buffer. The variable table is searched for variable A. Suppose A is not found. A is entered into the variable table and its value is set at 0. Suppose the number representation in the variable table is eight byte floating point binary coded decimal. A 0 is represented as HEX(0000000000000000). The console output device usually displays only valid ASCII characters.

The PRINT command has to be responsible for seeing that the eight

byte floating point 0 is converted to an ASCII 0, which is a HEX (30). Part of the execution of PRINT causes the work buffer to be filled with:



which is ASCII 0, followed by Carriage Return (HEX(0D)), followed by Line Feed (HEX (0A)).

The Command Buffer is required for the execution of some command sequences since the interpreter needs the Work Buffer to execute verbs, such as INPUT and PRINT.

The Command Buffer is usually given a reasonable size of about 256 bytes. If the Work Buffer is filled with a command sequence of more than 256 bytes, then only the first 256 are transferred to the Command Buffer.

The Command Buffer is available to Language System software engineers for uses other than command sequence storage.

PROGRAM EXECUTION: NUMERICAL COMPUTATIONS

The applications programmer can cause the contents of the

- Program Text Region
- Variable Table
- Work Buffer
- Command Sequence Buffer
- Value Stack

to be altered by entering either BASIC program statements or commands into the Language System. The contents of the Variable Table and Work Buffer can be modified during the execution of a BASIC program. A diagram of the arrangement is seen in Figure 31.

Character string data can be fetched from any one of these five regions of microcomputer memory. Numeric data, on the other hand, can only be fetched from the Variable Table and Value Stack. Processing character string data is more complicated than processing numerical data. Therefore, numerical computation procedures will be explained first.

The basic principle conveyed in Figure 31 is that memory regions containing either program statements or commands can only be, from the applications programmer's standpoint, read during program execution. This statement is not entirely accurate since the application programmer

may be able to issue a BASIC statement in a program which causes more program text to be read into the program text region of memory. This is called an *overlay*.

When executed by the Language System, BASIC programs can cause both reads and writes into the Value Stack, the Variable Table, and the Work Buffer. The Language System's programs do the reading and writing to these memory areas on behalf of the application programmer's BASIC programs.

Stack frame formats used for BASIC program execution have the format:

CHAIN	FRAME	LOCATION	VALUE

Chain is a two byte binary number specifying the number of bytes occupied by CHAIN, FRAME, LOCATION, and VALUE.

Value is either a pointer or a value itself.

Frame is a one byte binary number which specifies the frame type. The values of FRAME are

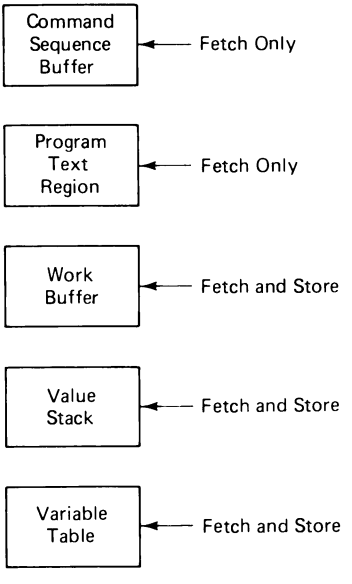


Figure 31: Diagram of microcomputer memory from the applications programmer's standpoint of whether a BASIC program causes information to be read or written from the various regions while it is executing.

FRAME	Frame description
00	Null
01	Four byte binary pointer pointing to a name in the variable table
02	A four byte binary pointer pointing to a character string value, followed by a four byte binary string starting position, followed by a four byte binary string length
03	Four byte binary pointer pointing to a numeric value in the variable table
04	Numeric value
05	Character string value

Location is a single byte number indicating where the data resides. Binary LOCATION definitions are

LOCATION	Location Definition	Permissible VALUE
00	Null	
01	Program Text Region	Pointer
02	Variable Table	Pointer
03	Work Buffer	Pointer
04	Command Buffer	Pointer
05	Value Stack	Numeric or character string

Null indicates that 00 is assigned before the values 01–05.

The best way to understand how stack frames for BASIC program execution are constructed is by example.

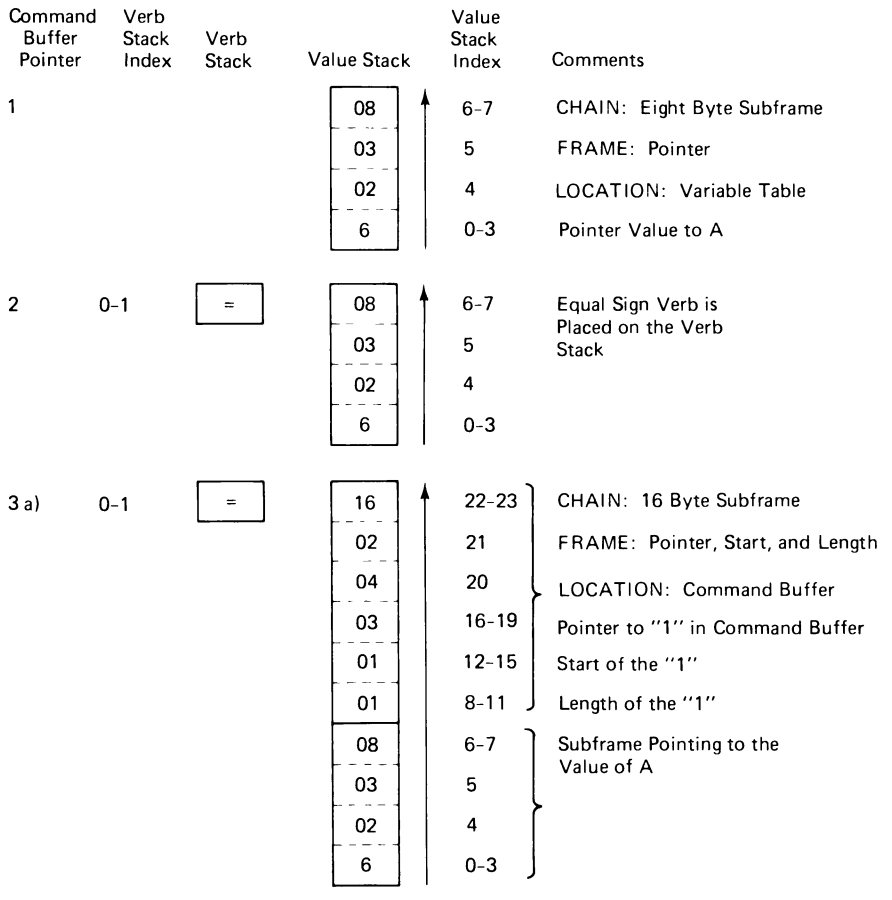
The BASIC command sequence,

A=1: B=A (CR),

Command Buffer, and Variable Table layout is shown in Figure 32. This diagram was made just after program resolution, but prior to execution. To increase clarity, the specified pointer values to the next variable in the Variable Table are somewhat imprecise. This imprecision resulted from the convention of numbering pointers to Variable Table fields beginning with 0.

Program execution begins with the Language System scanning the Command Buffer. Recall that the Verb stack expands from the top of memory toward the bottom of memory. The Value stack, however, expands from the end of the Variable Table toward the top of memory.

In order to understand just what occurs during program execution, diagrams of the direction(s) in which both the Verb and Value stacks expand during program execution are important.



Since a command sequence is being executed, the Language System processes text in the Command Buffer. If program statements were to be executed, then the Language System would turn its attention to the Program Text region of memory.

A is identified as a variable. A pointer to the value of A is placed on the value stack. When the scanner pointer to the Command Buffer is two, the equal sign is identified as a verb, and placed on the Verb Stack.

When the scanner pointer has three for a value, the "1" is identified as a number. "1" is a character string in the Command Buffer; "1" begins at pointer position 3, and has a length of one character. This information is placed in a value stack subframe whose length is CHAIN = 16. The pointer and the length and start of the character string occupy the VALUE portion of the stack subframe.

The ":" at scanner position 4 triggers an "execute" of the "="

verb. The "=" verb examines the character string "1" and realizes that a conversion from the ASCII "1" to a numeric 1 is required. The "=" pushes a "convert from ASCII" verb on the verb stack, and then invokes a "convert from ASCII" verb.

The remaining steps in the execution of "=" are:

Command Buffer Pointer	Verb Stack Index	Verb Stack	Value Stack	Value Stack Index	Comments
3 b)	0-1	=	12	18-19	CHAIN: 12 Byte Subframe
			04	17	FRAME: Numeric
			05	16	LOCATION: Value Stack
			1	8-15	Eight Byte Numeric "1"
			08	6-7	
			03	5	Subframe Pointing to the Value of A
			02	4	
			6	0-3	
c)		———	———		Assignment Execution Completed

Command Buffer

1	2	3	4	5	6	7	8	9
A	=	1	:		B	=	A	0D

Variable Table Expands

Bytes	Value	Comments
28-29	14	Pointer to the Next Variable in the Variable Table
20-27	0	Eight Byte Number Value of B
19	8	LENGTH
18	00	TYPE
16-17	BΔ	NAME
14-15	0	Pointer to the Next Variable in the Variable Table
6-13	0	Eight Byte Number Value of A
5	8	LENGTH
4	00	TYPE
2-3	AΔ	NAME
0-1	28	Pointer to End of Variable Table

Figure 32: Diagram of microcomputer memory just after program resolution has occurred, but before execution begins for the BASIC command sequence A=1: B=A (CR).

The CHAIN information is required so that a verb can determine where the next noun information on the value stack begins, and what the length of the VALUE portion of the subframe is.

In summary, sufficient information exists on the Value Stack for any verb processing module to locate and analyze subframes required for its successful execution.

Execution of the command sequence seen in Figure 32 resumes.

Command Buffer Pointer	Verb Stack Index	Verb Stack	Value Stack	Value Stack Index	Comments
4					Skip Over the ":"
5					Skip the Blank
6			08	6-7	CHAIN: Eight Byte Subframe
			03	5	FRAME: Pointer
			02	4	LOCATION: Variable Table
			20	0-3	Pointer to Value of B
7	0-1	=	08	6-7	The "=" Verb is Pushed Onto the Verb Stack
			03	5	
			02	4	
			20	0-3	
8	0-1	=	08	14-15	CHAIN: 8 Byte Subframe Length
			03	13	FRAME: Pointer
			02	12	LOCATION: Variable Table
			6	8-11	Pointer to Value of A
			08	6-7	
			03	5	Subframe Pointing to the Value of B
			02	4	
			20	0-3	
c)		———	———		(CR) Triggers Execution of the Assignment Code Block. The Assignment has the Responsibility to Pop Both the Verb and Value Stacks

Comparison of processing A=1 to B=A revealed that the only difference was that A=1 required both moving one number from the Value Stack to the Variable Table and moving an ASCII number to an internal number format conversion, while B=A required a move of eight bytes between two different locations in the Variable Table.

Two important points of program execution are:

1. The variable attributes of CHAIN, FRAME, and LOCATION need to be placed on the Value Stack along with frame VALUE.
2. The Language System verb processing modules must have sufficient "intelligence" to determine which type of data needs to be processed.

Step by step execution of the BASIC program:

```
10 DIM A(2):D=A(1)*(C-B)  (CR)
```

should clarify the process of numerical computation program execution. This program is atomized and placed in the Program Text Region of memory to:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
FF	00	10	93	A	(	2	)	:	D	=	A	(	1	)	*	(	C	–	B	)	0D

where HEX(93) is the atom for DIM. DIM, not DIM(, must be atomized because character strings can have their lengths changed by using a statement, such as 10 DIM A$64, which would change the default length of A$ (often 16) to a length of 64 bytes.

When program resolution has been completed, but before execution begins, the variable table is:

Bytes	Value	Comments
66-67	52	Pointer to Next Variable in the Variable Table
58-65	0	Value of B
57	8	LENGTH
56	00	TYPE
54-55	BΔ	NAME
52-53	38	Pointer to Next Variable in the Variable Table
4-51	0	Value of C
43	8	LENGTH
42	00	TYPE
40-41	CΔ	NAME
38-39	24	Pointer to Next Variable in the Variable Table
30-37	0	Value of D

Bytes	Value	Comments
29	8	LENGTH
28	00	TYPE
26-27	DΔ	NAME
24-25	0	Pointer to Next Variable in the Variable Table
16-23	0	Value of A(2)
8-15	0	Value of A(1)
6-7	2	Maximum Dimension
5	8	LENGTH
4	02	TYPE: Single Dimensioned Numeric Array
2-3	AΔ	NAME
0-1	66	Pointer to the End of the Variable Table

Program resolution was initiated by entering the RUN command.

The scanner pointer must now point to the Program Text Region of computer memory to begin execution of BASIC statements.

Program execution proceeds:

Program Text Pointer	Verb Stack Index	Verb Stack	Value Stack	Value Stack Index	Comments
1-3		———	———		Skip Over Line Number
4-9		———	———		Skip Over DIM: DIMs Processed at Program Resolution Time
10		———	08	6-7	CHAIN: Eight Byte Subframe
			03	5	FRAME: Pointer
			02	4	LOCATION: Variable Table
			30	30	Pointer to Value of D
11	0-1	=	08	6-7	= Verb Pushed on Verb Stack
			03	5	Subframe Pointing to the Value of D
			02	4	
			30	0-3	
12-13	0-1	=	08	14-15	CHAIN: 8 Byte Subframe Length
	2-3	Array	01	13	FRAME: Variable Name Pointer
			02	12	LOCATION: Variable Table
			2	8-11	Pointer to A(

Program Text Pointer	Verb Stack Index	Verb Stack	Value Stack	Value Stack Index	Comments
12-13-Continued			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
14 a)	0-1	=	16	30-31	CHAIN: 16 Byte Subframe
	2-3	Array	02	29	FRAME: Pointer and Length
			01	28	LOCATION: Program Text Region
			01	24-27	Pointer to ASCII "1"
			01	20-23	Start of ASCII "1"
			01	16-19	Length of ASCII "1"
			08	14-15	Pointer to Name of A(
			01	13	
			02	12	
			2	8-11	
			08	6-7	Subframe Pointer to the Value of D
			03	5	
			02	4	
			30	0-3	
14 b)	0-1	=	12	26-27	CHAIN: 12 Byte Subframe
	2-3	Array	04	25	FRAME: Numeric
			05	24	LOCATION: Value Stack
			1	16-23	Numeric "1"
			08	14-15	Pointer Pointing to Name A(
			01	13	
			02	12	
			2	8-11	
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	

Program Text Pointer	Verb Stack Index	Verb Stack	Value Stack	Value Stack Index	Comments
14 c)	0-1	=	08	14-15	CHAIN: 8 Byte Subframe
			03	13	FRAME: Pointer
			02	12	LOCATION: Variable Table
			8	8-11	Pointer Value of 8 Pointing to A(1) in Variable Table
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
15		The ")" is Skipped Over but Triggered the Array Evaluation in Step 14			
16	0-1	=	08	14-15	The Multiplication Verb, "*" is Placed on the Verb Stack
	2-3	*	03	13	
			02	12	Pointer Pointing to Value of A(1) inVariable Table
			8	8-11	
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
17	0-1	=	08	14-15	The "(" is Placed on the Verb Stack
	2-3	*	03	13	
	4-5	(	02	12	Subframe Pointing to the Value of A(1)
			8	8-11	
			08	6-7	Subframe Pointing to Value of D
			02	5	
			02	4	
			30	0-3	

Program Text Pointer	Verb Stack Index	Verb Stack	Value Stack	Value Stack Index	Comments
18	0-1	=	08	22-23	CHAIN: Eight Byte Subframe
	2-3	*	03	21	FRAME: Pointer
	4-5	(	02	20	LOCATION: Variable Table
			44	16-19	Pointer to Value of C
			08	14-15	Subframe Pointing to the Value of A(1)
			03	13	
			02	12	
			8	8-11	
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
19	0-1	=	08	22-23	Push "−" on the Verb Stack
	2-3	*	03	21	Subframe Pointing to the Value of C
	4-5	(	02	20	
	6-7	−	44	16-19	
			08	14-15	Subframe Pointing to the Value of A(1)
			03	13	
			02	12	
			8	8-11	
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
20 a)	0-1	=	08	30-31	CHAIN: Eight Byte Subframe
	2-3	*	03	29	FRAME: Pointer
	4-5	(	02	28	LOCATION: Variable Table
	6-7	−	58	24-27	Pointer to Value of B

Program Text Pointer	Verb Stack Index	Verb Stack	Value Stack	Value Stack Index	Comments
20 a) -Continued			08	22-23	Subframe Pointing to the Value of C
			03	21	
			02	20	
			44	16-19	
			08	14-15	Subframe Pointing to the Value of A(1)
			03	13	
			02	12	
			8	8-11	
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
b)	0-1	=	12	26-27	CHAIN: 12 Byte Subframe
	2-3	*	04	25	FRAME: Numeric Value
	4-5	(	05	24	LOCATION: Value Stack
			0	16-23	Eight Byte Value of C-B
			08	14-15	Subframe Pointing to the Value of A(1)
			03	13	
			02	12	
			8	8-11	
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
21	0-1	=	12	26-27	The ")" Causes the "(" to be Popped Off the Verb Stack
	2-3	*	04	25	
			05	24	Value of C-B Located on the Value Stack
			0	16-23	

Program Text Pointer	Verb Stack Index	Verb Stack	Value Stack	Value Stack Index	Comments
21– Continued			08	14-15	Subframe Pointing to the Value of A(1)
			03	13	
			02	12	
			8	8-11	
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
22 a)	0-1	=	12	18-19	(CR) Causes Processing of the "*" by the Language System
			04	17	
			05	16	Value of A(1)*(C – B) Located on the Value Stack
			0	8-15	
			08	6-7	Subframe Pointing to the Value of D
			03	5	
			02	4	
			30	0-3	
22 b)		——	——		(CR) also Causes the "=" to be Processed by the Language System
					The Language System "=" Processing Code Block is Responsible for Popping Both the Verb and Value Stacks

Now that the execution of statements has been diagrammed, a detailed explanation of the steps of execution will be given. The scanner pointer must be directed toward the program text region of memory since a program, as opposed to a command sequence, is being executed.

When the RUN command was keyed, it was entered into the Work Buffer. From there, the RUN command was transferred to the Command Buffer for execution. Execution of the command caused the two line BASIC program to be resolved. Resolution involved a sequential scan of

the entire program for the purpose of allocating storage for the variables in the variable table. The variables A( ), D, C, and B were stored in the same order as they were discovered in the program.

Execution of the BASIC program begins with the Language System looking for successive meaningful symbols. Because the BASIC program survived lexical and syntax analysis and resolution, the Language System expects meaningful program text.

While scanning program text positions 1–4, the Language System will identify a line number. No processing of this is required. At position 4, a DIM atom is identified and the Language System skips to end of the line at position 9, and then begins to process the symbol(s) at position 10.

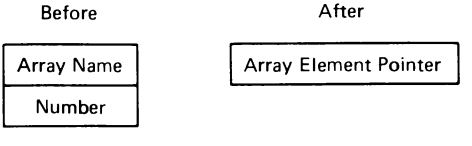
The variable D is identified and the Language System searches the variable table for D. The Language System creates a value stack subframe pointing to the *value* of D. The Language System searched the variable table for the *name* D, but then created a pointer pointing to the value of D. The length of the value stack subframe is always CHAIN. This knowledge is very important to the Language System software engineer or the applications programmer since a program can be written to 'chain' through the value stack examining value stack subframes. This traversal has to begin with the stack subframe located at the top of the value stack because the first CHAIN is located there.

At scanner pointer position 11, "=" is identified as a verb and placed on the verb stack. The Language System always examines the next token to decide whether to perform a stacking operation, or do one or more computations.

The array name A( ) is found at positions 12–13. A pointer to the name of A( ) is placed on the value stack. A value of A( ) cannot be pointed to since the index of A( ) has not yet been determined. Discovery of an array name also causes a Language System generated verb called "Array" to be pushed on the verb stack.

At scanner position 14 the number 1 is found. This number is a one digit ASCII character. The Language System converts the character string beginning at position 14 in the program text region of memory having a length of one character to an eight byte numeric 1 and places the numeric 1 on the value stack along with its header information.

The next token is a ")" and matches the Array verb on the verb stack. This informs the Language System that the array reference is to be evaluated. As diagrammed below in a rough conceptual way, a single dimension array reference has the before and after stack frames:



The Language System compares the index value against the maximum array dimension to check if a legal reference was made. If the index is outside the range of allowable subscripts (here 1 and 2), then the statement is displayed on the console output device; an arrow extends from the next line toward the offending index; and an error code is shown. Program execution is terminated.

Advanced BASIC Language Systems allow BASIC programs to intercept any error which might stop program execution. Notice that an out-of-bounds index might not necessarily stop program execution. Some applications programmers might use the error detection mechanism of the Language System to sense a failing operation and then take action to correct the situation.

Index values in BASIC are truncated to integers. X(3.45) is a specification of X(3).

At step 14c, the array name pointer and index subframes are replaced by a pointer to the indexed array element. In this case, the pointer is to A(1).

The ")" is skipped over and the "*" is identified as a verb and placed on the verb stack at step 16.

The "(" is identified as a verb at position 17 and is placed on the verb stack.

In step 18, C is found in the variable table. A value stack subframe, including a pointer to the value of C, is placed at the top of the value stack.

In step 19, "−" is identified by the Language System as a verb, and it is placed on the verb stack.

At step 20a, B is identified as a variable. The variable table is searched for the name B, B is found, and a subframe containing a pointer to the value of B is placed on the value stack.

The next symbol at step 20 is a ")". This causes the Language System to invoke the code block which processes the "−" verb. The result of this is seen in step 20b.

Processing the ")" in step 21 causes only the "(" to be popped from the verb stack.

Discovery of the carriage return at scanner position 22 first causes the "*" multiply verb to be executed. The execution leaves the product at the top of the value stack.

The "=" verb can be interpreted by the Langugage System. This event causes both the verb and value stacks to be moved back to the positions they occupied at the time execution of the statement began. This is most important. Much of the time, execution of a statement or command will begin with the value and verb stacks empty. This will not be the case when a statement is enclosed in a FOR/NEXT loop, or in a subroutine reached with a GOSUB statement or command.

Rules for processing arrays may still be confusing. For this reason the two commands PRINT (A(I,J)) and PRINT A((I+J)) will be executed in a rough conceptual manner.

The command PRINT (A(I,J)) has the command buffer atomization:

1	2	3	4	5	6	7	8	9	10
A0	(	A	(	I	,	J	)	)	0D

and execution steps:

Step	Verb Stack	Value Stack
1	A0	———
2	A0 (	———
3-4	A0 (Array	A(
5	A0 (Array	A(I
6-7 a)	A0 (Array	A(I J
7 b)	A0 (	A(I, J)
8 a)	A0 (	A(I, J)
8 b)	———	———

The important steps in this process is to observe that A(and Array are pushed onto the value and verb stacks respectively in step 3–4 and A(I,J) located when ")" was the next token which matched Array on the verb stack.

PRINT A((I+J)) has the command buffer atomization

1	2	3	4	5	6	7	8	9	10
A0	A	(	(	I	+	J	)	)	0D

and execution steps:

Step	Verb Stack	Value Stack
1	A0	
2-3	A0 Array	A(
4	A0 Array (	A(
5	A0 Array (	A(I
6	A0 Array (+	A(I
7a)	A0 Array (+	A(I J
7 b)	A0 Array (	A(I + J
8 a)	A0 Array	A((I + J)
8 b)	A0	A((I + J))
9		

There are two important points in this example. First, discovery of the array reference A(at scanner pointer values 2–3 resulted in the Language System placing the verb "Array" on the verb stack at the same time that A(was placed on the value stack. The second important point is that the ")" in step 8 was identified as the closing right parenthesis of a numeric expression because a corresponding left parenthesis was located on the verb stack.

GOSUB/RETURN AND FOR-TO-STEP/NEXT PROCESSING

Both the verb and value stacks expand and contract during the execution of commands and statements. In most cases the top of stack pointers for the verb and value stacks will be the same values at the end of execution as they were at the beginning for each line of code executed. Several examples of such statements are:

```
A=B*(C+D)
```

or

```
PRINT A,8*(A+2)
```

BASICs contain at least two statements, GOSUB and FOR-TO-STEP, for which the verb and value stack pointers are not returned to their original positions after execution of these commands or statements. For each of these statements or commands it takes a second verb to cause the verb and value stack pointers to be returned to the values they had at the beginning of the execution. The verbs GOSUB/RETURN and FOR-TO-STEP/NEXT operate in pairs. Some BASICs have verb pairs, such as WHILE/ENDWHILE. The principle of how these verb pairs work is the same. Execution of the first of the pair causes the verb and value stacks to be loaded with some information. Execution of the second of the verb pair may cause the information already placed on the verb and value stacks to be removed. This removal is not a certainty; some condition might have to be met before the information is removed from the two stacks. An example is:

```
FOR A=1 TO 3 STEP .5
: NEXT A
```

Execution of NEXT A removes the information placed on the verb

and value stacks only when FOR A=1 TO 3 STEP .5 was executed at a time when A is greater than or equal to 3.

The BASIC program:

```
10 GOSUB 20: STOP
20 PRINT A: GOSUB 30: RETURN
30 PRINT B: RETURN
```

is atomized to:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
FF	00	10	9A	FF	00	20	:	95	0D	FF	00	20	A0	A	:	9A

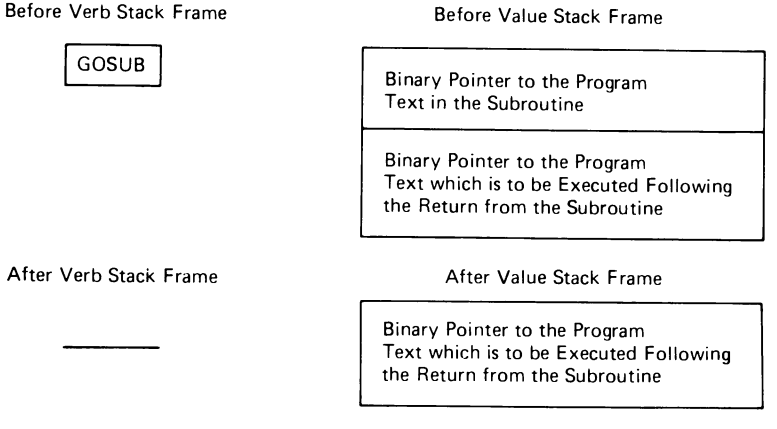
18	19	20	21	22	23	24	25	26	27	28	29	30	31
FF	00	30	:	9B	0D	FF	00	30	A0	B	:	9B	0D

where FF is the hexadecimal header of a packed decimal line number, and 9A is the hexadecimal atomization of GOSUB, 9B of RETURN, A0 of PRINT, and 95 of STOP. These hexadecimal atomization values are given in Appendix A. The hexadecimal 0D's are the carriage returns at the end of each line.

Program execution is initiated by entering the RUN command. Before program execution the RUN command causes program resolution take place, and A and B are entered in the variable table.

A series of rough conceptual diagrams of both the verb and value stacks during execution of this program will make more comprehensible the processing of the GOSUB/RETURN verb pairs.

The GOSUB verb has the before/after stack frames

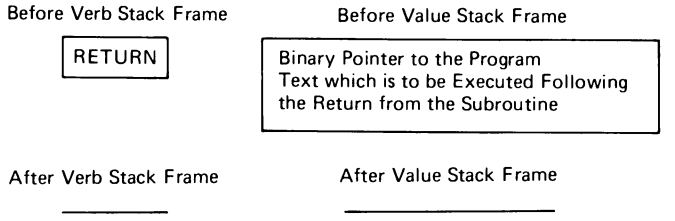


Execution of this program is

Interpreter Scanner Pointer	Verb Stack	Value Stack	Comments
1-3	———	——	Line Number Label Discovered and Skipped Over
4	GOSUB	——	GOSUB Atom Placed on Verb Stack
5-7	GOSUB	14	Line Number Reference to 20 was Discovered. The Program Text Region was Searched for Line Number Label 20. Label Line Number 20 was Located in Positions 11-13. 14 Points to the Beginning of the Statement on Line 20.
8 a)	GOSUB	14 9	The Next Program Statement Following the GOSUB Begins at Program Text Region Position 9.
b)	———	9	GOSUB is Processed and the Interpreter's Scanner Pointer is set to 14
14	PRINT	9	PRINT Verb is Placed on the Verb Stack
15a)	PRINT	9 A	A Pointer to A in the Variable Table is Placed on the Value Stack. Carriage Return is the Next Symbol and Triggers a Verb Execution
b)	———	9	The Value of A is Printed on the Console Output Device
16			The ":" is Skipped Over
17	GOSUB	9	GOSUB is Placed on the Verb Stack
18-20	GOSUB	9 27	Line Number Reference 30 is Identified The Program Text Region of Memory is Searched for Line Number Label 30. Line Number 30 is Found at Positions 24-26 in the Program Text Region of Memory. The Pointer to the Beginning of the Statement on Line 30 has Value 27
21 a)	GOSUB	9 27 22	The Statement Following this GOSUB Begins at Position 22 in the Program Text Region of Memory

Interpreter Scanner Pointer	Verb Stack	Value Stack	Comments
21 b)	———	9 22	The GOSUB is Executed and the Interpreter's Scanner Pointer is Set to 27
27	PRINT	9 22	PRINT is Placed on the Verb Stack
28 a)	PRINT	9 22 B	A Pointer to the Value of B Located in the Variable Table is Placed on the Value Stack
b)	———	9 22	The Next Symbol is ":" and Triggers an Execution of the Verb. The Value of B is Printed on the Console Output Device
29		9 22	The ":" is Skipped by the Interpreter
30 a)	RETURN	9 22	The RETURN is Placed on the Verb Stack. The Next Symbol is a Carriage Return and Triggers an Execute. The Interpreter's Scanner Pointer is Set to Value at the Top of the Value Stack
b)	———	9	
22 a)	RETURN	9	The RETURN is Placed on the Verb Stack. The Carriage Return is Next and Triggers an Execute
b)	———	———	Both the Verb and Value Stacks are Popped and the Scanner Pointer of the Interpreter is Set to 9
9 a)	STOP	———	The Next Symbol is a Carriage Return and Triggers an Execute
b)	———	———	The STOP is Executed by the Interpreter. The Language System is Placed in the Entry State

The RETURN verb has the before/after stack frames



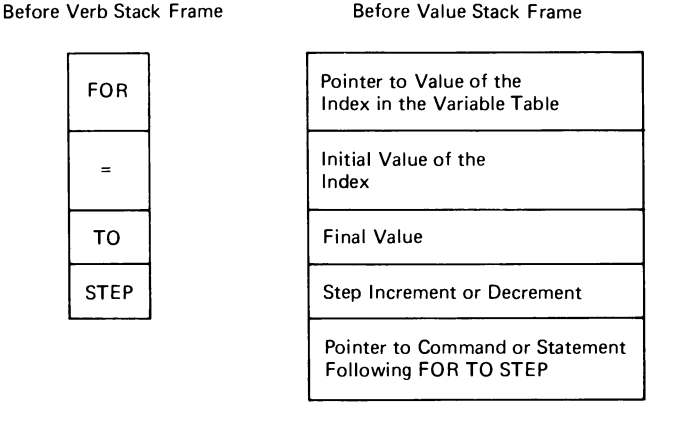
This means that the after value stack frame for the GOSUB verb matches the before value stack frame. This is why GOSUBs are usually paired with RETURNs.

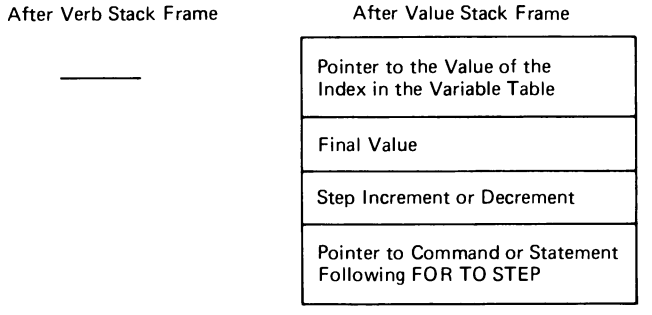
The word "usually" was required in the last sentence because some BASICs define a RETURNCLEAR verb. Execution of this verb causes the binary pointer on the value stack to be popped, but the interpreter's scanner pointer is not set to this value. Instead, the interpreter executes the next sequential instruction.

A called subroutine normally returns control to the program, whether it is the main program or another subroutine which called it. The RETURNCLEAR verb is valuable because it allows this normal return transfer of control to be altered. This is a useful feature when a serious error occurs within a subroutine. In this case a GOTO might be used to pass program control to another program.

Value stack frames that give a detailed analysis of the program execution performed in the previous example would have to include: CHAIN, FRAME, LOCATION, and VALUE. In this previous example the LOCATION would be 01, the Program Text Region of memory; FRAME would be 02, Binary Pointer, for the pointers contained on the value stack.

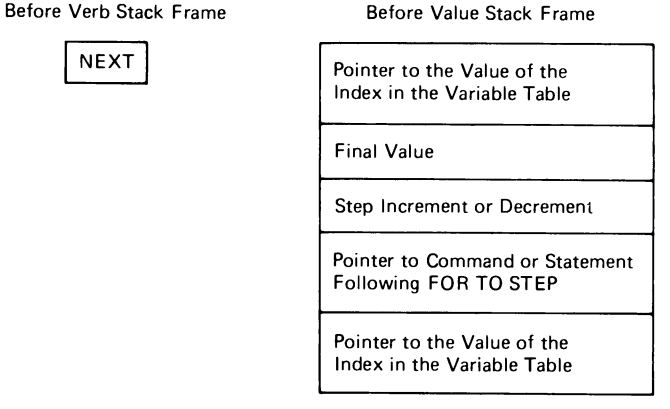
The FOR-TO-STEP verb has before/after stack frames:





The index's initial value, final value, and step increment or decrement can be values either on the stack or pointers to variables whose values are stored in the variable table.

The NEXT verb has before stack frames:



The NEXT verb has two alternate after stack frames. Which stack frame is selected depends on whether the final condition of the FOR-TO-STEP has been satisfied.

If the final condition has not been satisfied, then stack frames are the same as the FOR-TO-STEP after stack frames. If the final condition has been satisfied, then the NEXT after stack frames are:

The NEXT verb has the responsibility of both updating the index and making the required comparisons. The top and last elements of the value stack frame are the same so that NEXT can verify it is processing the correct FOR TO STEP. Using these two elements, the commands:

```
FOR A=1 TO 2: NEXT B
                  ↑ ERR
```

would be checked during execution. This check is quite important as an error would not be caught at syntax analysis time. Syntax analysis examines each command and statement, line by line.

The command sequence:

```
FOR A=1 TO 2: NEXT A
```

is stored in the Command Buffer as

1	2	3	4	5	6	7	8	9	10
9E	A	=	1	B2	2	:	9D	A	0D

Upon keying carriage return, this command sequence is analyzed for syntax errors, the command sequence is resolved, and the interpreter is invoked with its scanner pointer pointing to the beginning of the command buffer.

9E is the hexadecimal atomization of FOR, B2 of TO, and 9D of NEXT. 0D is a carriage return.

The step by step execution of this command sequence is

Interpreter's Scanner Pointer	Verb Stack	Value Stack	Comments
1	FOR	——	FOR Verb is Pushed on the Verb Stack
2	FOR	A	A Pointer to the Value of A which is Located in the Variable Table is Pushed on the Value Stack
3	FOR =	A	= Verb is Pushed on the Verb Stack
4	FOR =	A 1	The ASCII 1 is Converted to a Number and this Number is Pushed on the Value Stack
5	FOR = TO	A 1	TO Verb is Pushed on the Verb Stack

Interpreter's Scanner Pointer	Verb Stack	Value Stack	Comments
6 a)	FOR = TO	A 1 2	The ASCII 2 is Converted to a Number This Number Represents the Final Value. It is Pushed on the Value Stack
b)	FOR = TO STEP	A 1 2 1 8	The Next Symbol is ":"; The Interpreter Realizes from this that there is No STEP so it Creates a STEP Verb and an Increment of 1 and Pushes these, Respectively, on the Verb and Value Stacks. The Interpreter also Pushes a Pointer Pointing to the Command which Follows the FOR TO STEP onto the Value Stack
c)	———	A 2 1 8	The FOR TO STEP is Executed. The Initial Index Value is no Longer Needed so it was Removed from the Value Stack Frame
7	———	A 2 1 8	The ":" is Skipped Over by the Interpreter
8	NEXT	A 2 1 8	NEXT Verb is Pushed on the Verb Stack
9 a)	NEXT	A 2 1 8 A	A Pointer Pointing to the Value of A which is Located in the Variable Table is Pushed on the Value Stack. This Pointer Value is Checked to see if it is the Same as the Pointer Value of the Index. If it is Not, an Error is Flagged and the Language System Enters the Entry State
b)	———	A 2 1 8	The Comparison Between the Index and the Final Value is Made. The Comparison is of False Condition. Variable A is Incremented and the Interpreter's Scanner Pointer is Set to 8. The Carriage Return was the Next Symbol and Triggered the NEXT Verb's Execution

Interpreter's Scanner Pointer	Verb Stack	Value Stack	Comments
8	NEXT	A 2 1 8	NEXT Verb is Pushed on the Verb Stack
9 a)	NEXT	A 2 1 8 A	A Pointer Pointing to the Value of A which is Located in the Variable Table is Pushed on the Value Stack. This Pointer Value is Checked to Insure that it is the Same Pointer Value as the Index's
b)	———	——	A Comparison of the Index and Final Value is Made. The Index is Greater Than or Equal to the Final Value so True Condition is Found. Variable A is Incremented by 1, the Step, and the Scanner Pointer is Advanced to 10. 0D Signals the End of the Command Sequence and the Language System Enters the Entry State

Most BASICs always increase or decrease the index after the comparison regardless of how it turns out.

The FOR-TO-STEP/NEXT and GOSUB/RETURN are part of a family of verbs used for program flow control. Also included in this class is the GOTO, IF THEN, IF THEN ELSE, ON GOTO, ON GOSUB, ON GOTO ELSE, and ON GOSUB ELSE verb sequences. They all work in a manner similar to the FOR-TO-STEP/NEXT and GOSUB/RETURN. Care should be exercised when searching the program text region of memory for line number labels which are different from line number references.

A line number label will occur at either the beginning of the program text region of memory, or will be preceded by a hexadecimal 0D, a carriage return.

The LOCATION for the pointer value 8 has value 04 indicating the Command Buffer. Its frame TYPE is 02, a binary pointer.

CHARACTER STRING PROCESSING

Processing character strings is more complicated than processing numeric variables or flow control verbs. One of the reasons for this is that character strings can be read from the

1. Program Text Region
2. Variable Table
3. Work Buffer
4. Command Buffer
   and, but undesirably, the
5. Value Stack.

Character strings can be written to the

1. Variable Table
2. Work Buffer
   and, but undesirably, the
3. Value Stack.

Character strings can be very long. Placing them on the value stack is expensive because of duplicated string storage; it is also often unnecessary.

Character strings are defined in terms of three pieces of information:

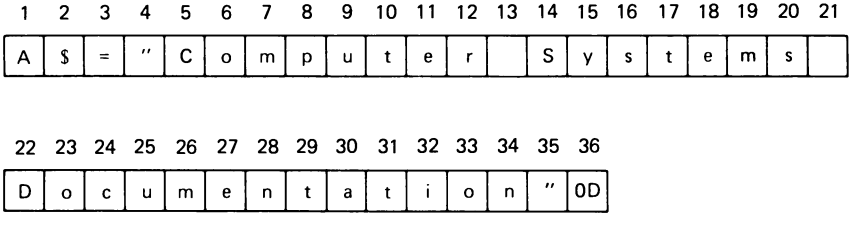
1. A pointer pointing to the beginning of the string.
2. The starting position of the string.
3. The length of the string.

1 is usually a binary pointer pointing to the beginning of the string; 2 and 3 are both numbers.

The command

```
A$="Computer Systems Documentation"
```

is stored in the Command Buffer



The LOCATION of the string "Computer Systems Documentation" is the Command Buffer. Its TYPE is a character string. The pointer to the string has a value of 5. The "C" is the start of the character string and has numeric position value 1. The length of the string is 30. This can be verified by counting or by calculation of $34-5+1=30$. The beginning " and ending " are not part of character strings.

Some BASICs allow " to appear between those of the beginning and ending"s. This is accomplished by removing one " when two " (" ") appear together. The command PRINT "This is a " " would cause This is a " to be printed on the console output device.

In summary, strings are defined by a LOCATION and a triple in the form (Binary pointer, number 1, number 2). Number 1 must be in the range 1 to the length of the string. Number 2 must specify a length which does not run beyond the terminating ". The triple (5,10,7) specifies the string (sometimes called a substring):

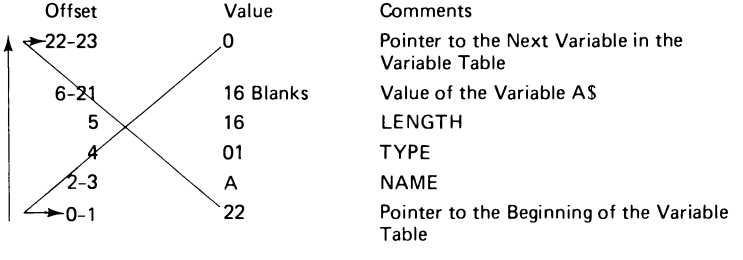
Systems

The triple (5,22,6) defines the string

mentat

The triple (5,28,4) is invalid. Position 28 contains the i, and a string of length 4 would be ion". This is invalid for the reason that the length overruns the terminating ".

The variable table structure for the command containing A$ is:



The value stack expands backward through memory. Working a detailed example of execution of the string command in this "backward" mode would be confusing. For this reason a rough step by step command execution will be shown allowing the value stack to expand in a forward direction.

The interpreter's scanner pointer is pointing to the first location in the Command Buffer.

Scanner Pointer	Verb Stack	Value Stack	Length	Comments
1-2	———	08	2	CHAIN: 8 Byte Subframe Length
		01	1	FRAME: Variable Name Pointer
		02	1	LOCATION: Variable Table
		2	4	Pointer to A$ Name in Variable Table

The value stack subframe points to the name of A$ in the variable table. The LENGTH of A$ will have to be consulted when the assignment is made and thus a pointer to its value is insufficient.

Extraordinary measures should be taken so that character strings themselves do not appear on the value stack as these strings can be many thousands of bytes long. If at all possible only use pointers, starting values, and string lengths to describe character strings.

Scanner Pointer	Verb Stack	Value Stack	Length	Comments
3	=	08	2	The "=" Verb is Placed on the
		01	1	Verb Stack
		02	1	Subframe Pointing to the Name A$
		2	4	in the Variable Table
4-35 a)	=	08	2	
		01	1	Subframe Pointing to the Name A$
		02	1	in the Variable Table
		2	4	
		16	2	CHAIN: 16 Byte Subframe
		02	1	TYPE: Character String Pointer
		04	1	LOCATION = Command Buffer
		5	4	Pointer to Begining of Character String
		1	4	Start Position of String
		30	4	Length of String
b)	———	———		The Carriage Return Triggers an Execute of the =

Character strings are usually moved, one by one, from left to right, from the source to the destination. In this case the source is the string "Computer Systems Documentation" and the destination is A$. Here the length of the destination is shorter than the length of the source. Thus, if PRINT A$ were executed,

Computer Systems

would be printed. This string contains 16 characters.

If the destination length is longer than the source, it is customary to fill the unused trailing character positions with blanks.

The string function STR(, MID$(, SEG$ are the same functions but with different names depending on the BASIC used. MID$(has the form:

MID$(String variable name, Start position of string, Length of string).

The string variable name in some BASICs can be an array name; for example A$( ).

Let

A$ = "ABCDEFGH"

then:

MID$(A$,3,2) is "CD"
MID$(A$,1,4) is "ABCD"
MID$(A$,5) is "EFGH"
MID$(A$,10,7)
↑ ERR because A$ has length of 16
MID$(A$,7,4) is "GH "

Realization has come to some designers of BASIC that if MID$(is allowed to appear on the left hand side of the equal sign, then verbs such as LEFT$(and RIGHT$(are unnecessary. Several examples are (always beginning with A$ = "ABCDEFGH")

MID$(A$,2,3) = MID$(A$,5) gives A$ = "AEFGEFGH"
MID$(A$,4,4) = MID$(A$,2,2) gives A$ = "ABCBCFGH".

The command:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
M	I	D	S	(	A	S	,	4	,	4	)	=	M	I	D	S	(	A	S	,

22	23	24	25	26
2	,	2	)	CR

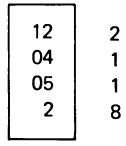
is processed:

Step	Verb Stack	Value Stack	Length	Comments
1-5	MID$(			
6-8	MID$(	08	2	CHAIN: 8 Byte Subframe
		01	1	FRAME: Pointer to Name of A$
		02	1	LOCATION = Variable Table
		2	4	Pointer to the Name A$
9-10 a)	MID$(	08	2	Pointer to the Name A$ in the Variable Table
		01	1	
		02	1	
		2	4	
		16	2	CHAIN: 16 Byte Subframe
		02	1	FRAME: Character String Pointer and Length and Start
		04	1	LOCATION: Command Buffer
		9	4	Pointer to the "4"
		1	4	Start of the "4"
		1	4	Length of the "4"
9-10 b)	MID$(	08	2	Pointer to the Name A$ in the Variable Table
		01	1	
		02	1	
		2	4	
		12	2	CHAIN: 12 Byte Subframe
		04	1	FRAME: Numeric Value
		05	1	LOCATION: Value Stack
		4	8	Eight Byte Numeric "4"
11-12 a)	MID$(	08	2	Pointer to the Name A$ in the Variable Table
		01	1	
		02	1	
		2	4	
		12	2	Numeric Value of 4 on Value Stack
		04	1	
		05	1	
		4	8	

Step	Verb Stack	Value Stack	Length	Comments
11-12 a) — Continued		16	2	CHAIN: 16 Byte Subframe
		02	1	FRAME: Character String Pointer and Length and Start
		04	1	LOCATION: Command Buffer
		11	4	Pointer to the Second "4"
		1	4	Start of the "4"
		1	4	Length of the "4"
11-12 b)	MID$(	08	2	Pointer to the Name A$ in the Variable Table
		01	1	
		02	1	
		2	4	
		12	2	Numeric Value of 4 on Value Stack
		04	1	
		05	1	
		4	8	
		12	2	Numeric Value of 4 on Value Stack
		04	1	
		05	1	
		4	8	
11-12 c)	———	16	2	CHAIN: 16 Byte Subframe
		02	1	FRAME: Character String Value Pointer Length and Start
		02	1	
		6	4	LOCATION: Variable Table
		4	4	Pointer to Value of A$
		4	4	Start Within A$
				Length of String Within A$
13	=	16	2	Pointers to the Start of the Substring
		02	1	Within A$ and Length of the Substring
		02	1	
		6	4	
		4	4	
		4	4	
14-18	=	16	2	Pointer to the Start of the Substring
	MID$(	02	1	Within A$ and Length of the Substring
		02	1	
		6	4	
		4	4	
		4	4	

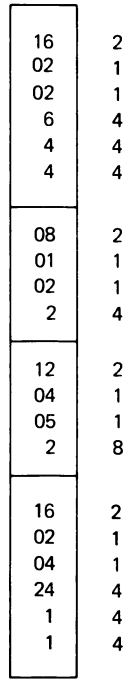
Step	Verb Stack	Value Stack	Length	Comments
19-21	=	16	2	Pointers to the Start of the Substring
	MID$(	02	1	Within A$ and Length of the Substring
		02	1	
		6	4	
		4	4	
		4	4	
		08	2	Pointer to the Name of A$ Located in
		01	1	the Variable Table
		02	1	
		2	4	
22-23 a)	=	16	2	Pointers to the Start of the Substring
	MID$(	02	1	Within A$ and Length of the Substring
		02	1	
		6	4	
		4	4	
		4	4	
		08	2	Pointer to the Name of A$ Located in
		01	1	the Variable Table
		02	1	
		2	4	
		16	2	Pointers and Length to the "2" Beginning
		02	1	in Position 22 of the Command Buffer
		04	1	
		22	4	
		1	4	
		1	4	
22-23 b)	=	16	2	Pointers to the Start of the Subtsring
	MID$(	02	1	Within A$ and Length of the Substring
		02	1	
		6	4	
		4	4	
		4	4	
		08	2	Pointer to the Name of A$ Located in
		01	1	the Variable Table
		02	1	
		2	4	

22-23 b) – Continued



Numeric Value of 2 on Value Stack

24-25 a)



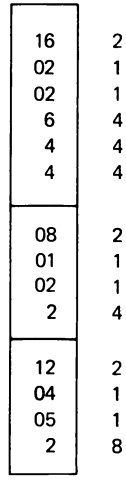
Pointers to the Start of the Substring Within A$ and Length of the Substring

Pointer to the Name of A$ Located in the Variable Table

Numeric Value of 2 on Value Stack

Pointers and Length of the "2" Beginning in Position 24 of the Command Buffer

24-25 b)



Pointers to the Start of the Substring Within A$ and Length of the Substring

Pointer to the Name of A$ Located in the Variable Table

Numeric Value of 2 on Value Stack

24-25 b) – Continued		12	2	Numeric Value of 2 on Value Stack
		04	1	
		05	1	
		2	8	
24-25 c)	=	16	2	Pointers to the Start of the Substring
		02	1	Within A$ and Length of the Substring
		02	1	
		6	4	
		4	4	
		4	4	
		16	2	Pointers to the Start of the Substring
		02	1	Within A$ and Length of the Substring
		02	1	
		6	4	
		2	4	
		2	4	
26	———	———		The CR Triggers Evaluation of the "="

Steps 11–12c and 24–25c are important because the MID$(verb must check the LENGTH stored with A$ to ensure that the new start and length are valid.

Expressions such as MID$(A,2+B*(COS(C+D)) are permitted. Inclusion of the variable FRAME and LOCATION in stack frames are required for evaluation of expressions of the type given in the previous sentence.

Storing intermediate number results on the value stack was permitted in numerical computations. In fact, this operation was quite necessary. The question arises of whether it is ever necessary to store intermediate results with character string manipulation. The answer is "Almost, but not quite".

Let A$="ABCD" and B$="EFG". The verb "concatenate" is often denoted by "&". PRINT A$&B$ would cause:

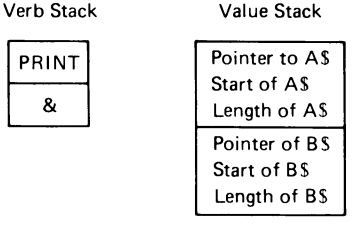
ABCDEFG

to be printed. PRINT B$&A$ would cause:

EFGABCD

to be printed.

A rough conceptual diagram of the verb and value stacks at a time immediately previous to evaluation of the PRINT and & in PRINT A$&B$ is:



Two choices are available:

1. Evaluate the & and move "ABCDEFG" onto the value stack, or
2. Evaluate the & and PRINT in one step and thus move the "ABCD" then the "EFG" to the work buffer for output.

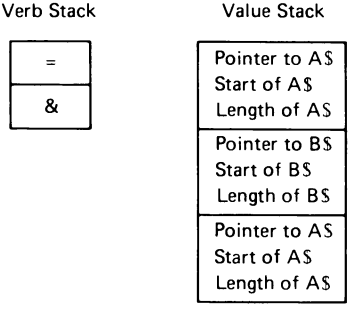
Choice number 1 is simpler, but number 2 may be preferable from a storage requirements standpoint. In either case, the stack frames are the same; the only difference is in processing them.

PRINT "AB"&"CD"&"EF" is also allowable so the & processing verb, which searched backwards through the verb stack, would have to be reasonably complicated.

An even more complicated problem with character string concatenation occurs with the command:

A$ = B$&A$

which has verb and value rough conceptual stack frames:



The simplest procedure would be to place the concatenated B$&A$ on the value stack. It could then be moved to A$.

Some BASIC implementers would, in this case, shift A$ to the right by three positions to get:

ABCABCD

then move B$ into A$, which gives:

EFGABCD

Such techniques do not require intermediate storage of character strings, but the tradeoff is increased complexity of the Language System's interpreter.

Because of the intermediate storage problem with character strings, most BASICs do not allow parentheses in string operations. However, seemingly complex string operations such as

A$,STR(B$( ),20,C) = AND D$ & HEX(FF00EE11) ADDC(05)

where ADDC is a binary add with a carry, and AND is a bitwise logical, are easily evaluated by an interpreter using the techniques previously described in this chapter.

BASICs contain a READ statement which works in a somewhat different manner than other BASIC instructions.

The BASIC program

```
10FORA = 1TO2
 :READ A$
 :NEXTA
 :DATA"ONE","TWO"
```

is atomized:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
FF	00	10	9E	A	=	1	B2	2	:	98	A	$	:	9D	A	:	97

19	20	21	22	23	24	25	26	27	28	29	30
"	O	N	E	"	,	"	T	W	O	"	0D

where FF is the hexadecimal delimiter for a packed line number, 9E for FOR, B2 for TO, 98 for READ, 9D for NEXT, and 97 for DATA. The

atomization table is found in Appendix A.

Immediately before the Program Text Region of memory is a memory area called the "Bookkeeping" area. Several different types of tables are kept in this area. These tables pertain to the BASIC program which directly follows them. One such table located in this area is called the DATA pointer table. The DATA pointer table has the form:

READ Index
Location
Pointer to Indexed Datum
Length of Datum

When the above program is RUN and the first READ executed, the DATA pointer table reads:

1
02
20
3

where 1 is the index, an eight byte number. 20 is a binary pointer pointing to the variable table which is LOCATION 02. 3 is the length of the character string "ONE". DATA statements can be used in commands so allowance must be made to specify a LOCATION.

Whenever a READ is executed, the READ index is incremented. A search is begun for a DATA element. If a valid element is found, then a pointer to its start and its length are placed in the DATA pointer table. If a valid data element is not found, then a READ error is signaled.

When the second read in the BASIC program is executed, the DATA pointer table would read:

2
02
26
3

The index, LOCATION, and text pointer are pointing to the second data item, the "TWO".

The RESTORE command/statement is often of the form:

	Examples
1. RESTORE	RESTORE
2. RESTORE <expression>	RESTORE 2
3. RESTORE LINE = <Line Number, [expression]>	RESTORE LINE = 10,2

where the angular brackets denote required information and the square brackets indicate optional information. The value of the expression in form 2 must be equal to, or greater than, 1.

When a RESTORE is executed, the DATA pointer table is filled with the index and other information pointing to a valid data item.

READ/RESTORE/DATA are important because some information required for their processing is obtained from the Bookkeeping area of memory.

Many BASICs allow GOSUB's in marked or labelled subroutines. An example of a marked GOSUB is:

```
10 GOSUB'5
  : STOP
20 DEFFN'5
  : RETURN
```

The interpreter processes GOSUB'5 by searching the program text region of memory for '5 in order to locate the entry point of the subroutine.

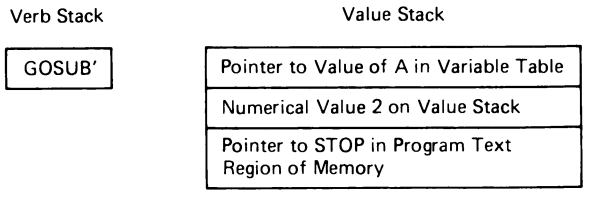
An example of a similar labelled subroutine is:

```
10 GOSUB FIVE
  : STOP
20 DEFFN FIVE
  : RETURN
```

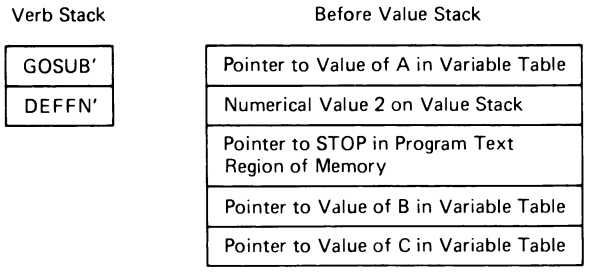
Arguments often can be passed into, but not out of, marked subroutines. An example is:

```
10 GOSUB'1 (A,2)
  : STOP
20 DEFFN'1 (B,C)
  : RETURN
```

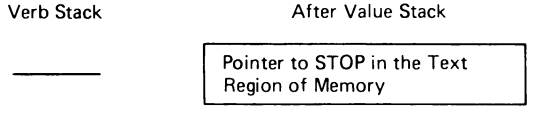
In this example the GOSUB'1 creates the stack frames:



The DEFFN' creates the stack frames:



and causes a "create" and execute, which copies the values of A into the value of B, and the value 2 into the value of C. The DEFFN' causes creation of the "after" stack frames:



This "after" stack frame is precisely what the RETURN expects.

BASICs generally do not allow arguments to be passed back from subroutines. Consider the *compiled* BASIC program:

```
10      GOSUB'3 (4)
 :      PRINT 4
 :      STOP
20      DEFFN'3 (A)
 :      A = 2  <----  The Value of A is Passed Back
 :      RETURN
```

might cause 2 to be printed rather than the intended value of 4.

BASICs, particularly interpreted ones, cannot handle the problem of passing back variable values gracefully because of the executing verb nature of the DEFFN', the line by line execution mode, and the structure of the variable table.

SUMMARY

A Language System comprises five states:

1. Language System initialization
2. Entry phase
3. Resolution phase
4. Interpreted Execution phase
5. Language System self test

Interpreted execution means that a BASIC program is executed by a Language System program called the interpreter. The interpreter scans BASIC program text and invokes appropriate Language System modules which cause the program to be executed. The interpreter uses two stacks: the verb stack and the value stack. The value stack frame format used for BASIC program execution contains the variable attributes of CHAIN, FRAME, LOCATION, and VALUE.

The work buffer stores commands and statements at the entry phase. Execution of a command sequence is processed in the command buffer by the Language System. Program statements are executed in the program text region of memory.

Character string data can be fetched or read from the program text, variable table, work buffer, command buffer, and value stack regions of memory; numeric data can only be fetched or read from the variable table and the value stack.

Two important points of program execution are:

1. The attributes of CHAIN, FRAME, and LOCATION need to be placed on the value stack along with a pointer to variable value or the value itself.
2. The Language System verb processing modules must have sufficient "intelligence" to determine which type of data needs to be processed.

Because character strings can be very long, processing them is more complicated than processing numeric variables or flow control verbs. Placing them on the value stack is expensive because of duplicated string storage. If at all possible, use only pointers, starting values, and string lengths, in addition to the CHAIN, FRAME, and LOCATION, to describe character strings in the value stack.

The verbs GOSUB/RETURN and FOR-TO-STEP/NEXT operate in pairs. Execution of the first of a verb pair causes the verb and value stacks to be loaded with some information. Execution of the second of the verb pair may cause the information on the verb and value stack to be removed.

The RETURNCLEAR verb is valuable because it allows control to be returned to some program other than the calling program.

READ/RESTORE/DATA are important because some information required for their processing is obtained from the Bookkeeping area of memory, located immediately before the program text area.

9

Compiled BASICs

A BASIC compiler is a computer program which converts a BASIC program into a program written in another language. The BASIC Language System's interpreter is one type of program that executes the applications programmer's program. One of the major goals of the compiler is to eliminate the interpreter's complex analysis of the applications program text during execution. This can be done by converting the source program to "Reverse Polish". The relative advantages and disadvantages of the complied BASIC and the BASIC Language System's interpreter are evaluated in this chapter.

Some seeming objections to interpreted execution, as it was described in the previous chapter, are:

- The work required to analyze the program text and place it on the verb and value stacks appears to be excessive. This is particularly true when evaluating a complex arithmetic expression in the middle of a loop.
- Searching the variable table for a value each time the variable is accessed appears wasteful in the time it consumes.
- Searching the program text region of memory each time a line reference appears to be a slow process.
- Requiring processing programs for those verbs not used in a program seems wasteful with memory.
- Retaining variable names and array bounds in the variable table also appears wasteful with memory, since they may not be used during program execution.

Compiled BASICs seek to remedy some of the seeming problems of interpreted BASICs.

The compiler approach to BASIC is to process source BASIC program statements and produce an output program that performs the functions of the source BASIC. The output BASIC may not very closely resemble the source BASIC.

The two major goals of the compiler are to:

1. Stop searching for variables and line references during execution.
2. Stop the complex analysis of the program text during execution.

There are many different kinds of compilers that, with varying success, would accomplish these goals. The second goal can reasonably be met by converting the source program to what is called "Reverse Polish". How this conversion to Reverse Polish is carried out is best explained by a rough conceptual diagram. One stack and one queue are used:

1. The operator stack.
2. The output queue.

The statement:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	E	=	(	A	+	B	)	*	(	C	-	D	)	0D

is processed by a compiler

Step	Operator Stack	Output Queue	Comments
1	—		The Line Number is Skipped Over
2	—	Variable E	Label and Push Variable on Output Queue
3	=	Variable E	Push Operator on Operator Stack
4	= (	Variable E	Push "(" Operator on Operator Stack
5	= (	Variable E Variable A	Label and Push Vaiable or Output Queue
6	= (+	Variable E Variable A	Push "+" Operator on the Operator Stack

Step	Operator Stack	Output Queue	Comments
7-8 a)	=	Variable E	Label and Push the Variable E on the Output Queue
	(	Variable A	
	+	Variable B	
b)	=	Variable E	The ")" Causes a Process, Similar to an Interpreter "Execute", which Moves the Operator to the Output Queue
		Variable A	
		Variable B	
		Operator +	
9	=	Variable E	Push "*" on the Operator Stack
	*	Variable A	
		Variable B	
		Operator +	
10	=	Variable E	Push Operator "(" on the Operator Stack
	*	Variable A	
	(	Variable B	
		Operator +	
11	=	Variable E	Label and Push Variable C on the Output Queue
	*	Variable A	
	(	Variabel B	
		Operator +	
		Variable C	
12	=	Variable E	Push Operator "–" on the Operator Stack
	*	Variable A	
	(	Variable B	
	–	Operator +	
		Variable C	

Step		Operator Stack	Output Queue	Comments
13-14	a)	=	Variable E	Label and Push Variable D on Output Queue
		*	Variable A	
		(	Variable B	
		−	Operator +	
			Variable C	
			Variable D	
	b)	=	Variable E	The ")"Caused a Process which Moved "−" to the Output Queue
		*	Variable A	
			Variable B	
			Operator +	
			Variable C	
			Variable D	
			Operator −	
	c)		Variable E	The End-of-Line Causes the Remainder of the Operator Stack to be Transfered to the Output Queue
			Variable A	
			Variable B	
			Operator +	
			Variable C	
			Variable D	
			Operator −	
			Operator *	
			Operator =	

This part of the compile process is similar to both syntax analysis and interpreter execution. It appears that the verb is placed on the value stack when it is to be executed. This is a rough analogue to what the compiler does.

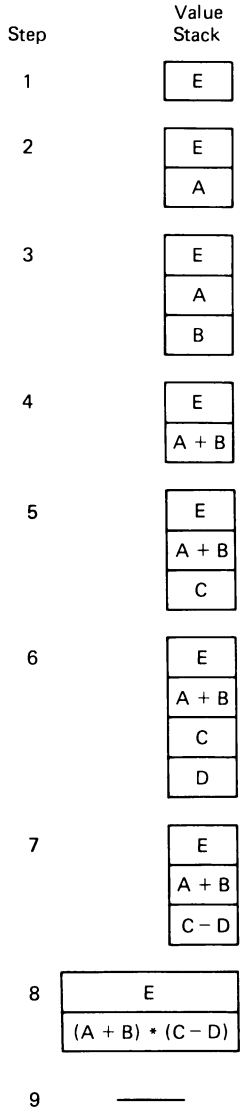
The output queue can be written

1	2	3	4	5	6	7	8	9
E	A	B	+	D	C	−	*	=

and said to be in "Reverse Polish".

Execution of the Reverse Polish queue requires that a value stack, but not a verb stack, be maintained. The reason is that a verb is immediately executed when it is encountered.

The execution steps of this queue are:



The reader might ask the question, "Wouldn't BASIC be simple to implement and efficiently executable if statements and commands were

written in Reverse Polish?" The answer is: "Yes." The FORTH language requires commands and statements to be written in Reverse Polish. FORTH maintains a value stack and one other stack for GOSUB/RETURN and FOR–TO–STEP/NEXT-type statement/command return information.

Construction of the variable table for a compiled BASIC program is carried out while the program text is being separated onto the operator stack and output queue and then recombined on the output queue.

The compile steps for the BASIC program

1	2	3	4	5	6	7	8	9	10
A	=	A	*	2	/	B	+	2	0D

are roughly diagrammed

Compile Step	Operator Stack	Output Queue	Variable Table Name	Variable Table Offset	Variable Length
1	—	A	A	0	8
2	=	A	A	0	8
3	=	A A	A	0	8
4	= *	A A	A	0	8
5 a)	= *	A A 2	A 2	0 8	8 8
b)	=	A A 2 *	A 2	0 8	8 8
6	= /	A A 2 *	A 2	0 8	8 8

Compile Step	Operator Stack	Value Queue
7 a)	=	A
	/	A
		2
		*
		B
b)	=	A
		A
		2
		*
		B
		/
8	=	A
	+	A
		2
		*
		B
		/
9	—	A
		A
		2
		*
		B
		/
		2
		+
		=

Variable Table Name	Variable Table Offset	Variable Length
A	0	8
2	8	8
B	16	8

Variable Table Name	Variable Table Offset	Variable Length
A	0	8
2	8	8
B	16	8

Variable Table Name	Variable Table Offset	Variable Length
A	0	8
2	8	8
B	16	8

Variable Table Name	Variable Table Offset	Variable Length
A	0	8
2	8	8
B	16	8

At step 1, the variable table is searched for the variable name A. A is not found, so it is entered into the variable table. A has a length of eight bytes. A begins at relative position 0 in the variable table. Constants and literal strings are treated much like variables. The compiler searches

the variable table for "2" at step 5. It is not found, so its name is entered into the variable table and its value begins at relative location 8 in the table. "2" is assigned a length of eight bytes.

At step 7 the variable B is processed. B is not found in the variable table, so it is added to the variable table and its value begins at relative position 16. B's value occupies eight bytes.

At steps 3 and 9, both A and "2" were found in the variable table by the compiler. Thus, the compiler did not have to add a new variable in the variable table for these two symbols.

By making only pointer references, compilers usually remove all references to variables by name. In this example, A is referred to by 0, "2" by 8, and B by 16. The variable table may be viewed:

Relative Location	Contents	Comments
0	00000000	Value of A
8	00000002	Value of "2"
16	00000000	Value of B

The symbolic compiled program queue is:

A	A	2	*	B	/	2	+	=

but the compiler would change this to:

Push 0	Push 0	Push 8	*	Push 16	/	Push 8	+	=

where "push" indicates that the value or pointer to the value of the position in the variable table is pushed onto the value stack.

The hardware of some computers is not stack oriented. Reverse Polish queues are not evaluated according to one circumscribed method. Some compilers produce very efficient output code while the output code of other compilers must be interpreted on a host computer.

The above example gives a very rough picture of what a compiler does. One discrepancy between how a compiler actually works and what is shown in the example it is the location of "2" in the variable table. Compilers allocate space at the beginning of the variable table to those variables that must be assigned initial values other than 0 or blanks. The allocation is done in this fashion because the compiled program module need not contain the entire variable table. Only those variables which are assigned nonstandard (other than 0 or blanks) need to have values included in the variable table.

The variable table for compiled BASIC must be organized in a manner similar to that of a Language System's variable table. A compiler's variable table usually begins directly after the compiled program and expands toward the bottom of memory.

Compiled BASICs require the BASIC programmer to run a program, called the BASIC compiler, which accepts as input a BASIC program, and produces as output the BASIC object code (the text in the output queue), a listing of the BASIC program, and a listing of compiler messages. Requests to run the BASIC compiler are made to a program called an *Operating System.*

An Operating System has as its primary functions to:

1. Schedule program execution.
2. Perform Input/Output for programs.
3. Assist with file oriented commands.

Once BASIC compilations are completed, another program called the *Linkage Editor* must be run; this is accomplished by making a request to the Operating System. A primary function of the linkage editor is that of linking the code required to execute some of the BASIC program's verbs. Functions such as COS(, SIN(, . . . and even MID$(are not generally compiled into the BASIC object code. Subroutine calls are made to many of these called functions. These subroutines reside in a file of systems subroutines, sometimes called the *Systems Library*. Only those subroutines which are called by the BASIC program are linked to it. This is different from a Language System because that computer code, which is used to evaluate all verbs, is always resident in a computer running a Language System.

When a BASIC object program has undergone a successful linkage edit, a program *loader* is run. This loads the BASIC object program and its system supplied ancillary routines into computer memory in preparation for a BASIC program execution by the computer.

A request to the Operating System to run the BASIC program can now be made. In summary, the steps of

1. compile
2. linkage edit
3. load
4. run

must be made in sequence to cause a BASIC program to execute in a BASIC compiler environment.

Should the BASIC program cause itself to be terminated on an error condition, then control is returned to the Operating System. The Operating System has the responsibility of informing the user what caused the pro-

gram to terminate. Language Systems simply display the offending statement with an arrow pointing to the offending symbol within the statement. Many Operating Systems do not have the capability of reconstructing BASIC source code from BASIC object code. The source symbols are often not kept with the object code, so reconstruction of the source code is not possible.

Writing a program to reconstruct source code from object code is a difficult matter. The key to solving this problem is to analyze object code queue output by looking at both ends of the queue.

Operating System compiled BASICs are orders of magnitude more complicated than Language System interpreted BASICs.

Writing a BASIC compiler requires about 5,000 to 20,000 lines of computer code. The linkage editor and loader are less complicated to write, but still require many thousands of lines of computer code. Depending on its level of sophistication, the Operating System may require many thousands of lines of code to implement. Operating System modules are usually kept small, and are fetched off disk only when they are needed to provide a service for an executing BASIC program.

Requests for services from the Operating System are made by entering *Job Control Language* (JCL) commands. Issuing JCL commands within an executing BASIC program is sometimes very difficult. JCL's can be very complicated and can resemble a computer language. Some of these JCL's are not unlike an interpreted BASIC Language System. In fact, almost all JCL's are interpreted. BASIC Language Systems incorporate most operating system JCL functions within the BASIC language. JCL verbs are added to the BASIC language in Language Systems.

Compiled BASICs meet the goal of reducing the amount of work expended on language overhead during program execution.

Compiled BASICs are only partially successful in reducing the amount of computer memory used during program execution. The amount of memory required for a Language System executing any sized BASIC program is always the same. Neither the length of the BASIC program, nor the statements it contains, influence the size of a Language System. When a BASIC program is executing, Operating Systems use very little memory space and they load only those verbs required for execution into memory. Speed of execution is partially obtained by duplicating code for verbs, such as +, −, * and /. Practical experience has shown that memory requirements are less for compiled BASIC short programs, but greater for large BASIC programs compared to an interpreted BASIC Language System. A rough diagram of this relationship is shown in Figure 33. Language System overhead is great for small programs while compiler code duplication is great for longer programs.

Developing a program on an Operating System compiled BASIC takes much longer than it does on a Language System interpreted BASIC, making compiled BASIC computer code more expensive to use than computer interpreted BASIC code. Some of the reasons for this difference are accounted for in these comparisons:

	Compiled BASIC	Interpreted BASIC
1	Edit-compile-linkage edit-load-run program text revision cycle	Edit-run program text revision cycle
2		Source statement in error displayed
3		Single step source execution possible
4		Interactive variable examination and change used in combination with single step execution.
5		Extensive run time error checking (i.e. array bounds checking)

Comparisons 2–5 on the compiled BASIC side were left blank for the reason that while most compiled BASICs do not provide the services

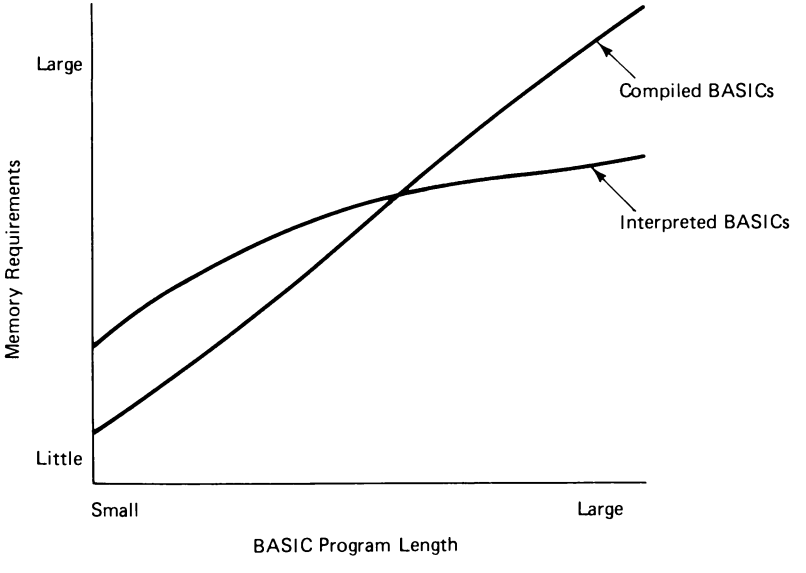


Figure 33: Rough diagram plotting the total memory requirement for compiled and interpreted BASICs as a function of the length of BASIC programs. Overhead of the BASIC interpreter is great for small BASIC programs. Code duplication becomes a significant factor for large compiled BASIC programs.

routinely provided by interpreted BASICs, there is no reason, other than complicated programming, why they cannot provide such services.

Language Systems or Operating Systems BASICs that offer such program development services are called "friendly."

Operating System BASICs often make it difficult for job control language statements to be issued within an executing BASIC program. They also often fail to provide good program development services. For these reasons, Operating Systems have been viewed by some application BASIC programmers as "unfriendly".

Compiled BASIC programs exceed Language System BASICs in speed of execution and in reduced memory requirements for short BASIC programs. Compilers are usually so large that total memory requirements for Operating System BASICs are equal to those of a Language System.

Interpreted BASICs can be made to run about the same speed as compiled BASICs. This is accomplished by employing several different processors (within the computer) to analyze the BASIC text, and to search for variables and line number reference labels.

In most cases, a systems disk must be mounted each time an Operating Systems BASIC is running. A Language System is completely resident at all times, so all disk drives can be used for applications programs and data.

The point of these comparisons between Operating Systems and Language Systems BASICs is to emphasize that there are two distinct categories of BASICs. There are advantages to both types of systems. The greatest advantage of Operating Systems is that they allow languages other than BASIC to be run on a computer. A Language System restricts the programmer to two languages. One of these might be BASIC, or COBOL, or ADA, or FORTH, or ?!; the other would be microcode or machine language.

SUMMARY

The compiler approach to BASIC is to process source BASIC program statements and produce an output program which performs the functions of the source BASIC. The output BASIC may not very closely resemble the source BASIC.

Two major goals of the compiler are to

1. Stop searching for variables and line references during execution.
2. Stop the complex analysis of program text during execution.

The second goal can reasonably be met by converting the source program before execution to what is called "Reverse Polish". Converting

to Reverse Polish requires an operator stack and an output queue. This compile process is similar to both syntax analysis and interpreter execution. Execution of the Reverse Polish output queue requires that a value stack, but not a verb stack, be maintained. The reason is that a verb is immediately executed when it is encountered. The construction of the variable table for a compiled BASIC program is carried out while the program text is being separated onto the operator stack and output queue and then recombined on the output queue. By making only pointer references, compilers usually remove all references to variables by name. A compiler's variable table commonly begins directly after the compiled program, and expands toward the bottom of memory.

Requests to run the BASIC compiler are made to a program called an Operating System. The Operating System:

1. schedules program execution
2. performs Input/Output for programs
3. assists with file oriented commands.

After a request has been made to the Operating System, the following steps:

1. compile
2. linkage edit, which links code required to execute some of the BASIC program's verbs
3. load, which loads the BASIC object program in preparation for execution
4. run

must be made in sequence to cause a BASIC program to execute in a BASIC compiler environment.

Compiled BASICs do meet the goal of reducing the amount of work expended on language overhead during program execution. Compilers are usually so large that total memory requirements for Operating System BASICs are equal to those of a Language System. Memory requirements are less for compiled BASIC short programs, but greater for large BASIC programs. Language System overhead is rather substantial for small programs, while compiler code duplication incurs larger costs for longer programs.

The greatest advantage of Operating Systems is that they allow languages other than BASIC to be run on a computer. The BASIC Language System restricts the programmer to two languages at a time.

10

Verb Failures, User-Defined Verbs, and BASIC Line Editor

The information presented in this chapter deals with two different classes of verbs, the methods of handling verb failures, three types of interactive input verbs, and a BASIC line editor.

VERBS AND VERB FAILURES

Each software module should:

1. perform a simple intended function;
2. minimize the possibility of performing benign or adverse unintended functions; and
3. provide adequate warning in the event of failure.

Each verb should return a *status* when it returns to the interpreter. The values of the status may range from indication of a successful termination to notification of minor difficulties discovered. If a verb execution totally fails, then control may be returned to some part of the Language System other than the interpreter. Possible verb execution failures must be checked at run time.

Verbs in Language Systems fall into two classes:

1. BASIC Language System implemented verbs.
2. User-defined machine language or microcode implemented verbs.

The methods of returning verb status and handling total verb failures are slightly different for the two classes of verbs.

The intended function of the BASIC program:

```
10 INPUT "A = ",A: B=1/A: PRINT "1/A = ";B:GOTO10
```

is to compute the reciprocal of a number and then print this value. An adverse unintended function would be that of stopping the program's execution. If the value of 0 is entered for A, then:

```
 A = ? 0
10 INPUT "A = ",A: B=1/A: PRINT "1/A = ";B:GOTO10
                           ↑ ERR
:
```

would appear and the Language System would return to the entry state. Division by zero caused the "/" verb to fail.

Some BASICs incorporate an ERROR flag and ERR function which are used to allow control to be retained within a BASIC program rather than have execution stopped. The BASIC reciprocal program can be rewritten:

```
10 INPUT "A = ",A: B=1/A: ERROR C=ERR: IF C 27
   THEN STOP "ERR other than 27": PRINT "A was
   0, please enter another value.": GOTO10
20 PRINT "1/A = ";B: GOTO10
```

If the "/" verb does *not* return a failure status, then all of the statements following the ERROR flag on the same line are skipped. If the statement B=1/A fails for any reason, then the remaining statements following the ERROR flag are executed.

Some BASIC Language Systems allow a BASIC programmer to define a verb in terms of either microcode or machine language. An example of such a user-defined verb is

```
EXECUTE #A,(440AA000440C,B$( ))C$( )
```

where A is a variable whose value points to a data path, 440AA000440C is a sequence of microcode instructions, B$( ) is an array which contains microcode register contents, and C$( ) is a source or destination data array. The microcode sequence usually can be placed in a character string and the character string can then be executed in an alternate form of an

EXECUTE verb. (EXECUTE is a BASIC verb which helps the user-defined machine language or microcode verb to execute.) If the EXECUTE verb fails, then verb failure is handled in the normal way by the Language System. Status is returned to the user in B$( ) in the form of dumped computer registers. Some of B$( ) may also be used to input information for the user-defined verb.

Most user-defined verbs are directed to providing software for special purpose input/output devices. These device drivers usually must be written in either machine language or microcode. BASICs provide an easy way to integrate such code into the Language System without making changes in the Language System itself.

Operating System BASICs usually require that device drivers be integrated into the operating system. This is often a complex and time consuming task.

INPUT VERBS

BASICs often have three types of interactive input verbs, which are also BASIC Language System implemented verbs.

1. INPUT
2. LINEINPUT
3. INKEYS or KEYIN

INPUT is used to interactively input numbers into numeric variables. Numbers must be in a valid format or an INPUT error is signaled. LINEINPUT is used to interactively input character string values into a character string variable. INKEYS or KEYIN is used to intercept single characters from an input device interactively.

BASIC LINE EDITOR

BASIC text is edited line-by-line. Entry of a line number is used to recall a line of BASIC text. This line can contain, of course, multiple lines of BASIC code. Some of the functions of a BASIC line editor should include:

1. Movement of the cursor to the right or left
2. Movement of the cursor up or down in a multiple line display line
3. Deletion of characters
4. Insertion of characters
5. Erasing all characters to the right of the cursor
6. Concatenation of two lines of BASIC text.

The editor must also be able to edit command sequences.

The editor can be expanded to edit character strings entered in a LINEINPUT statement.

Input to a Language System or BASIC is on a line basis for BASIC program text entry and edit and for INPUT and LINEINPUT. This means that the Language System does not need to know anything about what is happening in the edit phase until a carriage return is keyed which ends the line of text.

In a nontimesharing Language System editing can be done by the same computer which executes the interpreter. For a Timesharing Language System it is highly desirable to have a separate computer (microprocessor) handle both input and editing. Implementation of an edit, INPUT, or LINEINPUT becomes a matter of checking whether a full line is ready to be moved to the Work Buffer of the Language System. If a full line is not ready, control is passed to the next partition.

BASIC Language Timesharing Systems take great advantage of inexpensive hardware and costly software by distributing the intelligence of the Language System.

Arithmetic, mathematical elementary function, graphics, and audio verbs are often best handled by special purpose microprocessors. Implementation of these verbs on the main Language System computer becomes a matter of implementation of a communications interface with a special purpose microprocessor.

INKEYS or KEYIN cannot be handled on a line basis since each keystroke output must be intercepted and placed in the variable table.

SUMMARY

The BASIC Language System has two classes of verbs: the BASIC Language System implemented verbs, and those implemented by the user-defined machine language or microcode. Most user-defined verbs are directed to providing software for special purpose input/output devices. Each verb should return a status when it returns to the interpreter. If the verb execution totally fails, then control may be returned to some part of the Language System other than the interpreter. EXECUTE is a BASIC verb which helps the user-defined machine language or microcode verb to execute. If the EXECUTE verb fails, then verb failure is handled in the normal way by the Language System.

BASICs often have three types of interactive input statements which are also BASIC Language System implemented verbs:

1. INPUT which is used to interactively input numbers into numeric variables.

2. LINEINPUT which is used to interactively input character string values into character string variables.
3. INKEYS or KEYIN which is used to intercept single characters from an input device interactively.

BASIC text is edited line by line. Entry of a line number is used to recall a line of BASIC text. Command sequences must also be able to be edited. A Timesharing Language System should have a separate computer (microprocessor) to handle both input and editing. Nontimesharing Language System editing can be done by the same computer which executes the interpreter.

11

Timesharing Language Systems

The purpose of Chapter 11 is to explain to the reader how a Timesharing Language System works. In many situations a computer that has the ability to perform several different functions at the same time is desirable. A Timesharing Language System makes this possible.

Most microcomputer systems are used as stand alone systems. Only one BASIC program is executing at one time or one user is using the Language System in its Entry state.

Timesharing Language Systems can be so written as to create the appearance that one physical computer system looks similar to several independent computer systems. This capability is desirable for several reasons:

1. Two or more users can use the same computer simultaneously either interacting with BASIC programs or with the Language System.
2. Two or more BASIC programs can run independently, perhaps cooperatively, on the same computer.

BASIC programs which service such input/output as communications systems must be constantly on the alert for incoming messages. Inclusions of such programs within a large applications program would be undesirable since these communications programs would have to be called after every few BASIC statements within the applications program. These repeated calls are required so that incoming messages are not lost due to inattention. Practice has shown that these communications programs are best left continually running in a separate part of the Language System. These communications BASIC programs are frequently rich with

user-defined verbs which handle the communications low level protocol.

A second example, demonstrating the usefulness of having two programs executing simultaneously, occurs where one program is listing the contents of a file while the other program, the Language System itself, is being used to help a user develop a BASIC program. In other words, the user is both listing a file and developing a BASIC program simultaneously on the same computer.

Understanding how a Timesharing BASIC Language System works is more difficult than comprehending how a single Program Text memory region Language System works. For this reason, it is important to concentrate on understanding only those aspects of the Language System which pertain directly to Timesharing.

The essence of how a Timesharing Language System works is that there are several dissimilar copies of the Other Systems Tables (see Figure

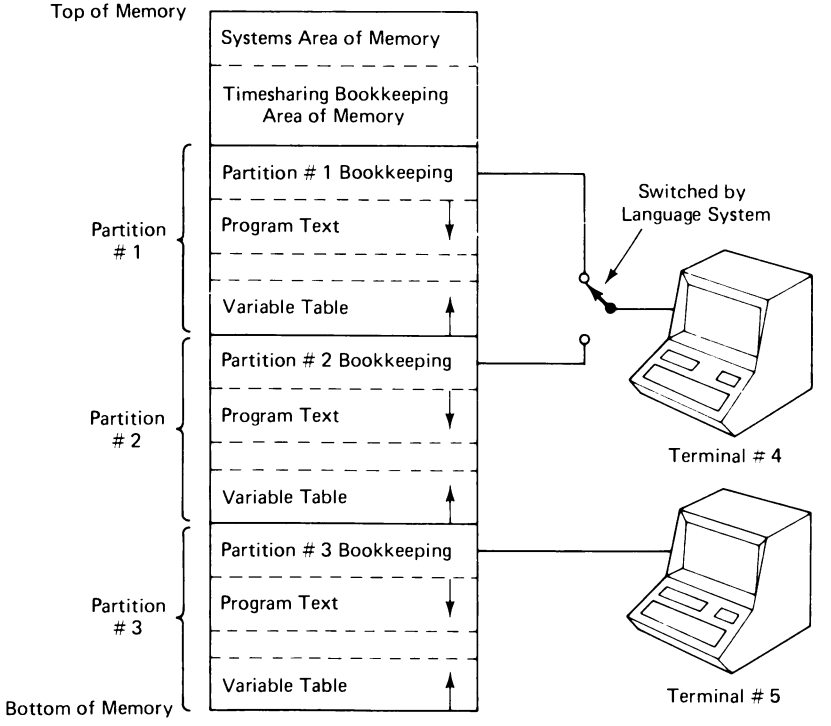


Figure 34: Rough diagram of the memory layout of a Timesharing Language System. A terminal, assigned at partition configuration, is assigned to each partition. Partitions one and two are both assigned to Terminal #4. Terminal #4 is first assigned to Partition #1 but can later be released to Partition #2.

34) through the variable table region of memory. Each of these dissimilar copies is called a *partition*. Each copy has the same overall organization but the contents of each copy differs as a result of the programs being executed or entered in that partition.

The Language System also reserves about a 3K byte memory area at the end of the Systems Area, called the *Timesharing Bookkeeping memory area*. A rough diagram of this memory layout for three partitions is seen in Figure 34. The DATA pointer table resides in the partition Bookkeeping areas of memory; actually, there is one DATA pointer table for each partition. Separate verb stacks and Command Sequence Buffers reside in each partition.

A terminal must be attached to each partition. This terminal is used for console input and output. Verbs such as PRINT, LIST, INPUT, LINEINPUT, or INKEY either attempt to send their output or seek their input from this terminal. If a terminal is attached to more than one partition, then the partition wishing to either send to, or receive from, this terminal will wait until the terminal is attached. A terminal cannot be attached to more than one partition at a time.

SIMPLE TIMESHARING

Explanation of how Timesharing works with Language Systems is best approached by analysis of several examples. The three partitions seen in Figure 34 contain the three programs:

```
5 $RELEASE TERMINAL TO 2
10 A = 1
                              } Partition # 1 Program Text
20 GOTO 10

30 B = 2
                              } Partition # 2 Program Text
40 GOTO 30

50 C = 1
                              } Partition # 3 Program Text
60 GOTO 50
```

Partition #1 and #2 are attached to the same terminal. The terminal is initially attached to Partition #1. When the RUN command is entered, the statement $RELEASE TERMINAL TO 2 causes the terminal to be

attached to Partition #2. The program will continue to run since it does not need a terminal for input or output. The two statements at line numbers 10 and 20 are repeatedly executed.

Terminal #4 is now attached to Partition #2. The program in Partition #2 is caused to begin execution by entering the RUN command.

Terminal #5 is a distinctly different physical terminal from #4. Entry of the RUN command causes the program residing in Partition #3 to begin execution.

All three programs are now executing. It appears that all three programs are executing simultaneously but they are not. Rather, the Timesharing Language System jumps from partition to partition executing at least one, but, perhaps, several lines of program text from each partition.

A more detailed rough diagram of computer memory showing the essential parts of these three partitions is seen in Figure 35. Sample values are entered in some of the tables. The Timesharing Bookkeeping Tables govern control of what partition is currently executing BASIC program text. Control is successively rotated from Partitions 1, 2, 3, 1, 2, 3, 1, ... At least one line numbered statement or group of statements is executed in each partition each time control is passed to that partition. All statements following the line number are executed. This means that more than one statement may be executed before control is returned for another examination of the Timesharing Bookkeeping Tables.

Figure 35 shows execution taking place in Partition #2. Only 15 milliseconds of a maximum of 30 milliseconds has been used executing statements in Partition #2. Statements will continue to be executed in Partition #2 until the time exceeds the maximum alloted time. When this occurs, BASIC program text statements located in Partition #3 will be processed until the time limit is again exceeded.

The Partition Systems Tables contain the pointer to the next instruction to be executed. In this example, this pointer is in terms of the program text coordinate of the next instruction to be executed.

The essence of Timesharing is that multiple copies of a single partition Language System are kept in memory. Control is rotated through each partition on a timely basis.

The maximum time limit of 30 milliseconds for executing statements in any partition was selected on the basis of how long a wait would be acceptable to either an applications program operator or a BASIC programmer. The appearance of "instantaneous" response should be maintained.

The Timesharing example given in Figure 35 is very simple. However, Timesharing, in practice, is considerably more complicated. As an example, one of the partitions, say Partition #3, could be used for program development while the other two are running BASIC programs. In this

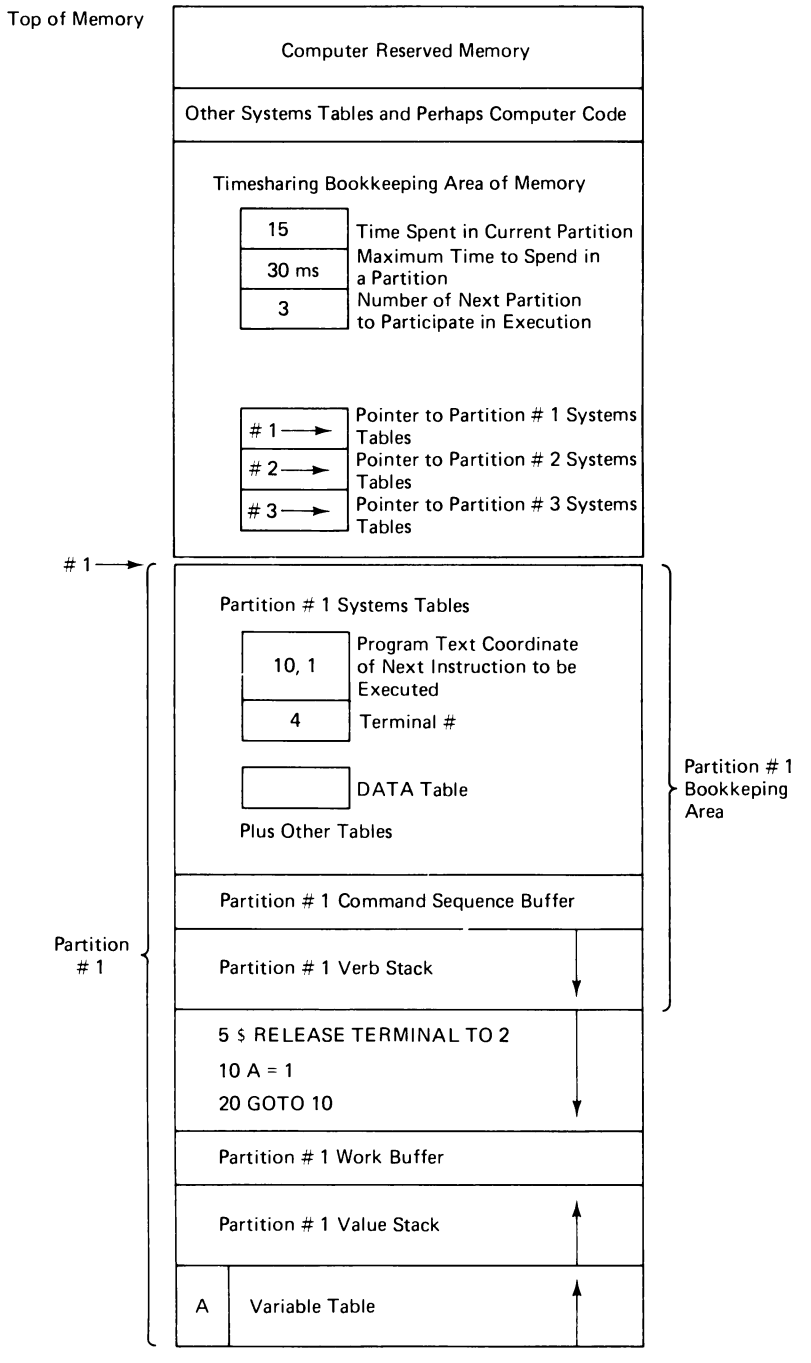


Figure 35: Rough diagram of a three partitioned Timesharing Language System. Example program text coordinates are given for each of the three partitions. Example values are also entered in Timesharing Bookkeeping Tables.

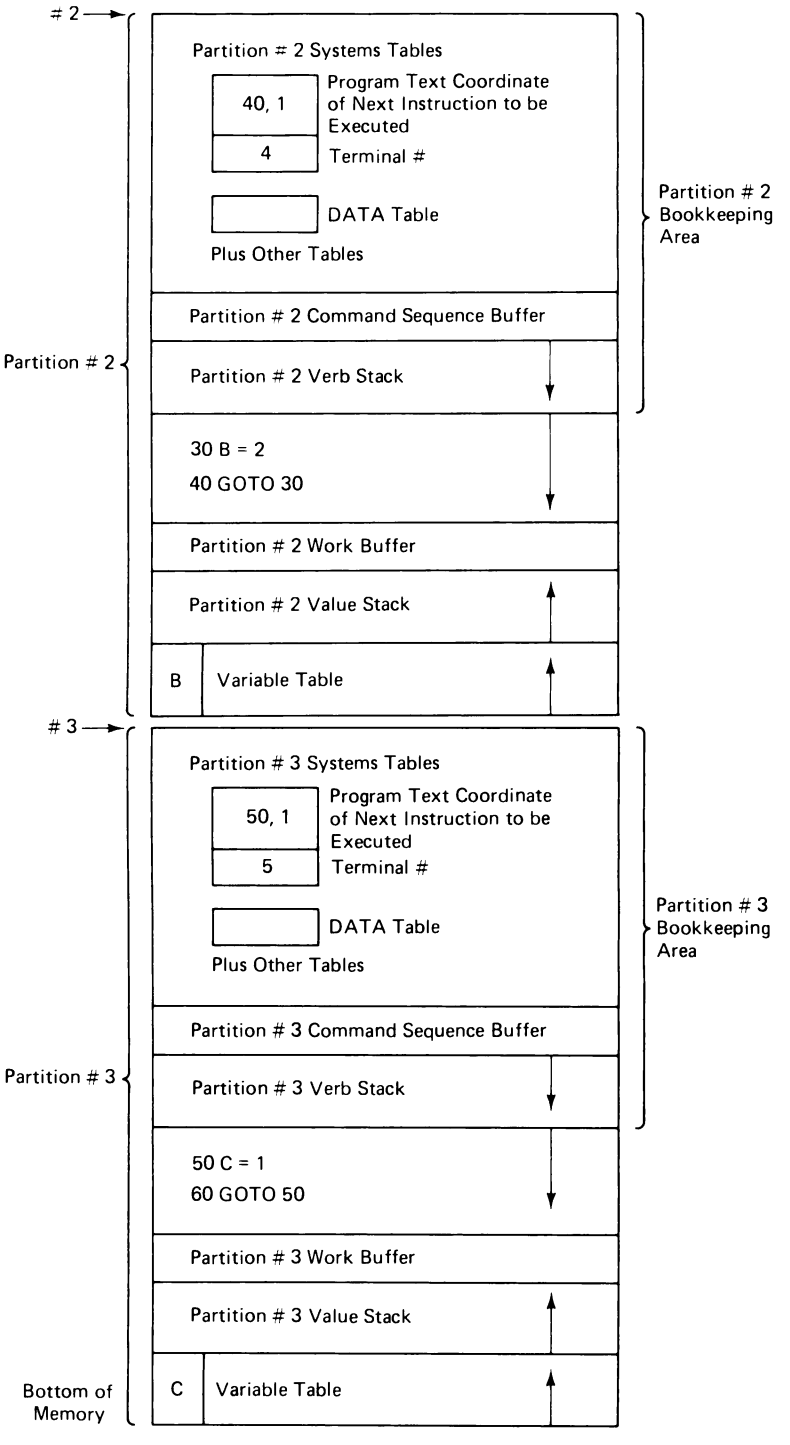


Figure 35: Concluded.

case Partition #3 would be in the entry state. Tables have to be maintained on the states of each partition.

The $ preceding the $RELEASE has a hexadecimal atomization of EA given in Appendix A. Statements such as $RELEASE are sufficiently infrequently encountered in program text that they are not atomized. $RELEASE TERMINAL TO 2 is stored in computer memory as

EA	R	E	L	E	A	S	E		T	E	R	M	I	N	A	L

	T	O		2	0D

PARTITION INTERACTION

The basic idea of how a Timesharing Language works is not particularly difficult to comprehend. The next step is to explain how BASIC can access variables in other partitions and use BASIC program text which resides in other partitions.

Wang Laboratories 2200 series MVP, LVP, SVP computers run a BASIC Language Timesharing Language System. Wang Laboratories is a current leader in development of such Timesharing Language Systems. For this reason Wang Laboratories verb names and variable naming conventions will be used to explain how partition interaction works.

A partition can declare itself *global.* This means that BASIC programs in other partitions may be able to access both variables and program text in the global partition. There may be more than one global partition. Each global partition has a unique name.

The statement:

```
10 DEFFN @PART "GLOBAL"
```

would cause a partition to declare itself global with a name "GLOBAL" when the DEFFN @PART verb was executed. The DEFFN @PART is an executable BASIC statement and must be executed so that the partition is given the name following the @PART.

Global variables are preceded by an @. Some examples of global variable names are:

```
@A, @A$, @A(B), @A$(B)
```

Global variables are distinct from regular variables. Global variables

are only entered into the variable table if they are explicitly defined in a DIM or COM statement. References to global variables in a nonglobal partition will, of course, not be entered into the variable table in the nonglobal partition.

A SELECT @PART verb is used to select a global partition in a BASIC program residing in a nonglobal partition. More than one partition can declare itself global so long as a unique partition name is used. An example of the SELECT @PART verb is:

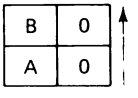
```
20 SELECT @PART "GLOBAL"
```

An example will make the use of global variables, global partitions, and DEFFN and SELECT @PART reasonably clear.

The BASIC program:

```
10 SELECT @PART "GLOBAL"
20 A=1
30 B=@A
```

is located in Partition #1. Its variable table just after program resolution but before the program is run is of the form:

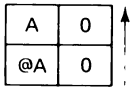


where the arrow shows the direction of expansion of the variable table. Zeros are the null values of scalar numerics.

The BASIC program:

```
10 DIM @A
20 A=2
30 @A=3
40 DEFFN @PART "GLOBAL"
```

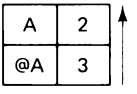
is located in Partition #2. Its variable table is of the form:



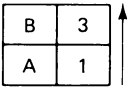
just after resolution but before the program is run.

The program in Partition #2 must be run before the program in Partition #1 is run. The reason is that the global @A must be entered into the variable table of Partition #2, and Partition #2 must be defined as global so that the SELECT @PART "GLOBAL" statement in Partition #1 can be satisfied from the execution of the DEFFN @PART "GLOBAL".

When the BASIC program in Partition #2 is run, its variable table becomes:



When the program in Partition #1 is run, its variable table becomes:



This means that when the statement B = @A was executed, the value of @A was retrieved from the variable table located in Partition #2. The variables A in both Partitions #1 and #2 are distinct variables.

Global variables can be defined in a partition which is never made global by a DEFFN @PART. This effectively gives the possibility of having a second set of variables since those variables preceded by @ are distinct from those not beginning with the @. The only difference is that all of the @ variables would have to be declared in DIM or COM statements.

Within the Systems Tables for each partition there is a table composed of:

Program text coordinate of next instruction to be executed	10,1
Current partition #	1
Global partition #	1
DATA partition #	1
Originating partition #	1
Terminal #	4

At program resolution time, the numbers in the above table represent the state of the table for Partition #1. When statement 10 was executed, the contents of the table became:

Program text coordinate of next instruction to be executed	20,1
Current partition #	1
Global partition #	2
DATA partition #	1
Originating partition #	1
Terminal #	4

This is the way global variable references are satisfied. When @A was discovered, this table was consulted and the variable table in global Partition #2 was searched for its value.

Only the Global partition number entry was used to resolve global variable references. Table entries of Current partition #, DATA partition #, and originating partition # are used for sharing program text and DATA for READ's.

How sharing program text works can be explained by use of three BASIC programs. Partition #1 contains the program:

```
10 A=1: B=2
20 SELECT @PART "SHARE": ERROR $BREAK 10: GOTO 20
30 GOSUB'1
40 STOP
```

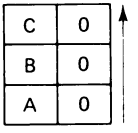
which has the variable table:

C	0
B	0
A	0

after program resolution but prior to execution. Partition #2 contains the program:

```
10 A=2: B=3
20 SELECT @PART "SHARE"
30 GOSUB'1
40 STOP
```

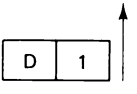
which has the variable table:



after program resolution but before execution. Partition #3 contains the program:

```
10 D=1
20 DEFFN @PART "SHARE": STOP
30 DEFFN'1
40 C=A+B
50 RETURN
```

which has the variable table:



after execution but before it has been referenced by the programs in either Partitions #1 or #2.

RUN's are executed in the order of Partition #1, #3, then #2. Even though the program in Partition #3 is not running, the program in Partition #1 will continue to wait at Statement #20 for the error condition (of not having a DEFFN @PART executed in another partition) to clear. The $BREAK 10 releases Partition #1 for 10 time slices of 30 milliseconds before the Language System returns to execute another statement in this partition. If the ERROR handling procedure was not used, then Partition #1 would be placed in the entry state; and an error code pointing to the SELECT @PART would be displayed on the console output device.

When the B=2 statement is executing the Systems Table for Partition #1 contains:

Program text coordinate of next instruction to be executed	20,1
Current partition #	1
Global partition #	1
DATA partition #	1
Originating partition #	1
Terminal #	4

When the SELECT @PART "SHARE" command is executed, the Systems Table for Partition #1 contains:

Program text coordinate of next instruction to be executed	30,1
Current partition #	1
Global partition #	3
DATA partition #	1
Originating partition #	1
Terminal #	4

which means that the function of the SELECT @PART "SHARE" was to search partition systems tables to find which partition was defined the global share.

When the GOSUB'1 has just finished executing, the systems table in Partition #1 contains:

Program text coordinate of next instruction to be executed	30,1
Current partition #	3
Global partition #	3
DATA partition #	1
Originating partition #	1
Terminal #	4

The 30,1 is pointing to the DEFFN statement in Partition #3. The current partition will now be 3, since program control is being transferred to this partition.

In Partition #3, only D appears in the variable table since no variables following a DEFFN @PART are entered into that partition's variable tables.

When RETURN at Statement #50 in Partition #3 has finished execution, the systems table in Partition #1 contains:

Program text coordinate of next instruction to be executed	40,1
Current partition #	1
Global partition #	3
DATA partition #	1
Originating partition #	1
Terminal #	4

Execution of RETURN returns control to calling Partition #1. The local variables following the DEFFN @PART refer to those variables located in the calling partition's variable table. For this reason the variable table in Partition #1, just after execution of the C = A + B, contains:

C	3
B	2
A	1

Statements 30–50 in Partition #3 appear to be located in Partition #1. Another way of looking at these statements is as an extension of Partition #1.

The program located in Partition #2 is similar to the one located in Partition #1. The execution steps are similar as is the systems table in Partition #3. When this program finishes execution, its variable table will contain:

C	5
B	3
A	2

The important part of this example is the sequence of instruction execution. The programs in Partitions #1 and #2 share the same code in Partition #3. A possible sequence of code execution is:

Program text Coordinate	Originating Partition	Current Partition
10,1	1	1
10,2	1	1
10,1	2	2
20,1	1	1
10,2	2	2
30,1	1	1
20,1	2	2
30,1	1	3
30,1	2	2
30,1	2	3
40,1	1	3
40,1	2	3
50,1	2	3
50,1	1	3
40,1	2	2
40,1	1	1

This could occur since the Language System round-robin rotates through the partitions executing BASIC statements. The STOP in line 20,2 in Partition #3 stopped execution of BASIC statements in the partition, so no time slice is given for execution of statements.

The important point in this example is that Partitions #1 and #2 are "simultaneously" executing code from Partition #3. This code should not modify values in the variable table in Partition #3 for the reason that would be unclear which Partition, #2 or #3, made the modification. Code which can be shared in this manner is called *reentrant*.

The function of the DATA partition # pointer can be explained from the execution of two programs. The first program is in Partition #1 and is:

```
10 READ A$
20 PRINT A$
30 SELECT @PART "KFAB"
40 GOSUB'1
50 DATA "ONE","TWO"
```

The second program is located in a global Partition #2 and is:

```
10 DATA "ALPHA"
20 DEFFN @PART "KFAB": STOP
30 DEFFN'1
40 READ A$
50 PRINT A$
60 RESTORE
70 READ A$
80 PRINT A$
90 RETURN
```

The systems table in Partition #1 is:

Program text coordinate of next instruction to be executed	10,1
Current partition #	1
Global partition #	1
DATA partition #	1
Originating partition #	1
Terminal #	4

when the program in Partition #1 begins execution.

When the DEFFN' statement in Partition #2 is executed, the systems table in Partition #2 contains:

Program text coordinate of next instruction to be executed	40,1
Current partition #	2
Global partition #	2
DATA partition #	1
Originating partition #	1
Terminal #	4

When the RESTORE at line number 60 has just finished execution the systems table in Partition #1 reads:

Program text coordinate of next instruction to be executed	70,1
Current partition #	2
Global partition #	2
DATA partition #	2
Originating partition #	1
Terminal #	4

Execution of the RESTORE causes the DATA partition number to be set equal to the Current partition number. Thus these two programs would cause:

	Comments
ONE	Statement 20,1, Partition #1 caused this.
TWO	Statement 50,1, Partition #2 caused this.
ALPHA	Statement 80,1, Partition #2 caused this.

When control is passed back to the BASIC program in Partition #1, the systems table in Partition #1 contains:

Program text coordinate of next instruction to be executed	50,1
Current partition #	1
Global partition #	2
DATA partition #	2
Originating partition #	1
Terminal #	4

Restoration of the DATA partition pointer to 1 would require that a RESTORE be issued in Partition #1.

Global variables can be used in nonglobal partitions. Whenever a SELECT @PART statement is executed, all references which follow this statement will refer to global variables located in the variable table in the global partition. An example should make this clear. Partition #1 contains the program:

```
10 DIM @A
20 @A=1
30 SELECT @PART "PANTEX"
40 PRINT @A
```

while Partition #2 contains the program:

```
10 DIM @A
20 @A=2
30 DEFFN @PART "PANTEX"
```

The program in Partition #2 is run before the program in Partition #1. The PRINT statement located at Line 40 will cause

```
2
```

to be printed rather than 1. The reason for this is that the SELECT @PART statement caused the global partition pointer to be changed from 1 to 2.

Timesharing BASIC Language Systems require several other verbs to solve problems which arise with Timesharing. One problem is that two BASIC programs located in two different partitions both may wish to use the printer at the same time. If it were the case that both programs could do this, then their respective output line might be interleaved on the hardcopy.

The verb $OPEN allows a partition to "hog" ("Hog" is a technical term meaning "to have exclusive use") an output device. The verb $CLOSE allows a hogged device to be released for use by a program in another partition.

The $PSTAT function returns information, including the partition system table, for examination of any partition's state. #PART returns the number of the partition of the currently executing BASIC program.

The $INIT statement or command allows a BASIC program to determine the number and sizes of partitions of the computer.

Timesharing Language Systems are relatively new. They are considerably simpler than Timesharing Operating Systems. Benchmark trials

have shown that Timesharing Language Systems have outrun Timesharing Operating Systems by a large amount.

SUMMARY

Timesharing Language Systems can be written where the appearance is created that one physical computer system looks similar to several independent computer systems.

The essence of how a Timesharing Language System works is that there are several copies of user's portion of microcomputer memory which extends from the Other Systems Tables region of memory to the variable table. Each of these copies is called a partition. Each partition has the same overall organization, but the contents of each partition differ as a result of the programs being executed or entered in that partition.

The Language System reserves about a 3K byte memory area at the end of the Systems Area for the Timesharing Bookkeeping memory area.

A terminal must be attached to each partition. This terminal is used for input and output information. A terminal cannot be attached to more than one partition at a time.

The programs in the partition do not execute simultaneously. The Timesharing Language System jumps from partition to partition executing at least one, and perhaps several lines of program text from each partition.

A partition can declare itself global. This means that BASIC programs in other partitions may be able to access both variables and program text in the global partition. There may be more than one global partition. Each global partition has a unique name. Global variables are preceded by an @ and are distinct from regular variables. Global variables can be used in nonglobal partitions.

12

Language System Code and its Systems Verbs

Three topics will be covered in this chapter: The Language System code, systems verbs, and the computer power up stage. The Language System code examines and changes the Language System tables. The systems verbs are code blocks which reside in the Language System code area in memory and are not usually accessible by the user's BASIC program. When the power is first applied to the computer, the Language System code is in either read-only memory or in random-access memory.

LANGUAGE SYSTEM IMPLEMENTATION CODE

A Language System is completely described in tables beginning at the systems tables area of memory. The computer code which examines and changes these tables may reside at different locations in computer memory. The location depends on the implementation of the Language System.

One alternative is to place the Language System computer code in the same memory space as all of the Language System tables.

A rough diagram of this arrangement is seen in Figure 36.

A second alternative is to place this code in a different memory space than that used to contain the Language System tables. A rough diagram of this arrangement is seen in Figure 37.

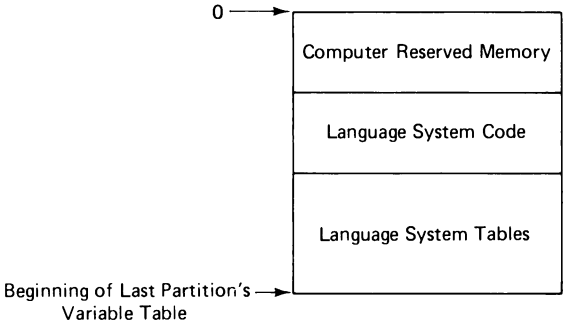


Figure 36: Example layout of microcomputer memory where the Language System implementation code is kept in the same memory space as the Language System tables.

The Language System tables include all Bookkeeping tables, BASIC programs, verb and value stack, variable tables, and all other tables required by the Language System.

One fundamental principle applies to the Language System Code area:

- Data operated on by the Language System Code is kept in an area of memory separate from the Language Systems Code.

A computer system which intermingles data and computer code is called a *vonNeumann machine*. Language Systems should *not* intermingle data and computer code.

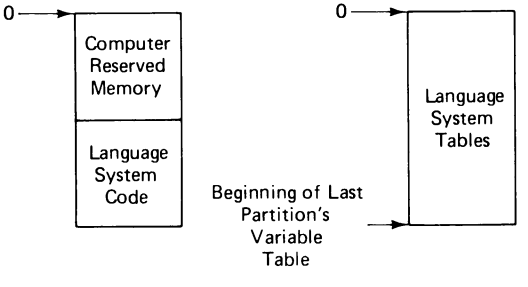


Figure 37: Example microcomputer memory layout where the Language System code is kept in memory space separate from the Language System tables.

SYSTEMS VERBS

Systems verbs are short code blocks of computer code which are written as subroutines in microcode or machine language. These code blocks reside in the Language System code area in memory. These verbs are usually not directly accessible by a user BASIC program, and so are called systems verbs.

Some examples of systems verb functions are:

1. Search the program text region of memory for a line number label and return a pointer to the position following the label.
2. Search the variable table for a reference to a character string or numeric scalar variable and return a pointer to its value.
3. Search the variable table for a reference to an array variable. Return a pointer to the value of this variable or indicate an out-of-bounds reference.

•
•
•

Many systems verbs' function is to create value stack frames for BASIC verbs to process. Many others function to aid the syntax analyzer in analyzing Work Buffer text.

Making changes to verbs does not create a major unfavorable impact on the Language System for the reason that *all* data references are made to the Language System tables area of memory.

Each verb should have a simple intended function. Each verb should be able to contain about a maximum of 30 lines of machine language or microcode instructions. Complex systems verbs should be constructed by calling elementary systems verbs.

Systems level programs, such as the interpreter, syntax analyzer, or partition execution scheduler also reside in the Language System code region of memory. Since these programs are more complex than the simpler verbs, they will reside toward the bottom of this region. These more complex systems programs will consist mostly of GOSUB-like statements to other systems verbs.

A Language System code value stack will have to be established to contain RETURN-like addresses to code blocks calling verbs.

A rough diagram of this arrangement is seen in Figure 38.

A critically important feature of the Language System code region of memory is that its contents do not change when the Language System is in operation.

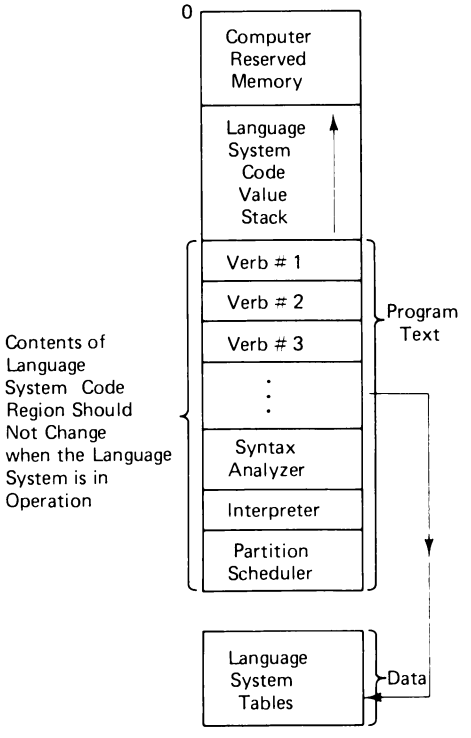


Figure 38: Rough diagram of a Language System with emphasis on the structure of the organization of the Language System code region of memory.

This statement must be tempered. The contents of this region are not supposed to change when the Language System is in operation, but momentary power fluctuations or failing memory chips can cause the contents of the code region of memory to be altered.

One of the states of the Language System is the self-test state. A redundancy measure of the computer code in the Language System code memory region should be computed and stored with the computer code in this region. The redundancy measure could be a simple arithmetic modulo sum of the code, or possibly a more complex cyclic redundancy check. The purpose should not be just to report that the code in this area is likely in error, but to report the chip in which the error was detected. This requires design of a more complicated redundancy measure scheme. This information would be displayed on the console output device in the event a failure occurred. Troubleshooting hardware costs can be reduced using this technique.

The hardware self test should be invoked upon system initialization or when a power failure condition is detected by the microcomputer.

POWER UP

When the power is first applied to a Language System computer system, either:

1. The Language System Code region is in read-only memory and thus contains the Language System at power up, or
2. The Language System Code region is in random access memory and must be loaded from a peripheral external device.

Option #1, experienced has shown, has several disadvantages:

- Language System software cannot be developed on the Language System computer directly since the read-only memory cannot be easily modified.
- Design and implementation errors in the Language System can be expensive to fix.

Containment of the Language System Code in read-only memory has been, and still is, a viable option. This is particularly a useful practice for reducing cost for mass distributed Language System computer systems.

If option #2 is selected, then a permanent storage device must be selected from which the Language (short for Language System code) will be loaded into the Language System code region of memory.

The permanent storage device is most likely some form of a disk but may be a tape or some other permanent storage device such as a bubble or read-only memory cartridge. The Language may even be loaded over a communications link.

Mass storage devices often have their media formatted by the Language. Disks often have a directory of file names and associated pointers pointing to where the contents of the file reside on disk.

The Language System must have a *bootstrap* program which loads the Language into the Language System Code region. This bootstrap program should be contained in read-only memory and should be able to load the Language System code region of memory from files formatted for regular user use. This is important because the file containing the Language can be copied using Language System commands.

When the computer system is powered up, a message to the effect:

KEY ESCAPE

should appear on the console output device. ESCAPE is the name of a special key on the console input device. When ESCAPE or a similar key is depressed, the bootstrap loader should make a check of itself to reasonably assure itself that it is correct. A modulo sum or cyclic redundancy check of its text is an appropriate measure. Memories do fail, so this check is important to make.

If the partial redundancy measure of program correctness fails, then a message should be printed on the console output device explaining this failure. If the bootstrap program appears to be accurately written in memory, then the bootstrap loader should inquire on the console output device:

KEY LOGICAL DEVICE ADDRESS

The response required to this prompt is entry of a logical address of a device which contains the code for the Language System.

An example configuration of a microcomputer system is shown in Figure 39. Letters A, B, C, . . . are used for the logical device addresses. The importance of allowing a selection of devices is that the Language System can be loaded from a different device in the event of a device failure, or even from a different medium.

When the Language System has been loaded, a self test is invoked. Successful completion of the self test will result in a:

READY

or similar message being displayed on the console output device.

For Language System development inclusion of a special key sequence, for example

ESC	S	Y	S	S	A	V	E

which causes the contents of the Language System code region of memory to be stored back in Language System files is valuable.

Once the Language System has been loaded from files, the medium containing the files, say a flexible diskette, can be removed from the physical device since it will not be referenced until the next power up. A "systems disk" does not need to be mounted in any drive when a Language System is in operation. Most Operating Systems require use of a "systems disk" for the reason that Operating Systems contain many more lines of code than do Language Systems.

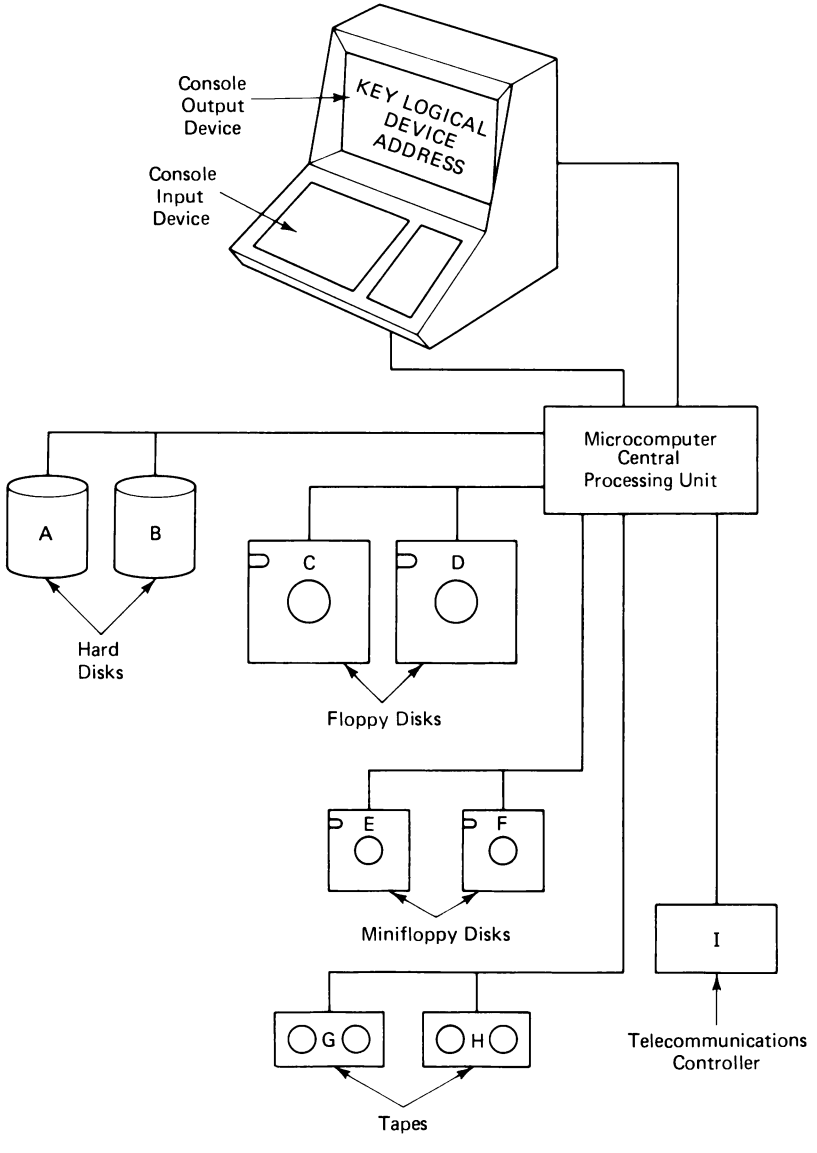


Figure 39: Example of logical device addresses which can be used to load a Language System. The bootstrap loader must be able to recognize the physical unit or the logical structure of the Language System's file must be the same for all devices.

SUMMARY

The Language System code is that computer code which makes BASICs work. The Language System's code examines and changes the various Language System tables. This Language System computer code can be placed in the same memory space as the Language System tables, or in a different memory space.

Systems verbs are short code blocks of computer code written as subroutines in microcode or machine language and are located in the Language System code area of memory. These verbs are usually not directly accessible by a user BASIC program. Many of these verbs create value stack frames for BASIC verbs to process. Others aid the syntax analyzer in analyzing work buffer text. The Language System code region of memory also contains the Language System's programs, such as the interpreter, syntax analyzer, or partition execution scheduler.

When the power is first applied to a Language System computer, either the Language System code region is in read-only memory (i.e., in memory at power-up time), or the Language System code region is in random access memory (i.e., must be loaded from a peripheral external device). If the Language System code resides in random access memory, a bootstrap loader is required to load the Language System code into the Language System code region of memory from a permanent storage device. The bootstrap loader should be contained in read-only memory. After the Language System code self-test, the Language System code initializes the Language System which includes the systems tables through the variable table. When the Language System code has been loaded, self-test successfully passed, and the Language System tables initialized, a "READY" or similar message will be displayed on the console output device. Once the Language System has been loaded from the file, the medium containing the file, such as a flexible diskette, can be removed from the physical device since it will not be referenced until the next power up.

13

How to Write a Language System

Knowledge of how Language Systems are written contributes to knowledge of how BASICs work.

Language Systems are considerably simpler than Operating Systems. Distinctions between Language and Operating Systems may become blurred in a complex Timesharing Language System. Clearly there is an "operating system" managing partition program execution in a Timesharing Language System. Some of the features and differences of Language and Operating Systems are enumerated in the chart on page 184.

Implementation of Operating Systems requires writing many hundreds of thousands of lines of code in most cases. Implementation of an Operating System requires writing:

1. the operating system proper
2. compilers
3. linkage editors
4. loaders, and
5. system utilities

Operating systems proper vary considerably in the amount of code required to implement them. Control Data's SCOPE 2 Operating System contains about 700,000 to 1,000,000 lines of computer code. Digital Equipment Corporation's RSX-11M Operating System required about 300 man years to develop.

A reasonable guess is that a compiler requires about 50,000 bytes of memory. Another reasonable guess is about 10,000 to 15,000 lines of code are required to implement the compiler. The UCSD p-System BASIC

Operating System	Language System
Complex software system	Simple software system
Many lines of computer code required to implement a operating system	Few lines of computer code required to implement a language system
On-line systems disk required for operation	System entirely contained in computer memory
Multiple user languages such as BASIC, FORTRAN, COBOL, ADA, PASCAL, Assembler can be run	Two user languages are the only languages allowed. These are the high level and machine language.
Operating system job control language is separate from user languages	Job control language, which includes input/output verbs, is included in the single user high level language
Edit-compile-linkage edit-load-run program development cycle may be required	Edit-run program development cycle
Assembly language usually requried	No assembler required, only machine language or microcode and the high level language
Minimum RUN time system memory requirements	Constant system memory requirements
Fast execution for compiled language programs	Usually slow execution speeds

compiler is written in about 11,000 lines of PASCAL code. Compilers are often written in languages other than assembler.

Linkage editors are usually about one-half to four-fifths the size of a compiler, so 5,000 to 12,000 lines of code is a justifiable estimate of implementation size. A loader is approximately about one half the size of a linkage editor so a rough estimate has its implementation requiring about 2,500 to 6,000 lines of code.

Systems utilities, such as file copies, cannot be accurately estimated for the reason that there can be arbitrarily many different utilities. An editor is considered a utility and can be written in about 6,000 to 10,000 lines of code. Such an editor would be a very complete editor as opposed to a simple BASIC line editor.

A conservative estimate of the number of lines of code required for implementation of a reasonably complete Operating System-based computer software system is:

Line of Code	Function
10,000	Operating System proper
10,000	One compiler
5,000	Linkage Editor
2,500	Loader
6,000	Editor
33,500	Total

This estimate of the total number of lines required to implement an Operating System software system is, as was mentioned previously, somewhat conservative. Lines of code required to implement an assembler (about the same as a compiler) and library routines (such as SIN, COS, . . .) have not been included in this estimate.

The largest Timesharing Language System, on the other hand, requires about 21,000 lines of code to implement. Some smaller BASIC systems can be implemented in about 2,700 lines of code. This includes a simple line editor, all of the library routines, and some utility verbs.

Operating System and compiler code is expensive to produce. An experienced systems programmer may produce about 5 lines of such code per day. Thus an experienced systems programmer may produce about 2,000 lines of such code per year. This means that production of the example Operating System software system could take roughly 17 man years of labor.

In 1980 in America a fully burdened systems programmer costs about $70,000 per year. This price includes such costs as the building required to house the programmer and the cost of the programmer's computer.

This means that it could cost about 1.4 million dollars, conservatively estimated, to produce the Operating System software system! Any person or corporation which might harbor the intention to write a new Operating System software system should keep this cost estimate in mind.

The amount of work required to write a new Language System is considerably less than that required to write an Operating System software system. Several reasons account for this:

1. Not as many lines of code are required.
2. An edit-run program development cycle can be used to bring up a Language System on the target computer.

An Operating System software system is usually required to write an Operating System software system. As an example, a Digital Equipment VAX computer and ADA compiler are currently required to write computer code for Intel's iAPX 432 microcomputer.

A Language System can be bootstrapped up on a target computer system. The edit-run capability can be used to produce computer code at a daily rate in excess of 5 lines of code per day. The cost of a microcomputer system is also considerably cheaper than a large Operating System computer system required to develop an Operating System.

One of the main attractions of a microcomputer system is its low cost. For this reason only a strategy on how to write a Language System will be examined.

A minimal microcomputer system on which to develop a Language System should include peripherals:

1. A keyboard console input device.
2. A cathode ray tube console output device.
3. Two floppy or minifloppy disk units.
4. A printer with a minimum printing rate of 100 characters per second.

The overall strategy for bringing up the Language System includes:

1. Writing a text editor which has the capability of editing all of microcomputer memory. Insertions, changes, and deletions of characters in either hexadecimal or ASCII should be supported.
2. Writing a bootstrap loader and saver which will load and save the Language System code region of memory on a flexible diskette.
3. Implementing the Language System in machine language or microcode.

   Number three can be broken down into:

   A. Writing a BASIC line editor for entry, recall, and edit of BASIC text for the command sequence buffer and program text region of memory.
   B. Writing verbs of SAVE and LOAD which will allow the contents of the program text region of memory to be SAVEd in a disk file or LOADed from disk files. SAVE and LOAD are both commands and thus are executed from the command sequence buffer.
   C. Writing enough of the "monitor" part of the Language System to allow the system to enter the states of Language System Initialization, Entry Phase, Language System Self Test, and Execution. Writing a complete Resolution Phase can be left to later.
   D. Writing the remainder of the Language System.

   The important point to keep in mind is that there are two editors.

One editor is used to edit both the Language System code region of memory and systems tables. Edit means examine and possibly change the contents of these regions of memory. The second editor enters, recalls, and edits information contained in either the command sequence buffer or program text region of memory. The editing takes place in the work buffer.

The second editor, at least, can be written with BASIC verbs. Much of BASIC can be written in BASIC.

There are also two loaders and two savers. The first loader/saver loads and saves microcode or machine language program text located in the Language Systems code region of memory in disk files.

The plural, "files," is used because systems verbs should be decomposed into classes. String handling, mathematical elementary function, numerical matrix, and so forth are possible classes of verbs. Verbs in each class are kept in a separate file. This facilitates development of a Language System by a team effort. Members of the team can thus make changes to their class of verbs with minimal impact on the work of other team members.

The second loader/saver loads and saves BASIC program text located in the program text region of memory in user named disk files.

There is a fundamental principle involved in any type of text development. This fundamental principle is:

- Enter text only once. Recall and edit after the initial entry.

Application of this principle supposes that LOAD and SAVE have been implemented.

The requirement of two disks for a minimal system is for making back up copies of disks. A command to the effect:

COPY <disk #1> TO <disk #2>

should be implemented at the beginning of Language System project so that back up copies of the increasing expensive Language System software can be made.

The printer is required for three reasons:

1. A printed back up copy of the Language System can be kept.
2. Some Language System errors are best diagnosed by studying a printed listing of program text.
3. Development of printer verbs.

Systems verbs are developed towards the beginning of the Language System project. Toward the end of the project, some verbs can be written in terms of other BASIC verbs. The numerical MAT verb such as MAT

A=0, MAT A=B*C, MAT A=I, and so forth are a class of verbs which lend themselves to be written in BASIC. At the end of a Language System project, it is found that BASIC is largely written in BASIC.

One principle in implementation deserves continued emphasis:

- Program text must be kept physically separate from data accessed by the program.

This principle applies to quality applications systems code as well as to Language Systems code.

Figure 40 contains a diagram of this principle applied to a BASIC Language System. The essential importance of this figure is that computer code, no matter what the language level, manipulates data stored in tables.

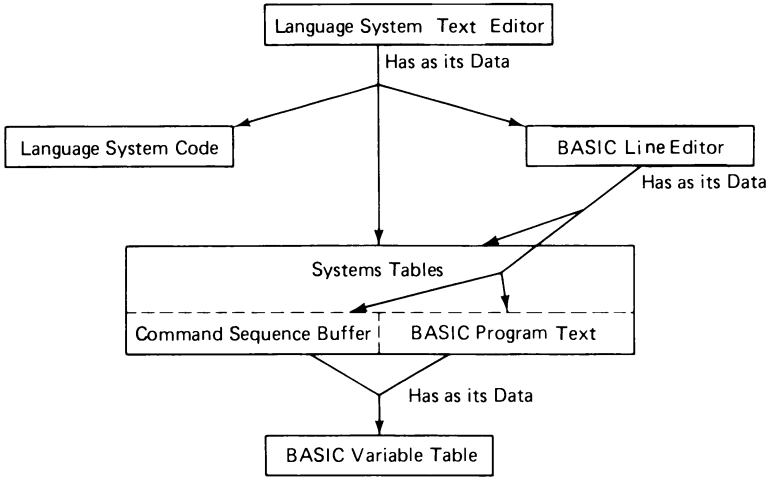


Figure 40: Diagram emphasizing that data must be kept separate from computer code. The Language System Text Editor has all regions, excluding its own, of microcomputer memory as its data area. The BASIC line editor edits text located in the Command Sequence Buffer and BASIC program text region of memory but also has some of its tables in the Systems Tables.

SUMMARY

The amount of work required to write a new Language System is considerably less than that required to write an Operating System for two reasons. First, not as many lines of code are required. Second, an edit-run program development cycle can be used to bring up a Language System on the target computer.

A minimal microcomputer system on which to develop a Language System should include the following peripherals:

1. A keyboard console input device.
2. A cathode ray tube output device.
3. Two floppy or minifloppy disk drives.
4. A printer with a minimum print rate of 100 characters per second.

The overall strategy for bringing up the Language System is:

1. Write a text editor which has the capability of editing all of microcomputer memory.
2. Write a bootstrap loader and saver which will load and save the Language System code region of memory on a flexible diskette.
3. Implement the Language System in machine language or microcode.

An important principle for implementation is that program text must be kept physically separate from data accessed by the program.

14

Conclusions and References

How do BASICs work? How do FORTRANs, COBOLs, PASCALs, ADAs, . . . work? They all work about the same way: they are composed of many different types of tables. Such systems are called *table driven.* Most of the newer systems are stack oriented as compared to link list oriented. IBM language implementations are mostly link list oriented.

Why are BASICs so popular? BASICs are easy to learn but more important:

- New verbs can be added to the langauge when technological advances require their addition.

Most high level computer languages have a set number of verbs and cannot adapt. Most high level computer languages are insufficiently rich in verbs that they require an Operating System to add verbs (JCL statements) so that they can work. BASICs include Operating System verbs as part of it.

BASICs are relatively inexpensive to write. Hardware manufacturers have to competitively market complete computer systems. Development of expensive Operating System software systems drive up the price of their product. Language Systems, on the other hand, do about everything an Operating System does, but can be produced with less cost. Language Systems only run two languages: The high level and microcode or machine language. COBOL or FORTRAN might be selected as the high level language for a Language System, but proponents of both languages are not particularly receptive to the idea of addition of new verbs

to their language. This is one reason why BASICs are selected by many software system implementers—no language standards.

BASIC Language Systems are still expensive to develop. For this reason some of their implementers encrypt their Language Systems on disk. The Language System is deciphered when the bootstrap loader loads the system into memory. The implementation information presented in this book was obtained from a wide variety of sources. This information was not stolen from any Language System software vendor.

Some literature on Language System implementations is beginning to appear. A more general book on this subject than this book is:

- Writing interactive compilers and interpreters by P. J. Brown, John Wiley and Sons, 1979.

Another excellent source of Language System implementation techniques is:

- Dr. Dobb's Journal of COMPUTER Calisthenics & Orthodontia.

Dennis Allison, happy lady and friends authored an excellent early article on Tiny BASICs in a 1976 issue of this journal.

Wang Laboratories 2200 series computers software engineers have written the most elegant and sophisticated Timesharing Language System. Their

- BASIC—2 language reference manual

presents an excellent conceptual overview of both single user and Timeshared Language Systems.

The Microsoft Corporation has produced simple Language Systems for a variety of inexpensive microcomputers. Their recent release of the Language System for the Radio Shack Extended (16K) BASIC for the Color Computer is a significant step forward in Language System design. Microsoft recently completed the BASIC for Convergent Technologies Intel 8086 based software system. System layouts for the Operating System are well documented in the publication

- Convergent Technologies, Technical summary, Information processing systems, Convergent technologies, 1980

Convergent Technologies product is an Operating System software system as opposed to a Language System. Their product is a very advanced Operating System and may not suffer the problems of other Operating System BASICs.

BASICs and Operating Systems do not get along well. If the BASIC works well, then the Operating System usually does not. And conversely.

About 30 beginning business BASIC students using Stanford BASIC significantly degraded the performance of an Amdahl V6 IBM compatible big computer system.*

How do BASICs work? We feel that the contents of this book give you a good idea of how they work. The better BASICs work, the lower the cost of applications software. BASICs are beginning to incorporate high level record access methods and sort procedure verbs. Applications programmers can write more complex systems faster which means less expensively.

BASICs become the native machine language in a Language System. BASICs are well suited to this task. Perhaps better suited to this task than any other language.

*Large mainframe computer compiler design is described in *Principle of Compiler Design*, A. V. Aho and J. D. Ullman (Addison-Wesley, 1977).

APPENDIX

The BASIC keyword atomization table included in this appendix is used by BASIC-2 which was developed by Wang Laboratories. The table on the following page serves as an example of a typical BASIC text atomization. This table is referenced: Technical note #2602, dated July 7, 1976 and authored by Bruce Patterson.

Only 128 text direct text atomizations are possible using this scheme. The $ (EA) is used to delimit a special class of Language System verbs such as $RELEASE, $BREAK, ...

The number of verbs in BASICs is steadily increasing. Perhaps two byte atomization might be more appropriate today.

TECHNICAL NOTE #2602
Author: Bruce Patterson
Date: July 7, 1976
Subject: BASIC-2 TEXT ATOMIZATION

In order to conserve memory and optimize program line interpretation, BASIC-2 atomizes program text when RETURN (EXEC) is pressed. Most BASIC words are replaced by single byte codes, called text atoms (see following page). The text atom is an 8-bit code with high order bit on; the lower 7-bits specify the particular BASIC word. Line numbers and line number references are stored in pack decimal form (2 bytes) preceded by the text atom FF_{16}.

Most atoms can be used to enter the associated BASIC word for Console Input or INPUT operations. However, $E5_{16}$ and $E6_{16}$ are interpreted differently. $E5_{16}$ represents line erase for CI, INPUT, and LINPUT operations. $E6_{16}$ represents the statement number key and causes a new line number to be generated for CI mode; $E6_{16}$ is ignored by INPUT and LINPUT.

Programs saved on disk are stored in atomized form.

BASIC-2 TEXT ATOMS

80 LIST
81 CLEAR
82 RUN
83 RENUMBER
84 CONTINUE
85 SAVE
86 LIMITS
87 COPY
88 KEYIN
89 DSKIP
8A AND
8B OR
8C XOR
8D TEMP
8E DISK
8F TAPE

90 TRACE
91 LET
92 FIX(
93 DIM
94 ON
95 STOP
96 END
97 DATA
98 READ
99 INPUT
9A GOSUB
9B RETURN
9C GOTO
9D NEXT
9E FOR
9F IF

A0 PRINT
A1 LOAD
A2 REM
A3 RESTORE
A4 PLOT
A5 SELECT
A6 COM
A7 PRINTUSING
A8 MAT
A9 REWIND
AA SKIP
AB BACKSPACE
AC SCRATCH
AD MOVE
AE CONVERT
AF [SELECT] PLOT

B0 STEP
B1 THEN
B2 TO
B3 BEG
B4 OPEN
B5 [SELECT] CI
B6 [SELECT] R
B7 [SELECT] D
B8 [SELECT] CO
B9 LGT(
BA OFF
BB DBACKSPACE
BC VERIFY
BD DA
BE BA
BF DC

C0 FN
C1 ABS(
C2 SQR(
C3 COS(
C4 EXP(
C5 INT(
C6 LOG(
C7 SIN(
C8 SGN(
C9 RND(
CA TAN(
CB ARC
CC #PI
CD TAB(
CE DEFFN
CF [ARC] TAN(

D0 [ARC] SIN(
D1 [ARC] COS(
D2 HEX(
D3 STR(
D4 ATN(
D5 LEN(
D6 RE
D7 [SELECT] #
D8 % [image]
D9 [SELECT] P
DA BT
DB [SELECT] G
DC VAL(
DD NUM(
DE BIN(
DF POS(

EO LS=
E1 ALL
E2 PACK
E3 CLOSE
E4 INIT
E5 HEX
E6 UNPACK
E7 BOOL
E8 ADD
E9 ROTATE
EA $ [stmt]
EB ERROR
EC ERR
ED DAC
EE DSC
EF SUB

FO LINPUT
F1 VER(
F2 ELSE
F3 SPACE
F4 ROUND
F5 AT(
F6 HEXOF(
F7 MAX(
F8 MIN(
F9 MOD(
FA reserved
FB reserved
FC reserved
FD reserved
FE reserved
FF packed-line-number

Annotated Glossary of Technical Terms

Address: A number which points to each byte location in a microcomputer memory.

Alphanumeric character string array variable names and values: Names consist of a $ which is preceded either by a single alphabetic letter or by a single alphabetic letter with a single digit 0 through 9 and parentheses, which signify a one or two dimensional array. Each byte in an alphanumeric character string array variable contains one of 256 different binary values (HEX(00) through HEX(FF)); these values are stored in memory on a character by character basis. Examples of alphanumeric character string array variable names are: A$( ), A0$( ), A1$( ), . . . Z8$( ), Z9$( ).

Alphanumeric scalar character string variable names and values: Names consist of a $ which is preceded either by a single alphabetic letter or by a single alphabetic letter with a single digit 0 through 9. Each byte of an alphanumeric scalar character string variable value takes on one of 256 different binary values (HEX(00) through HEX(FF)); these values are stored in memory on a character by character basis. Examples of alphanumeric scalar character string variable names are: A$, A0$, A1$, . . . Z8$, Z9$.

Applications programmer: An individual who designs, produces, and implements computer programs for use by application systems users.

Array mapping function: Used to compute a single offset pointer from knowledge of coordinates of an array element. The offset pointer points to the value of the array element.

Assembly Language: A low-level high-level language which allows writing symbolic machine language. If a machine's language is high-level, an assembly language is not needed. One of the major purposes of this book is to explain how high-level languages can be developed into a computer's machine language.

Assembler: A computer program which translates an assembly language program into machine language.

BASIC compiler: A computer program which converts a BASIC program (source code) into a program written in another language (object code). The other language can be the machine language or another high-level language. For example, Softech's BASIC compiler is written in PASCAL and the BASIC programs are compiled into p code. P code, the machine language for abstract computers, is interpreted on the host computer.

BASIC Language System: Also referred to as the Language System. A set of programs which process the applications programmer's program and commands. These programs can be contained in the computer's read only memory or can be loaded from a permanent storage device, such as a flexible diskette, into random access memory. Language System programs are written in either machine language or preferably microcode.

Binary coded decimal (BCD): Uses the hexadecimal digits of 0 through 9 to represent decimal digits. A diagram of the binary weighting for all 16 combinations of the hexadecimal digits is

8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
4	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

The extra six hexadecimal digits A, B, C, D, E, F have been used to denote signs for binary coded decimal numbers. IBM uses B or D for − and A, C, E, or F for +.

Bootstrap Loader: A read only program that loads the Language System code into the Language System code's region of memory. The Lan-

guage System should be stored in several different files since it can be developed by different teams working in parallel. Each team keeps its portion of the Language System in a separate file so that its modifications to the Language System will not interact with the information in the files of other teams.

The development of a Language System also requires a "bootstrap saver" which takes the Language System from the microcomputer control memory and places it back in the files from which it came. The Language System development teams use a "systems editor" to edit the Language System during its development. Language Systems implementation teams use "edit-run" program development cycles in much the same way as applications programmers use the "edit-run" capability of a Language System to rapidly produce software products.

Buffer: A temporary storage area in the Language System used for performing BASIC program input/output. See Command Buffer and Work Buffer for more information on individual buffers.

Byte: A sequence of adjacent binary bits that can be treated as a unit within a binary fixed word. In most computers, a byte is eight bits long. The preferred word for an eight bit byte is **octet**.

Calling program: A program which calls a subroutine in memory. The routine is called the called program.

Character string: Any combination of characters, letters, or hexadecimal digit pairs. It can be BCD, ASCII, etc. Interpretation is left up to the user.

Code blocks: Short program subroutines which generally contain no more than 10 to 20 statements.

Command: An instruction that is entered into the microcomputer from the keyboard and is executed immediately after a carriage return is entered. This instructs the Language System to perform a certain immediate operation or operations.

Command buffer: Located in the systems area of memory. Contains command sequences that are executed by the interpreter. The command buffer is usually given a reasonable size of about 256 bytes.

Compiler: See BASIC compiler.

Console input device: Generally a keyboard. Other devices can be used but care must be taken in selecting them.

Console output device: Usually a cathode ray tube. A printer is occasionally selected for the same purpose.

CR: Abbreviation for carriage return.

DATA pointer table(s): Resides in bookkeeping area of memory. Contains a pointer to the next DATA item to be read by a READ statement. Multiple DATA pointer tables need to be maintained in Timesharing Language Systems.

Data structures: Used in BASIC to denote numeric and alphanumeric values which participate in their appropriate operations. The data structures are: numeric scalar variables, alphanumeric scalar character string variables, numeric array variables, and alphanumeric character string array variables.

Data types: Same as Data Structures. See Data Structures.

Delimiters: Symbols which separate the parts of a statement. Some examples of delimiters are: :, ",), and CR.

Editor: There are two editors in the BASIC Language System. One editor is used to examine and change the contents of the Language System and the systems tables. Another editor is used to edit information contained in either the command sequence buffer or the program text region of memory.

Encrypt: Process of scrambling information according to a key so that it cannot be read without decrypting it. Decrypting requires knowledge of the key.

Entry phase: One of the five states of the Language System. The entry phase uses the work buffer to store commands, statements, or data which are entered from the console input device.

Execution phase: One of the five states of the Language System. The execution phase directly follows the successful resolution of a program.

Fetch: Process of reading information from computer memory.

Floating point: Numbers which consist of both an exponent and a mantissa (fraction) and signs for both the exponent and mantissa. Numeric values are usually stored in memory as floating point numbers.

Global partition: Division of main storage in a Timesharing System so that both variables and program text in one partition can be accessed by BASIC programs in other partitions.

Global variables: In a Timesharing System, a given set of variables are preceded by an @. Global variables are distinct from regular variables in that global variables can be used in nonglobal partitions.

HALT/STEP: Key that either stops the execution of the program or causes one statement or command to be executed each time the HALT/STEP key is depressed.

Hexadecimal: Numbering system with a base of 16 that incorporates the digits 0 through 9 and the alphabetic letters A through F.

Hog: Denotes exclusive use of a peripheral computer device.

Interpreter: A Language System program which executes the applications programmer's BASIC program.

Job Control Language (JCL): Commands that request services from the Operating System.

Language: Short version of Language System code.

Language System: Amalgamation of a small Operating System (sometimes called a monitor) and a high-level computer language.

Language System computer code: Examines and changes the Language System's tables. These tables range from the systems tables to the start of the variable table and are used to process the applications programmer's program.

Language System initialization: State in which the Language System initializes or gives first values to all of its tables and also checks on available microcomputer memory. This state follows memory loading.

Language System self test: One of the five states of the Language System. The self test may be invoked as a result of an abnormal computer condition, such as a momentary power loss or memory failure.

Lexical analyzer: Identifies verbs, variable names, arrays, numerical constants, and literals. Lexical analysis of commands and statements occurs in the work buffer area of memory.

Linkage editor: Part of an Operating System that links together both programs which have been compiled separately and call system library programs. When these programs have been linkage edited, they can be loaded into computer memory to be run.

Linked list: Information linked together through computer memory by means of pointers which point to the next information in the list.

Literal: A character string within quotes.

Loader: An Operating System program that takes the output of the linkage editor, i.e. linked computer programs, and loads it into computer memory for execution. Some computer systems allow a "compile and GO" procedure by which the loader accepts input from the compiler and bypasses the linkage edit step.

Machine Language: Native language of a computer. Machine language can be a high level language which is often interpreted in microcode. (*See* Microcode.) Machine language instructions do not usually check data structures as is always done in BASICs.

A machine language instruction set, such as that found in the IBM 360/370, has a large portion of its instructions implemented in microcode. Other instructions are "hardwired" in electronics. New supermini computers, such as DEC's VAX, have microcoded instructions but do not have a Language System in microcode. In Language Systems, BASIC is a sophisticated and advanced machine language.

MATREDIM: A language verb which allows redimensioning of both single and double dimensioned arrays.

Matrix: An array of numeric or string variables in rows and/or columns.

Memory locations: Storage units in microcomputer memory which each contain one byte, or eight bits.

Microcode: Base level machine language with instruction formats that can vary in form from computer to computer. Many microcode words are of fixed length: 19 or 32 bits are not uncommon, and some are only eight bits. Microcode instruction sequences are used to load microcode registers, set latches, send character strings down a bus, and so forth. Operations at this level can frequently occur in parallel.

BASIC Language Systems, unlike traditional machine language, can allow an applications programmer to issue microcode instructions via BASIC using an EXECUTE-type instruction.

Microcomputer memory: Electronic storage medium. Microcomputer memory is queried or set by certain programs in a BASIC Language System.

Microprocessor: A sophisticated processing unit. Generally a chip of silicon imprinted with transistors and circuits. One or more microprocessors is combined with memory, timers, and other components to make a microcomputer.

Numeric array variable names and values: Names consist of a single alphabetic letter or a single alphabetic letter with a single digit 0 through 9 and parentheses, which signify a one or two dimensional array. Numeric array variables have a range of numeric values; the values are usually stored in memory as floating point numbers. Numeric array variables participate in arithmetic operations. Examples of numeric array variable names are: A( ), A0 ( ), . . . Z8( ), Z9( ).

Numeric scalar variable names and values: Names consist of a single alphabetic letter or a single alphabetic letter with a single digit 0 through 9. Numeric scalar variable values are often stored in memory as floating point numbers. Numeric scalar variables participate in arithmetic operations. Examples of numeric scalar variable names are: A, A0, A1, . . . Z8, Z9.

Numeric variable: Refers to numeric scalar variable names and/or numeric array variable names. *See* Numeric Scalar Variable Names and Values and Numeric Array Variable Names and Values for more information.

Object code: A code in machine language or a high level language that has been translated by the compiler from the source code and can be loaded into memory and executed.

Offset: A pointer which takes on consecutive integer values.

Operands: Data and variables that are operated on.

Operating System: Schedules program execution, performs input/output for programs, assists with file oriented commands, and can be used to invoke compilers, linkage editors, loaders, and utilities. It executes a language called Job Control Language (JCL).

Operators: Synonymous with verbs. Symbols which indicate that action is to be performed. Examples of operators are: +, −, *, /, =, and >.

Overhead information: Information associated with storage of a variable in microcomputer memory. Provides knowledge of the variable type, variable length, array dimension, and location to Language System verbs.

Overlay: Program text which is read into the program text region of memory by an executing BASIC program.

Parity bit redundancy measure: Some microcomputer memory boards allow 9 bits instead of 8 to store each byte. The ninth bit is used as a parity bit to tell whether the 8-bit byte contains an even or odd number of bits. Some memory errors can be detected through the use of the parity bit.

Partition: In a Timesharing System, one of the several regions of microcomputer memory that can contain a BASIC program. Each partition appears to be a Language System and is considered a complete storage area.

Pointer: An index number which specifies a particular piece of information is located in the microcomputer memory.

Pop: Removal of an element from a stack.

Program resolution: Occurs when RUN key is pressed. The following process takes place: All non-common variables are removed from the variable table. Lexical and syntax analysis are redone. Line numbers are checked for validity. The variable table is constructed and initialized. If no errors are found during program resolution, the program is marked executable and program execution begins.

Push: Placement of a new element on a stack.

Queue: A list where information placed first in a queue is the information first removed from the queue. A queue is also referred to as First in - First out (FIFO).

Random-access memory (RAM): Computer memory which can be both read from and written into.

Read: To retrieve or fetch information from computer memory.

Read-only memory (ROM): Computer memory which can only be read.

Reentrant BASIC code: Language System code that can be simultaneously shared by other BASIC programs which reside in partitions of a Timesharing System.

Reverse Polish: Program notation in which an operator is written after its operands. Operators are executed immediately after they are encountered. Programs written in Reverse Polish are executed much faster than those that are not.

Resolution Phase: *See* Program Resolution.

Round-Robin: Programs in each memory partition are allowed to successively execute for a period of time.

Source code: Computer code written by applications programmers which may require translation into another computer language before it is run on a computer.

Stack: An array or contiguous list where information last placed on the stack is the first removed. Stacks are said to be Last in - First out (LIFO) information storage configurations. *Compare* Queue.

Stack frames: Groups of information placed on a stack as opposed to single pieces of information placed on a stack.

Stack subframe: Complete verb and value stack frames must be constructed before a verb processing code block can be invoked to execute the verb. Stack subframes are completed data plus header information (LENGTH, TYPE, LOCATION, . . .) which has been placed on a value stack.

Stack-oriented: Description of computers or languages which perform most of their operations on stacks. A stack-oriented computer or language can be compared to a list-oriented computer or language.

Statements: Resemble commands but must be preceded by a numeric label. A statement is stored in the program text region of memory.

Store: To write information into computer memory.

String variable: Refers to an alphanumeric scalar character string variable and/or an alphanumeric character string array variable. *See* Alphanumeric Scalar Character String Variable Names and Values and Alphanumeric Character String Array Variable Names and Values.

Syntax analyzer: Examines data types, verbs, and delimiters to make sure they appear in an order consistent with the BASIC Language definition.

Systems disk: Disk which contains a Language System.

Systems library: A library of computer programs used by an Operating System and its associated compilers, linkage editors, and loaders.

Systems verbs: Short blocks of computer code that are written as subroutines in microcode or machine language and that reside in the Language System code area in memory. These verbs are not usually directly accessible by a BASIC user program. Some of these verbs create value stack frames for BASIC verbs to process. Others aid the syntax analyzer in analyzing the work buffer text.

Systems tables: Located in microcomputer memory. Size is fixed. Contain information required to aid in the execution of a BASIC program.

Table driven: Description of software system in which complete information on the state of the system is contained in tables.

Text atom: Compressed representation of an ASCII verb name.

Timesharing: A central processing unit is shared by several different firms or individuals, each of whom may be unaware of the other users.

Tokens: Symbols representing verbs, variables, constants, and literals.

Value stack: Stack which contains data, not verbs.

Variable: A structure which can assume any appropriate value. Can include numeric scalar variable, alphanumeric scalar character string variable, numeric array variable, or an alphanumeric character string array variable.

Verb stack: Stack which contains verbs, not data. Compare with Value Stack.

Write: To store information.

Work buffer: Contains the input characters of statements, commands, or data and is located between the end of the program text region and the end of the variable table. BASIC program input/output is frequently done via the work buffer.

INDEX

IMPLEMENTING BASICs:
HOW BASICs WORK

Payne and Payne

Anyone with a little knowledge of BASIC can benefit from reading IMPLEMENTING BASICs: HOW BASICs WORK. This book enables you to:

- write better programs and
- write more advanced programs with less effort.

Here's your comprehensive guide which clearly explains the structure, functions, and capabilities of BASIC language. By using the stack-oriented method, you can develop table driven software and learn the specific techniques used in microcomputer language design and implementations that you need.

TABLE OF CONTENTS

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia 22090

0-8359-3044-0

Cover Design by Lee Forman

IMPLEMENTING BASICs

Payne & Payne

REWARD BOOKS