IRIS R8

BUSINESS BASIC MANUAL

Revision 02

NOTICE

Every attempt has been made to make this manual complete, accurate and up-to-date. However, all information herein is subject to change due to updates. All inquiries concerning this manual should be directed to POINT 4 Data Corporation.

P R E L I M I N A R Y

POINT 4 Data Corporation
2569 McCabe Way
Irvine, CA 92714
(714) 863-1111

REVISION RECORD

PUBLICATION NUMBER: SM-030-0012

Revision	Description	Date
01	Draft Version	10/01/82
02	Preliminary Release	02/01/84

LIST OF EFFECTIVE PAGES

Changes, additions, and deletions to information in this manual are indicated by vertical bars in the margins or by a dot near the page number if the entire page is affected. A vertical bar by the page number indicates pagination rather than content has changed. The effective revision for each page is shown below.

Page	Rev	Page	Rev	Page	Rev
Cover	-				
Title	02				
ii thru x	02				
1-1 thru 1-3	02				
2-1 thru 2-57	02				
3-1 thru 3-143	02				
Appendix Title	-				
A-1 thru A-14	02				
B-1 thru B-4	02				
C-1	02				
D-1 thru D-4	02				
E-1	02				
Comment Sheet	02				
Mailer	-				
Back Cover	-				

PREFACE

This manual describes IRIS data files and the statements which make up IRIS Business BASIC. Section 2 describes each type of IRIS data file, and discusses features common to all files. Section 3 presents each Business BASIC statement in alphabetical order.

In Section 3, each statement is presented on facing pages. The discussion of the statement is usually confined to the left pages with appropriate examples on the right page.

Five appendices are included in the manual. Appendix A includes twelve examples of BASIC programs. Appendix B lists BASIC error numbers and codes. Appendix C lists ASCII codes. Appendix D shows a listing of the BUILDXF program. Appendix E lists terminal control codes.

This manual assumes that the reader has had some previous programming experience in the BASIC language.

IRIS Business BASIC is upward compatible with Dartmouth BASIC as described in Kemeny and Kertz's BASIC Programming, First Edition. Programs written under Dartmouth BASIC will run without modification under IRIS Business BASIC, except that all appropriate statements must be executed.

Standard Writing Conventions

The following syntax conventions are used throughout the manual:

variable	simple or subscripted numeric or string variable
list	the elements of a list must be separated by commas; for example, an expression list might be: A,B,C*5,D+1
expression	a group of characters that may be evaluated to a simple numeric value; sometimes abbreviated expr
filename	an IRIS filename
filename string	a quoted literal or string variable containing a filename

A B	indicates that either A or B may be included, but not both
{}	braces indicate that the parameters they enclose are optional
*	an asterisk following information enclosed in braces indicates that the enclosed information may be repeated up to the length of the BASIC I/O and edit buffers
input	user input is underlined
<CTRL-X>	indicates a control character where X is an alpha key
<RETURN>	angle brackets around any word refer to a specific key on the keyboard; the RETURN key is shown

Related Manuals

Related manuals include:

Title	Pub. Number
IRIS Operations Manual	SM-030-0010
IRIS R8 User Manual	SM-030-0011
IRIS Installation and Configuration Manual	SM-030-0009

CONTENTS

APPENDICES

FIGURES

TABLES

Section 1
INTRODUCTION

BASIC is an easy-to-learn yet powerful programming language well-suited to interactive use on a time-sharing computer. The features of BASIC include the following:

- Simple grammar based on a small number of statements
- Facilities for handling strings, matrices and arithmetic expressions
- Built-in editing features that facilitate debugging and program modification
- Ease of translation by an interpreter. Use of an interpreter makes it possible to write and debug problems interactively. For example, a section of a program can be written and run; lines can be added, deleted, or modified; and the revised program can be rerun immediately without waiting for a compilation

IRIS Business BASIC, the version of BASIC described here, is designed to preserve the characteristics which have made BASIC practically the universal time-sharing language. It adds further capabilities which enhance its utility, especially for business applications. The extensions of Business BASIC are:

- Extended precision decimal arithmetic - provides up to fourteen decimal digits of accuracy
- PRINT USING - provides business oriented formatting of output
- Chaining - allows large applications to be segmented
- Signalling - allows communication between programs on different ports or within a single program
- Extended function set - provides special functions and facilities
- Error branching - allows BASIC programs to detect errors and attempt correction
- Provision for large strings and arrays - increases the usefulness of string and matrix operations

- Five types of data files: formatted, contiguous, text polyfile and indexed - provide random access data storage on the disc

Each extension of Business BASIC is discussed in detail below.

Extended precision decimal arithmetic overcomes two problems in most BASIC systems: limited precision and conversion errors. Most BASIC systems represent numbers internally in floating point binary form, typically to an accuracy of 21 to 24 bits. This results in six decimal digits of precision in which the sixth may be faulty because of errors introduced through the conversion from decimal to binary and back. Business BASIC provides four precision options: one-word integers (in the range $\pm$7999) and two-, three- and four-word floating point numbers which give, respectively, six, ten, and fourteen decimal digits of accuracy. Furthermore, all numbers are carried in decimal form and the arithmetic is entirely decimal, so that conversion and its inherent errors are eliminated.

The second major extension is PRINT USING, which simplifies report generation by providing COBOL-like picture formats. These are used to position column headings, line up decimal points, float dollar signs, insert commas, and provide the other controls required for technical and financial reporting.

Chaining allows programs to be segmented for execution in a system with a small memory. Small segments also permit faster swapping for more efficient system operation.

Signalling allows programs on different ports and program segments to communicate with each other.

The extended function set includes facilities for taking floating point numbers apart and putting them back together.

Error branching allows turnkey systems to be written where the BASIC program detects errors and attempts corrective action (instead of printing an error message), perhaps by asking for additional information from the user.

The usefulness of the string and matrix operations is increased by the provision for large strings and matrices, which are limited only by the program storage available and such features as substrings, string comparisons, MAT INPUT, and matrix inversion.

All files provide random access data storage on the disc. The five types of IRIS data files provide programming flexibility.

- Formatted files may store over 16 million bytes of information each, perform item-type checking and allow random addressing of items.

- Contiguous files store large amounts of data and provide fast access, without item-type checking.

- Text files have a capacity of over 16.7 million characters in an extended file. They provide a common data base between IRIS Business BASIC and other processors.

- Polyfiles may consist of up to 64 volumes and up to 63 directories. Each volume may reside on a different logical unit, thereby eliminating size restrictions imposed on other types of files.

- Indexed files offer up to 15 directories each and provide fast random access by data content. The user specifies the key length for each directory. Index-only files may also be created.

Polyfiles and indexed files are grouped under the heading "Keyed Files", because they both employ directories.

In addition to these extensions, IRIS Business BASIC provides the following features: direct execution (calculator mode), string processing, matrix algebra and the CALL statement.

Section 2
IRIS DATA FILES

IRIS file classes include system files, processor files, program files and data files. This section describes IRIS data files and how they are used.

2.1 DEFINITION OF TERMS ASSOCIATED WITH FILES

The use of terms associated with files often varies from system to system. This section defines important terms in order to prevent confusion.

BLOCK - the fundamental unit by which IRIS manages data on disc. Each block is 512 bytes long. Normally, files are accessed by record, leaving block manipulation to IRIS.

CHANNEL - a logical connection between a BASIC program and a file (or device). It allows the program to access the file using only an internal channel number for identification. This number is associated with the actual file (or device) in the statement which opens or builds it.

DIRECTORY - a method used to find records by key. The terms "directory" and "index" are used interchangeably. Directories are described in more detail in Section 2.7.4. (Note that this is not a logical unit directory.)

FILE - a collection of data which is stored on disc. Files are created, modified and retrieved by the use of system commands or application programs. (Files may also be stored on other magnetic media, but this is outside the scope of this section.)

ITEM - a logical segment of a record. A record usually consists of several related items. The terms "item" and "field" are used interchangeably.

KEY - a keyword or identifier used for identification when searching for a record in a keyed file. A key is a character string.

RECORD - a logical segment of a file. Each record consists of a variable number of bytes that can be used to store numbers and strings of characters. Each record may be accessed individually.

2.2 INTRODUCTION TO IRIS DATA FILES

Four IRIS file types are offered: formatted, contiguous, text and keyed. Keyed files include two types of files: polyfiles and indexed files. The major features of each file type are summarized below. Further details and more strict definitions of each file type may be found under each file heading later in this section.

IRIS provides two structurally different forms of data files: contiguously-allocated and randomly-allocated. These terms refer to the organization on the disc of the data blocks which make up the file. The data blocks of a contiguously-allocated file are physically contiguous to each other, and all the blocks of the file are allocated when the file is built. The data blocks of a randomly-allocated file are allocated dynamically as they are needed. They may or may not be physically contiguous on the disc.

A formatted file is randomly-allocated and formatted. When formatting a file, the user specifies whether each item in the file is a string or a decimal number of precision one, two, three or four. Every record has the same format, which is recorded in a format map in the file header. When a formatted file is accessed, the item type and type of variable are checked. If the types do not correspond, an error is generated.

A contiguous file is contiguously-allocated. Contiguous files offer the potential of faster access than formatted files because the position of the data record may be calculated by the system without accessing the file header blocks. Because contiguous files are not formatted and do not have item-type checking, they offer greater flexibility than files which are formatted.

Text files are randomly-allocated. A text file may be regarded as a string of up to 16 million ASCII characters terminated by a null character. Text files may be accessed sequentially, but IRIS allows the additional feature of random access.

There are two types of keyed files: polyfiles and indexed files. Polyfiles and indexed files are grouped as keyed files because they both allow random access by key.

Polyfiles consist of one or more associated files; each file is called a volume of the polyfile. Each volume is contiguously-allocated. Additional volumes may be added when necessary, so polyfiles need not be entirely allocated when they are created. They may employ an optional bit map to keep track of free records. Polyfiles offer practically unlimited storage, greater key size, and higher record capacity than any other type of file.

An indexed file is contiguously-allocated. Part or all of every indexed file is reserved for a directory. When a key is created, a record may be associated with it. The record may then be identified by unique keys contained in a directory. Unused data records may be linked together to form a structure known as the free record chain.

2.3 DATA FILES ACCESS

This section describes the file handling statements, access method for each type of file, sequential and repeated record access, record locking and the channel functions.

2.3.1 OVERVIEW OF THE FILE HANDLING STATEMENTS

The following summary overviews the statements which are commonly used to manipulate files and briefly describes their functions:

BUILD	used to build the specified type of file from within a program
CLOSE	used to dissociate a channel from the file it was used to access; a file being built is deleted if the program run is stopped before the file is closed
KILL	used to delete specified files
MAT READ	used to read the elements of a matrix (or an entire string) from a file
MAT WRITE	used to write the elements of a matrix (or any variable) into a file
OPEN	used to associate a file (or a peripheral device) with the specified channel number
PRINT	used to output information to a file. When PRINTing to a file, the output is formatted as it is for PRINT. A terminating semicolon is not a record unlock command; it is a formatting directive which suppresses the RETURN at the end of the line
READ	used to read information from a file
SEARCH	used for polyfiles and indexed files to manipulate directories and manage record allocation
WRITE	used to write information into a file

The syntax of each of these statements may be found in Section 3. The BUILD, MAT READ, MAT WRITE, READ, SEARCH and WRITE statements are discussed as they apply to each file type later in this section.

2.3.2 RECORDS AND ITEMS FOR CONTIGUOUS AND TEXT FILES

The syntax of the READ, WRITE, MAT READ and MAT WRITE statements are shown below:

READ #channel{,record{,item{,delay}}};variable list{;}

WRITE #channel{,record{,item{,delay}}};expression list{;}

MAT READ #channel{,record{,item{,delay}}};array or string variable{;}

MAT WRITE #channel{,record{,item{,delay}}};array or string variable{;}

The syntax of these statements allows specification of the record and item numbers. However, contiguous files do not employ records in the strict sense and text files do not employ records and items in the strict sense. These terms have different meanings for contiguous and text files.

For contiguous files, the item refers to the byte displacement from the beginning of the record. For example, the following statement reads data into variable A from a contiguous file open on channel 0:

```
100 READ #0,R,24;A
```

This statement reads record R (R contains the integer value of the record number), beginning at byte 24. (Byte 24 is the twenty-fifth byte, because the record starts at byte zero.)

Numeric data fields must start at an even byte displacement within the file.

For text files, the record refers to the relative block number and the item refers to the byte displacement within that block. (Each "record" in a text file is 512 bytes long.) For example, the following statement accesses a text file on channel 1:

```
200 READ #1,3,15;A$
```

This statement reads block 3, beginning at byte 15.

For comparison, the following summary shows the meaning of "record" and "item" for each file type:

File type	Record	Item
text	block	byte displacement into the block
contiguous	record	byte displacement into the record
polyfile	record	byte displacement into the record
indexed	record	byte displacement into the record
formatted	record	item

2.3.3 SEQUENTIAL AND REPEATED ACCESS

When accessing sequentially numbered records, it is not necessary to specify the number of each record. Instead, the values shown below may be used as the record expression to access the records sequentially or repeatedly. If the record expression is omitted, sequential access is used by default.

Record Expr	Effect
none	Accesses the next sequential record and assumes item zero. The record number of the last access on that channel is incremented by one and used as the record number.
-1	Same as none, except that it allows specification of the item number.
-2	Repeats access of the same record. The record number of the last access on that channel is used as the record number.

Sequential access operates somewhat differently for text files, as described in Section 2.6.2.

If a statement which uses sequential or repeated access is executed after opening the file or after building a new file, record number zero is accessed.

The following lines demonstrate sequential access:

```
10 OPEN #2, "PAYROLL"
100 READ #2;A, B$
110 READ #2,6; X,Y$
120 READ #2,-1,5; A,B$
130 READ #2,-1,0; A,B$
140 READ #2,-1; A,B$
```

Line 10 opens the file named PAYROLL on channel 2. Line 100 accesses item zero of record number zero of that file. Line 110 accesses items zero and one of record number 6 of the file. Line 120 reads from record number seven of the file, because the previous access of the file on channel 2 referenced record six. Item five is read into variable A and item six is read into variable B$. Line 130 is similar to line 120; record number eight is read, since the previous access read record seven. Item zero is read into A, and item one is read into B$. Line 140 operates similarly, referencing record nine.

The following example shows repeated access of the same record. This technique is especially useful for updating a record:

```
200 READ #5; A,B,C
.
.
.
250 WRITE #5, -2,1; B+D
```

Line 200 reads items zero, one and two of the next sequential record of the file open on channel five. Line 250 updates the same record by writing the value B+D into item one.

The channel functions, described in section 2.3.5, may be used to determine the number of the record last accessed.

2.3.4 RECORD LOCKING

Whenever a READ, WRITE, MAT READ or MAT WRITE statement is used, IRIS locks the record being accessed unless the statement ends with a semicolon. When a given file may be accessed simultaneously by more than one user, record locking restricts access to the record to one user. Thus, several statements may be executed to update a record before the next user has access to the record.

For example, suppose two users named Smith and Jones are working on an inventory control system. Smith checks how many walnut desks are in stock by reading the appropriate record using a READ statement with no ending semicolon. The record is locked. If Jones attempts to access the same record, Jones' program is paused until Smith unlocks the record. Jones may then read an accurate count of the remaining inventory.

IRIS locks individual records (not entire blocks like some other operating systems). If two or more records are stored in one block, only the record being accessed is locked; other records in that block may be accessed. A record which is two blocks long or which extends over a block boundary is entirely locked and therefore is safeguarded from simultaneous access. One record per channel only may be locked at any one time.

For text files, the exact block number and byte offset at which the file is currently positioned is locked. The current position is the next byte to be read or written. A record-lock condition occurs in a text file only if a READ or WRITE is directed to the exact current position.

Upon reading or writing to a new record, the old record remains locked until the new record is successfully accessed. As a result, a channel which is waiting to access a locked record can itself have a record locked.

A record may be accessed without being locked by including a semicolon at the end of the READ, WRITE, MAT READ or MAT WRITE statement.

A record is automatically unlocked if the program is terminated (except by a CHAIN statement); if the file is closed; if another access is made to the same record (without re-locking it); or if any other record is successfully accessed on the same channel. A record may also be unlocked by executing a statement of the form

```
WRITE #c;;
```

where c is the number of the channel on which the record is locked.

2.3.4.1 Deadly Embraces

A deadly embrace may occur if two users are each trying to access the record which the other user has locked. Each user is paused indefinitely while waiting for the other user to unlock the desired record. Deadly embraces may also occur between three or more users.

2.3.4.2 Delay Clause

A delay clause may be included in READ, WRITE and PRINT statements to generate an error when a program is paused longer than a specified period of time because a record (or device) is locked. If the program is paused longer than the specified delay period, error 123 is generated. The syntax of the delay clause is shown under READ, WRITE and PRINT statements in Section 3. The delay is specified in tenths of a second.

The method of recovery from error 123 depends on the type of file (or device) involved, the statement used and whether access was random or sequential. For formatted, contiguous or keyed files, the aborted I/O may simply be re-executed. For text files, the statement should be re-executed only if random access was used. If the error occurred during sequential access to a text file, the operation might have been only partially completed and the file position might have changed. A record-locked error on any data file may always be followed by random access to the same record or to a different record.

In the following example, if record zero, item zero of the file named "TEMP" remained locked by another user longer than two-tenths of a second, the system would generate error 123 at line 30:

```
10 DIM A$[100]
20 OPEN #1, "TEMP"
30 READ #1,0,0,2;A$
40 PRINT A$
```

2.3.5 THE CHANNEL FUNCTIONS

The channel functions provide the current file size and current access position within a file. The function CHF(x) may be used in any arithmetic expression, like any other function. The function type and channel number are combined in the expression (x) by adding the function type (0, 100, 200 or 300) to the appropriate channel number. For example, the value 302 selects the function type 300 and channel number two.

Each of the channel functions and its result is shown below, where c is the number of the channel on which the file is open.

Function	Result
CHF(c)	For the file open on the specified channel, returns an integer one greater than the record number of the highest numbered record into which at least one item has been written. If the file is a contiguous file, the value returned is the number of records in the file, regardless of where data has been written in the file.
CHF(c+100)	Returns the number of the record last accessed on the specified channel. If the file has not been accessed since it was opened or built, returns -1.
CHF(c+200)	Returns the byte displacement into the block last accessed on the specified channel. This function is most useful on text files.
CHF(c+300)	Returns the record size in words.

If a device is open on the specified channel, CHF(c) yields zero and the other channel functions yield unpredictable results. Error 49 occurs if the specified channel number is illegal or is not open.

2.4 FORMATTED FILES

Formatted files have the following features:

- randomly-allocated
- formatted; that is, the type and length of each item in a record are specified
- format is recorded in a format map in the file header
- each item may be an ASCII string, a decimal number, or one or more binary words
- item type and length are checked whenever information and length are transferred to or from a formatted file
- every record of the file has the same format
- each record may be up to 1 block (256 words) long and may have up to 64 items
- records may not span block boundaries; an integral number of records is stored in each block
- maximum capacity is 32768 blocks

This section describes the use and creation of formatted files.

2.4.1 CHARACTERISTICS OF FORMATTED FILES

Records may be written in any order, and records may be skipped. However, if a skipped record is read from a block which has been allocated, a "Record not written" error is not returned. A "Record not written" error occurs only if the program attempts to read a record in a block which has not been allocated.

For example, suppose an allocated block contains records four, five and six. Records four and six have been written into. If the program attempts to read record five, zero values would be returned; a "Record not written" error would not occur.

2.4.2 ACCESSING FORMATTED FILES

This section includes examples of the use of WRITE, READ, PRINT, MAT READ and MAT WRITE statements on formatted files. The general syntax of these statements is shown in Section 3.

Error 15 is generated if the program variable type is different from the file item type, except when data is transferred between binary and numeric types.

Data are always translated to the precision of the destination variable/item. However, if a number outside the range +/-7999 is read into a 1% variable, the value +/-7999 is read or written and error 15 is generated.

For example, suppose a formatted file had the following record layout:

Item	Variable	Length/precision
0	A$	12 character
1	A1	1%
2	A2	2%
3	A3	3%
4	A4	4%

The following lines manipulate data in this file:

```
10 DIM 3%,A3,4%,A4
760 READ #C,R,4;A3
```

Line 760 reads item 4, a precision 4% variable, into A3, which is a 3% variable. This is accomplished by modifying the 4% value to fit into the 3% variable.

2.4.2.1 WRITE for Formatted Files

For formatted files, a single WRITE statement may write into sequential items of a single record only. An error results if a variable type does not match the item type in the file.

An explicit WRITE to record zero is used to define the format of a formatted file newly built from within a BASIC program, as described in Section 2.4.4.

The following examples show the use of the WRITE statement on formatted files.

```
200 WRITE #4, 19; F, Y1+3, "EXAMPLE", D-E
```

Line 200 sets item zero of record 19 of the file open on channel 4 to the value of F, item one of the same record to the value of (Y1+3), item two to the string value EXAMPLE, and item three to the value of (D-E). Items zero, one and three must be formatted as numerics; item two must be formatted as a string.

```
700 WRITE #C-1, 2*R, 8; 0, 2 A, M$[4,Q];
```

In line 700, the channel number is given by the value of the expression (C-1) and the record number is given by the value of the expression (2*R). The item number may also be an expression. The line sets item 8 of the specified record to 0, item nine to the value of (2 A), and item ten to characters four through the value of Q of the string M$. The final semicolon leaves the specified record unlocked while the statement is executed, as described in Section 2.3.4.

2.4.2.2 READ For Formatted Files

In formatted files, sequential items of a single record only may be read by each READ statement. An error results if a variable type is not compatible with the item type in the file.

The following examples show the use of the READ statement on formatted files.

```
400 READ #2, 6; D, W$, K[7, A-2]
```

Line 400 reads item zero of record 6 of the file open on channel 2 into variable D, item one (which must be a string) into string variable W$, and item two into the element of K at row 7, column A-2. The record is left locked.

```
500 READ #C[4]+1, R8, 5; F$[4], J, J;
```

In line 500, the channel number is given by the value of the expression (C[4]+1) and the record number is given by the value of R8. The item number may also be an expression. The line reads items five, six and seven of the specified record. Item five is read into F$ beginning at character four; characters one through three remain unaffected. Item six is read into J, then item seven is read into J, thereby overwriting the value of item six. This technique may be used when the value of item six is not required. The final semicolon leaves the specified record unlocked while the statement is executed, as described in Section 2.3.4.

Numeric expressions are allowed for the channel, record and item, but an item value from the file may not be read into an expression.

When reading values from a formatted file into string variables listed in a READ statement, the system terminates the READ to a given string variable when a zero byte is encountered in the file or when the string variable is full, whichever occurs first. The system then proceeds to the next item listed and begins to READ into it. If a string specified in a READ statement is larger than the item in the file, the system reads the item into the string and adds a zero byte, leaving some unused space in the string.

2.4.2.3 PRINT for Formatted Files

The PRINT statement can be used on formatted files, and results in output of one or more strings. Output must be to an ASCII string item. The use of PRINT on formatted files requires a thorough knowledge of how PRINT operates. POINT 4 suggests the use of alternate statements, such as LET USING.

2.4.2.4 MAT WRITE Statement for Formatted Files

When using MAT WRITE, the entire matrix, string or variable is written into a single item. The item type in the file must be binary.

If the item is too small, then the data are truncated; no error message is given. No data conversion takes place; the user's program must ensure that the data will later be read back into the same type of variable.

The following examples show the use of the MAT WRITE statement on formatted files.

```
910 MAT WRITE #1, 20; A
```

Line 910 writes all elements of the matrix A into item zero of record 20 of the file open on channel 1.

```
300 MAT WRITE #C, 2*R, 8; B
```

Line 300 writes all elements of the matrix B into item 8 of the record given by the value of (2*R) of the file open on channel C.

```
700 MAT WRITE #L, R, 3; B$
```

Line 700 writes the string B$ into item 3 of record R of the file open on channel L. Item three must be a binary field. The system WRITEs the entire contents of B$, including all zero bytes, to the end of B$ or until the item is full, whichever occurs first.

2.4.2.5 The MAT READ Statement for Formatted Files

When using MAT READ, the entire matrix or string is read into a single item. The type of the item must be binary. If the item is too small, then the data are truncated; no error message is given. No data conversion takes place; the program must ensure that the data is read into the type of variable which matches the data form.

The following examples show the use of the MAT READ statement on formatted files.

```
400 MAT READ #2, 21; M
```

Line 400 reads all the elements of the matrix M into item zero of record 21 of the file open on channel 2.

```
600 MAT READ #J, R+3, 7; N
```

Line 600 reads all the elements of the matrix N into item 7 of the record given by the value of (R+3) of the file open on channel J.

```
800 MAT READ #3, 10, 2; A$
```

Line 800 reads the string A$ into item 2 of record 10 of the file open on channel 3. Item two must be a binary field. The system READs the entire contents of B$, including all zero bytes, until A$ is full or to the end of the item, whichever occurs first.

2.4.3 CREATING A FORMATTED FILE USING FORMAT

The FORMAT processor, described in the IRIS R8 User Manual, may be used to create a formatted file. To create a formatted file using FORMAT, enter a statement of the following form at the IRIS system prompt:

```
#FORMAT filename
```

where filename is the name of the formatted file to be created. The cost, protection and logical unit on which the file will be created may also be specified ahead of the filename.

The processor will then prompt for the format of each item in a record. The appropriate codes are described in the IRIS R8 User Manual.

NOTE

For an item which will be accessed by a matrix statement (MAT READ or MAT WRITE), the item type must be binary. For example, a 5x10 matrix dimensioned at 3% has six columns (0 through 5) and eleven rows (0 through 10). The item for this matrix must be formatted B198 (6x11x3). A 29-byte string must be dimensioned for 30 bytes, which is an even number. Its item type would be B15.

Refer to the IRIS R8 User Manual for details on the FORMAT processor.

2.4.4 CREATING A FORMATTED FILE USING THE BUILD STATEMENT

To create a formatted file from within a BASIC program, the BUILD statement is used followed by an explicit WRITE to record zero. The explicit WRITE to record zero sets the format of the record; this is called "auto-formatting" the file. For detailed syntax of the BUILD and WRITE statements, refer to Section 3.

To build a formatted file, execute a statement of the following form:

```
BUILD #c, "FMTFILE"
```

where "FMTFILE" is the name of the formatted file to be created and c is the channel number expression. This statement must be followed by a WRITE statement of the following form:

```
WRITE #c, 0; X, Y, Z
```

where c is the channel on which the formatted file being created is open, zero is the record number and X, Y and Z are the items. This statement automatically stores the format of each specified item into the file's format map in its header block. The first WRITE to a record other than zero ends auto-formatting.

The cost, protection and number of the logical unit on which the file will be built may be specified within the quotation marks, ahead of the filename. For example, the following statement creates a formatted file named "FMTSAMPLE" with protection 33, and cost $10 on logical unit 3:

```
BUILD #C,"<33> $10.00 3/FMTSAMPLE"
```

The detailed syntax of the BUILD statement is listed in Section 3.

2.5 CONTIGUOUS FILES

Contiguous files have the following features:

- contiguously-allocated, so the entire file is allocated when it is first built
- every record must be of the same size; multiple records may be read or written by a single statement to have the effect of variable record size
- record lengths should be a factor of the block size to prevent records from spanning block boundaries. Records which span block boundaries may require more than one disc access to transfer
- unformatted; therefore, there is no item type checking. The program must ensure that data are read back into the same variable types as those from which they were written
- maximum record length is 32768 words
- maximum file size is limited by the size of the logical unit only

This section describes the use and creation of contiguous files.

2.5.1 CHARACTERISTICS OF CONTIGUOUS FILES

A contiguous file can be built only if an adequate number of physically contiguous blocks are available on the specified logical unit. All blocks are allocated initially. When the file is built or opened, all information required to calculate the disc address where the data transfer is to begin is stored in memory.

Items are addressed by record number and byte position. A single data transfer may span record boundaries because the address specifies only a starting location.

2.5.2 ACCESSING CONTIGUOUS FILES

This section includes examples of the use of the WRITE, READ, PRINT, MAT READ and MAT WRITE statements on contiguous files. The general syntax of these statements is shown in Section 3.

For contiguous files, the specified record number is the reference point at which the operation begins. The "item" refers to the byte displacement from the beginning of a record. The transfer continues, possibly across record boundaries (and possibly across disc block boundaries) until all the data on the expression list have been written. If the end of the file is reached, the data are truncated with no error message. Two or more records may be transferred in one command.

Reading or writing beyond the specified record is possible in contiguous files if the "item" value (byte displacement) is beyond the defined range or if the expression list contains more data than will fit in one record.

In order to calculate the record size of randomly accessed individual items, individual item types must be understood. For a numeric item, its size (and displacement to the next item) equals the number of bytes equivalent to the precision of the variable if a variable is written, or four words if an expression is written. Any numeric constant is considered an expression and is written in four-word precision. The following formulas may be used to calculate the size and displacement of a string item.

Source string	Number of bytes (n)	
A$	n = d+1	
A$(x)	n = d-x+2	(0 < x <= d)
A$(x,y)	n = y-x+1	(x <= y <= d)
"..."	n = c+1	

where

n - the number of string bytes written into the file, including a terminating zero byte, if written

A$ - any string variable

d - the dimension of the string variable A$

x and y - numeric expressions

c - the number of characters in a literal string

Writing into a file is always terminated if a zero byte is encountered in the source string, and the zero byte is written. Writing of the zero byte may be prevented only by the use of two subscripts on a string variable.

The above formulas for n can also be used to determine the byte displacement from the beginning of one item to the starting location of the next item to be written sequentially by the same statement. The value of n is determined by the dimensioned length of the string, not by the current length of the string.

If the next item written is numeric (or any word-oriented data), and the displacement results in an odd byte displacement from the beginning of the record, then one byte in the file is skipped so that the following field begins on a word boundary. The skipped byte is not changed in the file.

The system maintains a byte pointer during READ operations. A READ into a string variable terminates either on a zero byte in the file or by filling the string, whichever occurs first. A READ into a numeric variable is terminated by transferring the number of words appropriate to the precision of the variable. In both cases, the byte pointer in the file is moved n bytes (as defined above) from the beginning of the field just read. If the pointer does not land on a word boundary and the next variable in the expression list is numeric, the pointer is advanced to the next word boundary and the READ begins there.

Take care not to supply a byte number which is in the middle of a previously written numeric item.

2.5.2.1 WRITE for Contiguous Files

The following examples show the use of the WRITE statement on contiguous files.

```
10 DIM A$[13],D$[20],2%,B,E
100 WRITE #C,7,21;A$,B,D$[1,8],E
```

Line 100 writes the string value of A$ followed by a terminator character into the file open on channel C, beginning at the twenty-second byte of record seven. The values of the remaining variables are written sequentially into the record. No null is written following D$[1,8]. The following columns show how the byte displacement of the items in line 100 is determined (assuming A$ is filled):

Variable	Number of Bytes	Displacement
A$	n = 13+1 = 14 bytes	0-13
B	2% precision = 4 bytes	14-17
D$[1,8]	n = (8-1)+1 = 8 bytes	18-25
E	2% precision = 4 bytes	26-29

```
200 WRITE #0,7,26;E
```

Line 200 writes the value of variable E into the file open on channel 0, beginning at byte 26 of record 7.

2.5.2.2 READ for Contiguous Files

The following example shows the use of the READ statement on contiguous files.

```
10 DIM W$[10],2%,D

200 READ #2,6;D,W$
```

Line 200 reads record 6 of the file open on channel 2 into variable D starting with byte zero. A total of four bytes (bytes zero, one, two and three) are transferred to D. The line then reads bytes four and on into W$, continuing the transfer until W$ is filled or until a zero byte is encountered in the file.

2.5.2.3 PRINT for Contiguous Files

The following examples show the use of the PRINT statement on contiguous files. When PRINT is executed, the output is formatted as it is for PRINT.

```
170 PRINT #3, 21, 6; 254.6, D+E, F/6
```

Line 170 prints the value 254.6 beginning 6 bytes into record 21 of the file open on channel 3, followed by the values of (D+E) and (F/6) to successive bytes.

```
240 PRINT #C, J; X(1), Y(3,6); Z+W
```

Line 240 prints the value of element 1 of variable X beginning at byte zero of record J of the file open on channel C, followed by the values of element 3,6 of array Y and (Z+W) to successive bytes.

```
310 PRINT #C2+1, R-1; USING B$(15); P; J; M$;
```

In line 310, the channel number is given by the value of the expression (C2+1) and the record number is given by the value of (R-1). The byte displacement may also be an expression. The line prints the values of the variables P, J and M$ beginning at byte zero and proceeding successively, using the format specified in the format string B$(15). The final semicolon is treated as a formatting directive which suppresses the RETURN at the end of the line, not as a record unlock command.

2.5.2.4 The Matrix Statements for Contiguous Files

When using MAT READ or MAT WRITE, the entire matrix, string or variable is read in one statement. For contiguous files, the "item" is used as a byte displacement into the record; however, all transfers are word oriented. If an odd byte displacement is given, then the transfer begins at the next higher byte displacement.

If writing a string variable, then the entire string as dimensioned is written, including all zero bytes within that string. If the string's dimension is odd, then a zero byte is written at the end of the string bytes. As a general rule, the system always adds a zero byte if necessary to fill the last word.

The following formulas may be used to calculate the number of words that will be transferred.

For an array variable: (r+1)(c+1)p

For a string variable: INT [(d+1)/2]

where
    r - row dimension of the matrix
    c - column dimension of the matrix
    p - number precision of the matrix
    d - string dimension (number of bytes)

2.5.2.4.1 MAT WRITE STATEMENT FOR CONTIGUOUS FILES

The following examples show the use of the MAT WRITE statement on contiguous files.

```
910 MAT WRITE #1, 20; A
```

Line 910 writes all elements of the matrix A beginning at byte zero of record 20 of the file open on channel 1.

```
300 MAT WRITE #C, 2*R, 8; B
```

Line 300 writes all elements of the matrix B beginning at byte 8 of the record given by the value of (2*R) of the file open on channel C.

```
700 MAT WRITE #L, R, 4; B$
```

Line 700 writes the entire string B$, including zero bytes, beginning at byte 4 of record R of the file open on channel L.

2.5.2.4.2 THE MAT READ STATEMENT FOR CONTIGUOUS FILES

The following examples show the use of the MAT READ statement on contiguous files.

```
400 MAT READ #2, 21; M
```

Line 400 reads all the elements of the matrix M beginning at byte zero of record 21 of the file open on channel 2.

```
600 MAT READ #J, R+3, 8; N
```

Line 600 reads all the elements of the matrix N beginning at byte 8 of the record given by the value of (R+3) of the file open on channel J.

```
800 MAT READ #3, 10, 2; A$
```

Line 800 reads the entire string A$, including zero bytes, beginning at byte 2 of record 10 of the file open on channel 3.

2.5.3 CREATING A CONTIGUOUS FILE USING FORMAT

To create a contiguous file using FORMAT, enter a statement of the following form at the IRIS system prompt:

```
#FORMAT [records:words] filename
```

where

```
    records - number of records in the file
      words - number of words per record
   filename - name of the formatted file to be created
```

The cost, protection and logical unit on which the file will be created may also be specified ahead of the [records:words] specification.

FORMAT then displays the size of the file in blocks.

Refer to the IRIS R8 User Manual for detailed syntax and more information on FORMAT.

2.5.4 CREATING A CONTIGUOUS FILE USING THE BUILD STATEMENT

To create a contiguous file from within a BASIC program, the number of records and number of words per record is specified in a BUILD statement. For the detailed syntax of the BUILD statement, refer to Section 3.

To build a contiguous file, execute a statement of the following form:

```
BUILD #c, "[10:256] CONTIGFILE"
```

where

```
          c - channel number expression
         10 - number of records
        256 - number of words per record
 CONTIGFILE - name of the contiguous file to be created
```

The cost, protection and number of the logical unit on which the file will be built may be specified within the quotation marks before the [records:words] specification. For example, the following statement creates a contiguous file named "CONSAMPLE" with protection 33 and cost $20 on logical unit 4:

```
BUILD #C, "<33> $20.00 [10:256] 4/CONSAMPLE"
```

The detailed syntax of the BUILD statement is listed in Section 3.

2.6 TEXT FILES

Text files have the following features:

- randomly-allocated, so blocks are allocated to the file only when they are needed
- unformatted
- system regards text files as a single string of zero to 16,777,215 characters (or the size of a logical unit, whichever is less) terminated by a null byte (binary zero)

This section describes the use and creation of text files.

2.6.1 CHARACTERISTICS OF TEXT FILES

Characters in a text file are packed and span block boundaries. Each character is stored as a seven-bit ASCII code with the eighth bit set unconditionally to one. If the file contains any RETURN codes, the system recognizes them as end-of-line markers.

Although text files are not organized around data records in the same sense as other data file types, a fixed record length of 512 bytes (one disc block) is adopted to facilitate random addressing.

2.6.2 ACCESSING TEXT FILES

This section includes examples of the use of the WRITE, READ and PRINT statements on text files. The general syntax of these statements is shown in Section 3. The matrix statements may be used on text files. However, if non-string data is output to a text file, other programs and utilities may be unable to access the data correctly. For this reason, MAT READ and MAT WRITE are not discussed in this section.

For text files, the record refers to the block and the item refers to the byte displacement within that block.

Text file access is usually sequential and line oriented, where a line is defined as any string of characters up to and including a RETURN code. Text files may also be accessed randomly as a series of one-block records.

Sequential and repeated access of a "record" operates differently for text files than for other file types. The values shown below may be used as the record number expression for the designated access.

Record Expr	Effect
none	Accesses from the next byte to the next RETURN code or zero byte in the file (or until the specified variable is filled, whichever occurs first).
-1	Same effect as none.
-2	Has the same effect as -1 except that when READing, the transfer is terminated by the length of the variable being read into or by the end of the file, not by a RETURN code.

If a statement which employs sequential access (as described above) is executed after opening or building a text file, byte zero of block zero is accessed.

2.6.2.1 WRITE For Text Files

The following example shows the use of the WRITE statement on text files.

```
700 WRITE #1; A$, D$, C$
```

Line 700 concatenates the values of A$, D$ and C$ and writes them into the file open on channel 1 as a single string, with no RETURN code at any point. If this statement were the first executed after opening or building a text file, byte zero of block zero would be accessed. Otherwise the strings would be written immediately following the data last accessed. If line 700 were repeated, the file would show a continuous concatenation of the fields written.

The WRITE statement does not add any return codes to the output. The program may output zero, one or more return codes in the output variables.

For text files, all expressions in the expression list of the WRITE statement must be string expressions.

2.6.2.2 READ for Text files

The following examples show the use of the READ statement on text files.

```
400 READ #1;A$;
```

Line 400 reads from the first byte following the data last accessed up to the first RETURN found in the file open on channel one into the string variable A$. If this statement were the first executed after opening a text file, the data from byte zero of block zero to the first RETURN code would be read. The final semicolon disables record locking.

```
500 READ #0,-2;B$;
```

Line 500 reads from the first byte following the data last accessed until filling B$ from the file open on channel 0. The final semicolon disables record locking.

When the end of a text file is encountered, the system returns either an empty string or a "Record not written" error.

2.6.2.3 PRINT for Text Files

The following examples show the use of the PRINT statement on text files.

```
840 PRINT #1,-1;A$
```

Line 840 prints the contents of A$ followed by a RETURN code to the file open on channel 1. If this statement were the first executed after opening or building a text file, byte zero of block zero would be accessed. Otherwise the string would be written immediately following the data last accessed.

The following codes may be included in PRINT statements:

Code	Meaning
\207\	ring bell
\214\	form feed
\215\	carriage return

2.6.3 CREATING A TEXT FILE USING A TEXT EDITOR

To create a text file using EDIT, enter a command of the following form at the IRIS system prompt:

```
#EDIT, filename
```

where filename is the unique name of a file to be created. EDIT prompts for further commands with an asterisk. Enter XEND to exit the editor. The cost, protection and logical unit on which the file will be created may also be specified ahead of the filename.

The IRIS R8 User Manual describes the use of EDIT in detail.

2.6.4 CREATING A TEXT FILE USING THE BUILD STATEMENT

To build a text file from within a BASIC program, execute a statement of the following form:

```
BUILD #c, +"TEXFILE"
```

where

```
        c - channel number expression
        + - indicates that the file is to be a text file
  TEXFILE - name of the text file to be created
```

The cost and protection of the file and the logical unit on which it will be built may be specified ahead of the filename, within the quotation marks. For example, the following statement creates a text file named "TXSAMPLE" with protection 33 and cost $10 on logical unit 4:

```
BUILD #c, +"<33> $10.00 4/TXSAMPLE"
```

The detailed syntax of the BUILD statement is listed in Section 3.

2.7 KEYED FILES

Keyed files have the following characteristics:

- unformatted
- contiguously-allocated
- allow random access by key
- may have two parts: the data portion and the directory portion
- directories are accessed via the SEARCH statement; data records are accessed in the same way as contiguous files
- keyed files may have more than one directory

2.7.1 TYPES OF KEYED FILES

There are two types of keyed files: indexed files and polyfiles.

2.7.1.1 Indexed File Features

Indexed files have the following characteristics in addition to those of keyed files:

- maximum key size is 30 bytes; maximum number of 30-byte keys per file is 1830
- key size must be specified in word increments (multiples of two bytes)
- maximum number of data records is 65,535
- require the use of search mode seven (described in Section 2.7.5.7) to redistribute directory keys; this process necessitates the suspension of timesharing
- the size of a given directory may not be changed
- employ a free record chain to keep track of free records; it is possible to disrupt this chain by improper programming
- data records start at a variable location depending on the size of the directory

2.7.1.2 Types of Indexed Files

Indexed files may include up to 15 directories, as specified by the user, within the data file itself. In addition, indexes may be set up and used independently of any data file. Files used in this way are called "index-only" files. The key length may be different for each directory.

2.7.1.3 Polyfile Features

Each polyfile is made up of one or more associated contiguous files, called "volumes" of the polyfile. Although each volume is contiguously-allocated and must be allocated in its entirety when built, additional volumes may be added to the polyfile when necessary. Polyfiles offer practically unlimited storage, greater key size and higher record capacity than any other type of file.

The volumes may reside on different logical units, except that no volume may reside on logical unit zero. The volumes are associated by the master volume, which holds linkage information in its header block. The master volume is always volume zero of the polyfile.

Polyfiles have the following characteristics:

- key size may be specified in exact number of bytes; maximum key size is 121 bytes
- keys are redistributed automatically, so search mode seven is not needed
- directories may be enlarged by dynamically adding new directory extension volumes
- a bit map keeps track of free records; this bit map cannot be disrupted by user programming
- data records always begin at record zero
- maximum number of keys per polyfile depends on the key size:
  - 121-byte keys = 8 million keys per polyfile
  - 60-byte keys = 20 million keys per polyfile
  - 32-byte keys = 40 million keys per polyfile
- maximum number of data records is 4,128,768
- maximum volume size is 65536 blocks (including the header) or the size of the logical unit, whichever is smaller

Polyfiles can not be transferred by the MAGTAPE utility or any other program which uses the $MTAS file transfer function.

2.7.1.4 Types of Polyfile Volumes

Three types of polyfile volumes exist:

- data volume
- base directory volume
- directory extension volume

It is possible to reserve space on an LU for a polyfile or to link a file to a polyfile via the polyfile CALL (91), without structuring the space or file as a data, directory or directory extension volume. This reserved space or linked file is called an unstructured volume of the polyfile.

2.7.1.4.1 POLYFILE DATA VOLUME

A polyfile may have a maximum of 64 data volumes, numbered 0 to 63.

The record length must be the same throughout every volume of a given polyfile. Records may cross block boundaries and are numbered sequentially through ascending volumes. For example, if volume two of a polyfile has 100 records, volume six has 200 records, and volume seven has 300 records, then records 0-99 will be found in volume two, 100-299 in volume six and 300-599 in volume seven.

New data volumes cannot be inserted into the existing data volume numbers because the system does not renumber records. It is best to allow the system to assign data volume numbers or number them as low as possible.

Polyfiles may employ a bit map to keep track of free records. The entire polyfile must be either mapped or unmapped; mapped and unmapped data volumes may not be combined in the same polyfile.

2.7.1.4.2 POLYFILE DIRECTORY AND DIRECTORY EXTENSION VOLUMES

Each directory volume may have a maximum of 15 directories. When a directory volume is filled with keys, a directory extension volume may be used. An extension is used to extend a given directory volume only; it provides a pool of additional disc blocks which are available to any directory in the associated base directory volume.

Section 2.7.4 describes the structure of keyed file directories.

2.7.2 MANAGING FREE RECORDS

Indexed files use a free record chain to keep track of free records. Polyfiles may employ a bit map to keep track of free records. (Record allocation may be managed by the program itself, if the programmer chooses to develop a free record structure.)

2.7.2.1 Indexed File: Free Record Chain

When an indexed file with internal directories is created, the data records are linked together in the free record chain. To allocate a record, the user obtains the number of an unused record from the free record chain and uses that record for the new file entry. To release a record, the user specifies the number of the record to be released using search mode one, and the record is returned to the free chain.

The free record chain is maintained on a Last-In-First-Out (LIFO) basis. The file header contains the number of the first record on the chain. The first word of that record contains the record number of the next free record, which contains the record number of the next free record, and so on.

When a program requests a free record, the first record on the chain is allocated to the program and the record number contained in its first word is inserted into the file header.

WARNING

The free record chain can be destroyed when data written to one record overwrites portions of the next record.

2.7.2.2 Polyfiles: Bit Map

Polyfiles use bit maps to allocate records in data volumes and disc blocks in directory volumes. Bit maps are optional for data volumes, but are required in directory volumes. All data volumes in a given polyfile must be mapped or all of them must be unmapped.

Each bit in a map block is used to map a block or record (in a data volume each bit maps a record number; in a directory each bit maps a disc block). Any map block can map up to 4096 units.

Each polyfile bit map also maps itself; thus, when allocating a new record, the system does not have to search through an entire bit map to find a bit which indicates a free record.

2.7.3 CHARACTERISTICS OF DATA PORTION OF KEYED FILES

The data portion of an indexed file or data volume of a polyfile has the same characteristics as contiguous files. Section 2.5 describes contiguous files.

2.7.4 STRUCTURE OF KEYED FILE DIRECTORIES

In keyed file directories, each key in the master level is linked to the corresponding level below it, as shown in Figure 2-1. Each key in the subsequent levels is linked to the corresponding level below it. Finally, each key in the fine level references a data record by its real record number.

Keys within each block are sorted in ascending order. To locate a given key, the system scans the master, coarse and fine levels in turn for a key of a matching or greater value. Each time a key of matching or greater value is found, the system follows the appropriate pointer to the subsequent level.

Directories for indexed files and polyfiles have one major difference. Indexed files have only three levels: master, coarse and fine. Polyfiles must have at least a one-block master level and a one-block fine level, but may have any appropriate number of coarse levels between.

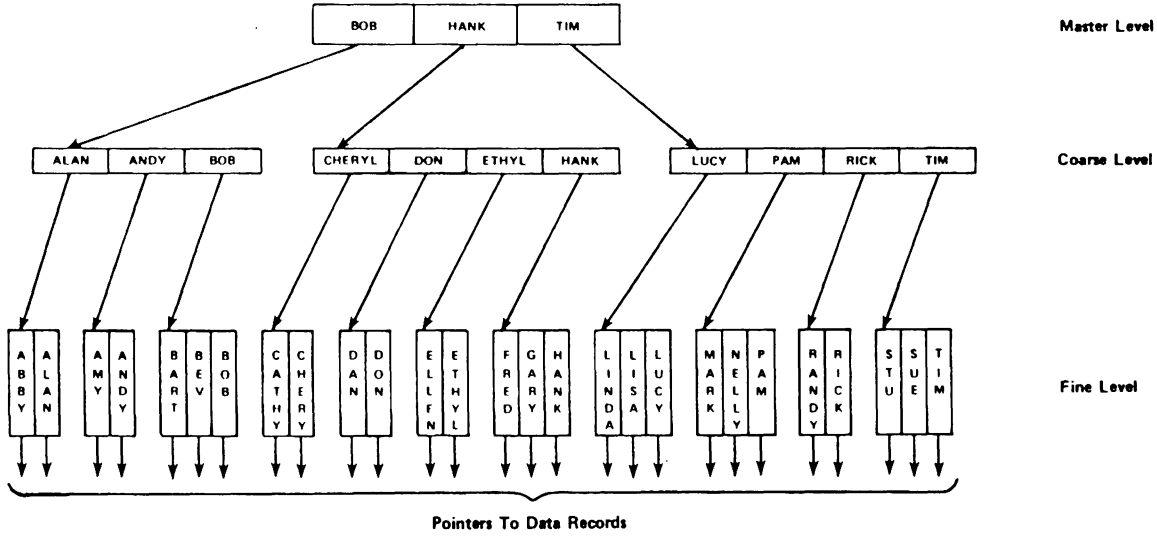


Figure 2-1. Directory Structure of Keyed Files

2.7.4.1 Indexed File Directories

Each directory of an indexed file consists of three levels: master, coarse and fine. The master level is always one disc block long. The sizes of the coarse and fine levels depends on the number of data records in the file.

The keys in each level are sorted in ascending alphabetic order. Each entry in the master level contains the address of the corresponding block of coarse level entries and the highest key held by that block. Each entry in the coarse level contains the address of the corresponding block of fine level entries and the highest key held by that block. Each entry in the fine level contains a numeric value, which is usually a record number. The numeric value need not be unique in each entry.

The maximum number of data records that can be indexed depends on the longest key length to be used. The following table summarizes the number of keys per block of the directory and the maximum number of indexed data records for various key lengths:

Longest Key Length K (number of words)	Keys per Block N=254/(K+1)	Max Number of Indexed Data Records =(N^3-N)/2
1	127	65534*
2	84	65534*
3	63	65534*
4	50	62475
5	42	37023
6	36	23310
7	31	14880
8	28	10962
9	25	7800
10	23	6072
11	21	4620
12	19	3420
13	18	2907
14	16	2040
15	14	1680

*maximum number of records in any file

The system uses two words from each block of each directory. Each key has a one-word pointer associated with it. Therefore, the number of keys per block may be calculated using the following equation:

N=INT 254/(L+1)

where N is the number of keys per block and L is is the length of each key in words. The string supplied for each key may be up to 2L bytes long.

Because the master level is always one block long, it may contain N keys. Each key in the master level points to one block on the coarse level, so the coarse level may have a maximum of N blocks.

Theoretically, there could be N 2 blocks in the fine level pointing to (N 3)-1 data records (one dummy key is inserted by the system at the end of each level).

In practice, however, the insert and delete key algorithms may cause every other block in the fine or coarse levels to contain only one key. As a result, the number of blocks in the fine level (F) must be:

```
F=2*R/(N+1)
```

where R is the number of data records to be indexed. In addition, the number of blocks in the coarse level (C) must be:

```
C=F/(N-1)
```

where F is the number of blocks in the fine level and N is the number of keys in the master level.

If a fraction results from the calculation, the number of blocks is rounded up.

There must be at least two blocks in the fine and coarse levels.

2.7.4.2 Polyfile Directories

Each directory of a polyfile begins with a one-block master level and a one-block fine level. When the fine level requires more keys than can be indexed by the master level, the master level is split to create an intermediate level, and a new one-block master level is formed. The maximum number of levels allowed is 127.

2.7.5 USE OF THE SEARCH STATEMENT

Indexed file and polyfile directories are accessed via the SEARCH statement. The SEARCH statement has seven modes which may be used to manipulate keys and directories.

The syntax of the SEARCH statement is shown below:

SEARCH #channel, mode, directory expr; K$, R, S

where

channel - channel number expression giving the number of the channel on which the appropriate file is open

mode - search mode expression, as defined in the following subsections

directory expr - expression giving the number of the appropriate directory associated with the file or which may be set to defined values to provide miscellaneous directory information, as defined in the following text

K$ - any string variable which contains the appropriate key

R - any variable which will contain or receive the record number of the key being acted upon

S - a variable which receives error status as shown in Table 2-1 or which may be set to defined values to provide miscellaneous search functions, as defined in the following subsections

TABLE 2-1. SEARCH ERROR STATUS

Value	Meaning
0	no error, operation was successful
1	operation was unsuccessful (usually indicates key not found)
2	end of directory (when inserting a key, indicates directory is full)
3	end of data; all data records are allocated
4	file has no index
5	undetermined error or, for polyfiles, file structure error
6	directory number not in sequence
7	file is not contiguous
8	indexed file or polyfile volume is already indexed
9	the value of the record number (R) is negative or too large
10	too many directories: for indexed files, the limit is 15 per file; for polyfiles, the limit is 63 per volume/polyfile
11	for indexed files: master directory level exceeds one block; for polyfiles: volume not found (possible structure error)
12	for indexed files: directories exceed size of file; for polyfiles: volume too small
13	no such directory
14	file not indexed
15	data volume number is less than pre-existing data volume
16	data volume map request not consistent with pre-existing volumes
17	data volume does not have record length matching that of the polyfile

TABLE 2-1. SEARCH ERROR STATUS (Cont)

Value	Meaning
18	block/record out of range
19	record was not allocated (already released)
20	volume has no bit map

For indexed files, status 5 (undetermined error) occurs in four cases: if the system's Auxiliary Buffer Area (ABA) is less than 1004 words octal; if an illegal command is given; if K$ is dimensioned shorter than the specified directory's key; or if the file is not structured as expected (for example, if there are less directories than the specified directory number).

Errors 6 through 12 occur in mode zero operations only.

The record number and status variables (R and S) must contain legal values (0 through 32767, 0 through 2^31-1) on entry even if those values are not used in a given mode.

The modes provided by SEARCH are summarized below:

Mode	Description
0	directory definition
1	directory information
2	key match
3	next key
4	insert a key
5	delete a key
6	unused
7	reorganize directory (not used on polyfiles)

Each of these modes are described in detail in the following subsections.

2.7.5.1 Search Mode 0: Directory Definition

Search mode zero is used when setting up a new file. Line 50 is an example of search mode zero:

```
50 SEARCH #C,0,1;K$,R,S
```

The function of search mode zero depends on the value of the directory expression, as shown in Table 2-2.

TABLE 2-2. SEARCH MODE ZERO

Directory Expression	Effect
> 0	Defines the directory specified by the directory number expression. It sets the key length (number of words) equal to the value of R and the number of keys per block equal to the integer value of [254/(key length + 1)] for the specified directory. The directories must be specified in sequential order starting with directory one.
= 0	Organizes all directories. This mode freezes the directory configuration to that specified by previous mode zero commands, assumes a number of data records as given in R, and sets up the internal linkage for all directories. All mode zero commands are rejected after this statement is executed.

2.7.5.2 Search Mode 1: Directory Information

Search mode one is used to read the attributes of an existing file after it has been opened. Line 100 is an example of search mode one:

```
100 SEARCH #C,1,0,K$,R,S
```

The effect of search mode one depends on the values of the directory number expression and status variable S, as shown in Table 2-3.

TABLE 2-3. SEARCH MODE ONE

Dir. Expr.	Status Var.	Effect
> 0	any value	Returns the key length (number of words) of the specified directory into R. If the specified directory does not exist, R remains unchanged and S is set to five.
= 0	0	Returns the record number of the first real data record into R. The returned value is always zero for polyfiles.
	1	Returns the number of free records in R. For indexed files, the number of free records is taken from the free record chain. For polyfiles, the number of free records is the sum of all available records across all data volumes in the polyfile.
	2	Indexed files: returns the record number of a free data record and removes that record from the free record chain. Polyfiles: Allocates and returns the record number of the first available free record and marks it "in use" in the bit map (if provided).
	3	Releases the record specified by the value of R. For indexed files, returns the record to the free record chain. For polyfiles, marks the record "free" in the bit map (if provided). Sets S to 19 if the record was already free and to 20 if the file does not have that record number.

2.7.5.3 Search Mode 2: Key Match

Search mode two searches the specified directory for a match with the key specified by K$. Line 200 is an example of search mode two:

```
200 SEARCH #C,2,D;K$,R,S
```

If the key is found, the system returns the entire key in K$, returns the associated data record number in R, and sets S to zero. If the key is not found, K$ and R remain unchanged, and S is set to one.

A match is found if the compared strings are equal to the end of K$, even if the found key is longer than the original key. If the strings match, the entire found key is returned in K$. For example, if the key "SMITH" were compared to "SMITH,ALAN", a match would be found and "SMITH,ALAN" would be returned in K$.

A match is not found if the key in the directory is shorter than the key specified in K$.

2.7.5.4 Search Mode 3: Next Key

Search mode three is used to process sequential entries, starting from a selected point. Line 300 is an example of search mode three:

```
300 SEARCH #C,3,D;K$,R,S
```

Mode three searches the specified directory for the first key whose value logically exceeds the value specified by K$. If the key is found, the system returns the entire found key in K$, returns the associated data record number in R, and sets S to zero. If the key is not found, K$ and R remain unchanged, and S is set to two.

For example, suppose K$ is set to "SMITH", and the directory contains the entries SMITH,ALAN and SMITH,BETH. The first search would return SMITH,ALAN, because it is the first key which logically exceeds SMITH. A second search, with SMITH,ALAN in K$, would return SMITH,BETH.

If the directory being searched may contain an entry which is equal to the entire value of K$, mode two should be used for the first search.

2.7.5.5 Search Mode 4: Insert a Key

Search mode four is used to insert a key. Line 400 is an example of search mode four:

```
400 SEARCH #C,4,D;K$,R,S
```

Mode four searches the specified directory for a match with the key specified by K$. If the key is not found and there is space available in the directory, the system inserts the key into the specified directory, references the key to the data record number in R, and sets S to zero. If the key is not found but cannot be inserted, S is set to two. If the key is found, the system returns the associated data record number in R and sets S to one.

Before inserting a new key, the data record must have been allocated using search mode one and the data record number for the new key must be specified in R. The data should be written into the record before the key is inserted into the directory.

For indexed files, if a key cannot be inserted because of inadequate space in a directory, search mode seven may be used to reorganize the directory. Section 2.7.5.7 describes search mode seven. Search mode four may then be repeated.

2.7.5.6 Search Mode 5: Delete a Key

Search mode five is used to delete a key from a directory. Line 500 is an example of search mode five:

```
500 SEARCH #C,5,D;K$,R,S
```

Mode five searches the specified directory for a match with the key specified by K$. If the key is found, the system deletes the key from the directory, returns the data record number in R, and sets S to zero. If the key is not found, R remains unchanged and S is set to one.

Search mode one must be used after deleting the key to return the freed data record to the free record chain.

2.7.5.7 Search Mode 7: Reorganize Directory

Search mode seven reorganizes the specified directory by optimizing it for efficient packing. Search mode seven is usually used when a key insertion (mode four) has failed. Line 700 is an example of search mode seven:

```
700 SEARCH #C,7,D;K$,R,S
```

Mode seven usually frees sufficient room for the key to be inserted. However, if a directory is nearly full, mode seven may not free sufficient space; provisions for this should be included in the program.

Search mode seven is not used on polyfiles because the keys in polyfile directories are redistributed automatically.

2.7.5.8 Inserting a Record into a Keyed File

To insert a record into a keyed file, write the data into the file before inserting the key in the directory. The following program shows how to insert a record into a keyed file:

```
100 DIM B$[10], K$[10]
150 INPUT "ENTER KEY:  "B$
.
.
190 REM Insertion Routine
200 K$=B$
210 LET S=2
220 REM Get unused data record from free list
230 SEARCH #1,1,0; K$,R,S
240 IF S<>0 GOTO 460
250 WRITE #1,R; A,B,C
260 R1=R
270 REM Put key for insertion into directory #1
280 SEARCH #1,4,1; K$,R,S
290 IF S<>0 GOTO 500
300 GOTO 150
.
.
460 PRINT "ABNORMAL MODE 1 RETURN; S= "S
470 STOP
.
.
500 REM Put record back on free chain
510 LET S1=3
520 SEARCH #1,1,0; K$,R1,S1
530 IF S1<>0 GOTO 580
540 IF S=1 PRINT "DUPLICATE KEY"
550 IF S=2 PRINT "DIRECTORY FULL OR DAMAGED"
560 IF S=5 PRINT "UNKNOWN DIRECTORY DAMAGE"
570 GOTO 1000
580 PRINT "ABNORMAL MODE 1 RETURN; S1= "S1
1000 STOP
```

2.7.5.9 Deleting a Record from a Keyed File

To delete a record from a keyed file, delete the key from the directory before deleting the data records from the file. The following program shows how to delete a record from a keyed file.

```
100 DIM K$[20], B$[20]
110 INPUT "ENTER KEY:  "B$
120 K$=B$
.
.
500 REM Delete key and return record number to free list
510 SEARCH #1,5,1; K$,R,S
520 IF S<>0 GOTO 700
530 LET S=3
540 SEARCH #1,1,0;K$,R,S
550 IF S<>0 GOTO 800
560 PRINT "KEY DELETED; RECORD RETURNED TO FREE CHAIN"
570 GOTO 110
.
.
700 PRINT "KEY NOT FOUND; S= "S
710 GOTO 110
.
.
800 PRINT "ABNORMAL MODE 1 RETURN; S= "S
810 STOP
```

2.7.5.10 Finding and Updating A Record in a Keyed File

The following program shows how to find a given key in directory one, check for an exact or partial match, and update the associated data record.

```
100 DIM A$[30], B$[30], K$[30]
110 OPEN #0;"FMTSAMPLE"
120 INPUT "ENTER NAME: "B$
130 K$=B$
.
.
500 REM Find the key and associated record number.
510 SEARCH #0,2,1; K$,R,S
520 IF S=1 GOTO 1000
530 IF S<>0 GOTO 1500
540 REM Branch if exact or partial match.
550 IF K$<>B$ GOTO 900
560 READ #0,R; A$,N
570 REM Update variables, then write the record.
.
.
600 WRITE #0,R;A$,J
.
.
700 GOTO 120
.
.
900 PRINT "PARTIAL MATCH; FOUND KEY IS: "K$
910 PRINT "KEY BEING SEARCHED FOR WAS: "B$
.
.
1000 REM Exception Routine--no match of any kind.
.
.
1500 REM Error Exit.
1510 PRINT "ABNORMAL RETURN; S= "S
1520 STOP
```

2.7.6 ACCESSING THE DATA PORTION OF KEYED FILES

The use of the READ, WRITE, PRINT, MAT READ and MAT PRINT statements on keyed files is the same as their use on contiguous files. Section 2.5.2 describes how contiguous files are accessed.

2.7.7 CREATING KEYED FILES USING BUILDXF AND BUILDPF

BUILDXF may be used to build an indexed file; BUILDPF may be used to build a polyfile. BUILDXF and BUILDPF are both described in the IRIS R8 User Manual.

To create an indexed file using BUILDXF, enter the following command at the IRIS system prompt:

```
#BUILDXF
```

BUILDXF then prompts for information regarding the data records and directories. The prompts and appropriate user responses are described in the IRIS R8 User Manual.

To create a polyfile using BUILDPF, enter the following command at the IRIS system prompt:

```
#BUILDPF
```

BUILDPF then prompts:

```
BUILDPF - Build polyfiles Utility

POLYFILENAME [must have "LU/" (not 0)]:
```

The number of the logical unit must precede the name of the polyfile; the LU may not be LU zero. The name must end with an @.

BUILDPF then attempts to open the polyfile. If the polyfile is found, BUILDPF enters the polyfile extension mode (refer to the IRIS R8 User Manual). If the file is not found, BUILDPF prompts:

```
POLYFILE NOT FOUND

DO YOU WISH TO CREATE A NEW ONE? (Y/N)
```

If the answer is No, the system returns to the IRIS system prompt. If the answer is Yes, BUILDPF requests a record size:

```
RECORD SIZE (in words for the entire polyfile):
```

After the record size has been entered, BUILDPF prints the following:

```
VOLUME: 0
```

This reminds the user that the following parameters are for volume zero, the master volume. BUILDPF then requests a volume type:

```
VOLUME TYPES:  "B" Base Directory
               "E" Extension Directory
               "D" Data Volume

VOLUME TYPE:
```

Enter the appropriate letter. BUILDPF then requests information pertinent to the type of volume specified, as described in the IRIS R8 User Manual.

2.7.8 CREATING AN INDEXED FILE FROM WITHIN A BASIC PROGRAM

An indexed file may be built within an application program using the following five steps:

1. Create the filename string and BUILD a contiguous file of the proper size. Section 2.5.4 describes how to BUILD a contiguous file. Information on determining directory size is provided below.

2. Define each directory with a SEARCH #C,M,D;, where M=0.

3. Organize the directory structure with a single SEARCH #C,M,D; statement, where M=0 and D=0.

4. Determine the numbers of the first and last real data records with a SEARCH #C,M,D;K$,R,S statement, where M=1, D=0, and S=0, and a CHF(0) statement, respectively.

5. Link all real data records on the free record chain with SEARCH #C,M,D;K$,R,S statements, where M=1, D=0 and S=3.

The overall size of each directory may be calculated using the following formulas:

```
N = INT [254/(K+1)]

F = FNR [R*2/(N+1)] >= 2

C = FNR [F/(N-1)] >= 2

T = F+C+1
```

where

```
  R - number of indexed records
  K - key length (words)
  N - number of keys per directory block
  F - number of blocks in the fine level
  C - number of blocks in the coarse level
  T - total number of blocks used by the directory
FNR - a function defined as FNR(x) = INT(x) + SGN[FRA(x)]
```

Function FNR is used to round up any fractional part of a block in the fine or coarse levels.

Multiply the total number of blocks for all directories by 256, and divide the product by the data record length (words) to compute the equivalent number of records (which must not exceed 65534). Apply FNR to the result, and add this equivalent number of records for all directories to the number of desired data records when building the file. If this total number of records exceeds 65534, then the data and directories must be in separate files.

The BUILDXF processor, listed in Appendix D, is a good example of how to build an indexed file from within a BASIC program.

2.7.9 CREATING A POLYFILE FROM WITHIN A BASIC PROGRAM

Three steps are involved in creating each polyfile volume. Each of these steps is described in detail in the following subsections.

1. Build a contiguous file.

2. Transform the contiguous file into a polyfile volume via the polyfile call (91). The parameter "v" in CALL 91 assigns the volume numbers.

NOTE

Volume 0, the master volume, must be built first.

3. Structure the polyfile volume via search mode zero.

2.7.9.1 Step 1: Build a Contiguous File

Build the contiguous file with the same name as the polyfile, but omit the "@". Use the FORMAT processor (described in Section 2.5.3) or the BUILD statement (described in Section 2.5.4).

2.7.9.2 Step 2: Convert to Polyfile Volume

The contiguous file is then converted to a polyfile volume by CALL 91, which must be executed while the file is still in the BUILD mode. After the call is executed, channel C0 must be closed to make the volume permanent on the disc. When building the master volume (0), channel C1 should be closed. Use the following statements:

```
IF ERR 0 STOP
CALL C,C0,C1,V,S,P
```

where:

C - CALL number (91)

C0 - Channel number on which the file is open

C1 - Channel number on which the master volume is open

V - Volume number

S - Status after the CALL is completed

P - File parameter array (see Table 2-4)

NOTE

The variable S and the array P must be declared in a DIM statement; otherwise, the "IF ERR" branch will take effect and S will return error 17, 18, or 19 (see Table 2-5). Use the following format:

```
DIM S,P[n]
```

where n is a value of 10 or greater. This permits validation of the P array dimensions.

If CALL 91 detects any errors, error #38 (error detected by a called subroutine) is returned. CALL 91 checks to see that the filename matches and that the volume is three or more blocks in size.

If C0 is less than 0, then only file parameters are returned.

If C0 is greater than or equal to 0, and if:

V>0 - this volume is linked to volume 0.

V=0 - a master volume is created.

V<0 - CALL C assigns the next available volume number and returns the number of that volume in V.

If the volume is a data volume, then the record length of the volume must match that of the master volume.

When C0 is non-negative, the value of S indicates the following:

S=0 - volume is to be a base directory or directory extension

S<>0 - volume is to be a data volume

If S is returned from the call with value 0, then file parameters are to be found in the array P. The parameters, ordered by index, are shown in Table 2-4.

TABLE 2-4. FILE PARAMETERS

Contents of P	Description
0	VLU (Volume/Logical Unit)
1	BNR (Base Number of Records)
2	LU flag (0=installed; <>0=not installed)
3	ACNT
4	TYPE
5	NBLK
6	LRCD
7	NRCD
8	LDAT
9	LDAT+1
10	Keylength of 1st directory in volume (FMAP+4)
11	Keylength of 2nd directory in volume (FMAP+5)
:	
72	Keylength of 63rd directory in volume (FMAP+102)
76	Logical Unit number
77	DHDR

If the value of S is nonzero, an error is indicated. The possible status values for S are shown in Table 2-5.

TABLE 2-5. CALL 91 STATUS VALUES

Contents of S	Description
0	No error
1	Invalid channel number
2	File not being built
3	Illegal volume number
4	File C1 is not a polyfile
5	File name invalid
6	Invalid variable type
7	Invalid number
8	Volume already exists on the desired LU, possibly as part of another polyfile of the same name
9	File C0 not found (deleted?)
10	Not enough nodes to link into extended DFT
11	Volume already exists for this polyfile
12	Volume V not found
13	Account numbers do not match
14	Volume in extended DFT but not on disc
15	No available volume number for this polyfile
16	Volume V is not defined
17	P is not allocated as the next variable after S
18	P is not an array
19	P is not dimensioned P[10] or greater
20	File C0 is not contiguous
21	File C1 is open elsewhere
22	File C0 is already a polyfile
23	HSLAs do not match for assign operations
24	VLU or BNR do not match for assign operations
25	Cannot move volume 0
26	Illegal move operation
27	File C0 is not write protected
28	Illegal write enable operation

2.7.9.3 Step 3: Structure Polyfile

Search mode zero is used to structure a polyfile. For polyfiles, search mode zero requires that a volume number be given and that key sizes be given in bytes. The format is:

```
SEARCH #C,m,d;K$,R,S
```

where:

m=0,d+128 - Define volume number d to be a data volume; 0<=d<=63; R=number of records.

- If R=0, the number of records is computed and returned in R.
- If S=0, a bit map is not built.
- If S<>0, a bit map is built with R free records in it.

m=0,d+64 - Define volume number d to be an extension of the base directory volume in S; 0<=d<=63.

m=0,d - Define directory d: volume number in S, key length in R (in bytes); 1<=d<=63.

m=0,d=0 - Organize all directories for the volume number given in S.

2.7.10 CONVERTING AN APPLICATION FROM USING INDEXED FILES TO USING POLYFILES

An indexed file may be converted to a polyfile by using the steps outlined below. Programs which use indexed files may also be changed to handle polyfiles.

To convert an indexed file to a polyfile, build a polyfile with key and data record sizes matching the current indexed file. Then construct and execute a program which will:

- read each key and its associated data from the indexed file
- get a free record from the polyfile
- write the data record into the polyfile
- insert the key into the polyfile
- allow at least a 3% (precision) variable for the polyfile record number
- use separate variables for indexed file access and polyfile access to prevent R and S being greater than 65535 for indexed file access

To convert a program which uses indexed files to one which uses polyfiles, do the following:

1. Modify OPEN statements to reference the polyfile.
2. Remove all search mode seven statements from the program. Because polyfiles redistribute keys automatically, search mode seven has no effect on polyfiles. A program containing search mode seven may safely be used; however, mode seven is unnecessary and should be eliminated.
3. Check that any variable intended for polyfile record numbers is large enough to hold 3% (precision) numbers.

2.7.11 SPECIAL POLYFILE CALL (91) MODES

Three special modes of CALL 91 are available. They allow the following:

- an individual volume of an existing polyfile to be moved to another logical unit
- the polyfile master volume and nonzero volume headers to be updated after a logical unit number is changed during an INSTALL procedure
- writing to an individual volume of a polyfile that is open in read-only mode

These modes are described in detail in the IRIS R8 Polyfile Document.

Section 3
BUSINESS BASIC STATEMENTS

This section gives a detailed discussion of each Business BASIC statement. The discussions include information such as whether the statement may be entered at the keyboard for immediate execution, whether it may be executed within a program, its syntax, and a description of its effect. Examples illustrating correct use of the statement are shown on the facing page. The format used in these discussions is illustrated in Example 3-1.

The statements are presented in alphabetical order.

STATEMENT

KEYBOARD: YES indicates the statement may be entered at the keyboard without a line number and executed immediately.

PROGRAM: YES indicates the statement may be entered as part of a program and executed when the program is run.

SYNTAX: shows all legal forms of the statement using the following conventions:

variable = simple or subscripted numeric or string variable

xx list = each element (such as xx) of a list except the last must be followed by a comma

expression = a group of characters that may be evaluated to a simple numeric value; sometimes abbreviated expr

filename = an IRIS filename

filename string = a quoted literal or string variable containing a filename

A
B indicates that either A or B may be included, but not both

{} braces indicate that the parameters they enclose are optional

* an asterisk following information enclosed in braces indicates that the enclosed information may be repeated up to the length of the BASIC I/O and edit buffers

EFFECT: description of what the statement does when it is executed

NOTES: additional comments on using the statement

Example 3-1. Format of Statement Descriptions

BUILD#

KEYBOARD: NO
PROGRAM: YES

SYNTAX: BUILD #channel,FORMAT/CONTIG/+TEXT{,#channel,FORMAT/CONTIG/+TEXT}*

where FORMAT defines a formatted file as follows:

"{<prot>}{$ddd.cc}{LU/}filename string or string var{!}"

and CONTIG defines a contiguous file as follows:

"{<prot>}{$ddd.cc}[records:words]{LU/}filename string or string var{!}"

and +TEXT defines a text file as follows:

+"{<prot>}{$ddd.cc}{LU/}filename string or string var{!}"

EFFECT: Used to build a data file. Creates a new file or replaces an old file, identified by the filename string, on a specified channel. Several files may be created by one statement, and filenames not preceded by a channel expression are created on successive channels.

NOTES:

1. Refer to Section 2 for specific information on using BUILD to create each file type.

2. If the file's protection or cost are not specified, then the protection is 77 (maximum protection), and the cost is zero.

   An existing file of the same type on the user's own account may be replaced by following the filename with an exclamation mark (!). If the cost or protection is changed, both must be re-entered or the default is assumed.

3. A new formatted file is automatically formatted by data written to record zero (record zero must be specified explicitly).

4. Unless the channel on which a file is built is closed using a CLOSE statement, any new file created using a BUILD statement is deleted when the program is exited or by an abortive error because the system automatically clears all open channels. When CHAINing to another BASIC program the channels are not cleared, so new files created using BUILD are not deleted.

EXAMPLES:

```
120 BUILD #2,"NEWFILE",#H3,N$
```

This example builds a formatted data file with the name NEWFILE on channel two, and a data file with the name given in N$ on the channel number given by variable H3.

```
220 BUILD #3,"<33> $14.50 [200:250] LEDGER"
```

This example builds a contiguous data file called LEDGER, with 200 records of 250 words each, and opens it on channel three. The file's cost is $14.50 and its protection is 33.

```
320 BUILD #1,"<33> $10.00 [10:256] 1/EXAMPLE!"
```

This example builds a contiguous file called EXAMPLE on logical unit one, replacing an existing file of the same name. It is built on channel one with protection 33, costs $10.00 each time it is accessed and has 10 records of 256 words each.

A new text file may be built by a statement of the form

BUILD #c,+filename string. . .

where c is a channel number expression, and the filename is the name under which the text file is to be built. For example:

```
100 BUILD #3,+"TEXT6.4"
150 BUILD #5,"FFILE",+D6$
```

Line 100 builds a text file named TEXT6.4 on channel three. Line 150 builds a formatted data file on channel five and a text file, whose filename is given in D6$, on channel six. The "+" symbol is part of the statement syntax and is not included as part of the filename string.

CALL

KEYBOARD: YES
PROGRAM: YES

SYNTAX: CALL mnemonic or routine expr,variable[,variable]*

EFFECT: The CALL statement is used to execute a Business BASIC or user-defined subroutine.

NOTES:

1. The integer value of the routine expression or the routine mnemonic selects a specific subroutine. (Subroutine numbers greater than 99 are reserved for use by POINT 4.) CALLs $FINDF, $LOGIC, $RDFHD, $STRING, $TIME, $TRXCO and the polyfile CALL are discussed on the following pages.

2. The variables are used to pass argument values to and from the subroutine. Up to twelve such parameters may be used. Simple variables and string variables may be used. Expressions may not be used as parameters; subscripts may not be used in strings.

3. For examples of some uses of CALLed subroutines, see program 1 in Appendix A for use of CALL 99 (system time) and programs 6 and 7 in Appendix A for use of CALL 98 (transmit system command).

4. When assigning a mnemonic to a subroutine, POINT 4 recommends that the mnemonic begin with a prefix of your choice to differentiate it from POINT 4 mnemonics.

5. CALLS 79, 93, 94 and 95 are reserved. CALL 91 is the polyfile CALL. CALLs 82, 88, 96, 97, 98 and 99 are used for subroutines which have mnemonics.

EXAMPLES:

```
625 CALL 14,F2,P2,D$
980 CALL $LOGIC,L,P1,P2,R
```

CALL 91

KEYBOARD: YES
PROGRAM: YES

SYNTAX: CALL 91, file channel,master vol channel,v,S,P

where
v - volume number
S - status after the CALL is completed
P - file parameter array (see note 8)

EFFECT: Converts a contiguous file to a polyfile volume.

NOTES:

1. This CALL must be executed while the contiguous file is being built. For a description of the steps involved in creating a polyfile, refer to Section 2.7.

2. For further information on CALL 91, refer to Section 2.7.

3. The variable S and the array P must be declared using the following format in a DIM statement:

   S, P[n]

   where n is a value of 10 or greater. If S and P are not dimensioned as shown, the required IF ERR branch will take effect and S will return error 17, 18 or 19. This permits validation of the dimensions of array P.

4. CALL 91 checks to see that the filename matches and that the volume is three or more blocks in size.

5. If the file channel (the channel on which the file is open) is set less than zero, only file parameters are returned. If it is set greater than or equal to zero, the effect depends on the value of v as described below:

   v > 0 - links this volume to volume zero
   v = 0 - creates a master volume
   v < 0 - assigns the next variable volume number and returns the number of that volume in v

EXAMPLES:

```
IF ERR 0 STOP
CALL 91, C0, C1, V, S, P

IF ERR 1 STOP
CALL X, C0, C1, V, S, P
```

NOTES: (Continued)

6. If the volume is a data volume, then the record length of the volume must match that of the master volume.

7. If the file channel is non-negative, S should be set to zero to indicate that the volume is to be a base directory or directory extension volume or to a non-zero value to indicate that the volume is to be a data volume.

8. If S is returned from the CALL with the value zero, then array P contains file parameters. The parameters are shown in Table 2-4. If S is returned with a non-zero value, then an error is indicated. The possible status values for S are listed in Table 2-5.

9. After CALL 91 is executed, the file channel must be closed to make the volume permanent on the disc. When the master volume (0) is created, the master volume channel should be closed, and the new master volume accessed through the file channel.

10. CALL 91 offers three special modes which may be used to move an existing polyfile to a new logical unit, update headers after changing the number of a logical unit, and write to an individual polyfile volume in read-only mode. These special modes are described in Section 2.7.11.

CALL $FINDF (CALL 96)

KEYBOARD: YES
PROGRAM: YES

SYNTAX: CALL $FINDF, filename string variable,header variable

EFFECT: Calls a subroutine used to find a file.

NOTES:

1. The filename, in the form required in an OPEN statement, is passed to the routine in the filename string variable.

2. The routine returns the header block address of the file in the header variable. If the file is not found, zero is returned.

3. The file is not opened and the protection or file type is not checked.

4. This CALL is usually used to determine whether a given file exists.

5. $FINDF may also be CALLed by its subroutine number, 96.

CALL 95

CALL 95, U, R, D, T, Z OR V$

WHERE;

U - LOGICAL UNIT #

R - RDA

D - DISPLACEMENT INTO BLOCK

T = TYPE

( )/(-) n

where NO SGN = READ

" (-) = WRITE

and;

n = 1 = 1%

2 = 2%

3 = 3%

4 = 4%

5 = N/A

6 = BINARY

7 OR NOR = STRING

Z = READ - RECIEVES VALUE READ

WRITE - VALUE TO BE WRITTEN

V$ = READ - STRING TO BE READ

WRITE - STRING TO WRITE

CALL 89/81

CALL C, C2, A$, B$, E

C= 89

C2= 2 /1

A$ = ~~STRING~~ ITEM TO SEARCH FOR

B$ = STRING TO SEARCH

E = POSITION ITEM FOUND IN STRING

CALL 81

Mode 1 - FIND MATCH between 1ST char of A$ IN B$ - SET E TO LOCATION

Mode 2 - FIND MATCH between ENTIRE A$ IN THE B$. SET E TO LOCATION found. START AT 1ST char of B$ IF E = 0

EXAMPLES:

```
75 CALL $FINDF,A$,X
```

CALL $LOGIC (CALL 88)

KEYBOARD: YES
PROGRAM: YES

SYNTAX: CALL $LOGIC,operator,variable1,variable2,result variable
string var1,string var2,result string var

Operator - a numeric variable containing the number of the appropriate logical operator

where
1 - specifies AND
2 - specifies OR
3 - specifies XOR
4 - specifies NOT

EFFECT: Performs the specified logic operation (AND, OR, XOR, and NOT) on variable1 and variable2 and returns the result in the result variable or on string var1 and sting var2 and returns the result in the result string var.

NOTES:

1. All variables in the calling statement, except the operator variable, must be of the same type (either string or numeric). Numeric integers must be in the range 0 to 65535.

2. Although NOT requires only one operand, the second operand must be specified and is used as a dummy variable to satisfy syntax requirements.

3. Numeric values are converted to unsigned 16-bit integers before the logical operation is performed.

4. For strings, the operation is performed byte by byte until the dimensioned length of the shortest string is reached.

5. This subroutine may be used to copy strings which include multiple binary zeros by ANDing the string with itself, and to fill a string with binary zeros by XORing it with itself. This is possible because the subroutine does not recognize binary zero as a string terminator.

6. $LOGIC may also be CALLed by its subroutine number, 88.

EXAMPLES:

```
100 CALL $LOGIC, A, M, N, R

200 CALL $LOGIC, S, A$, B$, C$
```

```
10 LET P1=3 \ P2=22
20 LET A=1 \  CALL $LOGIC,A,P1,P2,R
30 PRINT "Logical AND of P1 & P2="R
40 LET A=2 \  CALL $LOGIC,A,P1,P2,R
50 PRINT "Logical OR of P1 & P2="R
60 LET A=3 \  CALL $LOGIC,A,P1,P2,R
70 PRINT "Logical XOR of P1 & P2="R
80 LET A=4 \  CALL $LOGIC,A,P1,P2,R
90 PRINT "Logical NOT of P1="R
100 DIM A$[15],B$[50]
110 LET A$="HELLO THERE"
120 LET A=1 \  CALL $LOGIC,A,A$,A$,B$
130 PRINT "A$ ANDed with itself is:"B$
140 LET A=2 \  CALL $LOGIC,A,A$,A$,B$
150 PRINT "A$ ORed with itself is:"B$
160 LET A=3 \  CALL $LOGIC,A,A$,A$,B$
170 PRINT "A$ XORed with itself is:"B$
180 LET A=4 \  CALL $LOGIC,A,A$,A$,B$
190 PRINT "The logical NOT of A$ is:"B$
RUN
Logical AND of P1 & P2 = 2
Logical OR of P1 & P2 = 23
Logical XOR of P1 & P2 = 21
Logical NOT of P1= 65532
A$ ANDed with itself is:HELLO THERE
A$ ORed with itself is:HELLO THERE
A$ XORed with itself is:
The logical NOT of A$ is:
READY
```

NOTES: (Continued)

7. The logic operators cause the binary quantities x and y to attain the values below:

x	y	x AND y	x OR y	x XOR y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

x	NOT x
0	1
1	0

For further information on logic operators, consult a text on boolean algebra.

CALL $RDFHD (CALL 97)

KEYBOARD: YES
PROGRAM: YES

SYNTAX: CALL $RDFHD, lu,r,n$,a,t,s,q,c,i,d,l,h

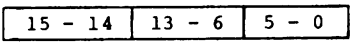
EFFECT: Calls a subroutine which reads the file header indicated by the record number r. The logical unit and record number must be specified. The subroutine returns the account number and the file's name, type, size, status, cost, total income, creation date, last access date and header disc address in the other variables.

NOTES:

1. lu,r,a,t,s,q,c,i,d,l, and h are any numeric variable names, and n$ is any string variable dimensioned as 15 characters or longer. Lu specifies the logical unit, r specifies a starting record number in the INDEX file, and the other variables receive information about the file as follows:

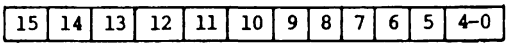
   n$ Filename

   a Account number word. The bits and their meanings are shown below:



Bits	Meaning
15-14	Privilege level
13-6	Account group number
5-0	Account user number

t File type word. The bits and their meanings are shown below:



Bit	Meaning
15	Not used
14	Read protected against users of lower privilege
13	Write protected against users of lower privilege
12	Copy protected against users of lower privilege
11	Read protected against users of the same privilege
10	Write protected against users of the same privilege
9	Copy protected against users of the same privilege
8	Runnable processor
7	Load active file when selected
6	Initiate input before first swap-in
5	May be locked in memory
4-0	File type

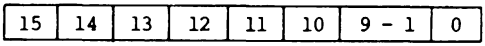
The value of t may be substituted in the following equations to extract the information shown below:

```
T1 = File Type = t-32*INT(t/32)
R  = R,L,I control digit = INT[(t -INT(t/512)*512-T1)/64]
P  = Protection digits = INT(t/512)
P1 = Protection digit #1 = INT(P/8)
P2 = Protection digit #2 = INT(P-P1*8)
```

The meaning and use of this information is described in the IRIS R8 User Manual.

s File size. Number of disc blocks used by the file.

q File status word. The bits and their meanings are shown below:



Bit	Meaning (if set)
15	File is being built; has not been closed yet
14	A file is being built to replace this one
13	File is to be deleted
12	File is mapped (ie, it is a formatted data file)
11	File has been opened with an open-lock
10	File can not be deleted
9-1	Not used
0	File is extended

If bits 14 or 13 are set, the file will be overwritten/ deleted when it is closed.

CALL $RDFHD (Continued)

c File cost to the dime.

i Total income to the file, to the dime. This is increased by the value of c each time a user on a different account accesses the file. Note: if the value of variable i was zero before the call, then the file's income will be cleared to zero by use of the call; otherwise, the file's income is not changed by the call.

d File creation date (hours after the base date). The age in hours may be calculated as Age = SPC(2)-D. SPC (18) returns the system base year.

l Last access date (hours after the base date). The hours since last access may be calculated as HSLA = SPC(2)-l. SPC (18) returns the system base year.

h Disc address of file's header.

2. All numeric values are returned in decimal. (LIBR lists the t, q, and h values in octal. The t, q, and h values returned by $RDFHD must be converted to octal for comparison with a LIBR listing.)

3. If the INDEX record specified by R does not contain an entry, then the next entry is automatically tried until an entry is found or the end of the INDEX is reached. If the end of the INDEX is reached, then R is set to -1, and all other variables remain unchanged. When a valid entry is found, R is set to the record number of the next INDEX entry, and all variables are loaded from information in the file's header as indicated above.

4. $RDFHD may also be CALLed by its subroutine number, 97.

EXAMPLES:

```
110 CALL $RDFHD,X,J,M$,P,K,O,L,S,N1,N2,H9,T2
320 CALL $RDFHD,S,B,B$,P,L,T1,T2,T3,T4,T5,T6,T7
```

CALL $STRING (CALL 82)

KEYBOARD: YES
PROGRAM: YES

SYNTAX: CALL $STRING, modeA, string expression

where modeA - a variable set equal to 1, 2 or 5

CALL $STRING, modeB, string expression, value

where modeB - a variable set equal to 3 or 6

CALL $STRING, modeC, value, string expression

where modeC - a variable set equal to 4 or 7

EFFECT: The subroutine $STRING provides the following functions: convert string to upper case, convert string to lower case, convert one or two characters to a numeric value, convert an 8-bit or 16-bit value to a character or characters, and read the I/O Buffer.

NOTES:

1. The variables shown by modeA, modeB and modeC in the syntax statements above represent the mode. The mode determines which function will be employed, as described below.

Mode	Function
1	convert string to upper case
2	convert string to lower case
3	convert a single character to an ASCII value
4	convert an ASCII value to a single character
5	read the Input/Output Buffer
6	convert two characters to a 16-bit number
7	convert a given value to two characters

2. When modes 1 and 2 are used, the subroutine converts alphabetical letters only to the appropriate case. Any characters already in the case being converted to or any nonalphabetic characters remain unchanged.

3. When mode 3 is used, the subroutine converts the first character of the string to an ASCII value in the range zero to 255, depending on the binary value of the character.

4. When mode 4 is used, if the specified value is greater then 255, the value modulus 256 is used in the conversion. The generated character overlays the first character of the string, if any, and the second character is overlayed with a null.

(Discussion of this statement is continued.)

EXAMPLES:

Modes 1 and 2:

```
10 DIM A$[100]
20 LET A$="ABCdef GhIj K!L@m3n$ ;:+= OPQRST uvwxyz"
30 INPUT "MODE: "M
40 CALL $STRING, M, A$  \  PRINT
50 PRINT A$
RUN
MODE: 1
ABCDEF  GHIJ  K!L@M3N$ ;:+=  OPQRST  UVWXYZ

READY
RUN
MODE: 2
abcdef  ghij  k!l@m3n$  ;:+=  opqrst  uvwxyz
```

Modes 3 and 4:

```
10 DIM A$[10],B$[10]
20 LET A$ = "Z"
30 LET M=3  !Mode 3
40 CALL $STRING, M, A$, A
50 PRINT A
60 LET M=4 !Mode 4
70 CALL $STRING, M, A, B$
80 PRINT B$
RUN
 218
Z
```

NOTES: (Continued)

5. When mode 5 is used, the returned string consists of the contents of the I/O Buffer up to the first <RETURN> found in the buffer. Because of this, when mode 5 is used, the CALL must precede any input or output within the program; otherwise, the contents of the I/O Buffer may be lost. The buffer contents are also destroyed if the program generates a BASIC error message.

6. When mode 6 is used, the subroutine converts the first two characters of the string to a value determined by the following equation:

```
(A * 256) + B = value
```

where A = ASCII value of the first character
B = ASCII value of the second character
value = the calculated result

The subroutine returns a 16-bit value in the range zero to 65535 which is calculated using the above equation and which depends on the characters and parity setting.

7. When mode 7 is used, the pair of generated characters overlays the first two characters of the string.

8. Only values in the range zero to 65535 may be converted to characters using this subroutine.

9. A program which uses mode 5 of $STRING may allow the user to enter

```
#DISPLAY REPORT
```

which uses the program "DISPLAY" on the file named "REPORT". See the example on the facing page.

10. $STRING may also be CALLed using its subroutine number, 82.

EXAMPLES:

Mode 5:

```
10 DIM C$[100]
20 LET M=5
30 CALL $STRING,M,C$
40 PRINT "THE PARAMETERS ARE: ";C$
50 CHAIN " "
EXIT
#SAVE ECHO
SAVED!!    CHECK CODE = CD1A
#ECHO THIS TEXT
THE PARAMETERS ARE:  THIS TEXT
```

Modes 6 and 7:

```
10 DIM A$[10],B$[10]
20 LET A$= "XX"
30 LET M=6  !Mode 6
40 CALL $STRING,M,A$,A
50 PRINT A
60 LET M=7  !Mode 7
70 CALL $STRING,M,A,B$
80 PRINT B$
RUN
 55512
XX
```

CALL $TIME (CALL 99)

KEYBOARD: NO
PROGRAM: YES

SYNTAX: CALL $TIME, string variable

EFFECT: Calls a subroutine which is used to read or set system time.

NOTES:

1. If CALL $TIME is issued with the string variable empty, the system time is returned in the string variable. The string variable must be dimensioned at least 23 bytes.

2. If CALL $TIME is issued from account 0,1 only with contents in the string variable, the system uses the string variable to set the time.

3. $TIME may also be CALLed by its subroutine number, 99.

EXAMPLE A:

From Any Account

CALL $TIME may be issued from any account to read the current system time.

```
10 DIM A$ (25)
20 A$ = ""
30 CALL $TIME, A$
40 PRINT A$
RUN
AUG 30, 1983  16:22:36
```

A$ was set to an empty string in line 20. A$ must be dimensioned at least 23 bytes.

EXAMPLE B:

From Account 0,1 Only

The system time may be adjusted by the system manager from account 0,1 by use of a CALL $TIME in a BASIC program. For example:

```
10 DIM A$ (25)
20 INPUT A$
30 CALL $TIME, A$
```

This program will work only from the system manager account (account #0,1). The string entered for A$ must represent the current time. The time may be represented in either of the following forms:

```
NOV 15, 1975   16:22:36
```

or

```
1975,11,15,16,22
```

where the "seconds" portion (the last colon and last two digits) is optional. All leading zeroes are also optional.

This usage differs from Example 1 above only in that A$ is not empty when entering CALL $TIME. BASIC Error #38 will result if the program in Example B is run on any account other than 0,1.

CALL $TRXCO (CALL98)

KEYBOARD: YES
PROGRAM: YES

SYNTAX: CALL $TRXCO, port no. var,command string var{,status var}{,priority no.}

EFFECT: Calls a subroutine which transmits a system command to any other port. Business BASIC programs and other processors may be run on another port by use of the CALL $TRXCO statement.

NOTES:

1. The value of the port number variable is used to specify the number of an interactive port. A constant may not be given for the port number. The port may or may not have a keyboard or any I/O device of its own. If the port has a terminal, it is called a "slave port" while being used in this manner; if it has no terminal, it is called a "phantom port". Phantom ports can be activated only by system commands transmitted from another port using CALL $TRXCO. Any program to be run or command to be executed on a phantom port should not require any operator interaction, because an operator can not interact with the port.

2. The command string variable may be any string variable which contains a legal system command (any command accepted at the system prompt). When CALL $TRXCO is executed, it logs on the specified port if necessary, and executes the specified command as though it had been typed at a keyboard of the specified port. Often, the command is to run a BASIC program.

(Discussion of this statement is continued.)

NOTES: (Continued)

3. The status variable is an optional numeric variable which receives error status. If included, this variable must be set to zero in the program before the CALL $TRXCO statement is executed. If an error is detected by the system while executing CALL $TRXCO, this variable is set to a value as defined below:

Status	Meaning
0	no error; successful operation
1	port no. variable is not a numeric variable
2	specified port (that specified by the value of the port number variable) does not exist
3	specified port is not interactive
4	specified port is the user's own port
5	command string var is not a string variable
6	specified port is compute bound or is actively outputting information
7	specified port has input in progress
8	string is longer than the specified port's input buffer
9	user is privilege level zero and therefore cannot use CALL $TRXCO
10	user is privilege level one and specified port is not a phantom port
11	user is privilege level one and specified port is in use by another account
12	user must be privilege level two to issue an abort

If the status variable is not includeed, then Error 38 occurs if an error is detected.

4. The priority no. is used to set a priority for a phantom port. The priority may be set in the range one to seven; the priority must be set less than or equal to the user's own priority. Priority is often used to assign a low priority so the operation on the phantom port runs as a background task.

5. A privilege level 2 or greater user may abort any operation in progress on the selected port by transmitting a backslash code (octal 334) in the command string variable. This is equivalent to entering <CTRL-C> at the keyboard. Note that an application program with error branching in effect may not respond to <CTRL-C>, so some other mechanism must be used to abort the program. The suggested technique is to include a SIGNAL 2 statement in the program to receive an abort signal.

CALL $TRXCO (Continued)

NOTES: (Continued)

6. Only privilege level one and higher accounts may use CALL $TRXCO; privilege level zero accounts may not use it. Privilege level one users can transmit commands only to a phantom port which is not in use or which is in use by the same account. Privilege level two users can transmit commands to any interactive port regardless of its status.

7. $TRXCO may also be CALLed by its subroutine number, 98.

EXAMPLES:

```
110 CALL $TRXCO,S,N1$,T

200 CALL $TRXCO,P,A$,S
```

The following program may be used to initiate a procedure on a phantom port. This program assumes that the user knows the number of an available phantom port and is running on a manager account.

```
100 DIM A$[100],B$[1]
110 INPUT "\215\PHANTOM PORT #, COMMAND "P,A$
120 LET V=0
130 LET V1=0
140 REM LOG OFF PORT IF IN USE
150 LET B$= "\334\"
160 CALL 98,P,B$,V1
170 REM WAIT FOR PORT TO LOG OFF
180 SIGNAL 3,20
190 REM WAIT IF PORT IS IN USE
200 IF V1<>0 GOTO 160
210 REM NOW ISSUE SYSTEM COMMAND
220 CALL 98,P,A$,V
230 SIGNAL 3,20
240 REM LOG OFF AND RESTART IF PORT WAS IN USE
250 IF V<> 0 GOTO 120
260 CHAIN ""
```

If lines 140, 150 and 160 are deleted, the above program may be run on a non-manager account. (Lines 130, 170, 180 and 190 could also be deleted.)

CHAIN

KEYBOARD: YES
PROGRAM: YES

SYNTAX: CHAIN quoted literal or variable string

EFFECT: Terminates running of the program in which it is included and transmits the quoted literal or variable string for execution.

NOTES:

1. There are two forms of the CHAIN statement: a long CHAIN and a short CHAIN. Both forms terminate running of the program in which they are executed. A long CHAIN chains to SCOPE and the commands in the statement are processed as though they were entered at the IRIS system prompt. A short CHAIN chains from one BASIC program to another BASIC program. The syntax of both types of CHAINs is the same. A short CHAIN is performed if the parameter string begins with the name of a BASIC program.

2. When CHAINing to another BASIC program (using a short CHAIN), the user's data channels remain open. This allows one program segment to open a set of data files, and succeeding program segments to access the same files on the same channels without requiring the filenames. All variables are cleared.

3. Several system commands may be given in a long CHAIN statement, and they will be executed in sequence. The commands must be separated by RETURN codes (enter <CTRL-Z> or \215\). In a long CHAIN, the first command may not be to run a BASIC program. For example, line 900 in the examples saves the program in its current state (including the values of all variables), and then starts running it again at line 910; this is called checkpointing a program. Statement 950 causes COPY to list the file LISTFILE (presumably a text file on Logical Unit #3) on the line printer, and then start the program PART3. The command string is executed from the Intermediate Input Buffer (IIB), so the length of the string must not exceed the size of IIB.

4. CAUTION! Do NOT type ahead while a program that may CHAIN to a system command is running. Since the IIB is used for both functions, the characters from the two sources may become intermixed. Type ahead is allowed during programs that CHAIN only to other BASIC programs. The IIB is not affected when CHAINing from one BASIC program to another BASIC program.

EXAMPLES:

```
520 CHAIN "PART2"
710 CHAIN "0/BYE"
840 CHAIN M$
900 CHAIN "0/SAVE TEMP!\215\BASIC\215\910 RUN"
950 CHAIN "0/COPY $LPT=3/LISTFILE\215\PART3"
990 CHAIN ""
1000 CHAIN "\230\MYPROG\215\INPUT1\215\INPUT2"
```

Statement 520 terminates running the current program and initiates execution of the BASIC program named PART2. An error message is printed by SCOPE if PART2 does not exist and the system returns to SCOPE. Any data channels that were opened by the current program remain open and can be referenced by PART2 without re-opening them.

The other examples show additional uses for the CHAIN statement. The port may be automatically logged off after all calculations are finished (and the results have been stored in data files or printed) by giving a BYE command as in statement 710. In statement 840, the string variable M$ must contain a BASIC program filename or a system command.

CHAIN looks first on the user's assigned LU or on the specified LU for the first filename given; it then looks on LU/0. Because of this, a "0/" should precede any system command as shown in lines 710, 900, and 950. Each system command given in a CHAIN statement is printed on the user's terminal, preceded by a ! symbol, just before it is executed, as if the user had entered the command at the system prompt. Nothing is printed for a direct CHAIN to another BASIC program.

Statement 990 shows the use of an empty string in the CHAIN statement to exit to the system, similar in effect to using the EXIT statement. All channels are cleared if the program chains to BYE or to system control mode as in examples 990 and 710.

Statement 1000 shows how to CHAIN to a BASIC program (or through multiple BASIC programs) and pass inputs as though they were entered on the keyboard. The "\230\" causes entry into SCOPE, thereby closing all channels.

CLOSE#

KEYBOARD: NO
PROGRAM: YES

SYNTAX: CLOSE #c{,#c}*

EFFECT: Dissociates the specified channel or channels from the file or device which was opened on that channel.

NOTES:

1. An open channel must be closed before another file or device can be opened or built on the same channel.

2. If a file is being built, closing the channel makes the file accessible to other users for the first time.

3. A file being built is deleted if the program run is stopped before the file is closed.

4. Attempting to close a channel which is not open generates an error. If a CLOSE statemment includes multiple channels and one of the channels generates this error, the following channels will not be closed.

5. Expressions may be used to provide the channel number, which may range from zero to the maximum channel number. The maximum channel number depends on the configuration of the system, but is usually nine.

EXAMPLES:

```
160 CLOSE #1

345 CLOSE #A-1,#4,#R
```

Line 160 above closes the file or device on channel one. Line 345 closes channels A-1, four, and R, where A-1 and R are expressions whose values each identify a channel number.

DATA

KEYBOARD: NO
PROGRAM: YES

SYNTAX: DATA {precision%,}constant{,constant}*

EFFECT: The DATA statement is used to supply constant numeric data of a specified precision within a program.

NOTES:

1. If a DATA statement does not contain a precision setting, then its data is stored with two-word precision. "n%" (n = 1, 2, 3, or 4) may be used in a DATA statement, in which case the numbers in that DATA statement are stored with precision n.

   Only one % symbol may be used in each DATA statement, and the n% must immediately follow the word DATA. All numbers in a given DATA statement are of the same precision. The % symbol in a DATA statement does not affect the dimension of a numeric variable specified in the DIM statement. Data from a DATA statement is stored in the precision of the variable into which it is read. Truncation or overflow (Error 15) may occur.

2. The data is read in sequence from the first to the last DATA statement and from left to right within each DATA statement. The system initially sets a pointer to the first item of data. As the READ statements request each data item, the pointer is moved to the next data item. The RESTORE statement may be used to reset the pointer.

3. DATA statements are not executed and may be placed anywhere in the program. Items in a DATA statement must be separated by commas, but no comma should follow the last item of data.

4. Although DATA statements may be entered in keyboard mode, they are useless because they can not be referenced by a READ statement. DATA statements must be entered with line numbers in order to be referenced by a READ statement.

EXAMPLE:

```
10 FOR J = 1 TO 4
20  READ Y
30  PRINT "THE SQUARE ROOT OF "Y"IS"SQR(Y)
40 NEXT J
50 DATA 3%,2,3.7,94.61,.0024
60 READ C,Z
70 LET E=C+Z
80 PRINT E
90 DATA 12,-32.4,9999,4E-16
```

In line 50 above, 2,3.7,94.61, and .0024 are stored as three-word precision numbers. In line 70, variable E is a two-word variable because the 3% specification in the preceding DATA statement affects only those numbers in the DATA statement.

DEF

KEYBOARD: NO
PROGRAM: YES

SYNTAX: DEF FN letter (dummy variable)=numeric expression

where
dummy variable is a single letter

EFFECT: This statement defines functions for use throughout the program in which they are defined.

NOTES:

1. Up to 26 functions, FNA through FNZ, may be defined in each program. Defined functions may be nested by using other defined functions within the definition. Up to 5 levels of nesting are allowed.

2. The DEF statement must be executed for the definition to become effective. The definition may be changed at any time by executing another DEF statement for the same function.

3. A defined function is used primarily when the same expression appears in several places in a program. A function is defined equal to that expression, and then the function is used in the program in place of the expression.

4. The variable name in the function definition is called a dummy variable because its name is independent of all other program variables. If there is a variable with the same name elsewhere in the program, it is not affected by the defined function nor does it enter into the evaluation. Also, the dummy variable is assigned the value of the argument in the function call only for the duration of the function evaluation. Any single letter may be used for the dummy variable.

EXAMPLES:

```
10 DEF FNR(B)=2*B-C/3
20 DEF FNC(D)=2*D-C/3
30 DEF FNN(L)=FNC(L)+FNR(L)-1
```

Because of the definition in line 10 above, the following two statements are identical in operation:

```
100 LET G=B+4*(2*Y*Z-C/3)-M
100 LET G=B+4*FNR(Y*Z)-M
```

The argument (Y*Z in lines 100) of the function call (FNR) may be any expression. The expression is evaluated, and the dummy variable in the definition (D in line 20) is assigned that value.

Lines 10 and 20 define equivalent functions.

DELETE

KEYBOARD: YES
PROGRAM: NO

SYNTAX: {beginning line} DELETE {ending line}

EFFECT: Deletes a group of statements.

NOTES:

1. If the beginning line number is omitted, then 1 is assumed. All lines from 1 through the specified ending line will be deleted.

2. If the ending line number is omitted, then 9999 is assumed. All lines from the specified beginning line through 9999 will be deleted.

3. DELETE with no arguments deletes the entire program.

4. An individual line may be deleted by entering its line number only.

EXAMPLES:

```
100 DELETE

DELETE

100 DELETE 200

150
```

DIM

KEYBOARD: YES
PROGRAM: YES

SYNTAX: DIM dimension list

where
dimension list consists of array, string, vector or variable declarations or precision settings in any order, with each element except the first preceded by a comma.

EFFECT: The DIM statement instructs the system to reserve the correct amount of storage space for a number, an array, or a string by specifying an upper limit on the amount of space that will be required.

NOTES:

1. DIM is used to specify the precision of variables and the maximum number of elements which may be stored in a one- or two-dimensional array or in a string.

   When a variable is first encountered in a program, it is set to the precision specified in the last DIM statement, or to the default of 2% if no DIM settings containing a precision setting have been encountered. Precision may be set at 1%, 2%, 3%, or 4%.

2. To specify the maximum number of elements that may be stored in a one- or two-dimensional array or in a string, the DIM statement is followed by a variable name and one or two expressions enclosed in brackets. For a two-dimensional array, the first expression specifies the highest row number, and the second expression specifies the highest column number. If the value of an expression is not integral, the integer portion of the value is used. Negative dimensions are not allowed. Since an array always includes a row zero and a column zero, an array dimensioned A[3,5] contains four rows and six columns for a total of 24 elements. Lines 20, 40 and 500 in the example include two-dimensional arrays in DIM statements.

3. A one-dimensional array is treated as a column vector; i.e., it has only one column (column 0). The expression in the DIM statement specifies the highest row number. Lines 10, 20, and 40 include one-dimensional arrays in DIM statements.

   A one-dimensional array which is not included in a DIM statement is automatically dimensioned 10 by 0. A two-dimensional array which is not included in a DIM statement is automatically dimensioned 10 by 10.

4. The dimension of a string variable specifies the maximum number of bytes that the string can store. Strings are stored two characters to a word. String variables must always appear in a DIM statement because they are not automatically dimensioned. Lines 20 and 40 include strings in a DIM statement.

EXAMPLES:

```
10 DIM A[15]
20 DIM B2[7,8],C4[40],D$[50]
30 INPUT B
40 DIM D[2,3],4%,G[15],H,B$[100],3%,R
50 LET C=B+M
60 READ X,Y,Z
70 DATA 4%,17,34.4669980257,2
500 DIM Q5[X, INT(Y)]
```

In the program above, variable B and array D in lines 30 and 40 are set to two-word precision because the default of 2% has not been changed in the preceding DIM statements. The specification of 4% in line 40 then causes all variables which are encountered for the first time to be set to four-word precision. Thus, vector G and variable H are both set to four-word precision. The % precision specification has no effect on strings, so B$ is dimensioned as a 100 character string. The specification of 3% then causes any new variables following it to have three-word precision. Thus, R, C, and M in lines 40 and 50 have three-word precision, but since variable B was already encountered in line 30, its precision remains at 2%. The variables X, Y and Z also have three-word precision, and they receive the values 17, 344.6699802 and 2, respectively. Variables or expressions may be used to dimension an array, as shown in line 500.

NOTES: (Continued)

5. The number of words required to hold data and variables in Business BASIC may be calculated from the following formulas, where "precision" is the precision at which the variable was allocated.

type	number of words
simple variable	2 + precision
array	4 + (number of elements)*(precision)
string	3 + INT [(DIM+2)/2]
DATA statement	3 + (number of elements)*(precision)

   The number of elements in an array dimensioned [R,C] is (R+1)*(C+1) including row zero and column zero.

6. Numeric arrays may be redimensioned when the program is run by execution of a second DIM statement. The total size of the array given by the new dimensions may not exceed the total size of the original array. The precision of the array is fixed by the initial DIM statement.

7. Strings may be dimensioned once only; arrays may be redimensioned if the total number of elements does not increase. If an array is redimensioned, it uses the same precision as the first time it was dimensioned, thus ignoring the current precision setting.

DUMP

KEYBOARD: YES
PROGRAM: NO

SYNTAX: {beginning line}DUMP{<prot>}{$ddd,cc}{LU/}filename{!}{ending line}

EFFECT: Saves the program currently in the user's active file in a text file.

NOTES:

1. The user may specify numbers of the beginning and ending lines, inclusive, to be saved. If the beginning line number is not specified, the first line is used; if the ending line number is not specified, the last line is used.

2. The user may specify the cost and protection of the file. If not specified, the protection is set to 77 (maximum protection) and the cost to zero.

3. The user may specify the number of the logical unit on which the text file is to be saved. If not specified, then the user's assigned logical unit is used.

4. An existing text file on the user's own account may be overwritten by following the filename with an exclamation point (!). If the cost or protection is changed, both must be re-entered or the default is assumed.

EXAMPLES:

```
DUMP <72> $2.58 4/CASHFLOW3
```

This command saves the user's program in source form in the text file named CASHFLOW3 on logical unit four and protects it against any access by lower privilege users.

A BASIC program in the user's active file may be listed on a line printer by giving the command of the form

```
DUMP $LPT
```

where $LPT designates the appropriate printer.

END

KEYBOARD: YES
PROGRAM: YES

SYNTAX: END

EFFECT: Terminates program execution.

NOTES:

1. END is similar to STOP (which usually terminates execution to indicate an error) and may be used anywhere in a program.

2. It is not mandatory that the last statement in a program be an END statement.

EXAMPLES:

```
1000 END
```

EXIT

KEYBOARD: YES
PROGRAM: NO

SYNTAX: EXIT

EFFECT: Causes BASIC to exit to SCOPE.

NOTES:

1. EXIT is useful while debugging. The user may EXIT a program, perform functions which do not disturb the active file, and then re-enter BASIC to find the variables unchanged.

2. The effect of EXIT is similar to using <CTRL-C> or CHAIN "" except that EXIT does not disturb the stored program or current value of the program's variables.

EXAMPLE:

```
#BASIC
NEW
LOAD ABC.
EXIT
#SAVE ABC1
```

FOR

KEYBOARD: YES
PROGRAM: YES

SYNTAX: FOR control variable = initial expr TO limiting expr [STEP step expr]
.
.
.
NEXT control variable

EFFECT: Creates a program loop and repeats it a predetermined (or calculated) number of times. The control variable, sometimes called the "index variable", must be the same in the FOR statement and in its matching NEXT statement.

NOTES:

1. The FOR statement assigns the value of the initial expression to the control variable, and saves the value of the limiting expression as a 4% number. If the initial value does not already exceed the limiting value, control passes to the statement following the FOR statement.

2. The value of the control variable exceeds the limiting value if it is greater than the limit (for a positive step value) or less than the limit (for a negative step value). If the initial value exceeds the limiting value, a search is made for the matching NEXT statement, and control is immediately transferred to the statement followng the NEXT statement without executing the statements within the loop.

3. When a NEXT statement is encountered, the value of the step expression (assumed to be +1 unless specified) is added to the control variable. The control value is then checked against the limiting value. If the control value does not exceed the limiting value, control is transferred to the statement following the FOR statement. If the control value does exceed the limiting value, control passes to the next statement in sequence following the NEXT statement.

(Discussion of this statement is continued.)

EXAMPLES:

```
110 FOR A = 11 TO 5
120   FOR B3 = 6 TO -4 STEP -2
130     FOR M = J TO K+4 STEP B-D
          .
          .
          .
250     NEXT M
300   NEXT B3
600 NEXT A
```

The FOR and NEXT statements simplify the creation of a loop, eliminating the need for multiple GOTO statements. For instance, the following two programs perform similar functions:

Program 1

```
10 FOR A=B TO C STEP D
20 . . .
 .
 .
 .
100 NEXT A
110 . . .
```

Program 2

```
10 LET A=B
15 IF A>C GOTO 110
20 . . .
 .
 .
 .
100 LET A=A+D
105 IF A<=C GOTO 20
110 . . .
```

There is one important difference between these two programs: changing the values of C and D within the FOR-NEXT loop has no effect on the limit or step values because they are evaluated only once when the FOR statement is executed; changing C and D within the other program affects lines 100 and 105. The above example assumes that the value of D is positive.

FOR (Continued)

NOTES:

4. The initial value of the control variable is assigned before calculating the limiting and step values; therefore, use caution if the control variable is used within expressions for the limiting or step values.

5. FOR-NEXT loops may be nested up to eight levels deep as shown in the examples on the facing page. Note that for legal nesting, the matching FOR and NEXT statements can be connected without crossing lines.

6. The FOR-NEXT stack is used to keep track of FOR-NEXT loops. When a FOR statement is encountered, the control variable is added to the FOR-NEXT stack. When a second FOR statement is encountered, its control variable is added to the top of the stack, and the original control variable is pushed down the stack. This process continues with up to eight FOR statements. When a NEXT statement is encountered, the control variable which is at the top of the stack is "popped off", or eliminated, and the remaining control variables move back up.

   As many as eight control variables may be on the FOR-NEXT stack at a time; if a ninth control variable is added to the stack, the stack overflows and an error message appears.

7. Two examples of illegal nesting are shown on the facing page. In the first example, the two loops interfere with each other. When the NEXT A statement is encountered, the system checks whether the last FOR statement encountered was a FOR A. It was not, so the FOR B loop is dropped, and the FOR A statement is found. The FOR A/NEXT A loop is processed to completion, but an error occurs when the NEXT B statement is encountered.

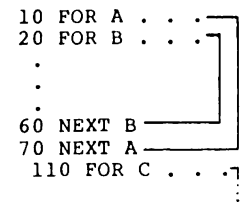
   The second type of illegal nesting involves use of the same control variable in nested loops. In this case the inner two loops are executed properly, but the outer loop is lost. When a FOR statement is executed, the system checks whether an existing loop uses the same control variable, and if so, the existing loop is dropped. Thus, the A loop created by line 10 in the example is aborted when line 30 is executed, and a "NEXT without matching FOR" error occurs at line 90.

   Because the previously existing loop is dropped, programs similar to the last program on the facing page can be properly executed.

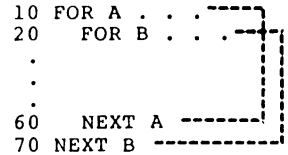
(Discussion of this statement is continued.)

EXAMPLES:

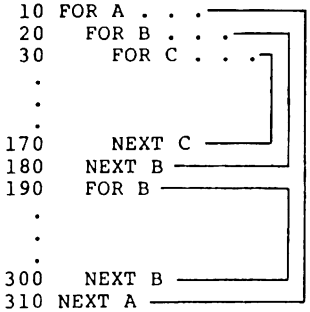
legal nesting



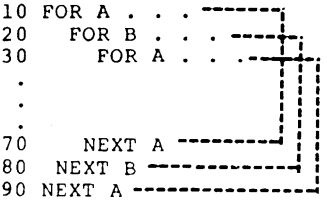
illegal nesting



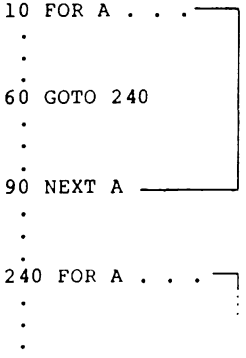
legal nesting



illegal nesting



legal nesting



FOR (Continued)

NOTES:

8. If the control variable (D in the example on the facing page) is an integer (1% precision), then the step and limit values are also evaluated as integers. The step and limit values are truncated to integers using simple truncation (not evaluated as though the INT function had been used). Thus, all arithmetic which must be performed by the system will be done using integers, and it will take approximately one-third as long to execute the looping function. Integers should be used whenever a short FOR-NEXT loop is used and maximum speed is desired.

   The difference between using an integer (1%) control variable and a non-integer control variable is illustrated by Example B on the facing page. Line 10 sets X to 2% precision. Since X is not an integer variable, the step value is not truncated and -1.2 is used. Line 50 sets A to 1% precision. Since A is an integer variable, the step value -1.2 is truncated to -1.

9. The maximum limit value when using an integer control variable is 7998; a limit of 7999 cannot be exceeded, and the loop would continue indefinitely after the value of the control variable reached 7999.

EXAMPLE A:

```
10 DIM 1%,D,3%
20 FOR D=1 TO 10 STEP 2.9
30   LET X=5/D
40   PRINT D;X
50 NEXT D
```

This program's output would be:

```
1 5
3 1.666666666
5 1
7 .7142857142
9 .5555555555
```

EXAMPLE B:

```
10 DIM 2%,X
20 FOR X= 10.2 TO 1.2 STEP -1.2
30   PRINT "X= "X
40 NEXT X
50 DIM 1%,A
60 FOR A= 10 TO 1 STEP -1.2
70   PRINT "A= "A
80 NEXT A
```

This program's output would be:

```
X=  10.2
X=  9
X=  7.8
X=  6.6
X=  5.4
X=  4.2
X=  3
X=  1.8
A=  10
A=  9
A=  8
A=  7
A=  6
A=  5
A=  4
A=  3
A=  2
A=  1
```

GOSUB

KEYBOARD: NO
PROGRAM: YES

SYNTAX: GOSUB subroutine line no.

EFFECT: The GOSUB statement transfers control to the specified line number.

NOTES:

1. The GOSUB and RETURN statements eliminate the need to repeat frequently used groups of statements in a program. Such a group of statements is called a subroutine. The subroutine must be exited using a RETURN statement.

2. A subroutine that has been entered with a GOSUB can itself contain a GOSUB statement. This nesting process can be carried out to eight levels. Each RETURN returns to the previous level. A RETURN statement cannot be executed without the previous execution of a GOSUB statement.

EXAMPLES:

```
10 DIM R$[10]
100 INPUT "Continue? (Enter YES or NO) "R$
110 PRINT
120 GOSUB 1000 !Check whether YES or NO was entered.
130 GOTO 500
140 GOTO 600
150 PRINT "You must enter YES or NO!"
160 GOTO 100
500 REM Perform the YES alternative
510 PRINT "The user entered YES"
520 STOP
600 REM Perform the NO alternative
610 PRINT "The user entered NO"
620 STOP
1000 REM Routine to check for YES or NO
1010 IF R$="YES" RETURN !return to next statement if YES
1020 IF R$="NO" RETURN 1 !return and skip next statement if NO
1030 RETURN 2 !return and skip 2 statements if neither
RUN
Continue? (Enter YES or NO) MAYBE
You must enter YES or NO!
Continue? (Enter YES or NO) YES
The user entered YES

STOP AT 520
RUN
Continue? (Enter YES or NO) NO
The user entered NO

STOP at 620
```

This program uses a subroutine to check whether the input string R$ is equal to "YES", "NO" or neither. If R$ is equal to "YES", the subroutine returns to the next statement following the GOSUB, which is "GOTO 500".

If R$ is equal to "NO", the subroutine returns to the program, skips one statement, and continues execution with the following statement, which is "GOTO 600".

If R$ is not equal to "YES" or "NO", the subroutine returns to the program, skips two statements, and continues execution with the following statement, which is "PRINT "You must enter YES or NO!""

GOTO

KEYBOARD: NO
PROGRAM: YES

SYNTAX: GOTO line number

EFFECT: This statement transfers control to the specified line.

NOTES:

1. GOTO must be followed by a line number to which control is to be transferred; there must be a statement in the program with that line number or an error will occur.

2. The statement is useful for jumping to another part of the program or for repeating a task indefinitely.

3. A GOTO should not be used to jump inside a FOR/NEXT loop because a "NEXT without matching FOR" error will occur when the NEXT statement is encountered (unless the FOR/NEXT loop is exited before the NEXT statement is encountered). A GOTO statement can be used to jump out of a FOR/NEXT loop.

4. A GOTO should not be used to jump inside a subroutine because a "'RETURN' WITHOUT 'GOSUB'" error will occur when a RETURN statement is encountered. A GOTO should not be used to jump out of a subroutine because the return address will be left on the stack, probably causing a "GOSUB STACK OVERFLOW" error.

EXAMPLES:

```
10 INPUT "Please enter a number from 1 to 5: "A
20 PRINT
30 IF A=1 GOTO 60
40 PRINT "A is not equal to 1"
50 GOTO 70
60 PRINT "A is equal to 1"
70 END
RUN
Please enter a number from 1 to 5: 3
A is not equal to 1

READY
RUN
Please enter a number from 1 to 5: 1
A is equal to 1
```

HELP

KEYBOARD: YES
PROGRAM: NO

SYNTAX: HELP {error number}

EFFECT: Prints a message describing the type of error encountered.

NOTES:

1. Some types of errors are serious enough that the program is stopped when the error is detected, and the word READY is printed following the error message. Other types of errors allow the program to continue with the next statement in sequence, thus allowing several errors to be detected in a single run.

2. If the error number is not specified, a message which describes the most recently encountered error, if any, is printed. If the error number is specified, a message which describes the specified error number is printed.

EXAMPLES:

```
ERROR #6
HELP
NO SUCH LINE NUMBER

HELP 30
USER FUNCTION NOT DEFINED
```

IF

KEYBOARD: YES
PROGRAM: YES

SYNTAX: IF logical expression statement{\statement}*

EFFECT: Provides conditional branching capabilities. The logical expression is evaluated and, if it is found to be true, the following statements on that line are executed.

NOTES:

1. Any BASIC statement is permitted as the conditionally executed statement, including another IF. Thus, the statement IF A>B IF C>D GOTO 100 transfers program control to statement 100 only if A is greater than B and C is greater than D. In these cases the second IF statement and its logical expression is executed only if the first is satisfied. Any assignment statement which immediately follows the logical expression must begin with the keyword LET.

2. If the test condition is false, control passes to the next line which begins with a line number (which may not be the next sequential statement if multi-statement lines are used).

3. The comparison symbols which may be used in the logical expression of an IF statement are listed below.

Symbol	Meaning
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<>	not equal to
=	equal to

4. IRIS Business BASIC does not use IF/THEN construction. A statement such as IF A>B THEN 500 may be entered, but the reserved word THEN has the same effect as GOTO and is converted to "GOTO" by BASIC.

5. Line 900 in the examples shows the advantage of using multiple-statement lines after an IF statement. If A is originally greater than B, the following statements on that line swap the value of A and B. If A is not originally greater than B, control transfers to the next numbered line, thereby skipping the remaining statements on line 900.

EXAMPLES:

```
10 IF A>B GOTO 100

200 IF B GOTO 100

70 IF LEN(A$)>10 LET C=D

800 IF A=5 PRINT "TRUE"

100 IF B$<>"YES" IF B$<>"NO" PRINT "Please enter YES or NO."

900 IF A>B LET T= A \ A=B \ B=T

170 IF A$,B$=C$ GOTO 1000
```

IF ERR

KEYBOARD: NO
PROGRAM: YES

SYNTAX: IF ERR error mode{statement}

EFFECT: The IF ERR statement sets or clears an error branch which may be used to trap non-abortive errors within the BASIC program instead of allowing them to cause an error printout; the error trapping allows the program to attempt corrective action. Escape (ESC, <CTRL-C>, and <CTRL-D>) is also trapped.

NOTES:

1. If the error mode is set to zero, error 99 is generated when error trapping is enabled and either <ESC> or <CTRL-C> is pressed. If the error mode is set to one, error 99 is generated when <ESC> is pressed and error 199 is generated when <CTRL-C> is pressed.

2. The statement is not executed at the time the IF ERR is encountered, but an error branch is set so that any non-abortive error or ESC causes the statement to be executed as if it were at the line where the error occurred. Therefore, unless the statement is a GOTO, GOSUB, RETURN or ON statement, the next statement executed will be the one following the statement in which the error occurred. If the statement is a GOSUB, then a normal RETURN from the subroutine will also return to the statement following the statement in which the error occurred, and a RETURN -1 will return control to the statement in which the error occurred.

3. If the statement is an assignment statement, it must begin with the keyword LET.

4. An IF ERR statement may be used anywhere in the program to set the error branch to point to its statement. An IF ERR with no following statement clears the switch so that any error will cause a normal error message printout.

(Discussion of this statement is continued.)

EXAMPLES:

```
10 IF ERR 0 LET E=E+1

20 IF ERR 0 GOSUB 1000

150 IF ERR 1 GOTO 9000

200 IF ERR 0
```

NOTES: (Continued)

5. SPC (8) may be used in the error handling subroutine to determine the type of error that occurred. IF ERR also causes ESC and <CTRL-C> to be trapped. In this case, SPC (8) returns 99 as the error type if the error mode is set to 0, and returns error 99 if ESC is trapped and 199 if <CTRL-C> is trapped if the error mode is set to 1. If the program is not copy-protected against the user, then the user can bypass the ESC trap by pressing <CTRL-Y> followed by ESC.

6. SPC (10) may be used in the error handling subroutine to determine the line number where the last error occurred. Thus, an error message may be printed giving the line number of the error if the program cannot recover from the error.

7. An IF ERR statement must be the first executable statement of a program in order to trap <ESC> and <CTRL-C> when chaining from one BASIC program to another BASIC program. Executable statements include all statements except REM and DATA.

INPUT

KEYBOARD: YES
PROGRAM: YES

SYNTAX: INPUT{@c,r;}{LEN limit;}{"lit prompt"}var list{,{@c,r{LEN limit;}{"lit prompt"}var list}*

or

INPUT{@c,r;}{LEN limit;}{"literal prompt"}string variable

where

c - column

r - row

var list - a list of vector or array variables in any order, with each element except the first preceded by a comma

EFFECT: This statement directs the system to accept data entered from the keyboard. The system temporarily suspends program execution, prints a question mark or the literal prompt, and awaits data to be entered by the user.

NOTES:

1. When the RETURN key is pressed after entering data in response to an INPUT statement, the cursor does not move, but remains at the same position.

2. To enter more than one number in response to an INPUT statement, either separate the numbers by commas or press the RETURN key after entering each number.

3. The standard question mark prompt character may be replaced by any prompt message given in quotes at the beginning of the statement. If there is nothing between the quotes, then input will be enabled with no prompt at all.

4. If the user's response to an INPUT statement is rejected, the system prints a backslash and question mark and outputs a bell. The user may then enter appropriate information and press RETURN. INPUT response is rejected if input for a numeric variable is required and the user enters nothing or enters illegal characters. However, if error trapping is enabled, then an empty input is accepted as a zero value and entry of an illegal character causes an error branch.

5. Two string variables or a numeric variable following a string variable may not be input using a single INPUT statement.

EXAMPLES:

```
110 INPUT A,B
120 INPUT C,D4,E
130 INPUT B[2]
140 INPUT "WHAT IS YOUR NAME? "N$
150 INPUT ""J
300 INPUT LEN 5;A$
500 INPUT @10,5;"AGE: "A,@10,10;"STATUS: "B
```

The following program inputs two numbers from the keyboard, adds them, prints the sum, and asks for two more numbers. Line 20 causes the program to stop if the first value entered is zero. The user may also press the ESC key at any time to abort the program run.

```
10 INPUT "ENTER THE FIRST NUMBER: "A,"ENTER THE SECOND NUMBER: "B
20 IF A=0 STOP
30 PRINT " THE SUM IS"A+B
40 GOTO 10
```

The following program accepts only the letters "Y" or "N" as input. The user may type "YES" or "NO", but the system acts as though <RETURN> were pressed as soon as the specified input length was reached. The excess letters ("ES" or "O") are queued up as type-ahead.

```
10 INPUT LEN 1; "Yes (Y) or No (N)? "A$
20 IF A$= "Y" GOTO 100
30 IF A$= "N" GOTO 200
40 PRINT "PLEASE ANSWER YES OR NO" \ GOTO 10
```

NOTES: (Continued)

6. The LEN clause may be used to limit the length of user input to the number of characters specified by "limit". For example, line 300 limits the length of user input for A$ to five characters. The LEN clause affects only the next read from the terminal. Setting the input length to zero or greater than the I/O Buffer size has the same effect as not including a LEN clause.

7. The @column,row clause may be used to specify the column and row at which the input prompt will be displayed (or the cursor position, if no input prompt is to be displayed). Note that the column is specified first, then the row.

KILL

KEYBOARD: YES
PROGRAM: YES

SYNTAX: KILL filename string{,filename string}*

EFFECT: Deletes the specified data files.

NOTES:

1. The filename string may contain a literal string or string variable which contains the filename of a data file. It may also include the number of the logical unit on which the file is stored (LU/). The effect is the same as if the KILL command were given in the system command mode. The user's account on which the file was created will be credited for the disc blocks.

2. An error occurs in three cases: if any of the specified strings do not contain a legal filename; if a specified file is write protected; or if a specified file does not exist.

3. If a legal command is given to kill a file that is open at the time on a data channel, the filename will be removed from the INDEX immediately, but the file will remain open on the channel. The file will be deleted later when the channel is closed by a CLOSE statement or cleared by program termination.

EXAMPLES:

```
KILL "FILE23"
KILL M$,"3/XPRL",D$
```

LET

KEYBOARD: YES
PROGRAM: YES

SYNTAX: LET destination variable = source expression or source string expression

EFFECT: This statement assigns a value to a variable.

NOTES:

1. In a LET statement, the symbol "=" should be read as "take the value of", not as "equals". For example,

   ```
   LET P=6
   ```

   should be read "LET P take the value of 6". Therefore, it is possible to have

   ```
   LET B=B+1
   ```

   which means let the new value of B take the existing value of B with one added to it.

2. The word LET is not required when entering an assignment statement except when the assignment statement follows the logical expression of an IF statement or error mode of an IF ERR statement. LET will be assumed as the statement type if no directive word is entered, and the word LET will be printed when the program is listed.

3. Any numeric expressions are allowed in subscripts.

4. Only one element of an array may be changed by a single LET statement.

5. When using a LET statement such as LET A=A$, only a literal string, subscripted string variable or string variable may be used; concatenated strings are not allowed.

EXAMPLES:

```
10 LET P=6

20 LET R2=Q+(T/5)

30 LET A[2]=C+5

40 LET B=B+1

50 D=P+5*Q-SQR(A[Z])

60 LET A$="THIS IS A STRING"

70 LET A$= B$,C$
```

```
10 DIM A$ [ 50],B$[ 50]
20 LET A$="THIS WAS A LONG STRING"
30 LET A$ [6,8]="IS"
40 LET A$ [11,15]=""
50 PRINT A$
60 LET A$ [5]="HELLO"
70 LET B$=A$
80 PRINT B$
90 PRINT A$ [10]
100 LET B$="=",B$
110 PRINT B$
```

This program's output would be:

```
THIS IS A STRING
THISHELLO
```

==========

LET ... USING

KEYBOARD: YES
PROGRAM: YES

SYNTAX: LET string variable = numeric expr list USING format string expr

EFFECT: Assigns a value to a string variable according to the format specified by the format string.

NOTES:

1. The value of the format string expression is used to specify the form in which the values represented in the expression list is to be printed. The format string expression may contain one or more format fields. It may also contain blanks (spaces) and any characters other than format control characters. The numeric expression may include numeric expressions, string expressions, commas, semi-colons, and TAB functions. The format fields which may be used are shown in Table 4-3, which follows the PRINT...USING statement.

2. Subscripted string variables may be used in the format string expression. However, the format string may not contain concatenated strings or concatenated string variables.

EXAMPLE:

```
10 READ A,B
20 DIM A$[20],B$[20]
30 LET A$= 12.345 USING "$$###.##"
40 PRINT A$
50 LET B$[1,4]=A+B USING "####"
60 PRINT B$
70 DATA 5, 10

RUN
 $ 12.34
  15
```

LIST

KEYBOARD: YES
PROGRAM: NO

SYNTAX: LIST

{first line} LIST {last line}

EFFECT: Lists the current program from the first line specified (or from the beginning) through the last line (or through the end).

NOTES:

1. If the first line is not specified, the system defaults to 1 or the beginning of the program; if the last line is not specified, the system defaults to 9999 or the end of the program.

2. The <ESC> key may be pressed at any time to terminate listing.

3. When a program containing FOR-NEXT loops is listed, statements within each loop are indented for easy identification of loops and nesting.

4. If a first line number is specified and there is no statement with the specified line number, then the listing starts with the next higher numbered line.

EXAMPLES:

Command:	Effect:
LIST	lists the entire program as it stands
180 LIST	lists from line 180 through the end of the program
270 LIST 600	lists lines 270 through 600 of the program

LOAD

KEYBOARD: YES
PROGRAM: NO

SYNTAX: LOAD {-}filename

EFFECT: Accesses a text file.

NOTES:

1. This command may be used to access a program that was saved as a text file by use of the DUMP command.

2. This command may be used to load BASIC source code which is in text file form.

3. If the filename is preceded by a minus sign "-" all end-of-line comments (!comment) are not loaded.

EXAMPLES:

```
LOAD CASHFLOW3

LOAD -CASHFLOW3
```

MAT

KEYBOARD: YES
PROGRAM: YES

SYNTAX: MAT destination matrix variable=source matrix variable

EFFECT: This statement sets all the elements of a destination matrix equal to the corresponding elements of a source matrix.

NOTES:

1. The destination matrix is automatically dimensioned the same as the source matrix and each element of the source matrix is copied to the same position in the destination matrix.

2. If the destination matrix existed prior to executing this statement, then it must be the same precision as the source matrix. The destination matrix will also be redimensioned automatically (an error occurs if the destination is not large enough).

3. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT statement has no effect on row zero or column zero.

EXAMPLES

```
200 MAT C = Y

500 MAT M2 = Q6
```

MAT INPUT

KEYBOARD: YES
PROGRAM: YES

SYNTAX: MAT INPUT matrix variable[[expr[,expr]]][,matrix variable[[expr[,expr]]]]*

EFFECT: This statement allows the entry of an entire matrix from the keyboard during program execution. The matrix may be dimensioned in the INPUT statement or given a new working size.

NOTES:

1. When entering the elements of each row, each element must be separated by a comma, and each complete row of data must be entered before pressing RETURN. For example, while entering the data necessary for the execution of

   ```
   190 MAT INPUT A[3,4]
   ```

   Four data items separated by three commas must be typed in before pressing RETURN. If too few or too many data items are entered, the system prints a backslash and requests the entire new line of data.

2. The matrix is filled from left to right across one row at a time. For example, element 1,1 is filled first, then element 1,2, then element 1,3 and so on.

3. Any matrix created by a MAT statement with a single dimension assumes a second dimension of one. For example, line 180 of the examples is equivalent to

   ```
   180 MAT INPUT R[5,1]
   ```

4. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT INPUT statement has no effect on row zero or column zero.

EXAMPLES:

```
170 MAT INPUT F
180 MAT INPUT R[5]
190 MAT INPUT C[E,J],F
```

MAT PRINT

KEYBOARD: NO
PROGRAM: YES

SYNTAX: MAT PRINT matrix variable{;|,matrix variable}*

EFFECT: This statement causes the system to print one or more entire matrices, row by row.

NOTES:

1. MAT PRINT simplifies programming with matrices by reducing the number of statements required to print a matrix. This is illustrated by programs A and B on the facing page.

2. A matrix may be printed in a "packed" form with up to 12 elements on a line by placing a semi-colon after the matrix variable. Otherwise, the matrix is printed with five elements per row.

   If the matrix variable is followed by a comma or semi-colon, an extra line feed is generated after printing the matrix to provide double spacing between matrices. More than one matrix may be printed in one statement by separating the matrix variable names by a comma or semi-colon.

3. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT PRINT statement has no effect on row zero or column zero.

EXAMPLES:

```
10 MAT INPUT A[3,4]
20 MAT INPUT B[2,3]
30 MAT PRINT A
40 MAT PRINT A;
50 MAT PRINT A,B
RUN
?1,2,3,4
?5,6,7,8
?9,10,11,12
?10,20,30
?40,50,60
 1             2             3             4
 5             6             7             8
 9             10             11            12

 1     2     3     4
 5     6     7     8
 9     10     11     12

 1       2       3       4
 5       6       7       8
 9       10       11       12

 10     20     30
 40     50     60
```

A. Sample Program Without MAT Statements

```
100 DIM A(3,3)
110 FOR R= 1 TO 3
120 FOR C= 1 TO 3
130 READ A(R,C)
140 NEXT C
150 NEXT R
160 FOR R= 1 TO 3
170 FOR C= 1 TO 3
180 PRINT A(R,C)
190 NEXT C
200 NEXT R
210 DATA 1,2,3,4,5,6,7,8,9
220 END
```

B. Sample Program With MAT Statements

```
100 DIM A(3,3)
110 MAT READ A
120 MAT PRINT A
130 DATA  1,2,3,4,5,6,7,8,9
140 END
```

MAT READ

KEYBOARD: YES
PROGRAM: YES

SYNTAX: MAT READ matrix variable{[expr{,expr}]}{,matrix variable {[expr{,expr}]}}*

EFFECT: This statement allows the computer to read an entire matrix from DATA statements. The matrix may be dimensioned in this statement or given a new working size.

NOTES:

1. MAT READ simplifies programming with matrices by reducing the number of statements required to read values into a matrix. This is illustrated by the two sample programs on the facing page.

2. The matrix is filled from left to right across one row at a time. For example, element 1,1 is filled first, then element 1,2, then element 1,3 and so on.

3. Any matrix created by a MAT statement with a single dimension assumes a second dimension of one. For example, line 415 in the examples is equivalent to line 420.

4. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT READ statement has no effect on row zero or column zero.

EXAMPLES:

```
400 MAT READ A

405 MAT READ B,C

410 MAT READ E[L,N],A

415 MAT READ F[7]

420 MAT READ F[7,1]
```

A. Program Without MAT Statements:

```
100 DIM A(3,3)
110 FOR R= 1 TO 3
120 FOR C= 1 TO 3
130 READ A(R,C)
140 NEXT C
150 NEXT R
160 FOR R= 1 TO 3
170 FOR C= 1 TO 3
180 PRINT A(R,C)
190 NEXT C
200 NEXT R
210 DATA 1,2,3,4,5,6,7,8,9
220 END
```

B. Program With MAT Statements:

```
100 DIM A(3,3)
110 MAT READ A
120 MAT PRINT A
130 DATA  1,2,3,4,5,6,7,8,9
140 END

RUN

 1          2          3
 4          5          6
 7          8          9
```

MAT READ#

KEYBOARD: NO
PROGRAM: YES

SYNTAX: MAT READ #channel,{record{,item{,delay}}};array or string variable{;}

EFFECT: Reads data from a single binary item of a formatted data file or, starting at a specified location, from a contiguous data file into an entire numeric array or into an entire string variable.

NOTES:

1. The MAT READ# statement functions exactly as the READ# statement except that an entire array (including row zero and column zero), or an entire string is read in one statement. No matrix or string subscripts are allowed. For contiguous files, the value of the item number expression is used as a byte displacement into the record, but all transfers are word-oriented; if an odd byte displacement is given (as in line 220 of the examples), then the transfer begins at the next higher even byte displacement. The entire array or string variable is filled by copying directly from the file unless the item (formatted file only), or the file (contiguous file only) ends before the entire array or string variable is filled, in which case the remainder of the array or string variable remains unchanged. For formatted files, the item type must be binary. No data conversion takes place, and it is the responsibility of the user's program to ensure that the data is read into the type of variable that matches the data form.

2. For a contiguous file, the byte displacement to the next item in the file is equal to twice the number of words transferred. The number of words transferred will be INT(d/2+1) for a string variable or (r+1)*(c+1)*p for an array variable, where:

   r = row dimension of array,
   c = column dimension of array,
   p = number precision of array, and
   d = string dimension (number of bytes).

EXAMPLES:

```
190 MAT READ #1,20;A

200 MAT READ #K+2,8;B

210 MAT READ #3,10;A$

220 MAT READ #1,R*2,3;B$
```

NOTES: (Continued)

3. If the item is not specified, the system defaults to item zero. If the record is not specified, the system defaults to sequential access.

4. A string variable might be used in a MAT READ# statement when, for example, it is necessary to read the full dimensioned size of the variable, ignoring terminators.

5. The terminating semicolon may be used as a record-lock command.

MAT WRITE#

KEYBOARD: NO
PROGRAM: YES

SYNTAX: MAT WRITE #channel,{record{,item{,delay}}};array or string variable{;}

EFFECT: This statement writes all data from a numeric array or from a string variable into the specified single binary item of a formatted data file or into a contiguous data file at a specified starting point.

NOTES:

1. The MAT WRITE# statement functions exactly as the WRITE# statement except that an entire array (including row zero and column zero) or an entire string variable is written by one statement. No matrix or string subscripts are allowed. In the case of a contiguous file, the value of the item number expression is used as a byte displacement into the record, but all transfers are word-oriented; if an odd byte displacement is given (as in line 220 of the examples), then the transfer begins at the next higher even byte displacement. If there is enough space in the file, then the entire array or string is written. However, if the item is too small (formatted file only), or the end of the file is reached (contiguous file only), then the data is truncated and no error message is given. In the case of a formatted file, the entire array or string is written into a single item whose type must be binary. No data conversion takes place, and it is the responsibility of the user's program to ensure that the data will later be read back into the same type of variable.

2. If an array variable is specified, then the entire array, including row zero and column zero, will be written. If writing a string variable, then the entire string as dimensioned, including the byte reserved for a terminator and any intermediate terminator bytes, will be written.

3. For a contiguous file, the byte displacement to the next item in the file is equal to twice the number of words transferred. The number of words transferred will be INT (d/2+1) for a string variable or (r+1)*(c+1)*p for an array variable, where:

   r = row dimension of array,
   c = column dimension of array,
   p = number precision of array, and
   d = string dimension (number of bytes).

EXAMPLES:

```
190 MAT WRITE #1,20;A
200 MAT WRITE #C,2*R,8;B
210 MAT WRITE #3,10;A$
220 MAT WRITE #L,R,3;B$
```

IN = (d/2+1)

(R+1) * (C+1) * P

R = Row Dim
C = Col Dim
P = Precision

so f(3) @ 2%
= (0+1) * (3+1) * 2
= 1 * 4 * 2
= 4 * 2
= 8

NOTES: (Continued)

4. A string variable might be used in a MAT WRITE# statement when, for example, it is necessary to write the full dimensioned size of the variable, ignoring terminators.

5. The terminating semicolon may be used as a record-lock command.

MAT ... =CON

KEYBOARD: YES
PROGRAM: YES

SYNTAX: MAT matrix variable=CON[(dimension expr[,dimension expr])]

EFFECT: Sets all of the elements of the specified matrix equal to one.

NOTES:

1. This statement simplifies the process of setting all elements of a matrix to one. This is shown on the facing page.

2. Any matrix created by a MAT statement with a single dimension assumes a second dimension of one. For example, line 155 of the examples is equivalent to line 160.

3. The matrix may be dimensioned or given a new working size by including dimension expression(s) in this statement.

4. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT ...=CON statement has no effect on row zero or column zero.

EXAMPLES:

```
150 MAT D = CON

155 MAT E = CON (8)

160 MAT E = CON (8,1)

165 MAT Z = CON (X,Y)
```

Line 155 and 160 above are equivalent to the following program:

```
160 DIM E[X,Y]
170 FOR I=1 TO X
180   FOR J=1 TO Y
190     LET E[I,J]=1
200   NEXT J
210 NEXT I
```

MAT ...=IDN

KEYBOARD: YES
PROGRAM: YES

SYNTAX: MAT matrix variable=IDN {(dimension expr,dimension expr)}

EFFECT: This statement establishes an identity matrix. The elements comprising the main diagonal are set equal to one, and all other elements are set equal to zero.

NOTES:

1. The identity matrix must be two-dimensional and should be square; i.e., the two dimensions should be equal. For example, line 220 of the examples assigns the matrix G:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

2. If the matrix is not square, then the main diagonal is assumed to start at the lower right corner. For example, line 250 of the examples assigns matrix B the value

```
0 1 0 0
0 0 1 0
0 0 0 1
```

3. A new working size may be specified by including dimension expressions in this statement.

4. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT ...=IDN statement has no effect on row zero or column zero.

EXAMPLES:

```
210 MAT F = IDN
220 MAT G = IDN (4,4)
230 MAT H = IDN (5,5)
240 MAT I = IDN (B4,B4)
250 MAT B = IDN (3,4)
```

MAT ... =INV

KEYBOARD: YES
PROGRAM: YES

SYNTAX: MAT destination matrix variable = INV (source matrix variable)

EFFECT: This statement inverts the elements of a source matrix according to the rules of matrix arithmetic and assigns the result to a destination matrix.

NOTES:

1. Only square, two-dimensional matrices may be used in this statement; i.e., the dimensions must be equal and non-zero. If the destination matrix existed prior to executing this statement, then it must be of the same precision as the source matrix.

2. A matrix may take on the value of the inverse of its former self, as in statement 200.

3. As a side effect of matrix inversion, the determinant value is evaluated according to the rules of matrix arithmetic. (The determinant value is defined as the sum of the products formed according to the rules of matrix arithmetic from a series of quantities arranged in an equal number of columns and rows.) After inverting a matrix, the DET function may be used to return the matrix's determinant value. This determinant value is available until another matrix is inverted or until a new run is initiated by the RUN command.

4. The facing page shows a matrix and the result of its inversion according to the rules of matrix arithmetic.

5. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT ...=INV statement has no effect on row zero or column zero.

6. For further information on matrix arithmetic, consult an appropriate text book.

EXAMPLES:

```
100 MAT B = INV (A)
200 MAT F = INV (F)
```

```
1  2  3    -1 6 6 6 7.5    3 3 3 3 3.5    -1 6 6 6 6.5
4  5  6 =   3 3 3 3 3     -6 6 6 6 6        3 3 3 3 3
7  8  9    -1 6 6 6 5.9    3 3 3 3 3      -1 6 6 6 6 .6
```

MAT ... =TRN

KEYBOARD: NO
PROGRAM: YES

SYNTAX: MAT destination matrix variable = TRN (source matrix variable)

EFFECT: This statement establishes a matrix which is the transposition of a specified matrix; i.e., it exchanges the rows and columns.

NOTES:

1. If the source matrix has the dimensions (M,N), then the destination matrix is dimensioned (N,M). A sample transposition, as commanded in line 450 of the examples, produces the following results.

R (source matrix)	Q (destination matrix)
1 2 3	1 4
4 5 6	2 5
	3 6

2. It is not necessary for the destination matrix to have been previously dimensioned; however, if the destination matrix does exist previous to executing this statement, it must be the same precision as the source matrix.

3. CAUTION! A statement of the form

```
120 MAT S = TRN (S)
```

   is illegal. It may be executed, but the result will not be an acccurate transposition.

4. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT ...=TRN statement has no effect on row zero or column zero.

EXAMPLES:

```
450 MAT Q = TRN (R)

460 MAT L = TRN (A)
```

MAT ... =ZER

KEYBOARD: YES
PROGRAM: YES

SYNTAX: MAT matrix variable=ZER {(dimension expr{,dimension expr})}

EFFECT: This statement sets all the elements of the specified matrix equal to zero. The matrix may be created or given a new working size in this statement.

NOTES:

1. This statement simplifies the process of setting all the elements of a matrix to zero. The facing page illustrates this.

2. Any array created by a MAT statement with a single dimension assumes a second dimension of one. For example, line 200 of the examples is equivalent to line 300.

3. A matrix may be dimensioned or given a new working size by including dimension expression(s) in this statement.

4. A matrix is defined as a two-dimensional array excluding row zero and column zero. Therefore, the MAT ...=ZER statement has no effect on row zero or column zero.

EXAMPLES:

```
100 MAT A = ZER

200 MAT B = ZER (15)

300 MAT B = ZER (15,1)

400 MAT Z = ZER (9,14)

500 MAT L = ZER (E,F)

600 MAT B = ZER (R,C)
```

Line 600 above is equivalent to the following program:

```
40 DIM B[R,C]
50 FOR I = 1 TO R
60   FOR K = 1 TO C
70     LET B[I,K] = 0
80   NEXT K
90 NEXT I
```

NEXT

KEYBOARD: NO
PROGRAM: YES

SYNTAX: NEXT control variable

EFFECT: Transfers control out of a FOR...NEXT loop when the value of the control variable exceeds the value of the limiting expression specified in the matching FOR statement.

NOTES:

1. The value of the index variable exceeds the limit value if it is greater than the limit (for a positive step value) or less than the limit (for a negative step value). If the initial value in the FOR statement exceeds the limit value, a search is made for the matching NEXT statement, and control is immediately transferred to the statement following the NEXT statement without executing the statements within the loop.

2. When a NEXT statement is encountered, the step value (assumed to be +1 unless specified) is added to the control variable.

   The control value is then checked against the limiting value. If the control value does not exceed the limiting value, control is transferred to the statement following the FOR statement. If the control value does exceed the limiting value, control passes to the next statement in sequence following the NEXT statement.

3. Refer to the FOR statement in this section for further information on FOR/NEXT loops.

EXAMPLES:

```
110 FOR A = 1 TO 5
120   FOR B3 = 6 TO -4 STEP -2
130     FOR M = J TO K+4 STEP B-D
        .
        .
        .
250     NEXT M
300   NEXT B3
600 NEXT A
```

ON

KEYBOARD: NO
PROGRAM: YES

SYNTAX: ON selecting expression $\begin{matrix}\text{GOTO}\\\text{GOSUB}\end{matrix}$ list of line numbers

EFFECT: Transfers control to one of several line numbers, depending on the integer value of the selecting expression.

NOTES:

1. When the ON statement is encountered, the selecting expression is evaluated and the resulting value is truncated (not rounded) to an integer. The result is used as an index to the list of line numbers. Control then passes to the selected line number and proceeds from that statement. For example, if the expression evaluates to 1, control passes to the first specified line number; if it evaluates to 2, control passes to the second specified line number.

2. If the integer value of the selecting expression is zero, a negative number, or greater than the number of line numbers listed, the GOTO or GOSUB is not executed; control is transferred to the statement immediately following the ON statement.

   The subroutine given control by an ON. . .GOSUB statement should be exited only with a RETURN statement.

3. The line numbers following GOTO or GOSUB must be separated by commas. There may be any number of line numbers listed as long as the statement fits on one line.

4. To illustrate the concept, statement 20 of the examples will transfer control to line 130, 140, 200, or 210 if the integer value of J is 1, 2, 3, or 4, respectively.

5. ON. . .GOTO is similar to the statement GOTO. . .OF implemented on some systems.

EXAMPLES:

```
10 ON LOG(R)+1 GOTO 95,407

20 ON J GOSUB 130,140,200,210

30 ON P+1 GOSUB 190,500,650

40 ON A GOSUB 400,400,350,410,430

60 ON J/2-5 GOTO 150,300,100,300,40




70 ON A+1 GOTO 100,200,300
 .
 .
 .
100 REM CONTROL PASSES TO THIS LINE IF A+1= 1
 .
 .
 .
200 REM CONTROL PASSES TO THIS LINE IF A+1= 2
 .
 .
 .
300 REM CONTROL PASSES TO THIS LINE IF A+1= 3
```

OPEN

KEYBOARD: NO
PROGRAM: YES

SYNTAX: OPEN #c{=mode},filename string{{,#c{=mode}},filename string}*

EFFECT: Opens a data file or peripheral device on a channel.

NOTES:

1. This statement opens an existing data file or peripheral device, identified by a filename string, on each channel identified by a channel number expression. Additional filenames not preceded by a channel number are opened on successive channels.

2. The mode indicates whether the file is to be opened in file maintenance mode for special manipulations. Mode zero is the default and indicates a normal open; it may be selected by omitting the clause. Any other value selects a file maintenance mode. All files listed in a given OPEN statement are opened in the specified mode until the mode is respecified.

EXAMPLES:

```
20 OPEN #0,"PRIME",B$

45 OPEN #A,D$[6,M-1],#4,"JOE5","$LPT"

70 OPEN #1=2, "INDEX"
```

In line 20 above, the file PRIME is opened on channel zero, and the file identified by the string in B$ is opened on channel one.

In line 45, the file identified by characters 6 through M-1 of D$ is opened on the channel specified by the value of A, the file JOE5 is opened on channel four, and the line printer is opened on channel five.

In line 70, the file INDEX is opened on channel one for file maintenance access by a program whose GUARD bits are set. The program must be guarded by the system manager before the file can be opened in mode two.

PRINT

KEYBOARD: YES
PROGRAM: YES

SYNTAX: PRINT {{'xx';}{@col,row;}numeric/string expr{,}}*
;

EFFECT: Prints text, numbers, and computational results on the user's terminal.

NOTES:

1. Most terminals have 80 columns or print spaces numbered zero through 79 across each line. The line is divided into five fields of 15 spaces each, starting at columns 0, 15, 30, 45 and 60. A comma in a PRINT statement causes a column tab; i.e., it causes spacing to the beginning of the next field.

   On terminals with a longer line, the comma tabs extend to the end of the line at every 15 character positions, and then wrap around the specified line length. A precision three or four variable or any expression may cause too many significant digits for 15 character spaces, so that the value printout and the comma tab occupy 30 spaces total.

   When a PRINT statement contains more than one expression, the expressions must be separated by commas or semicolons. A semicolon causes close packing (no column tabs). Each number is printed with either a leading minus sign or space, the value, and one trailing space. Therefore, the use of semicolons will print numbers in the closest readable form.

2. A verbatim message may be printed by enclosing it in quotation marks as shown in example 300. Semicolons are optional before and after literal strings except following the use of the TAB function.

3. A quotation mark may be included in a literal string by using two apostrophes, and a carriage return may be included by using <CTRL-Z> where the RETURN is desired. Any special characters may be included in literal strings.

(Discussion of this statement is continued.)

EXAMPLES:

```
100 PRINT A
140 PRINT 6*A,B,SQR(B)+C,
300 PRINT "THE ''BEST'' ANSWER IS"R
440 PRINT "THE SUM OF"X"AND"Y"IS"X+Y
610 PRINT
770 PRINT E;TAB(20);"*"
990 PRINT EXP(D+SQR(X))
```

If A=10, B=20, C=30, R=40, X=50, Y=60, E=70 and D=80, the output of this program would be:

```
 10
 60          20          34.472135954999
THE "BEST" ANSWER IS 40
THE SUM OF 50 AND 60 IS 110

70                *
6.5235543368236E+37
```

Statement 100 above prints the current value of variable A and then causes a carriage return and line feed.

Statement 140 prints the value of 6*A starting at column zero, the value of B starting at column 15 and the value of SQR(B)+C starting at column 30. The comma at the end of the statement causes spacing to column 45 where printing ceases with a carriage return.

Statement 300 shows how to enter and print quotation marks.

Statement 440 and its output is a good example of the use of literal strings.

The spaces before and after the printed numbers are actually part of the numeric value printouts as described previously. Each printout is followed by a carriage return and line feed unless this is suppressed by either a comma or a semi-colon at the end of the PRINT statement. Therefore, an empty PRINT statement, as in example 610, causes only a carriage return and line feed.

Statement 770 above prints the value of E, spaces to column 20, and prints an asterisk.

NOTES:

4. The TAB function may be used for further control of a printout. A field of the form

   TAB (expression);

   within a PRINT statement causes spacing to the column number specified by the integer value of the expression. If printing has already occurred beyond the specified column, no further spacing takes place. The print line is considered to be circular, so on a terminal with a 75 character print line, columns 75, 150, 225, etc. are the same as column zero. A negative value for the TAB expression causes an error. The TAB function may be used only in a PRINT, PRINT# or PRINT USING statement.

5. The resulting string from PRINT that goes into the I/O buffer is not printed each time but is buffered up until the buffer is filled, the user is swapped out, or another BASIC statement needs to use the I/O buffer for something other than PRINT (such as INPUT). The user can force printing with a SIGNAL 3,0 statement.

6. A sequence of two-digit terminal control codes, shown by xx in the syntax statement, may be included to specify special printing instructions. Terminal control codes which may be used are described in Appendix E. Note that the sequence of control codes is enclosed in single quotation marks.

7. The row and column may be included to specify the position on the terminal screen at which the information should be printed. The row and column may be represented by expressions.

8. A final semicolon may be included at the end of the expression list to suppress the output of RETURN/LINE FEED.

EXAMPLES:

```
170 PRINT 'CS'

250 PRINT @ 10,20;A$

560 PRINT 'CS';@10,5;A$ @X,Y;B$

840 PRINT 'MUMURB';"THE NUMBER "X" IS TOO LARGE";
```

PRINT USING

KEYBOARD: YES
PROGRAM: YES

SYNTAX: PRINT USING format string;expression list{;}

EFFECT: Prints text, numbers, and computational results in a format specified by a string.

NOTES:

1. The value of the format string is used to specify the form in which the expression list is to be printed. One subscript is allowed in the designation of the format string; if a second subscript is specified, it is ignored. The format string may contain one or more format fields which control the form of printout of the numeric expressions in the expression list. It may also contain blanks (spaces) and any characters other than format control characters. The expression list may include numeric expressions, string expressions, commas, semi-colons, and TAB functions.

2. Printing is accomplished by starting a scan of the expression list. Any string expressions are printed and tabulations are executed in response to commas, semi-colons, and TAB functions until the first numeric expression is reached. Then, a scan of the format string is begun and all characters other than format control characters in the format string are printed until the first format field is reached. The value of the numeric expression is then printed in the format specified by the format field. Next, all non-format characters in the format string are printed until either the end of the string or another format field is encountered. Format fields are separated by spaces. The scan is then resumed in the expression list, printing until the next numeric expression is reached. In this way, scans of the expression list and of the format string alternate until the expression list is exhausted. When the format string is exhausted, it is used over again starting at the beginning.

3. Table 3-1 lists and describes the types of format fields which may be used in format strings.

EXAMPLES:

```
10 DIM A$[10],B$[30]
20 LET A$="###.##"
30 PRINT USING A$;"ANSWER=";1.50*4
40 PRINT USING A$;8.006,300;TAB(40);C
50 LET B$= "+++## $$$.### -$#,###.##"
60 PRINT USING B$;7.6,5.4,-8500
70 PRINT USING B$ [15];X;Y
80 PRINT USING B$;X"TIMES"Y"="X*Y
90 LET B$="#.###^^^^ $****#.##"
100 PRINT USING B$;15360000;23.469
110 PRINT USING "####.##";X
```

If C = 20.5, X = 1000, and Y = 9.999, the output of these lines would be:

```
ANSWER=  6.00
  8.00              300.00                    20.50
  + 7                $5.400     -$8,500.00
 $1,000.00 $      9.99
+1000 TIMES $9.999 = $9,999.00
1.536E+07 $***23.46
1000.00
```

```
5  OPEN #1,"$LPT1"
10 INPUT A
15 PRINT #1;"THE ENTRY IS "A
20 PRINT #1;
30 PRINT #1; USING "$$#,###,###";A
40 PRINT #1; USING "$$*,***,***";A
50 GOTO 10
```

The output of this program to a printer is shown below.

```
THE ENTRY IS  2

 $         2
 $ ***  ***2
THE ENTRY IS  20000

 $    20,000
 $ **20,000
THE ENTRY IS  2222220

 $2,222,220
 $2,222,220
THE ENTRY IS  222

 $       222
 $ **** 222
THE ENTRY IS  222222

 $  222,222
 $* 222,222
```

TABLE 3-1. FORMAT FIELDS FOR PRINT USING

Format Field	Description
####	For each # in the format field, a digit (0-9) or blank (space) is substituted. Integers are right justified with leading blanks; a single zero is printed in the last position if the value is zero. The final zero digit may be suppressed by using "----" instead of "####" (see below). Only integers are represented; no decimal point or fractional portion is printed, and the sign is ignored if only #'s are given. If the data is too large, an asterisk is printed in each position.
###.##	Prints a decimal point where indicated. Digit positions (#) following the decimal point are filled; no blanks are left in these positions. If the fractional portion is too long, it is truncated to fit the format. Leading zeros in the integer portion are replaced by blanks except for a single leading zero preceding a decimal point.
Exponent Indicator (^)	Four consecutive carets (^^^^) indicate an exponent field and will be filled by E±nn where nn is two digits showing the exponentiation.
Signs (+,-,++,--)	A fixed sign (+ or -) may appear as the first symbol of a format field to indicate the following: + Outputs "+" if value is positive, "-" if negative. - Outputs " " if value is positive, "-" if negative. A floating sign (++... or --...) may appear as the first two or more symbols in the format field. Positions occupied by the second and subsequent signs can be used for numeric positions in the data, and the sign is printed immediately preceding the data. Comma separators may be used within a floating sign field, but the sign will not float into a comma position.

TABLE 3-1. FORMAT FIELDS FOR PRINT USING (Cont)

Format Field	Description
Fixed and Floating dollar sign ($)	A fixed $ may appear as the first or second character in the format field, causing a $ to be printed in that character position. The $ may appear as the second character if it is preceded by a fixed sign. A floating dollar sign ($$...) consists of at least two $ symbols beginning at either the first or second character position in the numeric field of the format string. It causes a $ to be placed in the character position immediately preceding the first digit. If the floating $ begins in the second character position, it is preceded by a fixed sign. Only one floating character (sign or $) is permitted in a given field. Comma separators may be used within a floating $ field, but the $ will not float into a comma position.
Separator (,)	The separator (,) places a comma in the position indicated except where leading zeroes (blanks) occur.
Asterisk (*)	The asterisk (*) specifies that asterisks should be printed in all leading positions which would otherwise print as blanks. Leading asterisks are commonly used for protection, as when printing checks. Comma separators may be used within an asterisk field, but blanks (not asterisks) will be printed in the comma positions.

PRINT

KEYBOARD: NO
PROGRAM: YES

SYNTAX: PRINT #channel{,record{,item{,delay}}};expression list{;}

EFFECT: Prints text, numbers, and computational results to a data file or to a peripheral device.

NOTES:

1. When the PRINT # statement is used, all output is in the form of an ASCII string identical to the string that would be printed on the user's terminal if an ordinary PRINT statement were used, but the string is output instead to whatever file or device is open on the specified channel. If nothing is open on the channel, or the specified channel number is greater than the number of channels on the system, then the output defaults to the user's terminal. This allows the destination for all output to be selected when the program is run.

2. The final semicolon at the end of the expression list may be included to suppress the output of RETURN/LINE FEED.

3. If printing is directed to a formatted data file, output must be to an ASCII string item within the record. If printing is directed to a peripheral device, the device must be capable of accepting an ASCII string; such devices include line printers and paper tape punches.

4. The record number may be omitted or -1 for sequential access, or may be -2 to reference the last record which was accessed, as described in Section 2.3.3.

5. When information is printed to any data file, the record is always left locked since the semicolon has special syntactical meaning in a PRINT statement. A WRITE #c;; statement should be used to unlock the record unless another transfer on the same channel will soon follow the PRINT# statement.

6. If the expression list includes terminal control codes such as 'CS' or "@5,10" and the output is directed to a file, the internal representation of each control code is written to the file.

7. Refer to Section 2 for examples of how to use PRINT# for each file type.

EXAMPLES:

```
10 OPEN #1, "EXAMPLE"
20 READ A,B,C,R,X,Y,E,D
30 PRINT #1;A
40 PRINT #1;6*A,B, SQR (B)+C
50 PRINT #1;"THE '' BEST'' ANSWER IS"R
60 PRINT #1;"THE SUM OF"X"AND"Y"IS"X+Y
70 PRINT #1;
80 PRINT #1;E; TAB (20);"*"
90 PRINT #1; EXP (D+ SQR (X))
100 DATA 10, 20, 30, 40, 50, 60, 70, 80
```

The output from this program to a file would be:

```
10
60                20              34.472135954999
THE "BEST" ANSWER IS 40
THE SUM OF 50 AND 60 IS 110

70                    *
6.5235543368236E+37
```

NOTES (Continued)

8. The delay may be included to generate error 123 when a program is paused longer than the specified period of time because a record or device is locked. The delay must be specified in tenths of a second. Setting the delay to -1 allows an unlimited delay period; setting it to 0 specifies no delay and no I/O retries. Note that the delay specifies the maximum amount of time to be spent retrying input or output. For example, a delay equal to 600 (60 seconds) allows 200 retries, given a .3 second delay between retries.

9. The item has special meaning when open file maintenance is in effect.

PRINT # USING

KEYBOARD: NO
PROGRAM: YES

SYNTAX: PRINT #channel{,record{,item{,delay}}};USING format string;expr list{;}

EFFECT: Prints text, numbers, and computational results, using a format string, to a data file or to a peripheral device.

NOTES:

1. This statement combines the features of the PRINT USING statement with the facilities of the PRINT# statement. All output is in the form of an ASCII string identical to the string that would be printed on the user's terminal if an ordinary PRINT USING statement were used, but the string goes instead to whatever file or device is open on the specified channel. The format fields which may be used are shown in Table 3-1, which follows the PRINT...USING statement. If nothing is open on the channel, or if the specified channel number is greater than the number of channels on the system, then the output defaults to the user's terminal. This allows the destination for all output to be selected when the program is run.

2. If printing is directed to a formatted data file, the selected item must be an ASCII string. If printing is directed to a peripheral device, the device must be capable of accepting an ASCII string; such devices include line printers and paper tape punches.

3. The record number may be omitted for sequential access, or may be -1 for sequential access or may be -2 to reference the last record which was accessed, as described in Section 2.3.3.

4. When output is directed to any data file, the record is always left locked since the semi-colon has special syntactical meaning in a PRINT statement. A WRITE #c;; statement should be used to unlock the record unless another transfer on the same channel will soon follow the PRINT# statement.

5. If the expression list includes terminal control codes such as 'CS' or "@5,10" and the output is directed to a file, the internal representation of each control code is written to the file.

EXAMPLES:

```
10 OPEN #1, "EXAMPLE"
20 DIM A$[10],B$[30]
30 READ C,X,Y
40 LET A$="###.##"
50 PRINT #1; USING A$;"ANSWER=";1.5*4
60 PRINT #1; USING A$; 8.006, 300; TAB (40) ;C
70 LET B$="++## $$$.### -$#,###.##"
80 PRINT #1; USING B$; 7.6, 5.4, -8500
90 PRINT #1; USING B$; (15);X;Y
100 PRINT #1; USING B$;X"TIMES"Y"="X*Y
110 LET B$="#.###^^^^ $****#.##"
120 PRINT #1; USING B$; 15360000; 23.469
130 PRINT #1; USING "#####.##";X
140 DATA 20.5, 1000, 9.999
```

The output from this program to a file would be:

```
ANSWER= 6.00
  8.00              300.00                    20.50
  + 7                $5.400      -$8,500.00
  +15 $**.***        $    9.99
+1000 TIMES $9.999 = $9,999.00
1.536E+07 $***23.46
1000.00
```

RANDOM

KEYBOARD: YES
PROGRAM: YES

SYNTAX: RANDOM numeric expression

EFFECT: The RANDOM statement allows the user to exercise control over the random number sequence generated by the RND function.

NOTES:

1. The RANDOM statement solves the following problems, which are common to most programs using random numbers:

   a. The program may be difficult to debug since each run produces different results.

   b. Successive runs of a debugged program may not behave independently if the "random" numbers are from a single pseudo-random sequence.

2. The RANDOM statement is often the first statement in a program which uses the RND function. The use of a non-zero expression, as in the first example, causes a certain sequence of pseudo-random numbers to be generated. Different non-zero expressions initiate different sequences, but each RANDOM statement with the same non-zero value initiates the same sequence.

3. Execution of a RANDOM statement with a zero value expression, as in the second example, causes the system clock to be used to initiate the random number sequence. Since the system clock changes each tenth second, the random number sequence which follows a RANDOM 0 statement is unpredictable.

4. For best results when using the RND function, the following procedure is recommended:

   a. Include a RANDOM statement with a non-zero expression at the beginning of the program while debugging.

   b. Once the program has been tested, change the expression in the RANDOM statement to zero.

EXAMPLES:

```
10 RANDOM 2

5 RANDOM 0

100 INPUT "SEED: "S
110 PRINT
120 RANDOM -S
200 INPUT "NUMBER OF VALUES: "N
210 PRINT
300 INPUT "MODULO: "M
310 PRINT
500 FOR I=1 TO N
510  LET R= RND (65536)
520  LET X= INT (R- INT(R/M)*M)
530  LET R1= INT (R+ 1E-06) !correct rounding error
540  PRINT R1;X;" / ";
550 NEXT I
```

NOTES (Continued)

5. The seed value set by RANDOM and numbers used in computation are 16-bit binary numbers.

   The sign of the value specified in the RANDOM statement affects the setting. If the value is positive, the four most significant decimal digits are placed in packed decimal notation. These four digits are placed in a 16-bit word which is interpreted as a 16-bit binary number when computing the pseudo-random number sequence. In effect, there are 9999 unique initial values. For example, the four most significant digits of RANDOM (12) are 1200, which is represented by 11000 octal in packed BCD. If the value is negative, the internal 16-bit value is set to the absolute value of the expression in binary. There are 65535 (2^16-1) unique initial values. This allows the user to pause the pseudo-random sequences, thus allowing two processes to maintain separate pseudo-random sequences.

   The program above shows how this may be accomplished. Given a seed value and a modulus, the example outputs N value pairs. The pairs are printed "seed value/". For example, if a seed of the fifth number is input for the seed value with the same modulus, the sequence of values produced by the second run will match those of the first, starting with the sixth output. Line 530 was required because of a rounding error when converting the 16-bit internal value.

READ

KEYBOARD: YES
PROGRAM: YES

SYNTAX: READ numeric variable list

where each element of the numeric variable list except the first is preceded by a comma.

EFFECT: The READ statement reads numbers from DATA statements and assigns the values to the corresponding specified variables.

NOTES:

1. The data is read in sequence from the first to the last DATA statement and from left to right within each DATA statement. The system initially sets a pointer to the first item of data. As the READ statements request each data item, the pointer is moved to the next data item. The RESTOR statement may be used to reset the pointer.

2. String variables may not be included in READ statements because strings may not be entered in DATA statements.

3. A READ statement may be executed from the keyboard if one or more DATA statements preceded by line numbers have been entered. If all the data in the DATA statements has been READ, a RESTOR statement may be used to set the pointer to the first data element.

EXAMPLES:

```
10 FOR J = 1 TO 4
20  READ Y
30  PRINT "THE SQUARE ROOT OF "Y"IS"SQR(Y)
40 NEXT J
50 DATA 2, 3.7, 94.61, 2.4E-03
60 READ C,Z
70 LET E=C+Z
80 PRINT E
90 DATA 12, -32.4, 9999, 4E-16
```

In line 50 above, 2, 3.7, 94.61 and 2.4E-03 are stored as two-word precision numbers. Variable E is a two-word variable.

The output of this program would be:

```
THE SQUARE ROOT OF  2 IS 1.414213562373
THE SQUARE ROOT OF  3.7 IS 1.9235384061671
THE SQUARE ROOT OF  94.61 IS 9.7267671916212
THE SQUARE ROOT OF  2.4E-03 IS 4.8989794855663E-02
 -20.4
```

READ

KEYBOARD: NO
PROGRAM: YES

SYNTAX: READ #channel{,record{,item{,delay}}};variable list{;}

where each element of the variable list except the first is preceded by a comma.

EFFECT: Reads item values from a data file or from a peripheral device into the variables listed.

NOTES:

1. The file or device to be accessed must have been previously opened on the channel specified by the channel number expression. The record number from which data are to be read may also be specified. A starting item number may be given if desired; otherwise, item zero is assumed.
2. The variables in the variable list are set to the values contained in the specified record of the specified file, starting with the specified item number or with item zero if no item is specified. The data in the file is not affected. The item number field is used as a byte offset for text or contiguous files (refer to Sections 2.3.2, 2.5 and 2.6).
3. In a formatted file, only sequential items of a single record may be read by each READ# statement, and an error results if a variable type does not match the item type in the file.
4. A semi-colon may be included at the end of the statement (as in the second example) to leave the record unlocked.
5. Numeric expressions are allowed in the file address (channel, record, and item numbers), delay and subscripts, but an item value from the file can not be read into an expression.
6. The record number may be omitted or -1 for sequential access or may be -2 to reference the last record which was accessed, as described in Section 2.3.3.
7. When reading into a string variable, a string terminator (zero byte) is stored after the last character read unless two subscripts are given on the string variable and there is enough data to fill the area specified by the subscripts. The rules for string assignment are not followed; the destination string is not closed up if the source string is too short, and overlaying a terminator byte has no effect.

EXAMPLES:

```
240 READ #2,6;D,W$,K[7,A-2]

415 READ #C[4]+1,R8,5;F$[4],J,J;
```

In line 240, item zero of record six of the file open on channel two is read into variable D, item one (which must be a string) is read into string variable W$, and item two is read into the element of array K at row seven, column A-2. The record is locked since there is no final semi-colon.

In line 415, the channel number and record number are given by the expressions C[4]+1 and R8, respectively. The string variable F$ is loaded from item five of that record starting at character position four in F$; characters one through three of F$ are not affected. Variable J will be set to the value in item six, but this value is replaced immediately by the value of item seven. This technique may be used if the value of item six is of no immediate interest. The final semi-colon unlocks the record.

NOTES: (Continued)

8. The delay may be included to generate error 123 when a program is paused longer than the specified period of time because a record or device is locked. The delay must be specified in tenths of a second. Setting the delay to -1 allows an unlimited delay period; setting it to 0 specifies no delay and no I/O retries. Note that the delay specifies the maximum amount of time to be spent retrying input or output. For example, a delay equal to 600 (60 seconds) allows 200 retries, given a .3 second delay between retries.

9. Refer to Section 2 for examples of how to use READ# for each file type.

10. The item has special meaning when open file maintenance is in effect.

REM

KEYBOARD: YES
PROGRAM: YES

SYNTAX: REM series of printable characters

EFFECT: The REM statement allows the insertion of a remark into a program.

NOTES:

1. REM lines are saved as part of the program. They appear when the program is listed, but they are not executed.

2. Comments may be appended to the end of any line by entering an exclamation mark and comment following the statements on the line.

EXAMPLES:

```
10 REM THIS PROGRAM ADDS NUMBERS
20 REM "#$%&'()
100 LET A=100 !set A to initial value
200 PRINT S !print the total
```

RENUMBER

KEYBOARD: YES
PROGRAM: NO

SYNTAX: {new beginning line no.} RENUMBER {increment}

EFFECT: Renumbers an entire program's line numbers.

NOTES:

1. All line numbers referenced in the program (such as in GOTO and GOSUB statements) are adjusted so that they remain pointing to the same statements.

2. The program is renumbered so that it will start with the specified beginning line number and subsequent lines are increased by the specified increment. If the increment is omitted, a value of 10 is assumed. If the line number is omitted, a value equal to the increment is assumed.

3. If a statement references a line number that does not exist, then the line number will be changed to zero and an error message will be printed giving the old line number of the statement where the error occurred. Because the error message gives the old line number, POINT 4 recommends SAVE-ing or DUMPing the program prior to renumbering.

4. CAUTION! Do not press <ESC> while renumbering, or the entire program will be lost.

EXAMPLES:

```
# BASIC SAMPLE
LIST
10 DATA 1,2,3,4,5,99
20 LET C=0
30 LET D=0
40 READ X
50 IF X=99 THEN 90
60 LET C=C+X
70 LET D=D+X
80 GOTO 40
90 PRINT C,D,C/D
100 END
100 RENUMBER 10
LIST
100 DATA 1,2,3,4,5,99
110 LET C=0
120 LET D=0
130 READ X
140 IF X=99 THEN 180
150 LET C=C+X
160 LET D=D+X
170 GOTO 130
180 PRINT C,D,C/D
190 END
RENUMBER 100
LIST
100 DATA 1,2,3,4,5,99
200 LET C=0
300 LET D=0
400 READ X
500 IF X=99 THEN 900
600 LET C=C+X
700 LET D=D+X
800 GOTO 400
900 PRINT C,D,C/D
1000 END
```

RESTOR

KEYBOARD: YES
PROGRAM: YES

SYNTAX: RESTOR

EFFECT: Resets the data pointer to the first item of data in a DATA statement, making it possible for the data to be re-read.

NOTES:

1. This statement may be used in keyboard mode to affect keyboard execution of the READ or MAT READ statements.

EXAMPLES:

```
10 FOR I=1 TO 4
20   READ X,Y
30   INPUT Z
40   PRINT X"+"Z"divided by "Y"="(X+Z)/Y
50   RESTOR
60 NEXT I
70 DATA 150,20,900,30
```

Line 50 in the example program resets the data pointer so that the same values are read for X and Y each time through the loop.

If the user entered 10, 20, 30 and 40 at the respective input prompts, the output from this program would be:

```
150 + 10 divided by 20 = 8
150 + 20 divided by 20 = 8.5
150 + 30 divided by 20 = 9
150 + 40 divided by 20 = 9.5
```

RETURN

KEYBOARD: NO
PROGRAM: YES

SYNTAX: RETURN {numeric expression}

EFFECT: Transfers control (from a subroutine) to the statement immediately following the matching GOSUB statement (in the main program) which originally transferred control.

NOTES:

1. Every subroutine must be exited using a RETURN statement. A RETURN statement may be used at any desired exit point in a subroutine, and there may be as many RETURN statements as needed in each subroutine.

2. A subroutine that has been entered with a GOSUB can itself contain a GOSUB statement. This nesting process can be carried out to eight levels. Each RETURN is to the previous level, but a RETURN statement cannot be executed without the previous execution of a GOSUB statement.

3. If any expression is included in the RETURN statement, then its integer value specifies transferring control to a number of statements forward (or backward if the expression is negative) from the normal return point. For example: RETURN 0 is the same as RETURN; RETURN +2 skips two statements and returns to the third statement following the GOSUB; and RETURN -1 returns to the GOSUB statement itself.

EXAMPLES:

```
10 DIM R$[10]
100 INPUT "Continue? (Enter YES or NO) "R$
110 PRINT
120 GOSUB 1000 !Check whether YES or NO was entered.
130 GOTO 500
140 GOTO 600
150 PRINT "You must enter YES or NO!"
160 GOTO 100
500 REM Perform the YES alternative
510 PRINT "The user entered YES"
520 STOP
600 REM Perform the NO alternative
610 PRINT "The user entered NO"
620 STOP
1000 REM Routine to check for YES or NO
1010 IF R$="YES" RETURN !return to next statement if YES
1020 IF R$="NO" RETURN 1 !return and skip next statement if NO
1030 RETURN 2 !return and skip 2 statements if neither
RUN
Continue? (Enter YES or NO) Maybe
You must enter YES or NO!
Continue? (Enter YES or NO) Yes
The user entered YES

STOP AT 520
RUN
Continue? (Enter YES or NO) NO
The user entered NO

STOP at 620
```

This program uses a subroutine to check whether the input string R$ is equal to "YES", "NO" or neither. If R$ is equal to "YES", the subroutine returns to the next statement following the GOSUB, which is "GOTO 500".

If R$ is equal to "NO", the subroutine returns to the program, skips one statement, and continues execution with the following statement, which is "GOTO 600".

If R$ is not equal to "YES" or "NO", the subroutine returns to the program, skips two statements, and continues execution with the following statement, which is "PRINT "You must enter YES or NO!""

RUN

KEYBOARD: YES
PROGRAM: NO

SYNTAX: {line number} RUN

EFFECT: Executes a BASIC program.

NOTES:

1. When the RUN command is given, the program in the user's active file is executed starting with the specified line number. If no line number is specified, execution begins with the lowest line number. All variables are initially assumed to be zero, all user functions are assumed undefined, and all arrays and strings assumed dimensionless until a statement is encountered to provide the necessary information.

2. If a line number is specified, the variables, user-defined functions, arrays, and strings remain as they were at the time the last run was stopped.

3. A run may be aborted at any time by pressing <ESC> unless error trapping is enabled (IF ERR 0 or IF ERR 1). If error trapping is enabled, the run may be aborted by pressing <CTRL-Y> followed by <ESC> (unless the program is protected against it).

4. A saved program may also be run using the IRIS processor RUN by giving the following command at the IRIS system prompt

   RUN Filename

   or simply

   Filename

   which brings a copy of the program identified by the Filename into the active file (if it is not read protected) and immediately begins running the program. This has the same effect as entering RUN within BASIC.

EXAMPLE:

```
10 READ X
20 IF X=99 GOTO 80
30 PRINT X"SQUARED="X*X
40 PRINT "THE SQUARE ROOT OF "X"IS" SQR(X)
50 PRINT
60 DATA 4, 9, 16, 25, 625, 99
70 GOTO 10
80 PRINT "ALL NUMBERS HAVE BEEN PROCESSED"
90 END
RUN

 4 SQUARED = 16
THE SQUARE ROOT OF 4 IS 2

 9 SQUARED = 81
THE SQUARE ROOT OF 9 IS 3

 16 SQUARED = 256
THE SQUARE ROOT OF 16 IS 4

 25 SQUARED = 625
THE SQUARE ROOT OF 25 IS 5

 625 SQUARED = 390625
THE SQUARE ROOT OF 625 IS 25

ALL NUMBERS HAVE BEEN PROCESSED
```

SEARCH

KEYBOARD: NO
PROGRAM: YES

SYNTAX: SEARCH #channel,mode,directory expr; K$, R, S

where

channel - channel number expression giving the number of the channel on which the appropriate file is open

mode - search mode expression; the function of each mode is summarized in Table 3-2

directory expr - expression giving the number of the appropriate directory associated with the file or which may be set to defined values to provide miscellaneous directory information, as defined in Table 3-2

K$ - any string variable which contains the appropriate key

R - any variable which will contain or receive the record number of the key being acted upon

S - a variable which receives error status as shown in note #2 or which may be set to defined values to provide miscellaneous search functions, as defined in Table 3-2

EFFECT: Used to search for and manipulate keys within indexed file and polyfile directories and to initialize and provide information about indexed files and polyfiles.

NOTES:

1. The SEARCH statement is described in detail in Section 2.7.5. The information provided here is a summary only. Read Section 2.7.5 to gain familiarity with the SEARCH statement.

(Discussion of this statement is continued.)

NOTES: (Continued)

2. S may receive a status value as defined below:

0	No error, operation was successful
1	Operation was not successful
2	End of directory (when inserting a key, indicates directory is full)
3	End of data (no free records available)
4	File has no index
5	Undetermined error (usually incorrect use of file) or, for polyfiles, file structure error
6	Directory number not in sequence
7	File is not contiguous
8	Indexed file or polyfile volume is already indexed
9	The value of the record number (R) is negative or too large
10	Too many directories: for indexed files, the limit is 15 per file; for polyfiles, the limit is 63 per volume/polyfile
11	For indexed files: master directory level exceeds one block For polyfiles: volume not found (possible structure error)
12	For indexed files: directories exceed size of file For polyfiles: volume too small
13	No such directory
14	File not indexed
15	Data volume number is less than pre-existing data volume
16	Data volume map request not consistent with pre-existing volumes
17	Data volume does not have record length matching that of the polyfile
18	Block/record out of range
19	Record was not allocated (already released)
20	Volume has no map

TABLE 3-2. SUMMARY OF SEARCH MODES

Mode	Directory Expression	Status (S)	Effect
0	>0		sets key length equal to the value of R and the number of keys/block = INT[254/(key length+1)] for the specified directory; must specify directories in sequential order, beginning with directory one
	=0		freezes directory configuration to that specified by previous mode zero commands; assumes a number of data records as given in R; sets up internal linkage for all directories
1	>0		returns key length of specified directory in R
	=0	0	returns record number of first real data record (always zero for polyfiles)
		1	returns number of free records in R
		2	returns the number of a free record
		3	releases record R; returns the record to the free chain or bit map, if provided
2			searches the specified directory for a match with K$; if found, returns found key in K$, associated data record number in R and the value zero in S
3			searches the specified directory for the first key whose value logically exceeds K$; if found, returns found key in K$, associated data record number in R and the value zero in S

TABLE 3-2. SUMMARY OF SEARCH MODES (Continued)

Mode	Directory Expression	Status (S)	Effect
4			inserts the key specified by K$, references the key to record number R and sets S to zero unless a key is found which matches K$ or there is insufficient room in the directory
5			deletes the key specified by K$, returns the associated data record number in R and sets S to zero (unless the specified key is not found); for indexed files, use mode one to return free record to chain
7			reorganizes the specified directory so key can be inserted using mode four; include error code in case directory is too full after reorganization to allow insertion; not used on polyfiles

SIGNAL 1

KEYBOARD: NO
PROGRAM: YES

SYNTAX: SIGNAL 1,port number expression,param1,param2

EFFECT: Sends a signal which consists of the integer values of expressions param1 and param2, to the port number given by the value of the port number expression.

NOTES:

1. The signal will be received by the addressee only if the program running on that port executes a SIGNAL 2 (Receive Signal) statement.

2. The signal resides in the signal list until a program at the destination port executes a SIGNAL 2 statement. However, a signal is ignored by the system if there is no user logged on at the destination port. An error occurs if the signal list is full at the time a SIGNAL 1 statement is executed, and the signal is lost. An error also occurs if a signal is set to a non-existent port.

3. To reduce the probability of the list being filled, the system scans the signal list each hour on the hour, and any signal that is more than one hour old is automatically deleted from the list. Also, the appropriate signal is deleted if the user at the destination port logs off.

4. The expressions param1 and param2 must evaluate to values not exceeding ± 32767. Their integer values are then placed in a signal list along with the integer value of the destination port number expression.

EXAMPLES:

```
250 SIGNAL 1,5,61,2140

365 SIGNAL 1,D,R+1,2*I-Q
```

In line 250, the values 61 and 2140 are sent as a signal to port number 5.

In line 365, the integer values of R+1 and 2*I-Q are sent to the port specified by the integer value of D.

See Supplement PS-6-33 for Sig 5, 6

SIGNAL 2

KEYBOARD: NO
PROGRAM: YES

SYNTAX: SIGNAL 2,port number expression,param1 var,param2 var[,delay]

EFFECT: Receives any signal which has been sent to the port on which this statement is executed (see SIGNAL 1).

NOTES:

1. The port number expression is set to the number of the port from which the signal was sent, and variables param1 and param2 are set to the signal values (the values of param1 and param2 from the SIGNAL 1 statement). If there is no signal to be received, the port number expression is set to minus one, and param1 and param2 remain unchanged.

2. A delay may be included when it is desirable to pause and wait for a signal. The value of the delay is specified in tenth-seconds. If a signal is received before the delay runs out, the program is immediately re-activated, and the port number and variables param1 and param2 are set to the signal values. If the delay runs out first, the program is re-activated, the port number is set to minus one, and param1 and param2 remain unchanged.

3. The port number expression and param1 and param2 variables must be simple variables or subscripted variables; expressions are not allowed. However, the delay may be an expression. The maximum value for the delay is 32767 which gives a delay of nearly one hour.

4. A user may send a signal to his or her own program by pressing the BREAK key on the keyboard (use <CTRL-B> if the BREAK key is disabled). When a SIGNAL 2 statement is executed, the port number is set to the user's own port number, and param1 and param2 are both set to zero. The SPC (6) function may be used by the program to determine its own port number.

5. A program may clear all signals addressed to its port by looping on a SIGNAL 2 statement until the value of the port number expression is less than zero.

6. If the addressee is in a "receive signal with time-out" statement, then that user's program is activated and receives the signal immediately.

EXAMPLES:

```
420 SIGNAL 2,P,A,B

610 SIGNAL 2,S,M[2,3],Y,30
```

In line 610, the program pauses for three seconds (30 tenths of a second) or until a signal is received. The port number of the sender (or -1 if no signal received) is put into variable S, and the two signal values are put into M[2,3] and Y.

SIGNAL 3

KEYBOARD: NO
PROGRAM: YES

SYNTAX: SIGNAL 3,delay

EFFECT: Allows a program to pause (defer further execution) for a specified period of time.

NOTES:

1. The delay is specified in tenth-seconds. All inputs (except ESCAPE) and all signals are saved until the delay is exhausted. An output in progress at the time the SIGNAL 3 statement is executed is allowed to finish. The maximum value for the delay is 32767, which gives a delay of nearly one hour.

2. This statement may be used whenever it is desired to pause before executing the next statement in the program. One example of this is a program which is to loop periodically. It also forces the output of the user's output buffer on demand. At the beginning of the pause, output to the terminal is started if the user's output buffer contains any data.

3. If the value of the delay is 0, then an immediate return is made to the next BASIC statement, but printout is initiated if any output is waiting in the user's output buffer.

CAUTION

4. The SIGNAL 3,0 statement must be used with great restraint. The only proper place for a SIGNAL 3,0 is following one or more PRINT statements where the following code will be compute-bound for a significant period of time. This causes the printout to occur while the computation progresses. Use of a SIGNAL 3 statement in cases where the next PRINT would occur within a few seconds may cause excessive swapping and may seriously degrade system throughput.

EXAMPLES:

```
660 SIGNAL 3,100
400 SIGNAL 3,A+42
```

Line 660 delays further program execution for ten seconds.

Line 400 delays further program execution for (A+42)/10 seconds.

SIZE

KEYBOARD: YES
PROGRAM: NO

SYNTAX: SIZE

EFFECT: Displays current size of a BASIC program by printing a message of the form "=x WORDS OUT OF m"

where
x is the current size of the program in decimal

m is the maximum size in decimal for a BASIC program on the system as it is configured

NOTES:

1. If the program has just been run, then the value x includes the program's variable storage space. After exiting to SCOPE with a <CTRL-C> and re-entering BASIC, the value of x represents the size of the program itself.

EXAMPLES:

```
#BASIC SAMPLE
LIST
10 DATA 1,2,3,4,5,99
20 LET C=0
30 LET D=0
40 READ X
50 IF X=99 THEN 90
60 LET C=C+X
70 LET D=D+X
80 GOTO 40
90 PRINT C,D,C/D
100 END
RUN
15 15 1
SIZE = 75 WORDS OUT OF 7774
<CTRL-C>
\
#BASIC SAMPLE
SIZE = 69 WORDS OUT OF 7774
```

STOP

KEYBOARD: YES
PROGRAM: YES

SYNTAX: STOP

EFFECT: Terminates the execution of a program.

NOTES:

1. The END and STOP statements have similar effects, and may be used anywhere in a program to terminate execution of the program. It is not mandatory that the last statement in a program be an END statement.

2. The END statement causes the simple message "READY" to be printed, while a STOP statement causes

   STOP AT line number of STOP statement

   to be printed. Pressing the ESC key aborts a program run at any time and causes a similar STOP message to be printed unless error branching is in effect (see the IF ERR statement).

EXAMPLE:

```
150 STOP
```

WRITE

KEYBOARD: NO
PROGRAM: YES

SYNTAX: WRITE #channel{,record{,item{,delay}}};{expression list}{;}

where the expression list consists of numeric or string expressions with each element except the first preceded by a comma.

EFFECT: Writes the values of the expressions in the expression list into a data file or to a peripheral device. The file or device to be accessed must have been previously opened on the channel specified by the channel number expression. The record number into which data is to be written may also be specified. A starting item number may be given if desired; otherwise, item zero is assumed.

NOTES:

1. The expressions following the first semi-colon are evaluated and the values are written into the specified record of the specified file, starting with the item number specified or starting with item zero if none is specified. Items not addressed in the file are not affected. The item number field is used differently for a text file or a contiguous file.

2. In a formatted file, only sequential items of a single record may be written into by each WRITE# statement, and an error results if a variable type does not match the item type in the file.

3. The final semicolon at the end of the statement may be used to leave the record unlocked.

4. The record number may be omitted or -1 for sequential file access or may be -2 to reference the last record which was accessed, as described in Section 2.3.3.

5. A locked record may be unlocked by a statement of the form

   line number WRITE #c;;

   which unlocks any record that may be locked on the specified channel without writing into the file.

6. Refer to Section 2 for examples of how to use WRITE# for each file type.

EXAMPLES:

```
195 WRITE #4,19;F,Y1+3,"JUNK",D-E

600 WRITE #C-2,2*R,8;0,2^A,M$[4,Q];
```

In line 195, item zero of record 19 of the file open on channel four is set to the value of F, item one of the same record is set to the value of the expression Y1+3, item two (which must be formatted as a string) is set to the string value JUNK, item three is set to the value of D-E, and the record is locked since there is no final semi-colon.

In line 600, the channel number and record number are given by the expressions C-2 and 2*R, respectively. The item number could also be given as an expression if desired. In this example, item eight of the specified record is set to zero, item nine is set to 2 ^ A, and item ten receives characters four through the value of Q of string M$. The final semi-colon unlocks the record.

NOTES: (Continued)

7. The delay may be included to generate error 123 when a program is paused longer than the specified period of time because a record or device is locked. The delay must be specified in tenths of a second. Setting the delay to -1 allows an unlimited delay period; setting it to 0 specifies no delay and no I/O retries. Note that the delay specifies the maximum amount of time to be spent retrying input or output. For example, a delay equal to 600 (60 seconds) allows 200 retries, given a .3 second delay between retries.

8. The record and item numbers have special meaning when open file maintenance is in effect.