REFERENCE MANUAL
FOR BUSINESS BASIC
LEVELS 3 AND 4

BFISD 5085

TABLE OF CONTENTS

LIST OF TABLES

OVERVIEW

This manual describes Basic Four Information Systems Division's (BFISD) Business Basic Language, Levels 3 and 4. These levels are used on BFISD's 1300 series of computers, which includes Models 200, 210, 410, 510, 610 and 730.

Programmers trained in the original Dartmouth University/General Electric Business BASIC language will discover major differences in BFISD's Levels 3 and 4, which provide extended capabilities.

It is recommended that newcomers to Basic Four products familiarize themselves with general information about the operating system, provided in Appendix A of this manual.

Information on the operating system, located in Appendix A, is provided as a convenience only. This manual is a language manual, not a system description.

Because this document contains information on two different release levels, certain portions do not apply to all users. However, the differences between Levels 3 and 4 are relatively few, and sections applying to only one level are clearly marked.

SCOPE

This reference manual is written as a tool for programmers in the everyday use of the systems described above. The explanations in this manual are presented in a simplified manner. All sections are structured to allow quick access of necessary facts for those seeking immediate answers to common questions, such as format or parameter selection.

The manual is directed toward users of Basic Four systems who develop, program and support business applications.

The information in this manual is presented in the following sequence:

- Section 1, "Introduction" - provides an overview of the purpose of the manual. Defines the intended audience, briefly describes the contents, defines style conventions, and lists related publications.

- Section 2, "Features of Business BASIC, Levels 3 and 4" - describes variables, constants and expressions, logical operations and output data formatting.

- Section 3, "Statement Formats" - explains each component of a statement and defines parameters, common parameter abbreviations, and input/output options. Also describes symbols, I/O options, compound statements and input terminators.

- Section 4, "Directives" - lists and describes each directive in alphabetical order.

- Section 5, "Functions" - lists and describes each function in alphabetical order.

- Section 6, "System Variables" - lists and describes each system variable in alphabetical order.

- Section 7, "Input/Output Options" - lists and describes each input/output options in alphabetical order.

- Section 8, "System Options" - lists and describes each system option in alphabetical order.

- Section 9, "Mnemonics" - lists and describes each mnemonic in alphabetical order, providing its applicable levels and devices.

- Section 10, "Disc Organization" - describes the organization of the disc.

- Section 11, "File Structures and Access" - discusses various aspects of Business BASIC files on Level 3 and Level 4 operating systems.

- Section 12, "Error Processing" - Lists each error and describes what the error number and message mean, and the procedures to follow for correction.

- Appendix A describes features of the Business BASIC Operating System as they relate to the BASIC language.

- Appendix B provides a table of internal character codes for use in converting characters from ASCII to hexadecimal and vice-versa.

- Appendix C is an alphabetical summary of the directives, functions, variables, I/O options and system options available in Business BASIC, Levels 3 and 4.

This manual uses standard style conventions established for all Basic Four Information System Division (BFISD) documentation. Symbols used are defined as follows:

Symbols used in the examples in this manual include the following:

{}	=	parameters enclosed in this type of bracket are optional. If these parameters are not entered, the system either does not use them, or sets default values for them. All parameters not appearing in these brackets are required by the system. Do not enter these brackets, only what they contain
()	=	parameters enclosed in parentheses are required. Parentheses are to be entered with the parameters they surround
[]	=	square brackets are to be entered with the parameters that appear within them. Square brackets are only used in the EDIT statement
""	=	parameters enclosed in quotation marks are required. Quotation marks are to be entered with the parameters they surround

NOTE

All of the above parameters are optional when enclosed in {} brackets. For example:

```
{"file ID"}
{(fileno)}
```

PARAMETER ABBREVIATIONS

Many directives use the same parameters, which appear in abbreviated form in the text. These parameters are defined as follows:

argument list	-	a list of one or more variables, constants or expressions
devno	-	the logical unit number of a device
discno	-	the number of a disc
file ID	-	a 1-6 character string (or a string variable containing same) that uniquely identifies a file
fileno	-	the logical unit number of a file
fileno/devno	-	the logical unit number of a file or device.
keysz	-	the size of a key in a keyed file; minimum=2, maximum=56 (if key is greater than 32,767, maximum=54)
logical expr	-	a comparison between variables and/or values, using a relational operator
numeric expr	-	a numeric variable or constant, or an expression containing any combination of both. Can also contain arithmetic operators
prog ID	-	the name of a program
recno	-	the number of records in a file
recsz	-	the size of each record of a file, in bytes
secno	-	sector number
stno	-	statement number
string expr	-	a string variable or literal, or an expression containing a combination of both. May also contain a "+" for concatenation

INPUT TERMINATORS

Input terminators are keys which notify the system that input has ended. The input terminator most commonly used is the CR character produced by pressing the RETURN key. Other field terminators are Control Bars (CB, sometimes called Motor Bars) I, II, III and IV. All operations in this manual are to be entered using the CR key. More information on input terminators can be found in the description of the CTL function in Section 4.

RELATED PUBLICATIONS

The following publications contain other information related to the Levels 3 and/or 4 Operating System:

610/730 OPERATOR GUIDE, BFISD 5042

OPERATOR TRAINING GUIDE, 200/410, BFISD 5045

MAGNETIC TAPE UTILITIES REFERENCE MANUAL, BFISD 5052

LEVEL 4 UTILITIES USER'S GUIDE, BFISD 5084

DATAWORD II OPERATOR'S GUIDE (LEVEL 1.1), BFISD 5065A

DATAWORD II REFERENCE MANUAL (LEVEL 1.2), BFISD 5104

SERIAL DEVICE MANUAL, BFISD 5060

OPERATOR'S GUIDE SYSTEM 610/730, BFISD 5042

SECTION 2 - FEATURES OF BUSINESS BASIC LEVELS 3 AND 4

OVERVIEW

This section discusses various aspects of the Business BASIC Language, Levels 3 and 4, including:

- o Variables, Constants and Expressions
  - -Numbers
  - -Simple Numeric Variables
  - -Subscripted Numeric Variables
  - -Arithmetic Expressions
  - -String Constants
  - -String Variables
  - -Subscripted String Variables
  - -String Expressions
  - -String Comparison
- o Logical Operations
- o Output Data Formatting
  - -Positioning
  - -Numeric Editing
  - -Non-Formatted Printing of Numeric Variables

CONSOLE MODE VS. PROGRAM MODE

The system can be utilized in two ways. First, commands can be entered directly into the system without statement numbers, which causes an immediate execution of the command upon striking of the RETURN or CR key. This type of operation provides immediate response to input. While these commands are being entered, the system is in Console Mode.

The second way to utilize the system is to enter commands with statement numbers. The system then checks the commands for syntactical accuracy, but makes no attempt to execute them. This is the process of creating a program, a series of commands to be executed in a specific order.

Once a program has been created, it can be invoked from Console Mode by use of the START or RUN directives. START is used to transfer enough memory to the user area to run the program; RUN is used when enough memory already exists in the user area.

Most directives can be used in either Program or Console Mode; those that cannot are listed below:

DIRECTIVE	PROGRAM MODE ONLY	CONSOLE MODE ONLY
DEF FN	X	
EDIT		X
EXECUTE	X	
GOSUB	X	
IOLIST	X	
LOAD		X
ON/GOTO	X	
RETURN	X	
SETTRACE		X
TABLE	X	

VARIABLES, CONSTANTS AND EXPRESSIONS

Business BASIC provides for use of numbers, strings, variables and other components of a computer language. These are discussed in the following paragraphs.

NUMBERS

A number is composed of digits and can be preceded by a sign and/or contain a decimal point. Because numbers can get extremely large, Business BASIC also provides another method of display, in which a number can optionally be modified by floating point notation (.1E-10). The number preceding the E is multiplied by 10 to the power following the E.

Example:

```
  3       3.000
003       3
  3.       .3E1
```

are all valid ways to represent the same number.

Numbers can range in magnitude from $-10^{60}+1$ to $10^{60}-1$. Numbers outside this range result in an ERROR 40. The system retains up to 14 significant digits. Integers and decimal places in excess of 14 digits return an ERROR 26.

If statement syntax calls for an integer (whole number) value, and the number used is not an integer, an ERROR 41 results.

SIMPLE NUMERIC VARIABLES

A simple numeric variable is denoted by a letter or a letter followed by a single digit, allowing for up to 286 simple numeric variables. B and Z7 are examples of names for simple numeric variables. A simple numeric variable requires 12 bytes of data area when it is assigned any value. Once assigned a value these bytes cannot be released without clearing the entire data area. A simple numeric variable can contain any valid number. All references to previously unassigned numeric variables yield a value of 0.

SUBSCRIPTED NUMERIC VARIABLES (DIM)

A subscripted numeric variable denotes an element of an array. (An array is a systematic grouping or arrangement.)

Arrays must be defined by use of a DIM statement before they are referenced (see DIM directive, Section 4).

ARITHMETIC EXPRESSIONS

Business BASIC uses common mathematical symbols, numeric variables and numeric constants to form arithmetic expressions. An arithmetic expression can be used wherever a numeric variable is valid, except to the left of an equal (=) sign. A string variable cannot be used in an arithmetic expression unless converted to numeric format (see NUM and ASC FUNCTIONS in Section 4).

Arithmetic expressions are evaluated according to the following hierarchy:

Order	Symbol	Meaning	BASIC	Math
1		Exponentiation	2^2	2^2
2	* and /	Multiply & Divide	2*2 and 2/2	2x2 and 2/2
3	+ and -	Add and subtract	2+2 and 2-2	2+2 and 2-2

If two symbols have the same order of precedence, operations are performed left to right. The order in which operations are performed can be changed by use of parentheses. If a set of parentheses appears within another set of parentheses, the innermost set is evaluated first and evaluation continues outward.

Examples (Note: constants used can be replaced by variables):

Math	BASIC	Result
10+20	10+20	30
10+20x10	10+20*10	210
(10+20)x10	(10+20)*10	300
$\frac{10+20}{10}$	(10+20)/10	3
2^2x3	2^2*3	12
$\frac{2+6}{4} \text{ x } \frac{2+3}{5}$	(2+6)/4*((2+3)/5)	2

STRING CONSTANTS

A string constant can be any length (subject to memory limits of the task) and can be represented in two ways. Characters that can be entered from the keyboard are enclosed in quotation marks (""). Characters that cannot be generated from the keyboard can be represented by their hexadecimal value. Hexadecimal string constants are enclosed in dollar signs. Two hexadecimal characters are required to represent each single character, e.g., 01 (see Appendix B for assigned values).

STRING VARIABLES

A string variable is identified by a single letter, followed by a dollar sign ($), such as A$, B$ or Z$; or by a single letter, followed by a single number and a dollar sign, such as A1$, A2$, A3$ or Z9$.

There is no limit, other than user memory, to the number of characters that can be stored in a string variable.

Example:

```
    A$ = "LOTSOFCHARACTERS"
```

SUBSCRIPTED STRING VARIABLES (DIM)

The DIM statement is used to assign a length and/or a string of the same characters to a string variable. The first parameter is the length of the string and the second parameter is the fill character. If the second parameter is omitted the fill character is a blank. If the second parameter is more than one character long, only the first character is used.

Examples:

```
    0300 DIM B$(5)
```

-B$ is 5 characters in length.

```
    0300 DIM B$(5,"*")
```

-B$ is 5 characters in length and is filled with asterisks (*)

STRING EXPRESSIONS

Business BASIC uses a mathematical symbol (+) with string variables and string constants to form string expressions. The plus sign represents concatenation.

Example:

```
    0010 LET A$="PEANUT"
    0020 LET B$="BUTTER"
    0030 LET C$=A$+B$
    0040 PRINT C$

    >RUN
    PEANUTBUTTER
```

The data area required by the system in execution of statements containing string concatenation is greater than the area normally required for storing the string(s); that is, the system requires data area overhead in handling string expressions.

STRING COMPARISON

When compared, strings of unequal length do not compare as equal. If two strings are equal for the length of the shortest string, then the longer string is considered greater in value.

Example:

```
0100 LET A$="SOME"
0110 LET B$="SOMEMORE"


    -B$ is greater in value
```

LOGICAL OPERATIONS

Statements of a program are executed in ascending statement number sequence. However, the program requires the ability to control the sequence of statement execution based on logical decisions. Thus, a fundamental feature of the Business BASIC language is the ability to alter the instruction execution sequence as a result of testing whether particular relationship exists. Testing by an IF statement determines, for example, if a numeric variable has reached a certain limit, or if the result of an arithmetic operation is within a specified range of values. Test criteria are established by the following relational operators

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

Compound conditions can also be specified with the use of AND and OR:

```
0100 IF A=1 AND B=2 OR C=3 THEN GOTO 0200
```

See the IF directive in Section 4.

OUTPUT DATA FORMATTING

The formatting of output data on a terminal or printer usually consists of two preparatory steps. The first step provides vertical and horizontal positioning as necessary to place an item of data in a specific area of the printed output. The second step provides for numeric value editing as necessary to vertically align a column of numbers (masking) and to add auxiliary characters such as dollar signs and commas.

Vertical and horizontal positioning are provided by the "positioning expression", which is used in association with a single parameter of a PRINT or INPUT statement. Positioning is effective for terminals and printers. The positioning expression precedes the parameter (output expression) as follows:

```
0020 PRINT (0,ERR=0100) @ (horizontal position
          expr {,vertical position expr})
          {,output expr}
```

where:

horizontal position	= a numeric constant, variable or arithmetic expression the value which defines the horizontal position at which the first character of the printed or displayed value is to be placed
vertical position	= a numeric constant, variable or arithmetic expression the value o which defines the line on which input is to be placed (applicable only to video display devices)
output expr	= an expression that defines the value to be printed or displayed (see individual statement descriptions for allowable forms)

The following are valid positioning expressions:

```
0010 LET A$="POSITION"
0020 PRINT @(5,10),A$
0030 LET R=2,B=5
0040 PRINT @(R*5,B+10),A$

>RUN


    POSITION


         POSITION
```

The horizontal position value, an absolute integer, indicates the horizontal character position where the first character of the output is to print or display. A value of zero indicates the first (left most) character position, and higher values indicate positions to the right. The VDT provides 80 (0-79) character positions, and printers provide 132 (0-131) character positions. Only the horizontal position can be specified on a printer.

The vertical position value is also an absolute integer value, but in this case, reference is made to the top line (line 0) of the VDT screen. The VDT has 24 lines (0-23). The vertical position value must not be greater than the number of lines on the display, or no display appears.

Editing of numeric values to be printed or displayed is provided by a form expression which includes a form operator (:) and a format mask (or the name of format mask). The form expression is appended to a parameter as follows:

```
     PRINT output expr:"###,##0.00+"
```

or

```
     PRINT output expr: A$
```

Following are numeric editing options:

output expr	= a numeric variable or an arithmetic expression that defines the value to be printed or displayed.
:	= the form operator
0	= a character that forces the printing of a digit or a zero in the position specified
#	= a character that is replaced by a digit of the expression, but that suppresses the printing of a leading or trailing zero in the specified position when there is no digit
*	= a "fill" character used in place of the first # to cause the printing of an asterisk in each leading zero position and following the data printed
$	= a "floating" character used in place of the first # or 0 to cause the printing of a dollar sign in place of the right most suppressed leading zero
,	= the point at which a comma is inserted if required (optional)
.	= the point at which a decimal point is inserted (optional)

Format masks can also be used in converting numeric data to string data:

```
       LET A$=STR(N:"000")
```

Any one of the optional elements below can be used to indicate the sign of the output value. The sign element can be placed at the beginning or the end of the format mask to establish the position of the output sign character and can be preceded by "B" (the letter) characters to force the insertion of blanks at the positions indicated:

```
  Example:    PRINT -1:"###,##0.00BB-"
                   1.00  -
```

Omission of the sign editing element causes the value to be output as an absolute value. Optional elements include:

```
  (mask)  = outputs the value masked as specified;
            enclosed in parentheses if negative, no
            parentheses if positive.

  +       = outputs + if the value is positive and -
            if the value is negative.

  -       = outputs a blank if the value is positive
            and - if the value is negative.

  DR      = outputs DR if the value is positive and
            CR if the value is negative.

  CR      = outputs two blanks if the value is
            positive and CR if the value is
            negative.
```

If the value of the number to be printed to the left of the decimal point exceeds the mask size, an error results. If there are more significant decimal places to the right of the decimal point than the mask allows, the number is rounded and truncated when output through the mask.

Examples:

```
   >A$="+##,##0.00"
   >A=.05

   >PRINT A:A$
       +0.05

   >PRINT 1000:A$
    +1,000.00

   >A=-50
   >PRINT A:A$
       -50.00

   >PRINT .005:A$
       +0.01
```

Most printing of numeric values is accomplished in a formatted manner. However, Business BASIC provides the ability to output numeric values in a non-formatted or free-form manner.

When a numeric value in a PRINT statement does not have an associated form operator (:), the manner in which the value prints is determined by the arithmetic mode. The number is rounded first according to the precision in effect, then output with a leading sign, if negative, or a blank.

If the exponent is greater than 14 or less than -14, or the program is in floating point mode, the value is printed as a floating point number, consisting of a sign (+ or -), followed by the fractional part of the value (shown as a decimal number with up to 14 positions), followed by the exponent of the value (in the form E+nn).

Examples:

```
    +.2531E+01
    -.17391621E-04
```

The system inserts one blank space before the first positive number prints.

I/O DIRECTIVES AND ALLOWABLE DEVICES

The following table lists the input and output directives available to the programmer, and the files and/or devices which can be specified for each:

File Type/Devices

	Disc Files					
	SERIAL/ INDEXED	DIRECT/ SORT	PROGRAM*	PRINTERS (LP,Pn)	MAG. TAPE UNIT (Mn)	TERMINAL (Tn)
OPEN	YES	YES	YES	YES	YES	YES
CLOSE	YES	YES	YES	YES	YES	YES
LOCK	YES	YES	YES	NO	NO	NO
UNLOCK	YES	YES	YES	NO	NO	NO
EXTRACT	YES	YES	NO	NO	NO	NO
FIND	NO	YES	NO	NO	YES	NO
GET	YES	YES	YES	YES	NO	NO
INPUT	YES	YES	NO	NO	YES	YES
PRINT	YES	YES	NO	YES	YES	YES
PUT	YES	YES	YES	NO	NO	NO
READ	YES	YES	NO	NO	YES	YES
REMOVE	NO	YES	NO	NO	NO	NO
WRITE	YES	YES	NO	YES	YES	YES
LIST	YES	NO	NO	YES	YES	YES
MERGE	YES	NO	NO	NO	YES	YES
FILE	YES	YES	YES	NO	NO	NO

* Program files may be accessed by READ/WRITE if the ISZ option has been used on OPEN. (Caution: this facility is reserved for utility program usage.)

OVERVIEW

Every BASIC program statement contains a statement number, directive and parameter(s). (Console Mode statements do not require a statement number.) This section discusses each part of the BASIC statement.

FORMAT

BASIC statements are in the following format

```
500  PRINT "EXPRESSION"
```

500	PRINT	"EXPRESSION"
Statement Number	Directive	Parameters
A number that uniquely identifies the statement and places it within the program in its proper sequence. Console Mode statements do not require a statement number	The type of operation being requested	The required and/or optional values used in association with the directive to further define the action to be taken Some directives need no parameters

STATEMENT NUMBERS

Each statement in a Business BASIC program begins with a statement number, an integer between 1 and 9999. Statement numbers should be assigned with enough gaps between them to allow insertion of additional statements, if any are needed.

Statements may be entered in any order, and are automatically sorted into ascending order. If a statement is entered without a statement number, it is executed immediately (Console Mode) and does not become part of the program.

If a new statement number is entered without a directive, the existing line with that statement number is deleted. If no statement already exists with that statement number, an ERROR 21 results.

When entering statement numbers, or any other numeric entry, leading zeros need not be entered (except when EDITing; then all leading zeros must be entered).

DIRECTIVES

The directive is the key element of the BASIC statement in that it instructs the system to perform specific operations such as PRINT or READ. Directives can be executed in both Console and Program Modes, unless otherwise noted in the description of the directive. All directives are available to both Levels 3 and 4, unless otherwise noted.

LET and THEN are optional directives and need not be entered (see the LET and IF/THEN directives in Section 4).

PARAMETERS AND I/O OPTIONS

The prime function of parameters is to define the precise steps required to perform the overall operation defined by the directive. The type of information included depends on the directive and on the options available as part of the statement capability. Some directives do not require any parameters.

Abbreviations for parameters are defined in the CONVENTIONS portion of Section 1.

For a detailed explanation of input and output parameters, see INPUT/OUTPUT OPTIONS in Section 7.

COMPOUND STATEMENTS

Multiple statements can be specified on one statement-numbered line. A semicolon is used to specify the continuation of statements on a line to form a compound statement.

Example:

```
1000 LET X=20; LET Z=50; GOSUB 2000
```

The following rules apply to compound statements:

1. Compound statements are acceptable in both Program and Console Modes. In Level 3, however, only the first part of a compound statement is executed when in Console Mode.

2. DEF, TABLE and IOLIST cannot be part of a compound statement.

3. A REMark or ESCAPE statement can appear only as the last part of a compound statement; neither can be followed by a continuation. Portions of a compound statement which follow a REMark or an ESCAPE are treated as REMarks.

4. Statements which transfer control cannot be followed by a compound, but can be followed in an IF statement by an ELSE, which then permits the addition of the following directives:

   GOTO, ON/GOTO, EXIT, EXITTO,
   END, STOP, RELEASE, RETURN,
   RETRY, START, RUN, EXECUTE

5. An ESCAPE check occurs at each semicolon during execution.

6. RETURN causes a return to the next statement in the compound sequence.

7. RETRY re-executes the appropriate statement within a compound sequence.

OVERVIEW

A directive is the key element of the BASIC statement in that it instructs the system to perform specific operations such as PRINT, READ, LOAD, etc.

Directives can be executed in both Console and Program Modes, unless otherwise noted in a directive's description.

All Directives are available to both Levels 3 and 4, unless otherwise noted.

Directives are presented in alphabetical order.

ADD	ADD
ADD R	ADD R
ADD E	ADD E
ADD C (Level 3 only)	ADD C (Level 3 only)
ADD L (Level 3 only)	ADD L (Level 3 only)
ADD S (Level 4 only)	ADD S (Level 4 only)

FORMAT

```
ADD "prog ID" {,ERR=stno}

ADD R "prog ID" {,ERR=stno} {,BNK=bank no.}

ADD E "OSSPOL" {,ERR=stno} {,BNK=bank no.}

ADD C ".CPLR" {,ERR=stno} {,BNK=bank no.}

ADD L ".LSTR" {,ERR=stno} {,BNK=bank no.}

ADD S ".SORT" {,ERR=stno} {BNK=bank no.}
```

DESCRIPTION

The ADD directive is used to add the file I.D. of a program to the dictionary, eliminating the necessity of a directory search during execution of a CALL, RUN or LOAD.

Variations of the ADD directive perform the following functions in addition to adding the program's file I.D. to the Dictionary:

ADDR -	Adds the specified program to memory, where it remains until it is DROPped.
ADDE "OSSPOL" -	Adds the error handler to memory
ADDC ".CPLR" -	Adds the compiler to memory (Level 3 only; the Level 4 compiler is permanently in memory)
ADDL ".LSTR" -	Adds the lister to memory (Level 3 only; the Level 4 lister is permanently in memory)
ADDS ".SORT" -	Adds the SORTSTEP module to memory (Level 4 only)

NOTE

It is not necessary to use ADD on Level 4 because of the directory caching feature. While ADD and DROP can be used on Level 4, the caching feature performs the same operations automatically, ADDing a file, then DROPping it as additional space is required.

When ADD is used, there is no way of determining the number of programs in the dictionary, which can ultimately lead to an ERROR 16, DISC OR PUBLIC PROGRAMMING DIRECTORY IS FULL.

Use of ADD is therefore recommended only on Level 3.

EXAMPLES

```
0100 ADD "SALT",ERR=0200

0100 ADD C ".CPLR",ERR=0200,BNK=2

0100 ADD E "OSSPOL"

0100 ADD L ".LSTR"

0100 ADD R "SENIC"

0100 ADD S ".SORT"
```

FORMAT

```
BEGIN
```

DESCRIPTION

The BEGIN directive resets the system by performing the following functions:

- o Resets the ERR and CTL system variables to zero
- o Resets uncompleted GOSUB and FOR/NEXT loops (clears the stack)
- o Resets precision to 2
- o Clears the user data area
- o Closes all OPEN files and devices

EXAMPLE

```
0020 BEGIN
```

FORMAT

```
CALL "prog ID" {,ERR=stno} {,SIZ=expr}
     {,argument list}
```

where:

argument list	one or more variables or expressions, separated by commas
SIZ= expr	available in Level 4 only, SIZ= is a number between 0 - 32K specifying the space needed for the CALLed program to run

DESCRIPTION

The CALL directive is used to transfer control and pass arguments to another program. Each variable in the argument list is referenced in the CALLed program by the name specified in the corresponding ENTER statement.

A program that is CALLed, but is not in the dictionary, is entered into the dictionary on a temporary basis and loaded into memory in the highest numbered bank with available space. Then, control is passed to it. When that program EXITs, its entry is dropped from the dictionary unless it is in use by another task (on Level 3 systems, it must also be the last Public program loaded in its bank, or it is not DROPped).

If a program that is CALLed is in the dictionary, but is not in memory, the program is temporarily ADDR'd, and control is passed to it.

If a program that is CALLed is both in the dictionary and resident in memory, the program simply has control passed to it.

When a CALLed program ends, control is returned to the statement following the CALL statement in the program originally issuing the CALL.

In Level 3, the CALL directive operates more quickly when the program has been previously ADDed, and quickest when a program has been ADDRed.

Programs using CALL should have provisions for handling execution of an ESCAPE statement, pressing of the ESCAPE key, and the occurrence of an error. If one of these conditions occurs, and the program is not designed to handle it, the CALLed program is EXITed, and the system enters Console Mode.

Arguments passed to a CALLed program can be returned to the CALLing program with or without a change in their values, depending on the manner in which the CALL argument list is used. In Table 4-1, "Y" denotes values which are subject to change upon returning from a CALLed routine, and "N" denotes variables which are used locally by the CALLed program and are not changed when control is returned to the CALLing program:

Table 4-1, CALL/ENTER Directives

CALL Argument	ENTER Argument	CHANGE	ACTION/RESULT
A	A	Y	A in CALLer is used/modified by reference to A in CALLed program
A	B	Y	A in CALLer is used/modified by reference to B in CALLed program
A+n (n=constant or numeric expression)	A	N	A in CALLed Program is set to value of CALLers A plus n. Original A of CALLer is preserved
A$	B$	Y	A$ in CALLer is used/modified by reference to B$ in CALLed program. Original A$ of CALLer can be changed
"XYZ"	C$	N	C$ in CALLed Program is set to "XYZ"
D(1)	E	N	E in CALLed Program is set to value of CALLers DO). CALLers DO) is not changed
D(ALL)	E(ALL)	Y	E(...) in CALLed Program is set to value of each element of CALLers D(...). CALLers D(...) changes each time E(...) changes. This is a special case to make an entire array common

For more information on the CALL directive, see "Public Programming" in Appendix A.

EXAMPLES

```
1000 CALL "MEACAB"

1000 CALL "MEABUS",ERR=2000,A$,B
```

FORMAT

```
CLEAR
```

DESCRIPTION

The CLEAR directive resets the system by performing the same functions as the RESET directive, and clearing the user data area.

Since CLEAR does not CLOSE any open files or devices, it is normally used when initializing a program that is to use files OPENed by a previously executed program.

CLEAR performs the following functions:

o Resets the ERR and CTL system variables to zero

o Resets uncompleted GOSUB and FOR/NEXT loops

o Resets precision to 2

o Clears the user data area

EXAMPLE

```
0020 CLEAR
```

FORMAT

```
CLOSE (fileno/devno {,ERR=stno} {,IND=index expr})
```

where:

index expr	=	used only for magnetic tape to indicate the position of the tape after the CLOSE. The following options are available:

IND=0,2	-	rewinds tape to load point
IND=1	-	rewinds tape to load point and takes tape off-line
IND=9	-	writes 2 file marks on tape, then rewinds tape

NOTE

If CLOSE has an IND=2 and is preceded by a WRITE RECORD, 2 file marks are written on tape. If CLOSE has no IND= and is followed by a WRITE RECORD, 1 file mark is written on tape.

DESCRIPTION

The CLOSE directive releases use of a file or device. CLOSING files and devices immediately after use is recommended, since the total number of open files and devices cannot exceed 7 at any one time on Level 3, or 8 at one time on Level 4.

Files and devices are also closed when a STOP, END or BEGIN directive is executed.

EXAMPLES

```
1200 CLOSE (1)

1200 CLOSE (1,ERR=0150,IND=0)
```

FORMAT

DEF FNx (argument list) = arithmetic expr

DEF FNx$ (argument list) = string expr

where

X	=	a function name that uniquely defines the DEF statement, where x is a letter (A - Z).
argument list	=	a list of variables where the position of each variable is correlated to a corresponding item positioned in the same relative location within the argument list of the statement using the DEF FNx directive

DESCRIPTION

The DEF statement is used to define up to 26 functions in a program. These functions are in addition to the predefined functions which are part of the Business BASIC language (see "FUNCTIONS" in Section 5).

The DEF FNx directive defines an arithmetic operation; the DEF FNx$ directive defines a string expression.

NOTE

FNx and FNx$ cannot be used in the same program; e.g., FNA and FNA$ cannot exist in one program.

Both DEF FN directives can only be used in Program Mode, and neither can be part of a compound statement.

Either DEF FN directive can contain strings and numbers in the argument list. The output (expression is limited to strings (DEF FNx$) or numbers (DEF FNx).

The Format parameters in the argument list are not "dummy" variables used only by the DEF function. They can also be referenced and used elsewhere in the program, though caution should be exercised since they may change when the DEF function is used.

When one of these DEF functions is called, the values of the arguments being passed are moved into the corresponding formal arguments of the DEF. For example:

```
>10 DEF FNS(X)=X*X
>20 LET X=-1
>30 PRINT X,FNS(10),X

>RUN
-1 100 10
```

-note that referencing the function FNS changed the value of its formal argument X, from -1 to 10

There are 26 user-defined functions available per program.

EXAMPLES

DEF FNx -
```
0010 DEF FNA(A,B)=(A+B)/A

0020 LET C=FNA(2,6)
```

- Statement 0020 assigns A=2, B=6 and C=(2+6)/2=4

DEF FNx$ -
```
0010 DEF FNA$(A$,B$)=B$+"-"+A$
1000 LET X$="SIDO",Y$="DOE"
1010 PRINT FNA$(X$,Y$)

>RUN
DOE-SIDO
```

FORMAT

DELETE {stno a} {,} {stno b}

where

stno a	the number of a statement to be removed, or the first in a series of statements to be removed
stno b	the number of the last in a series of statements to be removed

DESCRIPTION

The DELETE directive is used to remove one or more statements from a program. It cannot be used in a CALLed program.

EXAMPLES

DELETE	- removes all statements from the program
DELETE 10	- removes only statement 10 from the program (entering only "10" performs the same task)
DELETE 10,	- removes statement 10 and all following statements
DELETE 10,100	- removes all statements between 10 and 100, inclusive
DELETE ,100	- removes all statements through 100, inclusive

FORMAT	DIM array name (range of first dimension {,range of second {,range of third}})

where:

array name =	name of the numeric array (must be a single letter)
range of dimensions =	the number o.f elements in each dimension (first, second and third). The value of each dimension (subscript) must be an integer

DESCRIPTION

The DIM array statement is used to define an array. An array is a 1-, 2- or 3-dimensional grouping of numeric values, referenced by a common name (A, B,...Z) and the appropriate dimensions (subscripts).

A 1-dimensional array is commonly called a "list"; the statement DIM A(3) defines a list comprised of 4 elements, referenced as follows:

```
A(0), A(1), A(2), A(3)
```

A 2-dimensional array, called a "matrix", is referenced by the name and two subscripts; the statement DIM A(3,3) produces an array of 16 elements:

```
A(0,0), A(0,1), A(0,2), A(0,3),
A(1,0), A(1,1), A(1,2), A(1,3),
A(2,0), A(2,1), A(2,2), A(2,3),
A(3,0), A(3,D , A(3,2), A(3,3)
```

The statement DIM A(3,3,3) produces a 64 element array. When a DIM statement is executed, all elements of the array are set to zero. Previously defined arrays can be set to zero by executing another DIM statement. The area required for the array can be released by DIMing the array to zero. For example:

```
0010 DIM A(0)
```

EXAMPLES

```
0010 DIM A(0)
0010 DIM A(1)
0010 DIM A(2,2,2)
```

Both the simple numeric variable and an array with the same name can exist in the same program without conflict:

```
0200 DIM A(5)
0210 FOR 1=0 TO 5
0215 LET A=37
0220 LET A(I)=I*10; NEXT I
0230 PRINT A(5),A(4),A(3),A(2),A(1),A(0),A

>RUN
 50 40 30 20 10 0 37
```

FORMAT

DIM string variable name (length {,str expr})

where

string variable name	=	name of the string
length	=	length of the string (up to the limit of available user memory). Data area for the string variable is released if the length is 0
string expr	=	the character used to fill the string. If no character is specified, the string is filled with blanks.

DESCRIPTION

The DIM directive is used to assign a string comprised of a single character to a string variable. The character can be repeated within the string; the DIM directive also assigns the length of the string.

When an string is defined, it can be initialized with the fill character specified in the string expression. If no fill character is specified, the string is filled with blanks.

EXAMPLES

```
1200 DIM A$(5)      -assigns 5 blanks to A$

1200 DIM B$(5,"A")  -assigns "AAAAA" to B$

1200 DIM C$(5,"BC") -assigns "BBBBB" to C$
```

FORMAT

```
DIRECT "file ID", keysz, recno, recsz, discno, secno
       {,ERR=stno}
```

where:

keysz	the size of a key in a keyed file; minimum=2, maximum=56 (if key is greater than 32,767, maximum=54)
recno	the maximum number of records for the file (cannot exceed 8,388,608)
recsz	the size, in bytes, of each record in the file (cannot exceed 32,767)
secno	the sector number where the file is to begin

DESCRIPTION

The DIRECT directive is used to define files with records that can be directly accessed through a key. The key, which provides access for both READing and WRITEing the record, is usually made up of a data field itself, such as Employee Number or Customer Name, or a combination of fields. The key is established when the record is initially written into the file. Each key must be unique in order to identify its associated record.

Records of the file can also be accessed sequentially through IND (physical order), or in logically ascending order of the keys.

The Direct file structure is described in Section 5.

NOTE

When a Direct file is defined, the Scatter Index Table and the key area are initialized. Therefore, AN ACCIDENTALLY ERASED DIRECT FILE CANNOT BE RESTORED BY EXECUTING ANOTHER DIRECT STATEMENT. It is recommended that a backup of each file be kept.

EXAMPLE

```
DIRECT "HIT",10,100,50,0,200
```

-defines a DIRECT file named "HIT" with a key size of 10 bytes, 100 records of 50 bytes each at sector 200 of disc 0

FORMAT DISABLE discno

DESCRIPTION The DISABLE directive prevents access to files on the specified disc by making the disc drive unavailable to the entire system.

All files on the specified disc must be CLOSEd before the disc can be DISABLEd. The DISABLEd disc can only be ENABLEd by the task that DISABLEd it.

CAUTION

Disc drives must be DISABLEd before disk packs are removed

EXAMPLE DISABLE 0 - DISABLES disc number 0

FORMAT	DROP "prog ID" {,ERR=stno}
DESCRIPTION	The DROP directive is used to remove a program from the dictionary. In Level 3, the last program ADDRed must be removed first. If it is necessary to remove the program that is second-to-last in the bank, the last program in that bank must first be removed. This is known as the LIFO (Last In, First Out) Rule, and does not apply to Level 4. DROP cannot be used in a program to be CALLed; i.e., a CALLed program cannot DROP itself.
EXAMPLE	1200 DROP "ALINE" - removes "ALINE" from the Dictionary and from memory

FORMAT

```
EDIT stno {C[copy through value]}
     {D[delete through value]}
     {Rtreplace value]} {[insert value]}
```

where

copy through value	=	text in the original statement, after which a change is to occur
delete through value	=	text in the original statement that is to be deleted
replace value	=	text that is to replace existing characters in the original statement, on a character-by-character basis
insert value	=	text to be inserted into the original statement without replacing any of the existing characters

DESCRIPTION

The EDIT directive is used to add, delete or replace any character(s) or string of characters in any statement in a program.

EDIT is available in Console Mode only, except when used as part of an EXECUTE statement.

The "copy" option specifies the character(s) preceding that portion of the statement to be altered. The system scans from left to right when searching for the "copy through" characters. Therefore, text of this field must be unique, unless the copy through characters are the first occurrence of their type in the statement.

For instance, if the EDIT is to take place after the first period (.) in the original statement, the period by itself is sufficient as the copy option. But if there is more than one period in the original statement, and the EDIT is to take place after the second or subsequent period, the contents of the copy option must be unique.

There is, however another method which can be used. The copy option can be repeated to progress through the statement. For example:

```
0200  REM "THE ARK IS FULL.  PLEASE LEAVE"
>EDIT 0200 C[E]C[E]C[E]C[E]R[ USE TH] [E SKIS"]

0200  REM "THE ARK IS FULL.  PLEASE USE THE SKIS"
```

NOTE

For editing purposes, the statement number is part of the text. For example:

```
0020 REM "23"
>EDIT 20 C[2]R[4]

0024 REM "23"
```

The "delete" option is used with the "copy" option to specify the portion of the statement to be deleted. As the system scans the statement from left to right, the unique character before the first character to be deleted is entered as the "copy" option. The last character to be deleted is then entered as the "delete" option, and the system deletes all characters between, including the "delete" option character.

The "replace" option is used to specify the replacement character(s). It replaces characters in the original statement on a character-by-character basis. The "copy" option is often used with the "replace" option to position the changes.

The "insert value" option is used to specify characters or strings which are to be inserted into the original statement, without replacing existing characters.

All characters following the last character to be deleted, added, or replaced are automatically copied without use of the copy option. If a statement number is EDITed., a new statement is added to the program with the new statement number, and the old statement remains unchanged.

EXAMPLES

ORIGINAL STATEMENT: 1200 PRINT(1)"CHANGER"

1. "delete" EDIT:

```
   EDIT 1200 C["] D[H]
```

result:

```
   1200 PRINT(1)"ANGER"
```

2. "replace" EDIT:

```
   EDIT 1200  C[(] R[2]
```

result:

```
   1200 PRINT(2)"ANGER"
```

3. "insert" EDIT:

```
   EDIT 1200 C[)"] [CH]
```

result:

```
   1200 PRINT(2)"CHANGER"
```

Multiple EDITs are possible within a single EDIT statement. For example, the following EDIT is valid

```
ORIGINAL STATEMENT:   0150 PRINT(1)"ABCDEFGHI"

    EDIT command:

       >EDIT 150 C[ ] R[WRITE] C[(] R[2] C[B] R[X]
                 [Y] D[F] [MNO]

    result:

        0150 WRITE (2) "ABXYMNOGHI"
```

FORMAT ENABLE discno

DESCRIPTION The ENABLE directive reactivates a disc drive that was previously DISABLEd or RESERVEd. The disc drive must be ENABLEd by the same task that DISABLEd or RESERVEd it.

EXAMPLE ENABLE 0 - ENABLES disc number 0

END

FORMAT

END

DESCRIPTION

The END directive is used to terminate a program.

END performs the following operations:

- o Resets the program execution counter to the first statement of the program
- o CLOSEs all open files and devices
- o Performs a RESET operation
- o Returns the terminal to Console Mode

The termination point established by the END directive is also used to discontinue MERGE operations. Therefore, END should only be used at the end of a program.

END does not alter the contents of either the user data area, or the user program area.

All Basic Four systems have an AUTO-END feature which automatically ends every program; this makes use of the END statement optional. However, use of END is recommended, and is required when MERGE is used.

NOTE

END in a CALLed program performs an EXIT.

EXAMPLE

```
9999  END
```

FORMAT ENDTRACE

DESCRIPTION The ENDTRACE directive is used to terminate the listing of statements begun by execution of the SETTRACE directive.

EXAMPLES

```
 >ENDTRACE

0200  ENDTRACE
```

FORMAT

ENTER argument list

where:

argument list	=	one or more variable names, separated by commas. Must contain exactly the same number of elements as the variable list of the corresponding CALL in the CALLing program. Also, corresponding variables must be of the same mode (numeric, string or dimensioned array)

NOTE

Only one ENTER directive can be used per CALLed program.

DESCRIPTION

The ENTER directive defines a set of variables in a CALLed program that corresponds to a set of variable names in the argument list of the CALLing program.

ENTER is used for passing arguments (values) from the CALLing program to the CALLed program, and back again.

Arguments passed to the CALLed program can be returned to the CALLing program with or without a change in their values, depending on the manner in which the CALL argument list is used. In Table 4-1, "Y" denotes values which are subject to change upon returning from a CALLed routine, and "N" denotes variables which are used locally by the CALLed program and are not changed when control is returned to the CALLing program:

Table 4-1, CALL/ENTER Directives

CALL Argument	ENTER Argument	CHANGE	ACTION/RESULT
A	A	Y	A in CALLer is used/modified by reference to A in CALLed program
A	B	Y	A in CALLer is used/modified by reference to B in CALLed program
A+n (n=constant or numeric expression)	A	N	A in CALLed Program is set to value of CALLers A plus n. Original A of CALLer is preserved
A$	B$	Y	A$ in CALLer is used/modified by reference to B$ in CALLed program. Original A$ of CALLer can be changed
"XYZ"	C$	N	C$ in CALLed Program is set to "XYZ"
D(1)	E	N	E in CALLed Program is set to value of CALLers DO). CALLers DO) is not changed
D(ALL)	E(ALL)	Y	E(...) in CALLed Program is set to value of each element of CALLers D(...). CALLers D(...) changes each time E(...) changes. This is a special case to make an entire array common

EXAMPLES	1000	ENTER A$, B, C	passes parameters A$, B and C to the CALLed program
	2000	ENTER A(ALL)	passes the entire array of parameters to the CALLed program

CALLING SUBROUTINE

0010	CALL "SUB", 1, 2, 3	note that only three variables are used to call the subroutine

SUBROUTINE CALLED

```
0010  A=-1, B=-2, C=-3, D=-4
0020  ENTER A, B, C, D
```

however, the resultant values of A, B, C and D are:

```
A=1
B=2
C=3
D=-4
```

Although an ERROR 36 occurs at statement 20 due to a variable mismatch, the values passed are entered into the corresponding argument

FORMAT	ERASE "file ID" {,ERR=stno}
DESCRIPTION	The ERASE directive deletes an"entry from the disc directory. Since the file itself is not affected by ERASE, an Indexed or Program file that is accidentally ERASEd can be restored by execution of another INDEXED or PROGRAM statement (providing that area on the disc has not been reused). However, a DIRECT, SORT or SERIAL file cannot be restored in this manner since DIRECT and SORT statements clear the Scatter Index Table and key area upon redefinition; and redefinition of a SERIAL file clears header data, which has the effect of destroying all references to data records (see FILE directive in Section 4).
EXAMPLE	1000 ERASE "AGOOF" - deletes the file "AGOOF" from the disc directory

FORMAT

```
ESCAPE
```

DESCRIPTION

When executed in Program Mode, ESCAPE causes an interruption of the program, lists the ESCAPE statement, and places the terminal in Console Mode. Continuation of the program from this point is accomplished by entering RUN. Strategic placement of ESCAPE directives within a new program permits periodic examination of data, thereby simplifying program debugging.

When executed in Console Mode, ESCAPE causes the system to list the next statement (if any) in line be executed in the currently RUNning program.

ESCAPE can appear in a compound statement, but any statements that follow it are treated as a REMark.

EXAMPLES

```
2000  ESCAPE
```

EXECUTE

FORMAT

```
EXECUTE {stno} string argument
```

where

stno =	the statement number where the EXECUTEd command is to be inserted into the program
string argument =	a string expression that duplicates either a Console Mode command or a line of code from a program

DESCRIPTION

The EXECUTE directive can only be used in Program Mode. It cannot be used in Console Mode, nor can it be invoked in a Public Program.

EXECUTE provides a capability for generating or modifying program statements within a program.

EXECUTE can be used to build statements, and when used within a program, enables commands which are normally available only in Console Mode.

On Level 3, the compiler must be resident or an ERROR 51, "COMPILE OR LIST OPERATION WITHOUT COMPILER/ LISTER", results.

EXAMPLE

EXECUTE can be used to print the values in the variables A1$, A2$, A3$, etc.:

```
0010  FOR X=0 TO 9

0020  EXECUTE "PRINT(1)A"+STR(X)+"$"

0030  NEXT X
```

EXECUTE can also be used to edit other statements in the program:

```
0100  LET X=Q+3

0200  IF X=4 THEN GOTO 400
   .
   .
   .
0400  EXECUTE "EDIT 0100 C[+]R[5]"
```

FORMAT

EXIT {expr}

where:

expr	=	a value, 0-127, to which the error variable of the CALLing task is to be set (upon return to the CALLing program); or ERR if EXIT is used to pass program 'control when an error is encountered

DESCRIPTION

The EXIT directive is used to return control, and optionally pass an error code to the CALLing program.

The first statement executed after an EXIT directive is the statement following the CALL statement in the CALLing program. If the CALL was made from Console Mode, EXIT returns control to Console Mode.

EXIT ERR can be used to EXIT from a CALLed program when an error occurs.

EXAMPLES

```
9999  EXIT

9999  EXIT ERR
```

FORMAT

```
EXITTO stno
```

DESCRIPTION

The EXITTO directive transfers program control to a specified statement number within the program. It is used to exit from a FOR/NEXT loop without completing all the statements in the loop, or to clear the RETURN address from the top of the FOR/GOSUB stack. The top level of the FOR/NEXT/GOSUB stack is cleared, whether it is a NEXT address or a RETURN address.

The statement number referenced by the EXITTO statement must be a constant whole number, not a variable. If the specified statement number does not appear within the program, program control transfer to the next higher statement number that does exist in the program.

EXAMPLE

```
0010 FOR 1=1 TO 10

0020 IF A(I)=B THEN EXITTO 0040
   .
   .
   .
0050 NEXT I
```

In this example, when A(I)=B, control branches to statement 0040, and the top entry is cleared from the FOR/NEXT stack.

FORMAT

```
EXTRACT (fileno {,ERR=stno} {,END=stno} {,DOM=stno}
        {,IND=index value} {,KEY=key value}
        {,TBL=stno} {,SIZ=size}) {argument list}
        {,IOL=stno}
```

where:

argument list	string or numeric variables into which EXTRACTed data is to be inserted

NOTE

A comma is to be inserted before IOL= only when both IOL= follows an argument list

DESCRIPTION

The EXTRACT directive reads fields of data from a file into respective variable fields in the statement.

EXTRACT differs from READ in two ways: first, it prevents other users from accessing the record until another operation is performed on the file; second, it does not advance the record pointer to the next key in the file, but sets the forward pointer to the EXTRACTed record.

If an EXTRACT is used before a WRITE, the WRITE does not require a key; the EXTRACTed record is overwritten, and is released for access by other users.

If the information in a field is not required, an asterisk (*) can be substituted for the variable name to bypass processing of that field. The advantages of skipping fields are speed and a reduction of memory used by the program.

EXAMPLE

```
0300 EXTRACT (1,ERR=2000,KEY=A$)A,B
```

-reads and locks a record, setting record pointer to the EXTRACTed record

FORMAT

```
EXTRACT RECORD (fileno {,ERR=stno} {,END=stno}
              {,DOM=stno} {,IND=index value}
              {,KEY=key value} {,TBL=stno}
              {,SIZ=size}) {string variable}
```

where

string variable	a string variable into which the record is to be read

DESCRIPTION

The EXTRACT RECORD directive reads a full record from a file or device. If the SIZ= option is included, only the size specified is read. All field marks in the record are transferred as data.

EXTRACT RECORD differs from READ RECORD in two ways: first, it prevents other users from accessing the record until another operation is performed on the file; second, it does not advance the record pointer to the next key in the file, but sets the forward pointer to the EXTRACTed record.

If an EXTRACT RECORD is used before a WRITE RECORD, the WRITE RECORD does not require a key; the EXTRACTed record is overwritten, and is released for access by other users.

EXAMPLE

```
0200  EXTRACT RECORD(1,ERR=1000)A$
```

-reads and locks a record, setting the record pointer to the EXTRACTed record

FORMAT

FILE string

where:

string	=	a 20 byte string with the same format as the FID function

DESCRIPTION

The FILE directive can be used to define any file type by placing the parameters of the file into a 20-byte string. This string has the same format as the FID function (see FID function in this section). FILE can also be used to restore a file that has been accidentally ERASEd from the directory.

EXAMPLE

```
0010 OPEN (1,"ADOOR")
0020 LET F$=FID(1)
0030 CLOSE (1)
0040 ERASE "ADOOR"
0060 FILE F$
```

When statement 50 is added to the above program, a DIRECT, SORT or SERIAL file can be redefined without clearing the Scatter Index Table and key area, or the header area:

```
0050 LET F$(10,1)=IOR(F$(10,1),$40$)
```

FORMAT

```
FIND (fileno {,ERR=stno} {,END=stno} {,DOM=stno}
      {,KEY=key value} {,TBL=stno} {,SIZ=size})
      {argument list} {,IOL=stno}
```

where

argument list	=	variable into which fields of the record are to be read

NOTE

A comma is to be inserted before IOL=
only when IOL= follows an argument list
Find is designed to be used with a key

DESCRIPTION

The FIND directive is used to read data from a file into variables. FIND differs from READ and EXTRACT by not updating the key pointer position to the next highest key following a key that is not found. This difference makes FIND faster than READ and EXTRACT when the specified key is _not_ in the file. If the key _is_ in the file, about the same amount of time is required for any of the three directives.

If the information in a field is not required, an asterisk (*) can be substituted for the variable name to bypass processing of that field. The advantages of skipping fields are speed and a reduction of memory used by the program.

EXAMPLE

```
0200 FIND (1,KEY=K$,ERR=0500)A,B$
```

FORMAT

```
FIND RECORD (fileno {,ERR=stno} {,END=stno}
            {,DOM=stno} {,KEY=key value} {,TBL=stno}
            {,SIZ=size}) {argument list}
```

where

argument list	=	variable into which fields of the record are to be read

DESCRIPTION

The FIND RECORD directive is used to read a full record from a Direct file into variables in the same manner as a READ RECORD or EXTRACT RECORD. FIND RECORD, however, does not update the key pointer to the next highest key following a key that is not found. This difference makes FIND RECORD faster than READ RECORD or EXTRACT RECORD if the specified key is not in the file. If the key is in the file, the three directives are approximately equal in speed.

EXAMPLE

```
0200 FIND RECORD(1,KEY=K$,ERR=0500)A$
```

FORMAT FLOATING POINT

DESCRIPTION The FLOATING POINT directive is used to initiate the Floating Point Mode. This mode maintains maximum (14 digit) accuracy while permitting the generation of very large or very small values by using "E" to indicate a power of 10.

Numbers are output in Floating Point notation unless a mask is specified.

EXAMPLE

```
0010 FLOATING POINT
0020 FOR 1=0 TO 5
0030 PRINT 2^I;NEXT I

>RUN

.1E+01
.2E+01
.4E+01
.8E+01
.16E+02
.32E+02
```

FORMAT

```
FOR ctrl variable=start expr TO end expr
    {STEP expr}
```

where

ctrl variable	=	a simple numeric variable, whose value controls the FOR/NEXT loop. When the value of the control variable exceeds that of the end value, the loop is terminated.
start expr	=	a numeric value to which the control variable is set upon execution of the FOR statement
end expr	=	a numeric value. The FOR/NEXT loop is exited when the control value exceeds the end value
STEP expr	=	a numeric value which determines the amount that the control variable is advanced during each execution of the NEXT statement. The step size cannot be 0, but can be negative. If not specified, step size is 1

DESCRIPTION

The FOR/NEXT loop is used as a means for repetition of a series of statements in a program.

When a FOR statement is first executed, the control variable is set equal to the start value. The end value and step value are saved. The statements following the FOR statement are executed in sequential order until the NEXT statement is reached The control variable is then incremented by the step value and compared to the end value.

If the control variable is less than or equal to the end value, control passes to the statement following the FOR statement. This sequence is repeated until the control variable is greater than the end value. Execution then continues with the statement following the NEXT statement.

Except for available memory, there is no limit to the number of FOR/NEXT loops allowable in a program. FOR/NEXT loops can be "nested". However, each NEXT must correspond to its FOR, e.g.:

```
0100 FOR 1=1 TO 5
0110 FOR J=1 TO 5
0120 NEXT I
0130 NEXT J
```

is invalid.

FOR/NEXT loops can also be divided into two groups; one where the series of statements is repeated until the loop is terminated, and the other where the loop terminates before the specified number of executions is complete (see examples below).

EXAMPLES

(These examples are normal FOR/NEXT loops where the series of statements is repeated until the loop is terminated.)

FOR/NEXT loop:

```
0010 FOR 1=1 TO 5
0020 PRINT I,
0030 NEXT I
0040 PRINT "  FINAL VALUE = ",I

>RUN
 1 2 3 4 5  FINAL VALUE = 6
```

Nested FOR/NEXT loop:

```
  0010 FOR I=1 TO 2
  0020 FOR J=1 TO 3
  0030 PRINT 10*I+J,
  0040 NEXT J
  0050 PRINT 'LF'
  0060 NEXT I

  >RUN
   11 12 13
   21 22 23
```

(This example is a loop which terminates before its normal number of executions. Note the use of EXITTO rather than GOTO to escape the loop. This clears the FOR/NEXT loop stack.)

FOR/NEXT loop

```
  0010 REM "PROGRAM TO VERIFY THAT STRING INPUT IS
       NUMERIC
  0020 BEGIN
  0030 INPUT "NUMERIC? - ",A$
  0035 IF A$="END" THEN GOTO 0120
  0040 IF A$=" " THEN LET A$="0"
  0050 LET F$="Y"
  0060 FOR 1=1 TO LEN(A$)
  0070 REM "THE FOLLOWING LINE EXITS TO 100
  0080 IF POS(A$(I,1)="0123456789+- ")=0 THEN LET
       F$="N";EXIT TO 0100
  0090 NEXT I
  0100 IF F$="N" THEN PRINT "INVALID"
  0110 GOTO 0030
  0120 END
```

FORMAT

```
GET discno, secno {,ERR=stno} {,RTY=no. of retries},
    input string variable {,verify string variable}
```

where:

RTY	=	number of RETRYs if the GET is unsuccessful. Can be 0 - 254 (more than 254 is interpreted as 0). If no RTY is specified, the system defaults to 19 retries on removable disc systems, or 27 retries on fixed disc systems.
input string variable	=	predimensioned variable to receive data from the disc
verify string variable	=	optional verification string (must be the same size as input string variable), which performs internal comparison of strings to check for data integrity.

NOTE

GET is not recommended for use in applications programs

The GET directive transfers data from a sector on a disc into a variable.

```
0190 DIM A$(1024),A1$(1024)
0200 GET 0,1096,ERR=0500,RTY=49,A$,A1$
```

GOSUB

FORMAT

GOSUB stno

DESCRIPTION

The GOSUB directive, available in Program Mode only, calls an internal subroutine, transferring program control to the specified statement number. Statements in the subroutine are executed sequentially until a RETURN statement is found. Control then returns to the statement following the GOSUB.

Every subroutine referenced by a GOSUB directive must be ended by a RETURN or EXITTO statement (the EXITTO statement ends a subroutine without returning to the calling point, and clears the top level entry from the RETURN address stack).

EXAMPLE

```
0010  REM "EXAMPLE OF REPORT PROGRAM USING GOSUB"
0020  BEGIN
0030  OPEN (7)"LP"
0040  OPEN (1)"INVENT"
0050  LET P$="####0.00"
0060  GOSUB 1000
0070  READ (1,END=0500,ERR=0600)A,B,C,D
0080  LET L=L+1
0090  IF L>50 THEN GOSUB 1000
0100  PRINT (7)A:P$,g(10),B:P$,@(20),C:P$,@(30),D:P$
0110  GOTO 0070
0500  PRINT "END OF RUN"
0510  STOP
0600  PRINT "ERROR:  ",ERR:"000"," OCCURRED ON READ"
0610  STOP
1000  REM "SUBROUTINE TO PRINT HEADINGS"
1010  LET P=P+1,L=0
1020  PRINT(7)'FF',"ITEM",@(10),"QUANTITY",@(20),
1020: "COST",@(30),"PRICE",@(70),"PAGE",P,'LF'
1030  RETURN
```

GOTO GOTO

FORMAT

```
GOTO  stno
```

DESCRIPTION

The GOTO directive unconditionally transfers program control to the specified statement number. If the specified program number does not exist, the statement with the next higher number is executed.

GOTO can be used in Console Mode (followed by a RUN command) to direct program control to any statement number. This is useful in program debugging.

EXAMPLES

```
0100 OPEN (7)"LPn
0110 LET X=L+1
0120 GOTO 0500
0130 PRINT (7)"THIS"

>GOTO 0130
>RUN
THIS
```

FORMAT

```
IF  logical expr {AND logical expr} {OR logical expr}
    {THEN} statement a {ELSE statement b}
```

where:

logical expr	=	a comparison between variables and/or values, using a relational operator sign
statement a	=	the statement, such as GOTO 0250 or WRITE A$, to execute if the comparison in the logical expression is "true"
statement b	=	the statement, such as GOTO 0275 or WRITE B$, to execute if the comparison in the logical expression is "false"

DESCRIPTION

The IF directive allows conditional execution of BASIC statements based upon the result of a logical comparison between two or more data items.

The logical expression portion of the statement contains two expressions, either string or numeric, separated by a relational operator. The relational operators are:

=	equal to
<	less than
>	greater than
<> or X	not equal to
>= or =>	greater than or equal to
<= or =<	less than or equal to

Some examples of logical expressions are:

```
A=B    LEN(X$)<=16

C>=B   A/B=E
```

Several logical expressions can be evaluated in relation to each other by use of the AND and OR operators. An unlimited number of ANDs and ORs can be used in an IF statement, and they have equal precedence; the system evaluates them from left to right.

Parentheses can be used to change the order of evaluation. The action taken by the IF statement is determined by the "trueness" or "falseness" of the logical expressions.

Example:

```
0010 LET A=1,B=2,C=3
0020 IF A=1 OR B=2 AND C=0 THEN PRINT "20 IS
0020:TRUE"
0030 IF A=1 OR (B=2 AND C=0) THEN PRINT "30 IS
0030:TRUE"

>RUN

 30 IS TRUE
```

Statement 20 only prints if A or B is true, and C is true. Statement 30 prints if A is true, or if B and C are true

The THEN and ELSE clauses of the IF statement are conditionally executed based on the evaluation of the logical expression(s). If the expression(s) are evaluated as "true", the THEN clause is executed. If they are evaluated as "false", the ELSE clause is executed. If no ELSE clause exists, the next statement is executed.

Each THEN or ELSE clause can contain a single or compound BASIC statement. Any BASIC statement is valid, except for DEF, IOLIST, and TABLE.

IF/ELSE commands can be nested into a single statement, provided the IF and ELSE conditions appear in an alternating sequence:

```
    IF logical expr THEN statement

    ELSE IF logical expr THEN

    statement ELSE IF . . .
```

Example:

```
 0030 IF D1>10 THEN LET D2=10-ELSE IF
 0030:D1<8 THEN LET D2=9
```

EXAMPLES

```
0010 IF A=B THEN GOSUB 6000 ELSE GOTO 9999
```

It is not necessary to type the word "THEN" as part of a THEN clause, if another directive is involved (e.g., GOTO, GOSUB, etc.); the system adds it automatically. The system also adds zeros where applicable:

```
 10 IFA=BGOSUB60              -as entered (no
                               spaces, no THEN)
>LIST
0010 IF A=B THEN GOSUB 0060   -as the lister prints
```

FORMAT

```
INDEXED "file ID", recno, recsz, discno, secno
       {,ERR=stno}
```

where:

recno	-	the maximum number of records for the file (cannot exceed 8,388,608)
recsz	-	the size, in bytes, of each record in the file (cannot exceed 32,767)
secno	-	the sector number where the file is to begin

DESCRIPTION

The INDEXED directive defines a file comprised of records located in contiguously numbered sectors. These records can be READ or WRITEn either sequentially or randomly by record number (the first record is number 0).

Records defined in an Indexed file are all the same length. Fields within the records are delineated b» special characters called "field marks", which are inserted by the system.

NOTE

An Indexed file can be expanded to include a greater number of records by ERASEing the file, and then redefining it with a larger number in the recno field. The ERASE operation deletes information from the disc directory but does not alter the data in the area defined for the file. The file can be enlarged only if sufficient disc space exists immediately following the file.

EXAMPLE

```
0130 INDEXED "FINGER",100,50,0,200
```

-creates the Indexed file "FINGER" at sector 200 of disc drive 0 with 100 records of 50 bytes each

FORMAT

```
INPUT {(fileno/devno {,ERR=stno} {,END=stno}
      {,DOM=stno} {,IND=index value} {,KEY=key value}
      {,TBL=stno} {,TIM=time} {,SIZ=size})}
      {@(expr{,expr})} {,string constant} {,mnemonic}
      {,variable} {,IOL=stno}
```

where

@ expr, expr	=	horizontal and vertical positioning of the INPUT statement

NOTE

A comma is inserted before IOL= only when IOL= is used with a variable

DESCRIPTION

The INPUT directive is used for two-way communication between the operator and the program. An INPUT may contain string constants for output to the terminal device. The operator's response goes into the variables included as parameters in the INPUT statement.

If the information in a field is not required, an asterisk (*) can be substituted for the variable name to bypass processing of that field. The advantages of skipping fields are speed and a reduction of memory used by the program.

When the system executes an INPUT statement, a message (if one was specified) appears on the operator's terminal. The system then waits for the operator to respond. The operator enters the response, then presses a field terminator (usually RETURN), and the system stores the data as directed by the statement, and then sets the CTL (control) task variable to a value determined by the type of field terminator used. The following list identifies the available field terminators and the resulting CTL values:

Keys	ASCII Character	Control (CTL) Value
CR or LF	CR or LF (line feed)	0
Control Bar I	FS (field separator)	1
Control Bar II	GS (group separator)	2
Control Bar III	RS (record separator)	3
Control Bar IV	US (unit separator)	4

An INPUT, INPUT RECORD, READ or READ RECORD statement using the SIZ= option sets the CTL value to 5 if the number of characters INPUT or READ corresponds to the SIZ value specified.

The operator selects the key(s) to be pressed based on the directions given, or in accordance with pre-established operating procedures. If the programmer has directed the possible use of any terminator other than RETURN, the INPUT statement can be followed by a statement that selects program branching, depending on the type of terminator entered. The operator can thus be given the ability to determine the course of processing that ensues.

Under normal circumstances all entries typed at the terminal keyboard are received by the system, and are then immediately returned to the terminal for display or printing. However, in some applications (such as when entries must be masked before display), this immediate return of the entry is inhibited and the display results from execution of a subsequent PRINT statement. Inhibition of the immediate display (or printing) of input data is accomplished by using a device number other than zero in the INPUT statement.

The device number used must have been previously assigned to the terminal by means of an OPEN statement. An example of an INPUT statement that inhibits display of a keyboard input follows:

```
0010 LET F$=FID(0)
0020 OPEN (2)F$
0030 INPUT (2,ERR=0030)@(0,10),"ENTER QUANTITY
0030:SOLD-",B
0040 PRINT (0,ERR=0030)@(0,11),B:"00000"
>RUN

ENTER QUANTITY SOLD-
00123
```

An attempt to enter non-numeric variables results in an ERROR 26. This provides an easy method for verifying that data input is numeric.

Example:

```
0010 INPUT (0,ERR=0100)"ANY NUMBER? ",A
0020 PRINT "VALID"
0030 GOTO 0010
0100 PRINT "INVALID"
0110 GOTO 0010
>RUN
ANY NUMBER? 1
VALID
ANY NUMBER? A100
INVALID
```

INPUT VERIFICATION

Business BASIC provides the means to verify the maximum and minimum sizes of strings, the values of strings, and the maximum, minimum and number of decimal places of a numeric within an INPUT statement, as described below. Tests for verification occur from left to right within the parentheses.

Numeric Verification

```
INPUT {(file parameters)} N: ({-} range mask)...
```

where:

range mask	=	is a literal string of digits, with or without a decimal point, which specifies the maximum (inclusive) limit of N
minus sign (-)	=	specifies (if used) that the minimum limit of N is the negative value of the mask, inclusive; if not specified, the minimum is 0

Placement of the decimal point, or absence of it, specifies the maximum number of fractional digits allowed.

Examples:

```
0010 INPUT (0,ERR=0010)A:(249.99)
```

-the acceptable values of A are in the range of 0 through 249.99. Any value in excess of 249.99 or with more than 2 fractional digits generates an ERROR 48.

```
0010 INPUT (0,ERR=0010)A:(-999)
```

-the acceptable values for A are integers in the range of -999 through +999.

String Verification

```
INPUT {(file parameters)} N$: ({branchlist} {,}
      {LEN=Min,Max})
```

where:

branchlist	=	branchlist is one or more items whose syntax is: string literal =stmnt no. (e.g., "END" = 100). Branchlist items are separated by commas. If a true condition is found (i.e., N$ = string literal), statement execution is transferred to the specified statement number
Min Max	=	Min and Max specify the inclusive range of legal lengths for N$. Min must be less than or equal to Max, or an ERROR 20 results

If no branchlist is specified, or if the variable does not match any literal in the branchlist, the LEN= specification is checked. If LEN= is not specified, an ERROR 48 is generated.

An ERROR 48 is also generated if the length of the variable is not within the specified range and the variable does not match any literal in the branchlist (or if there is no branchlist). Otherwise, statement execution continues normally.

Examples:

```
0010 INPUT (0,ERR=0010)"L/N/C",A$:("L"=0200,
     "N"=0300,"C"=0400)
```

-if A$ = "L", program control is transferred to statement 200

-if A$ = "N", program control is transferred to statement 300

-if A$ = "C", program control is transferred to statement 400

-any other value for A$ takes the ERR branch and returns to the INPUT statement

(more)

```
0100 INPUT (0,ERR=0100)"FILE NAME",A$:(LEN=1,6)
```

-if the length of A$ is less than 1 or greater than 6, the ERR branch is taken

```
0050  INPUT "NEXT KEY OR CR",A$:(""=1000,
      LEN=8,10)
```

-if A$ = no entry, program control is transferred to statement 1000

-if the length of A$ is less than 8 or greater than 10, an ERROR 48, "INVALID INPUT", occurs.

INPUT RECORD

FORMAT

```
INPUT RECORD  (fileno/devno {,ERR=stno} {,END=stno}
              {,DOM=stno} {,IND=index value}
              {,KEY=key value} {,SIZ=size})
              {@(expr{,expr})} {string variable}
```

where:

string variable = name of the string into which the record is to be input

DESCRIPTION

The INPUT RECORD directive is used to input a full record from a file without the need to specify what fields comprise the record. Field marks are transferred as data.

INPUT RECORD is similar to the READ RECORD directive and is used in the same way. It inputs one record from a file or device into a string variable. Any field terminators are included in the record as data, and no field terminator is added to the end of the record.

The SIZ= clause must be used with an INPUT RECORD command when input is from the VDT, since a RETURN or Control Bar key is treated as part of the data, rather than as a terminator.

EXAMPLE

```
0010  INPUT RECORD(2,ERR=0100,SIZ=5)A$
```

IOLIST IOLIST

FORMAT

IOLIST argument list {,IOL=stno}

where:

argument list	a list defining data items to be input or output in subsequent I/O statements. The list can contain string variables, string constants, numeric variables, numeric constants, arithmetic expressions, string expressions, at-positions (@), mnemonics, or other IOL references

DESCRIPTION

The IOLIST directive, available in Program Mode only is used to define a set of variables that can be referenced in input and output statements. Use of the IOLIST directive saves both coding space and debugging time.

The list of variables established in the IOLIST directive is referenced by other statements using an IOL= clause. An IOL= clause can also appear in IOLIST statements.

The IOLIST statement cannot be part of a compound statement.

EXAMPLE

```
0050 OPEN (1)"AFILE"
0100 IOLIST A$,B,C$,D$,IOL=0110
0110 IOLIST E,F$,G$
0120 IOLIST A$,B:"###","ABC","05678",IOL=0110
0200 READ (1, KEY=A$)IOL=0100
0250 WRITE (1, KEY=A$)IOL=0120
0260 PRINT 'SB',@(0,1),IOL=0120
```

FORMAT

```
{LET} {numeric variable = numeric expr} {,}
      {string variable = string expr} {,...}
```

DESCRIPTION

The LET directive assigns a value to a variable. The value on the right side of the equal sign is assigned to the variable on the left side of the equal sign. Both sides of the equal sign must be the same data type, numeric or string.

The word LET is optional and need not be entered as part of the statement. The system automatically assumes LET if no other directive is recognizable. More than one LET assignment can be made in one statement by using commas between them. The LET verb occurs only at the start of the assignment list, if at all.

EXAMPLE

```
0010 LET A=2

0010 B=5,Q=2

0010 LET D1=P*Q; IF D1>10 THEN LET D1=12
```

FORMAT

```
LIST {(devno {,ERR=stno} {,TBL=stno}) {stno a} {,}
      {stno b}
```

where:

stno a = the number of the statement to be LISTed or the number of the first statement in a series of statements to be listed

stno b = the number of the last statement in a series of statements to be listed

DESCRIPTION

The LIST directive is used to print, or output on any output device (except MTC & MTR), any statement or any series of statements. The selected statement(s) are accessed from the user program area and are output in statement number sequence. The LISTed information includes statement numbers, directives and all parameters of each statement, including any REMark statement in the series. The LIST directive can be used as a statement in any program except a Public program.

When any statement in a list exceeds 79 characters in length (including the statement number), the portion in excess of 79 characters is listed on the next line. The continued portion of the statement is then preceded by the statement number, followed by a colon(:).

When LISTing to a disc file, the file must be an INDEXed file with at least as many records as there are lines in the program that are to be LISTed.

EXAMPLES

```
>LIST        -lists all statements

>LIST 10     -lists statement 10

>LIST 10,    -lists statement 10 and all following
              statements

>LIST 10,100 -lists statements 10 through 100

>LIST ,100   -list all statements through 100
```

```
0100 LIST (4,ERR=0070)0010,0100
```

-specifies that statements 10 through 100 inclusive are to be listed at device 4. Control transfers to statement number 70 in the case of an error

```
>LOAD "INZONE"

READY
>INDEXED "FINGER",100,80,0,1850
>OPEN (1)"FINGER"
>LIST (1)
```

-this routine sets up the Indexed file "FINGER", opens it, and copies the statements in "INZONE" to "FINGER"

FORMAT

```
LOAD "prog id"
```

DESCRIPTION

The LOAD directive, available in Console Mode only, is used to bring a program into memory.

When a LOAD command is issued, the current program in the user area is deleted, all FOR/NEXT/GOSUB/ SETERR/SETESC return addresses are.cleared, precision is set to 2, and the program is READ into the user area. The program can then be executed or modified. The execution of a LOAD command has no effect on the user data area.

If insufficient program area is available, an ERROR 19 (PROGRAM SIZE) displays. In Level 3, the program area is cleared prior to the attempt to LOAD. In Level 4, the program area is not cleared until it has been determined that the specified program can be LOADed.

Like RUN, LOAD conserves the values of the variables. For example:

```
>LET A=129
>LOAD "PGM"
>PRINT A

129
```

-if the program "PGM" uses A, its A value is set to 129 (unless a BEGIN or CLEAR is executed first)

EXAMPLE

```
>LOAD "INZONE"
```

FORMAT

```
LOCK (fileno {,ERR=stno})
```

DESCRIPTION

The LOCK statement prevents other users from accessing a file. This is especially useful when file is being updated.

A LOCKed file is released by an UNLOCK or CLOSE statement.

EXAMPLE

```
0100 LOCK (1,ERR=0200)
```

FORMAT

```
MERGE (fileno/devno {,ERR=stno} {,IND=value}
      {,TBL=stno})
```

where

value = the index number of the first record in the file which contains the lines of code to be added

DESCRIPTION

The MERGE directive is used to retrieve a program in LIST format from an INDEXED file on disc, or from any other input device (except MTC and MTR), and to add that program to the program currently existing in a user memory area.

The statements of the two programs are merged together. If both programs have a statement with the same statement number, the one in the MERGEing program replaces the existing one.

The addition of a statement with a statement number that does not exist in the current user program, causes that new statement to be inserted in the program in numerical order, according to its statement number. The MERGE Operation is terminated following the MERGEing of an END statement. If no END statement is present in the program being read, an ERROR 21 (STATEMENT NUMBER MISSING) is displayed upon reaching a record in the file that contains no statement number.

MERGE cannot be used in a Public program.

EXAMPLE

Follow these steps to perform a MERGE:

1. LOAD, then LIST the program to be MERGEd ("PGM1"):

```
>LOAD "PGM1"

READY
>LIST
0010 REM "LOADING PGM1"
0020 INPUT A$
0130 PRINT A$
0140 GOTO 0020
1000 END
```

2. OPEN an Indexed file ("TRUNK"), and temporarily store the program to be MERGEd in it in LISTed format:

```
>INDEXED "TRUNK",5,80,0,2096
>OPEN (1)"TRUNK"
>LIST (1)
>END
```

3. LOAD, then LIST the program into which "PGM1" is to be MERGEd ("PGM2"):

```
>LOAD "PGM2"
READY
>LIST
0010 REM "PGM2"
0015 OPEN (1)"BOX"
0030 IF LEN(A$)>3 THEN GOTO 0150
0040 READ (1,ERR=0150,KEY=A$)«
0050 PRINT "VALID"
0150 PRINT "INVALID"
0160 GOTO 0020
```

4. OPEN the Indexed file ("TRUNK"); then enter the MERGE command:

```
>OPEN (1)"TRUNK"
>MERGE (1)
```

5. LIST the combined programs:

```
>LIST
0010 REM "PGM1"
0015 OPEN (1)"BOX"
0020 INPUT A$
0030 IF LEN(A$)>3 THEN GOTO 0150
0040 READ (1,ERR=0150,KEY=A$)*
0050 PRINT "VALID"
0130 PRINT A$
0140 GOTO 0020
0150 PRINT "INVALID"
0160 GOTO 0020
1000 END
```

Statement 10 is listed in both programs, so the one in the MERGEing program survives.

NOTE

No error is signalled (nor is ERR= exit taken) for statements which are invalid in the MERGE file (except those with missing statement numbers).

FORMAT

NEXT control variable

where:

```
   control
   variable = the variable to be incremented (or
              decremented if the step value is
              negative)
```

DESCRIPTION

The NEXT directive is used with the FOR statement to create conditional looping within a program.

See FOR/NEXT in this section.

EXAMPLE

See FOR/NEXT in this section for examples.

FORMAT

```
ON expr GOTO stno a {,stno b} {,stno c}...{,stno n}
```

where:

expr	=	a numeric integer (or variable representing same), the value of which determines the next statement number to be executed
stno a	=	the statement number to be executed next if the value of the expression equals 0 or less
stno b	=	the statement number to be executed next if the value of the expression equals 1
stno c	=	the statement number to be executed next if the value of the expression equals 2
stno n	=	the statement number to be executed next if the value of the expression is equal to or greater than the relative position of the statement number in line, minus one

DESCRIPTION

The ON/GOTO directive is used to transfer program control to a specified statement number. The statement number selected depends upon the numeric value of the expression, and the relative position of the statement numbers after the GOTO determines which statement number is to be executed next. During execution, the value of the expression must be an integer.

The first statement number (stno a) is executed next if the value of the expression is equal to 0 or less (negative). The second statement number (stno b) is executed next if the value of the expression is equal to 1. Subsequent statement numbers represent branch locations for successive integer values of the expression. The last statement number (stno n) is used for all values equal to or greater than the number of statement numbers in the list, minus 1.

There is no limit to the number of statement numbers permitted in the list (other than restrictions due to memory).

EXAMPLE

```
0100 ON X GOTO 0200,0300,0400,0500
```

-if X=0 or less (negative), the next statement execute' is 0200

-if X=1, the next statement executed is 0300

-if X=2, the next statement executed is 0400

-if X=3 or more, the next statement executed is 0500

FORMAT

```
OPEN (fileno/devno {,ERRrstno} {,BLK=max buffer size}
    {,TRK=track number} {,SEQ=sequence number}
    {,ISZ=recsz}) "file/device ID"
```

where:

BLK (user buffer size) (Level 3 only)	= either 0 (no user-area buffer) or 1024 The BLK= option can be used with Indexed, Serial and Direct files to speed up sequential accesses by reducing the number of physical I/O operations to one per buffer, rather than one per record. The option assigns user memory for the buffer used exclusively by the specified file. A buffer can be shared between a CALLing and CALLed program, and the file can be accessed by either program. WRITES are prohibited unless the file is LOCKed
ISZ (record size)	= an arithmetic expression representing a temporarily redefined record size for a file. The file is accessed as if it were an Indexed file with a record size equal to the arithmetic expression. The ISZ= option is used with READ RECORD and WRITE RECORD to handle multiple records or partial records (e.g., the Scatter Index Table (SIT) and KEY areas for Sort and Direct files). The FID of a file opened with the ISZ= option reflects the new record size and number of records, but the disc directory is not affected.

The last record in a file OPENed with ISZ is short (less than the ISZ size) if ISZ is not evenly divisible into the file size, but an ERROR 2, END OF FILE, is not generated until there is no data to be read in the file. An ERROR 1 is generated when the last record is written if the record to be written is larger than the last record size available.

A file OPENed with ISZ is implicitly LOCKed from use by other tasks.

TRK	= used for magnetic tape cartridge only, TRK specifies the track (0-3) on the cartridge to be used for data transfer
SEQ	= used for magnetic tape cartridge and reel-to-reel tape, SEQ specifies the file on the track to be accessed

DESCRIPTION

The OPEN statement is used for two purposes: To permit a user to access a specified disc data file for subsequent input/output operations, or to allow a user to reserve a specified input/output device for his/her exclusive use. Each user is limited to access (OPENing) of a total of 7 files and/or devices at any given time on Level 3, and 8 on Level 4.

Additional files/devices can be OPENed by CLOSEing those files/devices that are no longer needed.

The terminal on which the user program is running is assigned a device number of zero by the operating system.

EXAMPLES

```
0010 OPEN (1)"AD00R"

0020 OPEN (2,ERR=0050)A$

0030 OPEN (3,TRK=1,SEQ=0)"C0"
```

FORMAT

PRECISION expr

where:

expr = an integer value between 0 and 14

DESCRIPTION

The PRECISION directive is used to change the number of places of rounding. PRECISION is always reset to 2 when a BEGIN, CLEAR, RESET, END, STOP, RUN or LOAD statement is executed.

EXAMPLES

```
0010 PRECISION 2
0020 LET A=.55555
0030 FOR 1=0 TO 5
0040 PRECISION I;PRINT A;NEXT I

>RUN
 1

 .6
 .56
 .556
 .5556
 .55555
```

Statement 20 involves no computation; therefore, no rounding takes place. If, however, statement 20 above is replaced with the following:

```
0020 LET A=0+.55555
```

then the stored value of A is 0.56, and the printout reflects the rounded value:

```
>RUN
 1
 .6
 .56
 .56
 .56
 .56
```

```
0100 REM "CODE 3-6
0200 PRECISION 2
0220 LET A=.5,B=.01,C=4
0230 LET D=A*B*C,E=B*C*A
0240 PRINT D,E

>RUN

.04 .02
```

PRINT PRINT

FORMAT

```
PRINT {(fileno/devno {,END=stno} {,ERR=stno}
      {,IND=index value} {,KEY=key value}
      {,DOM=stno} {,TBL=stno})} {@(expr{,expr})}
      {list} {,IOL=stno} {,}
```

where:

list	= one or more numeric or string constants or variables or arithmetic or string expressions defining the data items to be printed or displayed. Each such data item can employ a positioning expression and/or a form expression as required. Mnemonic constants can be inserted at points where I/O device control is required
	= a comma can be used at the end to suppress the otherwise automatic line feed (LF)

NOTE

A comma is inserted before IOL= only when IOL= follows an expression list.

DESCRIPTION

The PRINT directive is used to PRINT to a file or device. Use of PRINT suppresses automatic generation of a field mark (line feed (LF) character) following each data field. One line feed character is generated at the end of all data items. A comma (,) at the end of all items suppresses the terminating LF character.

The PRINT statement is normally used to output data to terminals and printers. In this capacity the PRINT statement makes full use of positioning and form expressions as required to produce printed reports and precisely arranged and edited displays.

The PRINT statement can include any number of parameters defining data items to be printed. If the expression for any data item is not preceded by a positioning expression, printing (or display) occurs immediately following the last character output.

EXAMPLE

```
0130 PRINT (3,ERR=0340)@(5),A$,@(35),B:X$
```

FORMAT

```
PRINT RECORD (fileno/devno {,END=stno} {,ERR=stno}
             {,SIZ=size limit} {,DOM=stno}
             {,IND=index value} {,KEY=key value}
             {,TBL=stno}) {@(expr {,expr})}
             {string variable}
```

DESCRIPTION

The PRINT RECORD statement provides a means of writing a full record to a file without the requirement of specifying all of the variables which comprise the record. All field marks are transferred as data and no additional terminator is supplied. If the length of the variable is shorter than the defined record size, the rest of the record is filled with hexadecimal zeros.

EXAMPLE

```
0130 PRINT RECORD(3,ERR=0340)A$
```

FORMAT

```
PROGRAM "file ID" prog size, discno, secno
        {,ERR=stno}
```

where:

prog size	= the maximum size of the program in bytes (can not exceed 32,767 bytes)
discno	= the disc (0-7) on which the program is to be SAVEed
secno	= the sector where the program is to begin

DESCRIPTION

The PROGRAM directive defines a program file. Program files differ from data files in that they are accessed by LOAD, SAVE, RUN or CALL, rather than READ or WRITE.

NOTE

A Program file can be expanded to include a greater number of bytes by ERASEing the file and redefining the program with a larger number in the size field. The ERASE operation deletes information from the disc directory but does not alter data in the area defined for the program. Redefinition can be performed only if sufficient disc space exists immediately following the Program file.

EXAMPLE

```
>20 PROGRAM "KOJAK",2000,1,1000,ERR=0100
```

-defines program "KOJAK", with a maximum size of 2000 bytes at sector 1000 of disc number 1

FORMAT

PUT discno, secno {,ERR=stno} {,RTY=no. of retries}, input string variable {,verify string variable}

where:

secno	=	sector number to begin writing to
RTY	=	number of retries if the PUT is unsuccessful. RTY can be 0 to 254 (more than 254 is interpreted as 0). If not entered, the number of retries equals 19 on removable discs or 27 on fixed discs
input string variable	=	variable containing data to be put on disc
verify string variable	=	optional verification string, the same size as the input string variable

DESCRIPTION

PUT is used to write data contained in a string variable to a sector on a disc.

PUT can only be used on a DISABLEd or RESERVEd disc.

CAUTION

Improper use of the PUT directive can cause extensive file damage. Data that is PUT into a sector overwrites existing data in that sector. Due to common misuse of this directive, PUT is not recommended for use in applications programs.

```
0200 PUT 0,1096,ERR=0500,A$,A1$
```

FORMAT

```
READ {(fileno/devno {,END=stno} {,ERR=stno}
     {,IND=index value} {,KEY=key value} {,TBL=stno}
     {,SIZ=size} {,DOM=stno} {,TIM=time})}
     {,@(expr{,expr})} {variable list} {,IOL=stno}
```

where:

```
   @ expr,
   expr          =    horizontal and vertical
                     positioning
```

NOTE

A comma is inserted before IOL= only when both IOL= and a variable list are used.

DESCRIPTION

The READ statement is used to read data from a file or device. The fields from the record are placed into the respective variables in the READ statement.

If a field contains non-numeric information, and the corresponding variable is numeric, an error results. A numeric field can be read into a string variable, and a field that has been written as a string, but contains only valid numeric data can be read into a numeric variable.

If the information in a field is not required, an asterisk (*) can be substituted for the variable name to bypass processing of that field. The advantages of skipping fields are speed and a reduction of memory used by the program.

For non-terminal devices, the differences between READ and INPUT are as follows:

o String constants cannot be used;

o Positioning expressions are not allowed;

o Mnemonic constants are not allowed.

Use of the READ directive varies somewhat, depending on the file type (see File Types in Section 5).

Direct File READ

A Direct file can be READ either with or without a KEY option. If a key is not specified, the directive READs the record with the next highest key value. When the READ operation is complete, the "next key" pointer is updated to point to the key following the key that has just been read (i.e., READing a Direct file without specifying a key causes the records to be retrieved in keysorted order).

If a record is READ with a key and the key is not found, an error occurs, and the key pointer is updated to point to the next higher key after the key that was not found.

If it is not desirable for the key pointer to be updated to the next higher key fter the key that was not found, FIND should be used instead of READ.

NOTE

Level 4 unlinked files do not use linked keys. Therefore, KEY= or IND= must be specified in the READ statement.

Examples

Reading and writing a Direct file:

```
0010 REM "PROGRAM 1 ?  WRITE DIRECT FILE"
0020 BEGIN
0030 DIRECT "AA",52,100,100,0,700,ERR=0500
0040 OPEN (1)"AA"
0050 INPUT (0,ERR=0600)"PRODUCT NUMBER OR END
     ",A$:("END"=1000,LEN=5,5)
0060 INPUT (0,ERR=0060)"QUANTITY-",A:(100)
0070 INPUT (0,ERR=0070)"PRICE-'»,B:(99999.99)
0080 WRITE (1,KEY=A$)A$,A,B
0090 GOTO 0050
0500 PRINT "ERROR:  ",ERR:"000"
0510 STOP
0600 PRINT "ERROR IN INPUT.  PLEASE RE-ENTER"
0610 GOTO 0050
1000 PRINT "END OF JOB"
1010 STOP
9999 END
```

(more)

```
0010 REM "PROGRAM 2 ?  READ DIRECT FILE IN
     SEQUENCE AND PRINT PRICE"
0020 BEGIN
0030 OPEN (1)"AA"
0040 READ (1,END=1000)A$,*,B
0050 PRINT "PRODUCT-",A$,"   PRICE: ",B
0060 GOTO 0040
1000 PRINT "ALL PRODUCTS AND PRICES PRINTED"
1010 STOP
9999 END
>
```

```
0010 REM "PROGRAM 3 ?  UPDATE PRICES"
0020 BEGIN
0030 OPEN (1)"AA"
0040 INPUT (0,ERR=0040)"PRODUCT NUMBER OR END:
     ",A$:("END"=1000,LEN=1,52)
0050 EXTRACT (1,ERR=0500,KEY=A$)*,A,B
0060 PRINT "OLD PRICE IS ",B
0070 INPUT (0,ERR=0070)"ENTER NEW PRICE
     ",B:(99999.99)
0080 WRITE (1)A$,A,B
0090 GOTO 0040
0500 IF ERR0 11 THEN GOTO 0600
0510 PRINT "INVALID PRODUCT ENTERED.  PLEASE
     RE-ENTER"
0520 GOTO 0040
0600 IF ERR<>0 THEN GOTO 0700
0610 PRINT "RECORD FOR THIS PRODUCT IN USE. WAITING"
0620 GOTO 0050
0700 PRINT "ERROR: ",ERR:"00","OCCURRED ON READ"
0710 STOP
1000 PRINT "END OF JOB"
1010 STOP
9999 END
```

Sort File READ

The READ statement for Sort files cannot specify any data fields.

The following example defines a Sort file of 50 keys, each of which contains 10 characters, then writes 50 keys to the file, READs the Sort file sequentially, and prints each key:

```
0010 REM "CREATE SORT FILE"
0020 SORT "SORT", 10,50,0,100
0030 OPEN (1)"S0RT"
0040 FOR 1=1 TO 50
0050 WRITE (1,KEY=STR(I:"00000")+"AAAAA")
0060 NEXT I
0070 CLOSE (1)
0100 REM "READ SORT FILE SEQUENTIALLY AND PRINT
     KEYS"
0110 OPEN (1)"S0RT"
0120 LET K$=KEY(1,END=0200)
0130 REM "    K$ CONTAINS THE KEY OF SORT FILE"
0140 PRINT "KEY=",K$
0150 READ (1)
0160 REM "READ IS NECESSARY TO ADVANCE TO NEXT KEY"
0170 GOTO 0120
0200 REM "END OF FILE"
0210 STOP
```

One use of a Sort file is to effect different sequences of a single Direct master file. In the following example, the Direct file "MASTR" is a customer master file in customer number sequence (customer number is a 5-digit number). Each record in the master file contains 5 fields: Customer Number, Customer Name, Address, Amount Due, and Amount Paid. A SORT file "NAME" has been created with a key consisting of 10 characters: the first 5 characters of both the customer,name and the customer number. The sample program prints an alphabetic listing of all the customers in the master file which have a non-zero amount due:

```
0010 OPEN (1)"MASTR"
0020 OPEN (2)"NAME"
0030 OPEN (7)"LP"
0035 REM "  K$ CONTAINS THE FIRST 5 CHARACTERS OF
     CUST NAME
0036 REM "  PLUS THE CUSTOMER CODE IN POSITION 6-10
0040 LET K$=KEY(2,END=1000)
0045 REM "  CUSTOMER CODE IS THE KEY TO FILE "MASTR"
0050 READ (1,KEY=K$(6,5))A$,B$,C$,D,E
0055 REM "  THE VARIABLE D CONTAINS THE AMOUNT DUE
0056 REM "  IF NOT ZERO, THE CUSTOMER WILL BE LISTED
0060 IF DO 0 THEN PRINT (7)"CUSTOMER CODE",A$,
     "NAME: ",B$," AMOUNT DUE:",D
0070 READ (2)
0080 GOTO 0040
9999 END
```

READing From Indexed or Serial Files and Peripheral Devices

READ statements for Indexed or Serial files or from magnetic tapes cannot include a DOM= or KEY= option. The IND= option can be used to select specific records.

Example:

```
0010 REM "PROGRAM TO PRINT LABELS
0020 BEGIN
0030 OPEN (1)"ADDRES"
0040 OPEN (7)"LP"
0050 READ (1,END=0100)A$,B$,C$,D$
0060 PRINT (7)'FF',A$
0070 PRINT (7)B$
0080 IF LEN(C$)>0 THEN PRINT (7)C$
0090 PRINT (7)D$
0100 GOTO 0050
0110 CLOSE (1)
0120 CLOSE (7)
0130 END
```

FORMAT

```
READ RECORD (fileno/devno {,DOM=stno} {,END=stno}
            {,ERR=stno} {,IND=index value}
            {,KEY=key value} {,TBL=stno}
            {,TIM=time} {,SIZ=size}) string variable
```

where

string variable	=	a string variable into which the record is read

DESCRIPTION

The READ RECORD directive provides a method of reading a full record from a file or device. All field marks in the record are transferred as data When the size option is included, only the size specified is transferred.

NOTE

When used with magnetic tape units (MTC/MTR), the string variable must be DIMed large enough to hold all the data. Unlike other devices, which expand a string to a length necessary to hold all the data, MTC and MTR remain as DIMed. Data that exceeds the defined length of the string is truncated, and an ERROR 1 is returned.

EXAMPLE

```
0100  READ RECORD(1,END=0900)A$
```

FORMAT

```
RELEASE {"task ID"}
```

where:

task ID = the task identifier corresponding to the task identifier used with a prior START command from the system control task (SCT). The Task ID is not used when RELEASE is executed by a task other than the SCT

DESCRIPTION

The RELEASE directive closes files and releases a task's memory. The SCT can RELEASE any task except its own; all other tasks can RELEASE only themselves or any ghost task.

EXAMPLES

```
0010 RELEASE "T1"

0010 RELEASE
```

FORMAT

```
REM {{"}remark{"}}
```

where

remark	= a comment

DESCRIPTION

A comment can be inserted at any point in a program by using the REM statement. Quotation marks are recommended in cases of multiple REM's in one statement, and to ensure that any blanks within a REMark are retained.

EXAMPLE

```
0100 REM "PROGRAM TO DISEMBARK ARK"
```

FORMAT

```
REMOVE (fileno, KEY=key value {,DOM=stno}
       {,ERR=stno})
```

DESCRIPTION

The REMOVE directive is used to delete the key of an existing record in a keyed file. Deletion of a key removes all references to the key and its associated data. The associated record is filled with hexadecimal zeros (00).

The system updates the key pointer to point to the key following the key that has just been REMOVEd.

The REMOVE statement must specify a key.

EXAMPLE

```
0100 REMOVE (1,KEY=K$)
```

FORMAT

```
RESERVE discno
```

DESCRIPTION

The RESERVE directive RESERVES a disc for exclusive use by the task executing it. Only the task that RESERVEd the disc can access the files on it.

The following rules apply to the RESERVE directive:

- A DISABLEd disc can not be RESERVEd
- A RESERVEd disc can be DISABLEd by the same task
- An ENABLE deRESERVEs a disc
- A PUT can be performed on a RESERVEd disc

EXAMPLE

```
0100 RESERVE 0
```

FORMAT

```
RESET
```

DESCRIPTION

The RESET directive performs a low-level system reset that affects only the task that issued the statement. RESET resets the ERR and CTL system variables to zero, and any GOSUB or FOR/NEXT loops that have not been fully executed are reset. Additionally, the RESET statement reestablishes the arithmetic mode at PRECISION 2. Execution of the RESET statement does not clear the user data area, nor CLOSE any OPEN files or devices, nor reset the program execution pointer which identifies the statement to be next executed.

EXAMPLE

```
0900 RESET
```

RETRY RETRY

FORMAT

RETRY

DESCRIPTION

The RETRY directive causes the transfer of program control to the statement where the last error occurred. RETRY must be preceded by an error branch in a program or an ERROR 27 occurs. RETRY cannot be executed unless an error occurred previous to the RETRY.

The RETRY branch address is cleared by a START, LOAD or RUN (with program name specified), and BEGIN, CLEAR and RESET statements.

EXAMPLE

```
0010 REM "PROGRAM TO INPUT NEW CUSTOMERS"
0020 BEGIN
0030 OPEN (1)"MASTER"
0040 LET P$="00000"
0050 INPUT (0,ERR=0210)'CS',"CUSTOMER NUMBER
     (CR TO END) ",N:(99999)
0060 IF N=0 THEN STOP
0070 LET N$=STR(N:P$)
0080 FIND (1,DOM=0120,KEY=N$)
0090 INPUT (0,ERR=0090)@(0,22),'RB',"CUSTOMER ON
     FILE (DEL TO DELETE, CR TO CONTINUE:,T$:("DEL"
     =0100,""=0050)
0100 REMOVE (1,KEY=N$)
0110 GOTO 0050
0120 SETERR 0210
0130 INPUT @0,1),"ADDRESS",A$:(LEN=0,30)
0140 INPUT @(0,2),"CITY",C$:(LEN=0,15)
0150 INPUT @(0,3),"STATE",S$:("CA"=0160,"AZ"=0160,
     "OR"=0160)
0160 INPUT @(0,4),"ZIP",Z:(99999)
0170 INPUT @(0,5),"BALANCE",B:(-99999.99)
0180 SETERR 0
0190 WRITE (1,KEY=N$,ERR=8000)N$,A$,C$,S$,Z,B
0200 GOTO 0050
0210 INPUT (0,ERR=0210)@(0,22),'RB','INVALID (CR TO
     CONTINUE) ",T$:(""=0220)
0220 RETRY
   .
   .
8000 REM "ERROR HANDLING ROUTINE"
```

FORMAT RETURN

DESCRIPTION Available in Program Mode only, the RETURN directive is used to terminate a GOSUB, SETESC or SETCTL routine. It returns program control to the statement following the GOSUB, SETESC or SETCTL.

EXAMPLE

```
0300  GOSUB 0950
0400  LET Z$="ZFRANC"
.
.
.
0950  LET A=50; LET B=A*C/2; PRINT B
0960  RETURN
```

FORMAT

RUN {"prog ID"}

DESCRIPTION

The RUN directive is used to execute a program. Execution begins at the lowest numbered statement.

If a program has been SAVEd on disc but is not now the current program in memory, it can be executed by providing its file identification as a RUN parameter, as follows:

```
>RUN "AMOK"
```

If a program name is specified, RUN automatically LOADs the program, clearing FOR/NEXT/GOSUB/SETERR/ SETESC return addresses, and resetting PRECISION to 2. RUN then executes the program, beginning at the lowest numbered statement. As with the LOAD, the user data area remains unchanged when RUN is executed. If insufficient user area is available, an ERROR 19 (PROGRAM SIZE) is generated.

RUN can also be used to continue execution of a program after it has been stopped by any condition other than END or STOP. The condition causing the STOP is usually either the occurrence of an error or an ESCAPE. RUN causes the program to continue execution at the statement causing the error, or, following an ESCAPE, at the next statement in sequence.

Programmed overlay of segmented programs can be accomplished by the use of the RUN statement as part of a program:

```
0400 RUN "PRGM"
```

All previously existing program statements in the program area are DELETEd, and the program statement pointer is set to one. Existing data in the data area is not changed and is usable by the incoming program.

On Level 3 systems, the user program area is cleared prior to the attempt to LOAD. On Level 4, the program area is not cleared until it has been determined that the specified program can be LOADed.

EXAMPLES

```
>RUN


0400 RUN "AMOK"
```

FORMAT

SAVE "file ID" {,prog size, discno, secno}

where:

file ID	=	the name of the program. File ID is optional on Level 4, required on Level 3
prog size	=	the maximum number of bytes to be reserved on disc to store the program (cannot exceed 32,767 bytes)
discno	=	the number of the disc (0-7) on which the program is.to be SAVEd
secno	=	the number of the sector where the program is to begin

DESCRIPTION

The SAVE directive is used to copy a program from user memory to a Program file on disc. The Program file must have been previously defined by a PROGRAM statement or must be currently defined by parameters of the SAVE statement.

When the SAVE directive includes the program size, disc number and sector number parameters, it automatically defines a PROGRAM file and saves the program on disc.

When the SAVE directive is used with only the file ID, the program currently in the user memory area is saved in a previously defined file.

It is recommended on Level 4 and required on Level 3 that the file ID always be used to prevent accidental saving of the wrong program. If the file ID is not used (Level 4 only), the program is SAVEd to the file ID which was last referenced (LOADed, SAVed, RUN).

If an attempt is made to SAVE a program into a file of insufficient size, an ERROR 19 results. The file is defined, however, and must be ERASEd prior to entry of a subsequent SAVE statement.

If a program file has been ERASEd, it can be recovered by executing the equivalent PROGRAM statement if the area where the program was on disc has not been overwritten.

SAVE cannot be used in a Public program.

EXAMPLES

```
>SAVE "STAMPS",3000,0,2057
```

is equivalent to

```
>PROGRAM "STAMPS",3000,0,2057
>SAVE "STAMPS"
```

FORMAT

```
SERIAL "file ID", av recno, av recsz, discno, secno
       {,ERR=}
```

where:

av recno	the average number of records in the file
av recsz	the average size, in bytes, of each record in the file
secno	the sector number where the file is to begin

DESCRIPTION

The SERIAL directive defines a Serial file.

The average record size and average number of records parameters are used to define the total space required for the file. However, they do not limit either quantity to that amount. For example, a Serial file defined with 100 records with an average size of 60 bytes can actually contain 200 records of 30 bytes, or 50 records of 120 bytes, or any other combination which totals 6000 bytes.

Rules for using Serial files:

1. The maximum record size for a serial file is 32,767 bytes.

2. The file must be LOCKed in order to WRITE to it; otherwise, an ERROR 13, ILLEGAL FILE USE/ACCESS, results.

3. Indices can be used to access records in a Serial file as they are in an INDEXED file, except that record-to record movement of the index can be in the forward direction only. To move to a previous record, the file must be CLOSEd, then reOPENed.

EXAMPLE

```
>0170 SERIAL "TRIX",40,50,1,540
```

-This example defines file "TRIX" for storage of a total of 2000 bytes of data (including field terminators and record lengths), breaking down to an average number of records of 40, and an average record size of 50 bytes at sector 540 of disc 1.

FORMAT

```
SETCTL stno
```

DESCRIPTION

Available in Level 4 only, the SETCTL directive is used to cause branching when the operator enters CTRL+Y. When the operator enters this combination, and SETCTL is in effect, branching occurs to the statement number specified in SETCTL, and the ERR variable is set to 126. If no SETCTL is in effect, and the operator enters CTRL+Y, the input is ignored and no action is taken.

CTRL+Y can be used to branch to a subroutine, much like the GOSUB directive. When a RETURN is encountered after the CTRL+Y causes branching, program control returns to the statement following the CTRL+Y.

EXAMPLE

```
0700  SETCTL 0950
```

FORMAT

```
SETDAY "string expr"
```

where:

```
  string expr  =  8 characters in the format
                  MM/DD/YY, MM-DD-YY, or any other
                  format.  The recommended format
                  is:

                       MM = Month
                       DD = Day
                       YY = Year
```

DESCRIPTION

The SETDAY directive is used to set the value returned by the system variable DAY. The argument can be any string expression with 8 characters. All other lengths result in an error.

EXAMPLE

```
SETDAY "06/01/81"
```

SETERR

FORMAT

SETERR stno

DESCRIPTION

The SETERR directive is used to branch to a general error routine. RETRY can then be used to return to the statement at which the error occurred for reexecution. This greatly simplifies the code required to handle errors.

The following rules apply to SETERR:

- If an error occurs within a statement that has no explicit error exit (an ERR= option takes precedence over a SETERR), a branch occurs (if a SETERR is in effect) to the specified statement. The specified statement can be the beginning of a routine for handling the error.
- The routine can be terminated with a RETRY statement, in which case program control returns to the error statement.
- SETERR is cleared by a RUN, LOAD, RESET, BEGIN, CLEAR, END or SETERR 0.
- When the system takes the SETERR or ERR= branch, it automatically performs a SETERR 0 and saves the statement number to RETRY (unless the error occurred on an ERR= branch and returns to the same statement where the error occurred). This allows limited error branching within an error routine without losing the original RETRY address. When the RETRY statement is executed, the SETERR is restored to its original value. This design prevents an error within an error routine causing an infinite loop.
- If an ERR= option (that does not branch to itself) is executed within an error routine, the RETRY address is set to that statement (losing the original RETRY address) and the SETERR is not reset.
- If a SETERR is used for handling errors in a routine, a SETERR 0 should be executed after completion of the routine. This protects future errors from falling under control of the first SETERR.

EXAMPLE

```
0010  SETERR 0100
```

FORMAT

SETESC stno

DESCRIPTION

The SETESC directive is used to prevent an operator from escaping out of a program. SETESC causes program control to transfer to the specified , statement number when ESCAPE is pressed. The operating system executes a GOSUB to the SETESC line number and begins processing. Following a RETURN, the operating system resumes processing at the point the SETESC branch was taken.

The SETESC branch does not occur when a statement contains an ESCAPE directive.

EXAMPLE

```
0050 SETESC 9000
0059 REM "ESCAPE KEY WILL BE PRESSED DURING EXECUTION
     OF 60"
0060 LET A=A+1,B=B+1,C=C+1
0070 GOTO 0060
9000 REM
9001 REM "ESCAPE ROUTINE"
9002 REM
9003 PRINT "YOU CANNOT ESCAPE"
9004 RETURN
```

SETTIME

FORMAT

SETTIME numeric expr

where:

numeric expr :an expression representing a value between 0 and 24 entered in decimal form (i.e., 13-50 = 1:30 p.m.). The following formula should be used to determine the proper format:

H + (M/60)   (S/3600)

where:

H = Hours
M = Minutes
S = Seconds

DESCRIPTION

SETTIME is used to change the value of the TIM system variable. The TIM variable is set to 0 whenever the system is LOADed.

EXAMPLE

```
0010 REM "PROGRAM TO SET TIME AND DAY"
0020 BEGIN
0030 INPUT (0,ERR=0030)"HOUR = ",H:(23)
0040 INPUT (0,ERR=0040)"MINUTES = ",M:(59)
0050 INPUT (0,ERR=0050)"SECONDS = ",S:(59)
0060 PRECISION 4
0070 SETTIME H+M/60+S/3600
0080 INPUT (0,ERR=0080)"MONTH= ",M:(12)
0090 IF M<1 THEN GOTO 0080
0100 INPUT (0,ERR=0100)"DAY = ",D:(3D
0110 IF D<1 THEN GOTO 0100
0120 IF POS(STR(M:"00")="04060911",2)0 0 AND D>30
     THEN GOTO 0100
0130 IF M=2 AND D>29 THEN GOTO 0100
0140 INPUT (0,ERR=0140)"YEAR = ",Y:(99)
0150 IF Y<1 THEN GOTO 0140
0160 IF FPT(Y/4)<>0 AND M=2 AND D>28 THEN GOTO 0100
0170 SETDAY STR(M:"00")+"/"+STR(D:"00")+
     "/"+STR(Y:"00")
0180 REM "PRINT THE DATE AND TIME"
0190 PRECISION 4
0200 LET T=TIM, H=INT(T), S=INT(FPT(T)*3600),
     M=INT(S/60), S=S-M*60
0210 PRINT "DATE IS",DAY
0220 PRINT "TIME IS",H:"00",":",M""00",":",S:"00"
0230 STOP
```

SETTRACE

FORMAT

SETTRACE {(fileno/devno)}

DESCRIPTION

The SETTRACE directive is used to initiate the listing of statements as they are executed. It is especially useful when debugging a program that appears to be branching in an unforeseen or undesirable manner. The resulting listing delineates the exact sequence in which program statements are being executed. The SETTRACE command can be used as a statement within the program at selected points until the program is debugged. The output of the SETTRACE is in the same format as the LIST command. Optionally, SETTRACE can be entered in Console Mode to begin the listing of executed statements. In any case, the listing continues until terminated by execution of an ENDTRACE, END or STOP.

If the file or device specified has not been OPENed or has been made unready, an error results on the SETTRACE statement. Also, if the device being used to trace the execution should fail, an error occurs and the statement being executed is displayed as the statement in error. This can be confusing since the listed statement may not actually be in error.

In Level 3, SETTRACE is automatically discontinued when the Error Handler is ADDE'd.

EXAMPLE

```
0010 FOR 1=1 TO 3
0020 LET A=I+1; NEXT I
>SETTRACE

>RUN
0010 FOR 1=1 TO3
0020 LET A=I+1 NEXT
0020 LET A=I+1 NEXT
0020 LET A=I+1 NEXT
END

>READY
```

FORMAT

```
SORT "file ID", keysz, recno, discno, secno
     {,ERR=stno}
```

where

keysz	-	the size of a key in a keyed file; minimum=2, maximum=56 (if key is greater than 32,767, maximum=54)
recno	-	the maximum number of records for the file (cannot exceed 8,388,608)
secno	-	the sector number where the file is to begin

DESCRIPTION

The SORT statement is used to define a Sort file.

When a Sort file is defined, the Scatter Index Table and key area are initialized. Therefore, if a Sort file is accidentally erased, it cannot be restored by executing another SORT statement.

When accessing a Sort file, the I/O directives used must not specify any data fields.

EXAMPLES

```
0100 SORT "ACUTE",15,100,1,350
```

```
0010 REM "PROGRAM TO INPUT CUSTOMERS"
0020 BEGIN
0030 GOTO 0070
0040 REM "THE TWO FILES ARE"
0050 DIRECT "CUST",5,1000,64,1,24
0060 SORT "ZIP",10,1000,1,1000
0070 OPEN (1)"CUST"
0080 OPEN (2)"ZIP"
0090 INPUT (0,ERR=0090)"CUSTOMER NUMBER",A:(99999)
0100 LET A$=STR(A:"00000")      NAME",A1$:(LEN=0,20)
0110 INPUT (0,ERR=0110)"CUSTOMERZIP",Z:(99999)
0120 INPUT (0,ERR=0120)"CUSTOMER
0130 WRITE (1,KEY=STR(A:"00000"))A,A1$,Z
0140 REM "FOLLOWING EXTRACTS RECORD TO ENABLE
     OBTAINING OF INDEX VALUE"
0150 EXTRACT (1,KEY=A$)*
0160 LET I=IND(1)
0170 WRITE (2,KEY=STR(Z:"00000")+STR(I:"00000"))
0190 GOTO 0090
```

```
0010 REM "PROGRAM TO READ ZIP FILE AND PRINT CUSTOMER
     NAME AND NUMBER"
0020 BEGIN
0030 OPEN (1)"CUST"
0040 OPEN (2)"ZIP"
0050 LET A$=KEY(2,END=0100)
0060 READ (1,IND=NUM(A$(6)))A,A1$
0070 PRINT "CUST #",A,@(15),"CUST NAME",
     A1$,@(33) 'ZIP",A$(1,5)
0080 READ (2)
0090 GOTO 0050
0100 END
```

FORMAT

```
START pages {,ERR=stno} {,BNK=bank no.}
      {,"prog ID"} {,"task ID"}
```

where:

pages =	the number of pages assigned to the task (min=3 on Level 3, min=4 on Level 4; max=128)
bank no.=	the bank number in*which the task is to be located (some tasks-, e.g., DataWord and magnetic tape utilities, are limited to banks numbered 7 or lower)
prog ID =	the name of the program
task ID =	the task identifier. Terminal-connected tasks are numbered TO, T1, etc.; ghost tasks are numbered GO, G1, etc.

DESCRIPTION

The START directive assigns memory to a task, closes files, and clears the program and data areas of the task.

After the system is LOADed, only the System Control Task (SCT) is active. The START command must be executed from the SCT to activate other tasks. For example:

```
START 28,"UR","T1"
```

executed either in Console Mode or Program Mode, initiates the program "UR" at terminal "T1" in a little less than 28 pages of memory (there are 2 1/2 pages of task overhead on Level 3 and 3 1/2 pages on Level 4, whenever a task is STARTed). The statement:

```
START 28,"T1"
```

assigns a little less than 28 pages of user memory to "T1", with no program initiated, and T1 is activated in Console Mode (actual usable pages are 25 1/2 in Level 3, and 24 1/2 on Level 4, due to overhead).

Once activated by a START command from the SCT, other terminals can use the START command to reassign memory to their tasks. Terminals other than the SCT can only START themselves and ghost tasks.

EXAMPLES

```
>START 40

>START 40,"CASPER","GO"
```

FORMAT

STOP

DESCRIPTION

The STOP directive is used to terminate a program at any point other than the end of that program. The termination process resets the program execution pointer to the first statement of the program, CLOSES all open files and devices, performs a RESET operation, and returns the terminal to Console Mode. The "logical" termination point established by the STOP statement does not discontinue MERGE operations, and for this reason the STOP statement should be used for all program terminations except the one that occurs at the end of the program.

Execution of the STOP statement does not alter the data content of either the user data area or the user program area.

STOP is identical in function to END, except that END is used to terminate program LOADing during a MERGE operation.

EXAMPLE

```
6510 STOP
```

FORMAT

TABLE hexadecimal string

DESCRIPTION

TABLE is a non-executing statement used to define the substitution values used to translate characters from one code to another during an input/output operation.

Any input/output instruction which specifies a TBL= option includes, in the processing of that data, a conversion of each data character using the procedure described below. For input, the conversion is performed before the check is made for an input field terminator. For output, field terminators are converted after they are supplied by the system.

The first two digits of the hexadecimal string are used as a mask byte which is ANDed with each input byte; the remainder of the hexadecimal string is the code comparison table and can be of any length, 256 bytes or less.

The mask byte is ANDed with each input byte to form a temporary result byte. The ANDing operation is done on a bit level. When a bit in the input byte is a 1 and the corresponding bit in the mask byte is 1, the same bit in the result byte is set to 1. If either the bit in the input byte or the mask byte is 0, the corresponding bit in the result byte is set to 0. The ANDing operation can be compared to a filtering process: 1 bit in the mask allows data to pass through, 0 bits stop data from passing through the filter. The following examples demonstrate the AND operation:

```
INPUT BYTE 'FA' =1111  1010 'A6' = 1010 0110
MASK BYTE  'A3' =1010  0011 ?7F' = 0111  1111
RESULT BYTE A2   1010  0010  26    0010  0110
```

The result byte is then used as a subscript to the code conversion table. If the value of the subscript is 0, the first byte in the table (excluding the mask) replaces the input byte. If the value of the result byte is the binary equivalent of 20, the 21st byte (including the mask) from the table replaces the input byte.

TABLE (Cont'd) TABLE (Cont'd)

NOTE

> Proper selection of the mask byte reduces the size of the table. If the mask byte is 0011 1111 ('3F'), as in the examples above, the result byte never exceeds 0011 1111 ('3F'), and the table does not need to be larger than 64 characters in length. If the result byte exceeds the size of the table, the system outputs the result byte.

EXAMPLE

The following paragraphs provide ah example of the method used to build a table for EBCDIC to ASCII conversion:

Assume that the data to be read and converted contains only upper case letters and no special characters or terminators.

The first step is to build a table of the character set to be converted, the binary value of each character in ascending order. This is shown by columns one and two in Table 4-2. By looking at the Binary column (Column 2) it can be determined that the first two bits provide no useful information since they are identical. There are also cases where they are not the same, but provide no information, as in the case of a parity bit. In the example, it is desirable to strip off the first 2 bits. The mask for this is 0011 1111, or $3F$.

Next, column 3, which is the decimal value after the masking operation, is filled. After completing this, columns 4 and 5, which are the ASCII characters and hexadecimal values that the EBCDIC characters are to be converted to, are filled. At this point, Table 2 can be built showing all possible masked decimal values and their corresponding hexadecimal value. There are usually numerous holes in the table (marked with an *). These holes must be filled with some hexadecimal values, such as blanks, or another hexadecimal value that is not in the output character set, such that they can be later removed. Once this table is complete, it can be written in BASIC by appending the mask byte to the front of the hexadecimal values.

TABLE (Cont'd) TABLE (Cont'd)

TABLE 4-2 - TABLE STATEMENT TABLE

COLUMN 1	COLUMN 2	COLUMN 3	COLUMN 4	COLUMN 5
EBCDIC CHARACTER	EBCDIC BINARY VALUE	MASKED DECIMAL VALUE	ASCII CHARACTER EQUIVALENT	OUTPUT HEX VALUE
A	1100 0001	1	A	C1
B	1100 0010	2	B	C2
C	1100 0011	3	C	C3
D	1100 0100	4	D	C4
E	1100 0101	5	E	C5
F	1100 0110	6	F	C6
G	1100 0111	7	G	C7
H	1100 1000	8	H	C8
I	1100 1001	9	I	C9
J	1101 0001	17	J	CA
K	1101 0010	18	K	CB
L	1101 0011	19	L	CC
M	1101 0100	20	M	CD
N	1101 0101	21	N	CE
O	1101 0110	22	O	CF
P	1101 0111	23	P	DO
Q	1101 1000	24	Q	D1
R	1101 1001	25	R	D2
S	1110 0010	34	S	D3
T	1110 0011	35	T	D4
U	1110 0100	36	U	D5
V	1110 0101	37	V	D6
w	1110 0110	38	w	D7
X	1110 0111	39	X	D8
Y	1110 1000	40	Y	D9
Z	1110 1001	41	Z	DA

```
Masked Decimal
Value                 0  1  2  3  4  5  6 7  8  9  10*11*12*13*14*
Output HexValue      AO C1 C2 C3 C4 C5 C6C7 C8 C9 AO AO AO AO AO
Masked DecimalValue  15*16*17 16 19 20 2122 23 24 25 26 27 28 29


Output HexValue      AO AO CA CB CC CD CE CFDO D1 D2 AO*AO*AO*AO*
Masked DecimalValue  30 31 32 33 34 35 36 3738 39 40 41 42 43
                                                  through 63

Output HexValue      AO*AO*AO*AO*D3 D4 D5 D6D7 D8 D9 DA AO
                                                  are all AO's
0100 TABLE 3FAOC1C2C3C4C5C6C7C8C9AOAOAOAOAOAO
           AOCACBCDCECFD0D1D2A0A0A0A0A0A0A0
           D3D4D5D6D7D8D9DAAOAOAOAOAOAOAOAO
           AOAOAOAOAOAOAOAOAOAOAOAOAOAOAOAO
```

Within the Basic Four system, the TABLE statement has most often been used in (but is not limited to) the conversion of ASCII (American Standard Code for Information Interchange Code) to EBCDIC (Extended Binary Coded Decimal Interchange Code), and vice-versa.

FORMAT

```
UNLOCK (fileno/devno {,ERR=stno})
```

DESCRIPTION

The UNLOCK directive allows other tasks to access files previously LOCKed. A LOCKed file automatically becomes UNLOCKed when the file is CLOSEd.

EXAMPLE

```
0200 UNLOCK (1,ERR=0200)
```

FORMAT

```
WAIT number of seconds.
```

where

number of seconds	=	an integer representing the length of time for the pause

DESCRIPTION

The WAIT directive is used to suspend task execution for a specified number of seconds. The pause can range from 0 to 255 seconds, and is accurate to within 1 second.

EXAMPLE

```
0200 WAIT 2
```

FORMAT

```
WRITE {(filno/devno {,DOM=stno} {,END=stno}
      {,ERR=stno} {,IND=index value} {,KEY=key value}
      {,SIZ=size} {,TBL=stno})} {@(expr {,expr})}
      {variable list} {,IOL=stno}
```

NOTE

A comma is inserted before IOL= only when both IOL= and an argument list are used.

DESCRIPTION

The WRITE statement functions in the same manner as the PRINT statement except that the system automatically appends a line feed (LF) ($8A$) character to each variable specified in the WRITE variable list.

NOTE

Mnemonic constants and positioning expressions, if included as parameters, are output as data to non-terminal devices.

DIRECT FILE WRITE

Unless an EXTRACT preceded the WRITE operation (see EXTRACT), a Direct file WRITE statement must include a KEY. The system then searches the Key/Scatter Index Table area to see if the key already exists in the file.

If the key already exists, the new record is written over the old record. The operation is then complete. If the key does not exist, the system must find space for the key and data. If the "last removed key pointer" entry in the header is not zero, the relative position specified by this pointer is used for the key, and data.

If the removed key pointer is zero, the system uses the relative position specified by the next available pointer. The pointers are also updated to include the new key in sorted order.

It is highly recommended that the key always be written as part of the data. Accidental erasures of the file are then recoverable, since the data area is not destroyed on redefinition.

For more information, see "FILE TYPES", Section 11.

FORMAT

```
WRITE RECORD (fileno/devno {,DOM=stno} {,END=stno}
             {,ERR=stno} {,IND=index value}
             {,SIZ=size} {,KEY=key value}
             {,TBL=stno}) {string variable}
```

where:

string variable	=	the string variable to be written

NOTE

IND= cannot be specified when WRITEing to a magnetic tape unit.

DESCRIPTION

The WRITE RECORD statement provides a means of writing a full record to a file or device without the requirement of specifying all of the fields which comprise the record. All field marks are transferred as data and no record terminator is written. If the field is smaller than the defined record size, the record is filled with hexadecimal zeros.

EXAMPLE

```
0100 WRITE RECORD(1)A$
```

OVERVIEW

FUNCTIONS are used to manipulate data. They perform a variety of operations, such as converting characters to different forms (ASCII, hexadecimal, etc.), checking for data integrity, returning information related to files, converting variables from strings to numerics and vice versa, and more.

Predefined functions which are part of the Business BASIC language are discussed in this section.

In addition to the predefined functions, 26 user-defined functions are available for each program (see "DEF FNx" Directive in Section 4).

FORMAT

ABS (numeric expr {,ERR=stno})

DESCRIPTION

The ABS function computes the absolute value of an argument. The argument is evaluated for magnitude alone; the sign (+ or -) is ignored.

EXAMPLES

```
0100 LET X=ABS(12)     - assigns the value 12 to X

0100 LET X=ABS(-6)     - assigns the value 6 to X
```

FORMAT	AND (string expr, string expr {,ERR=stno})

DESCRIPTION — The AND function returns a string that is the result of combining the bits of two string expressions according to the following rules:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

EXAMPLE — LET X$=AND($0F$,$DC$)

```
          then   $0F$  0000 1111
                 $DC$   11011100

  PRINT HTA(X$) = $0C$ = 0000 1100
```

FORMAT	ASC (string expr {,ERR=stno})
DESCRIPTION	The ASC function converts a single string character to a decimal number. If the string expression is longer than one character, the value returned is the ASC of the first character in the string.

EXAMPLES

```
0500 LET X=ASC("A")   - returns a value of 193 to X

0500 LET X=ASC("1")   - returns a value of 177 to X

0500 LET X=ASC($79$)  - returns a value of 121 to X
```

FORMAT

```
ATH (string expr {,ERR=stno})
```

DESCRIPTION

The ATH function converts the hexadecimal characters in the string expression to ASCII characters. The string must contain only the characters 0-9 and A-F. Each two characters in the argument string are converted to one character in the output string. If the string contains an odd number of characters, a zero is added to the left of the first character before the conversion.

EXAMPLE

```
LET X$=ATH("B0B1B2")   -  X$ = $B0B1B2$ = 012
```

FORMAT

BIN (numeric expr, length {,ERR=stno})

where:

length	= length of t-he string

DESCRIPTION

The BIN function returns a string containing the binary representation of the value of the argument. The string is the length specified, padded with hexadecimal zeroes to the left, if necessary.

If the length is too short to contain all the significant digits of the number, the string is right justified and truncated.

The leftmost bit is considered the "sign" bit. If "on' (1), the nuber is negative. Negative numbers are stored in two's complement (negative binary) notation

The Binary to Hexadecimal Conversion Table is as follows:

```
                                 1111
0000 :: 0  0101 :: 5   1010 :: A
0001 :: 1  0110 :: 6   1011 :: B
0010 :: 2  0111 :: 7   1100 :: C
0011 := 3  1000 :: 8   1101 :: D
0100 :: 4  1001 :: 9   1110 :: E
```

EXAMPLES

```
LET X$=BIN(50,2)       -  X$ is $0032$

LET X$=BIN(1024,2)     -  X$ is $0400$

LET X$=BIN(-50,2)      -  X$ is $FFCE$

LET X$=BIN(193,1)      -  X$ is $C1$
```

-To print the values of X as listed, enter: PRINT HTA(X$)

FORMAT

BSZ (bank no.)

where:

bank no. = a number between 1 and 15

DESCRIPTION

The BSZ function returns the number of bytes available in the specified bank. The bank number can be any number between 1 and 15, inclusive, provided the specified number corresponds to an existing bank. The number of pages available in a bank can be computed as the INT (BSZ(bank)/256).

EXAMPLE

```
X=INT(BSZ(2)/256)

      X = the number of available pages in bank 2
```

FORMAT

```
CHR (numeric expr {,ERR=stno})
```

DESCRIPTION

The CHR function converts the numeric expression to an ASCII character. The number must be between 0 and 255.

EXAMPLES

```
0100 LET X$=CHR(193)
```

- stores "A" in X$

```
0100 LET X$=CHR(177)
```

- stores "1" in X$

CPL (COMPILE)

FORMAT

CPL (string expr {,ERR=stno})

DESCRIPTION

The CPL function compiles the string expression. The string can contain any valid BASIC statement, with or without a line number. The CPL function converts the statement to a format that can be executed by the BASIC executor. The first two hexadecimal bytes of the output string contain the string length in binary format, provided the statement number is included.

EXAMPLES

```
A$=CPL("1 END")     A$ = $04000143$
                           |  |  |
                           |  |  |
                           |  |  | -compiled "END"
                           |  |
                           |  |-statement number in
                           |     binary
                           |
                           |-length of compiled
                             statement in binary
```

-To print the value of A$ as listed, enter PRINT HTA(A$)

FORMAT

CRC (string expr {,2-byte string})

where:

2-byte string	the seed or start string, hexadecimal value

DESCRIPTION

Used to check for data integrity, the CRC function computes checksums for a string variable. Creation of the checksum is based upon the unique bit pattern of the series of characters comprising the string.

EXAMPLES

```
0020 LET A$=CRC(B$)
0030 LET A=ASC(A$(1))*256+ASC(A$(2))
```

-returns a 2-byte string that contains characters with ASCII values between 0 and 255

The CRC function also allows the accumulation of the CRC of a large string without having the complete string in memory at one time.

```
C$=CRC(A$+B$) is equivalent to:

C$=CRC(A$), C$=CRC(B$,C$)
```

NOTE

If the CRC is to be used in conjunction with unformatted synchronous communications, the bytes must be in reverse order.

FORMAT	DEC (string expr {, ERR=stno})

DESCRIPTION

The DEC function converts a binary string expression into a signed decimal number. The leftmost bit is considered the "sign" bit. If "on" (1), the number is negative.

Negative numbers are stored in two's complement (negative binary) notation.

EXAMPLES

```
LET X=DEC($0032$)       -  X is 50

LET X=DEC($FFCE$)       -  X is -50

LET X=DEC($0400$)       -  X is 1024

LET X=DEC("A")          -  X is -63

LET X=DEC($00$+"A")     -  X is 193
```

FORMAT

EPT (numeric expr {, ERR=stno})

DESCRIPTION

The EPT function returns the exponent of the numeric expression.

EXAMPLES

```
LET X=EPT(55)            55*10^2     then: X=2
                        .
LET X=EPT(5.23)          523*10^1    then: X=1

LET X=EPT(-500)         -.5*10^3     then: X=3

LET X=EPT(0)            0*10^0       then: X=0

LET X=EPT(.00001)       .1*10^-4     then: X=-4
```

FORMAT

```
FID (fileno {,ERR=stno})
```

DESCRIPTION

The FID function returns information associated with the specified file number. If the file number refers to a device, a two-byte device name is returned. If the number refers to a disc file, 20 bytes of information about the file are returned. The FID for a disc file has the following format:

BYTES	NUMBER OF BYTES	DESCRIPTION
1-3	3	Starting sector of file
4-9	6	File name
10	1	File type
		00 Indexed file
4-9	1	01 Serial file
		02 Direct or Sort File
10		04 Program file
		08 Unlinked (or Unsorted) Direct or Sort file (Level 4 only)
		$0A$ Dictionary
11	1	*Key size plus pointers
12-14	4	Number of records
15-16	2	Bytes per record
17-19	2	Ending sector + 1
20	1	Disc number (plus Fileset in Level 4)

*where pointer size in bytes is 4 for fewer than 32K records, 6 for 32K records or more

EXAMPLE

```
>LET A$=FID(2)
>PRINT A$(4,6)          -displays the file name of the
                         file OPENned on channel 2
```

FORMAT

FNx {$} (argument list)

where

x	= any letter from A-Z
$	= specified for string functions
argument list	= values provided for use by the DEF statement

DESCRIPTION

Used with the DEF directive, FNx allows reference to specific functions not provided in Business BASIC (see DEF directive in this section).

EXAMPLE

```
0230 LET A=FNA(B,D)
```

FORMAT

```
FPT (numeric expr {,ERR=stno})
```

DESCRIPTION

The FPT function returns the fractional part of the numeric expression, rounded to the PRECISION in effect.

EXAMPLES

```
0200 PRECISION 3

0210 LET X=FPT(55.885)      X=.885



0200 PRECISION 2

0210 LET X=FPT(55.885)       X=.89

0215 LET X=FPT(55.884)       X=.88
```

FORMAT	GAP (string variable or literal)
DESCRIPTION	This function generates odd parity for the specified string variable or constant on a byte-for-byte basis The resultant string is the same length as the specified string.
EXAMPLE	0200 LET A$=GAP($0FDC$) - A$ is equal to $8FDC$

FORMAT

HSH (string expr {,2-byte string})

where:

2-byte string =	the seed or start string, hexadecimal value

DESCRIPTION

The HSH function is used by the system to:

- o determine the location of an entry in the Scatter Index Table (SIT) that corresponds to a particular key; and
- o perform a data integrity check when a program is loaded into memory. All Level 3 and 4 programs have a HSH at the end, following the Auto-End.

EXAMPLES

```
0600 LET A$=HSH(B$)

0600 LET A$=HSH(B$,C$)

0600 LET A$=HSH(B$(10,LEN(B$)-11))
```

computes the Hash of a program where B$ is the entire program in compiled format

FORMAT	HTA (string expr {, ERR=stno})
DESCRIPTION	The HTA function converts each ASCII character in the string to its hexadecimal equivalent. Each character in the string is converted to two characters in the output string.
EXAMPLES	LET X$=HTA("ABC") - X$ is "C1C2C3"
	LET X$=HTA("123") - X$ is "B1B2B3"

FORMAT

```
IND (fileno {,ERR=stno} {,END=stno})
```

DESCRIPTION

The IND function returns the index of the next record to be accessed on the specified file. For Indexed and Serial files, the value returned is the index of the next sequential record. For Direct and Sort files, the value returned is the index of the next higher logical key.

EXAMPLE

```
LET A$=IND(1,ERR=0500,END=1000)
```

FORMAT	INT (numeric expr {,ERR=stno})
DESCRIPTION	The INT function returns the integer part of the numeric expression. Any fractional digits are removed, and rounding does not occur.

EXAMPLES

```
0100 LET X=INT(5.84)        -  X is 5

0200 LET Y=INT(.333)        -  Y is 0

0300 LET Z=INT(-6.22)          Z is -6
```

FORMAT

```
IOR (string expr, string expr {,ERR=stno})
```

DESCRIPTION

The IOR function returns a string that is the result of combining the bits of the two string expressions according to the following rules:

```
0  IOR  0  =  0
0  IOR  1  =  1
1  IOR  0  =  1
1  IOR  1  =  1
```

EXAMPLE

```
LET X$=IOR($0F$,$DC$)

        then:   $0F$ = 0000 1111
                $DC$ = 1101 1100

           X$ = $DF$ = 1101 1111
```

FORMAT

```
KEY (fileno {,ERR=stno} {,END=stno} {,IND=recno})
```

DESCRIPTION

The KEY function returns a string containing the key of the next logical record to be accessed from the file. Key is for use with Direct or Sort files.

For more information, see "FILE STRUCTURES AND ACCESS", Section 11.

EXAMPLE

```
0075 LET A$=KEY(1,ERR=0500,END=2000)
```

FUNCTION

```
LEN (string expr {,ERR=stno})
```

DESCRIPTION

The LEN function returns the length of the string, including any non-printable or fill characters.

EXAMPLES

```
0010 LET A$="ABC"

0020 LET B$="DEFG"

0030 LET X=LEN(A$)     -  X is 3

0040 LET Y=LEN(A$+B$)  -  Y is 7
```

FORMAT

LRC (string variable or constant)

DESCRIPTION

Used to perform a data integrity check, this function computes a longitudinal redundancy check based on the string variable or constant specified.

The code generated is returned as a 1-byte string, and is equivalent to the exclusive ORing (XOR) of all bytes of the argument string. A Null argument returns 00.

EXAMPLE

```
>LET A$=LRC($1C4D27$)
>PRINT HTA(A$)
 $76$
```

LST

LST
(LIST)

FORMAT

LST (string expr {, ERR=stno})

DESCRIPTION

The LST function converts a compiled BASIC statement into LIST format. The string expression must contain valid compiled BASIC code, with a line number.

EXAMPLE

```
0100 LET A$=LST(B$)
```

- when B$ is a compiled BASIC statement the statement is converted into LIST format and placed in A$.

```
1000 LET A$="BASIC"
1010 PRINT A$," FOUR"
1020 LET X$=PGM(1010)
1030 PRINT LST(X$)

RUN
>BASIC FOUR
```

FUNCTION	MOD (numeric expr a, numeric expr b {, ERR=stno})		
	where:		
	numeric expr a	=	the number on which to perform the modulo calculation; the dividend
	mumeric expr b	=	the number representing the base or divisor

DESCRIPTION

The MOD function performs repeated divisions, the first-numeric expression divided by the second. The result returned is the remainder of the last division (not the quotient).

MOD divides integers. It does not use fractional values rounded to Precision.

EXAMPLES

```
0100 LET X=MOD(26,7)     - X=5, the remainder

0100 LET X=MOD(22,11)    - X=0, the remainder
```

FORMAT NOT (string expr)

DESCRIPTION The NOT function returns a string that is the result of taking the inverse of the string, bit by bit. The rules for the NOT operation are:

```
NOT 0 = 1
NOT 1 = 0
```

EXAMPLE

```
0100 LET X$=NOT($DC$)

     if:                      $DC$ = 1101 1100

     then:  PRINT HTA(X$) = $23$ = 0010 0011
```

FORMAT

NUM (string expr {,ERR=stno})

DESCRIPTION

The NUM function returns the numeric value of the characters in the string expression. All characters in the string must be numeric, or related to numbers e.g., "+", "-", ".", ",", "E" are legal.

EXAMPLE

```
0100 LET A$="224"

0200 LET B=NUM(A$, ERR=8000)
```

- B is 224. If A$ contains any invalid characters, program control transfers to statement 8000.

FORMAT PGM (stno)

DESCRIPTION The PGM function returns the compiled format of the. designated statement number. If the statement number does not exist in the program, the next higher statement is returned.

EXAMPLES

```
0100 LET A$=PGM(10)       -  A$ returns the compiled
                             form of statement 10

0100 LET A$=LST(PGM(10))  -  A$ returns the listed
                             format of statement 10
```

POS (POSITION)

<u>FORMAT</u>

```
POS (scan string  relational operator  target string
    {, step value} {, ERR=stno})
```

where:

scan string	= the string (in constant or variable form) being searched
relational operator	= one of the valid symbols of comparison:
	`=` `<>` or `X`
	`<` `<=` or `=<`
	`>` `>=` or `=>`
target string	= the string (in constant or variable form) to be searched
step value	= the increment defining the intervals at which the target string is examined for each subsequent comparison (default value is 1)

<u>DESCRIPTION</u>

The POS function is used to determine the position of specified character(s) less than, equal to, or greater than those within a specified string. The value returned is the offset of the first matching substring in the target string. A zero is returned if no substring is found that meets the requirements.

<u>EXAMPLE</u>

```
LET A$="ABCDEFGHIJKL"
        (target string)

then: LET X=POS("D"=A$)       - X is 4

      LET X=POS("D"<A$)       - X is 5

      LET X=POS("D">A$)       - X is 1

      LET X=POS("5"=A$)       - x is 0

      LET X=POS("DE"=A$,3)    - X is 4

      LET X=POS("DE"=A$,4)    - X is 0

      LET X=POS("DE"<A$,3)    - X is 7
```

FORMAT

PUB (bank no.)

where:

bank no.	the designated bank number

DESCRIPTION

The PUB function returns a string representing all of the Public programs in the designated bank. For each Public program, a string (11 bytes in length on Level 3, 16 bytes for Level 4) in binary format is returned. The format, with conversion code, is as follows:

BYTE	CONTENTS	CONVERSION CODE
1,2	start location	PRINT DEC(A$(1,2))
3,4	program size	PRINT DEC(A$(3,2))
5-10	program name	--
11	program types: 1=ADDR 3=ADDE 5=ADDS	PRINT ASC(A$(11,1))
12	fileset number	PRINT ASC(1$(12,1))
13-16	unused	

EXAMPLE

```
0100 FOR 1=1 TO N
0110 LET A$=PUB(I)
0120 IF LEN(A$)=0 THEN GOTO 0160
0130 FOR J=1 TO LEN(A$) STEP X
0140 PRINT A$(J+4,6)
0150 NEXT J
0160 NEXT I
```

-where:

N = the highest available bank

X = the STEP value for the PUB function (11 bytes for Level 3, 16 bytes for Level 4)

FORMAT

```
SGN (numeric expr {, ERR=stno})
```

DESCRIPTION

The SGN function returns the sign of the numeric expression. If the expression is negative, a -1 is returned; if it is positive, a 1 is returned; and if it is zero, a 0 is returned.

EXAMPLES

```
LET X=SGN(-77)    -  X=-1

LET X=SGN(6)      -  X=1

LET X=SGN(0)      -  X=0
```

FORMAT

STR (numeric expr {:mask} {,ERR=stno})

where

mask	see "NUMERIC EDITING", page 2-9

DESCRIPTION

The STR function converts the numeric expression to a string of characters. The length and format of the string is specified by a format mask. The mask can be expressed as a string constant surrounded by double quotation marks (""), or as a string variable.

EXAMPLES

```
LET X$=STR(100:"00000")          X$ is "00100"

LET A=100
LET X$=STR(A:"$##0.00")          X$ is "$100.00"


LET X$=STR(100)               -  X$ is "100"
```

FORMAT

XOR (string expr, string expr {, ERR=stno})

DESCRIPTION

The XOR function returns a string that is the result of combining the bits of the first string with the bits of the second string according to the following rules:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

The strings must be the same length

EXAMPLE

```
 LET X$=XOR($0F$,$DC$)

                      then:        $0F$ = 0000 1111
                                   $DC$ = 1101 1100

                PRINT HTA(X$) = $D3$ = 1101 0011
```

SECTION 6 - SYSTEM VARIABLES

OVERVIEW

A system variable is a function whose use is pre-defined by the operating system. System variables are used to determine the value of specific system operations, such as the time (TIM) and the date (DAY).

System variables are also used to determine the number of unused bytes in the user area of memory, the value of the last occuring error, the highest available sector number, and more.

FORMAT

```
CTL
```

DESCRIPTION

The CTL variable contains a number that indicates which field terminator was used to end the last input statement. The meaning of each terminator key is defined by the application.

The following chart shows the terminator keys that the operator can use and the ASCII and CTL values (CTL is set to five (5) if input is terminated because a "SIZ=" clause in an input statement was satisfied):

KEY	VALUE	ASCII CHARACTER	CTL VALUE
RETURN	\$8D\$	CR (carriage return)	0
CBI (or SHIFT+CTRL+L)	\$9C\$	FS (field separator)	1
CBII (or SHIFT+CTRL+M)	\$9D\$	GS (group separator)	2
CBIII (or CTRL+N)	\$9E\$	RS (record separator)	3
CBIV (or CTRL+O)	\$9F\$	US (unit separator)	4
none (SIZ=satisfied)	(none)		5

EXAMPLES

```
0100 PRINT CTL

0100 IF CTL=4 THEN GOTO 9000
```

FORMAT

DAY

DESCRIPTION

The DAY variable contains the current date as an 8-byte string, and is set by using'the SETDAY directive. The date is returned in the following format:

```
MM/DD/YY       -where MM=month

                      DD=day

                      YY=year
```

EXAMPLES

```
 >PRINT DAY

0100 LET X$=DAY
0200 PRINT X$(1,2)   -  prints the month
```

FORMAT	DSZ
DESCRIPTION	The DSZ variable contains the number of unused bytes remaining in the user memory area.
EXAMPLE	> PRINT DSZ

ERR (ERROR) | ERR (ERROR)

FORMAT

```
ERR {(code 1, code 2,...,code n)}
```

where

code	an error code

DESCRIPTION

The ERR variable contains the value of the last error that occurred. This can be a number from 0 to 127-

ERR can be used by itself, as demonstrated in Example 1 below, to display the previous error number.

ERR can also be used to branch to a specified statement number, based upon the error code of the previous error, as demonstrated in Example 2 below.

EXAMPLES

1.
```
0100 PRINT "ERROR CODE = ", ERR
0999 EXIT ERR
```

2.
```
0050 ON ERR(11,12,47) GOTO 100,200,300,400
```

-branch to 100 if error is other than 11, 12 or 47

-branch to 200 if error=11

-branch to 300 if error=12

-branch to 400 if error=47

The same operation can be written using a LET statement:

```
0050 LET E=ERR (11,12,47)
0060 ON E GOTO 100,200,300,400
```

HSA
(HIGHEST SECTOR AVAILABLE)

HSA
(HIGHEST SECTOR AVAILABLE)

FORMAT

```
HSA (discno) {,ERR=stno}
```

DESCRIPTION

The HSA variable contains the highest sector number available on the specified disc.

EXAMPLE

```
>PRINT HSA(0)
```

PSZ (PROGRAM SIZE)

FORMAT

```
PSZ
```

DESCRIPTION

The PSZ variable contains the number of bytes used by the resident program, not including data. If PSZ is referenced in a CALLed program, the value is the size of the CALLing program.

NOTE

PSZ contains the user program area overhead. Therefore, PSZ always equals at least 19.

EXAMPLE

```
>PRINT PSZ
```

SSN
(SYSTEM SERIAL NUMBER)

SSN
(SYSTEM SERIAL NUMBER)

FORMAT SSN

DESCRIPTION The SSN variable contains the system serial number, returned in a 9-byte string in Level 3, and a 19-byt string in Level 4.

EXAMPLE

```
>PRINT SSN
```

SSZ (SECTOR SIZE) SSZ (SECTOR SIZE)

FORMAT	SSZ (discno)
DESCRIPTION	The SSZ variable contains the number of bytes in a sector on the specified disc.
EXAMPLE	>PRINT SSZ(O)

SYS (OPERATING SYSTEM LEVEL)

<u>FORMAT</u>

```
SYS
```

<u>DESCRIPTION</u>

The SYS function contains the level of the operating system. SYS is available only on Level 4.2 systems and above. It provides an 11-byte string expression showing the operating system level.

<u>EXAMPLE</u>

```
 >PRINT SYS
LEVEL 4.2A
```

TCB (TASK CONTROL BLOCK)

FORMAT

TCB (n)

where:

n = a numeric value ranging from 0-9

DESCRIPTION

The TCB variable contains information that pertains to a particular task. Level 3 systems support 9 TCB variables. Level 4 adds a tenth TCB which contains information about the SELECTed state of a task (see SELECT in this section).

Each TCB variable is one or two bytes in length, and some TCB's must be converted into decimal or hexadecimal format to be useful. The following list shows the contents of each TCB and the appropriate equation for conversion, if required. Note that all 1-type values are divided by 256.

TCB(n)	Description	Byte Length	Conversion Equation
0	disc number	1	D=INT(TCB(0)/256)
1	sector number	2-	S=DEC(BIN(TCB(1),2)+ BIN(TCB(2)/256,1))
2	sector number	1-	
3	system status	2	S$=HTA(BIN(TCB(3),2))
4	current statement number	2	
5	statement number of last error	2	
6	statement number SETESC references	2	
7	statement number SETERR references	2	
8	undefined		
9	SELECTed state (0-63, 255)	2	F=ASC(BIN(TCB(9),2))

EXAMPLES

```
0200 LET S=DEC(BIN(TCB(1),2)+BIN(TCB(2)/256,1))
```

- provides the sector number

```
1000 INPUT (0,ERR=8000)@(5,10)'CL',A
```

```
8000 PRINT @(0,21),'CL',"YOU GOOFED.  ERR =   ",
     ERR,"AT LINE    ",TCB(5);INPUT*;RETRY
```

- displays the line number where the error occured

FORMAT

TIM

DESCRIPTION

The TIM variable contains the current system time in hours and fractional hours. It is continually updated by the system, and can be set by using the SETTIME instruction.

TIM can be translated into hours, minutes, and seconds, as in the example below.

EXAMPLES

```
0100 LET T=TIM

0200 LET H=INT(T)

0300 LET S1=INT(FPT(T)*3600)

0400 LET M=INT(S1/60)

0500 LET S=S1-M*60
```

- where H=hours, M=minutes, S=seconds

```
>PRINT TIM
```

FORMAT

TSK (bank no.)

where:

bank no. = the number (0-15) of the memory bank to be checked for currently residing tasks

DESCRIPTION

The TSK function returns a 6-byte string representing each of the tasks located in the designated bank. The string consists of 2 bytes each for starting location within the bank, length in bytes, and task name ("T1", "T2", etc.).

When zero is specified as the bank number (i.e., TSK(O)), the system returns a list of configured devices, except for discs. Each configured device is contained in a 6-byte substring, the format of which is as follows:

Bytes 1,2	-	device name in ASCII (e.g., "TO")
Byte 3	-	device status in ASCII:
		Code 0 = available (not in use)
		1 = ESCAPE was pressed on an available VDT
		2 = not available (currently in use)
		3 = defective port (ERR=5 on access) (Level 4)
Bytes 4,5	-	reserved for future use (currently assigned blanks - $A0A0$)

(more)

Byte 6 device type in hexadecimal:

00	- DataWord I terminal
01	- serial printer (non pitch selectable)
02	- DataWord I printer
03	- slave printer (non selectable pitch)
06	- 3270 channel A
07	- 3270 channel B
$0A$	- 27xx/37xx channel A, leased
$0B$	- 27xx/37xx channel A, switched
$0C$	- 27xx/37xx channel B, leased
$0D$	- 27xx/37xx channel B, switched
$0E$	- 27xx/37xx channels A or B Auto-Dial
44	- 3100 parallel printer
47	- matrix printer
48	- drum printer
49	- 32xx parallel printer
$4A$	- pitch selectable serial printer
$4B$	- pitch selectable slave printer
80	- VDT with attached printer
81	- Auto-Dial terminal
FF	- VDT

NOTE

The values defined for the 6th byte are currently applicable only to terminals, ghost tasks, and printer devices. The 4th and 5th bytes are reserved for future use.

EXAMPLES

TSK(O)

```
>PRINT TSK(O)
```

An over-simplified example of a program using the TSK(O) variable follows:

```
0100 LET A$=TSK(0)
0110 FOR 1=1 TO LEN (A$) STEP 6
0120 LET B$=A$(I,6)
0130 PRINT B$ (1,2)
0140 IF B$(3,D ="0" THEN PRINT "AVAILABLE"
0150 IF B$(3,D ="1" THEN PRINT "ESCAPE KEY
     PRESSED"
0160 IF B$(3,D ="2" THEN PRINT "ACTIVE/IN USE"
0170 NEXT I
0180 END
```

The above example does not use the device type byte (byte 6 of each substring), but can be used to indicate the actual device. For example:

```
0165 IF B$(6,1)=$4B$ THEN PRINT "PITCH SELECTABLE
     SLAVE PRINTER"
```

NOTE

Byte 6 of some of the substrings not listed may return a byte which does not reflect the actual device type

TSK(1-9)

If the System Control Task issues the following:

```
Start 20,BNK=2,"T1"
Start 30,BNK=2,"T2"
```

and B$ = TSK (2) the result is as follows:

B$ is 12 bytes long, the first task starting at location (HTA) of B$ is EC00, the task length is 1400 (20 pages, 5120 bytes), and the file name is "T1" (D4B1).

The second task's starting location (at B$(7)) is CE00, the task length is 1E00 (30 pages, 7680 bytes) and the file name is "T2" (D4B2).

Input/Output options are used to augment the execution of an I/O directive. Specified within the parentheses, immediately following the file number, these optional parameters can cause branching within the program. They can also set up controls to override system defaults, specify a record to be accessed, specify a length for the range of a variable, and more.

Multiple I/O options in a statement are separated by commas. Except for the ERR= option, the order of the options within the parentheses does not matter.

If an ERR= option appears in a statement following, for example, a DOM= option, program control may transfer before the ERR= option is reached.

BLK=
(USER BUFFER SIZE)

FORMAT

```
BLK=(n)
```

where:

n = 0 (no user-area buffer) or 1024 (character user-area buffer)

DESCRIPTION

Available in Level 3 only, the BLK= option can be used with OPEN statements for Indexed, Serial and Direct files to speed up sequential accesses by reducing the number of physical I/O operations to one per buffer, rather than one per record.

BLK= assigns user memory for the buffer used by the specified file. A buffer can be shared between a CALLing and CALLed program, and the file can be accessed by either program. However, WRITE access is prohibited, unless the file is LOCKed.

EXAMPLE

```
0450  OPEN (1,ERR=0800,BLK=1024) "FILENM"
```

FORMAT

DOM= stno

DESCRIPTION

The DOM= option transfers control to the specified statement if the key specified in an INPUT or REMOVE operation is not found in the file, or if the key specified in a WRITE operation is already in the file. If a DOM= option is not used, an ERROR 11, MISSING OR DUPLICATE KEY, is generated when the specified key is not found.

NOTE

Use of DOM= is recommendedi,n statements performing READs and WRITES. When DOM= appears in the syntax before ERR=, special branching occurs in cases of missing or duplicate keys.

Exception: when DOM= is used in a WRITE which is updating a record, DOM= is ignored, the record is not updated, and the ERR= branch is taken.

EXAMPLES

`0100 READ (2,KEY=A$)R$`	If the KEY is not in the file, an ERROR 11 occurs
`0100 READ (2,KEY=A$,DOM=500)R$`	If the KEY is not in file, the DOM= branch is taken, and ERR=11 is set
`0100 WRITE (2,KEY=A$)R$`	If the KEY is in the file, old data is WRITEn over
`0100 WRITE (2,KEY=A$,DOM=500)R$`	-If the KEY is in the file, the DOM= branch is taken, and ERR=11 is set. Old data is not WRITEn over
`0100 WRITE (2,KEY=A$,DOM=500,ERR=400)R$`	- If the KEY is not in the file, but another error occurs, branch to statement 400

FORMAT	END= stno
DESCRIPTION	The END= option transfers program control to the specified statement number when the end of the file is reached. If an END= option is not used, an ERROR 2, END OF FILE, is generated.

EXAMPLES	`0200 READ (1,END=0500)A$`
	
	`0200 LET K$=KEY(1,END=9000)`

FORMAT

```
ERR= stno
```

DESCRIPTION

The ERR= option transfers program control to the specified statement number if an error occurs while executing the statement. For the statement containing it, the ERR= option overrides a SETERR statement. Specific error control clauses, such as END= and DOM=, override an ERR= option. Errors greater than 99 are not trapped by an ERR= option; rather, they cause an immediate exit to Console Mode, due to the nature of these errors (exceptions: ERRORS 126, CTRL+Y KEY USED, and 127, ESCAPE, are trapped by ERR=).

NOTE

Use of DOM= is recommended in statements performing READs and WRITEs. When DOM= appears in the syntax before ERR=, special branching occurs in cases of missing or duplicate keys.

EXAMPLE

```
0200 READ (1,ERR=0500)A$
```

FORMAT

IND= expr

where:

expr	= a numeric expression that specifies the position of the record in a file, relative to zero

DESCRIPTION

The IND= option specifies the index (record number) of the record to be accessed by the input/output statement. The first record in a file has an index of 0.

IND= can be used with Indexed, Direct, Sort or Serial files. Use of IND= when READing Direct or Sort files speeds record access by using the relative (to 0) record number. However, files are not sorted when this method is used.

EXAMPLE

```
0200 READ(1,IND=10)
```

ISZ=
(ACCESS FILE AS IF INDEXED)

FORMAT

ISZ=recsz

where:

recsz = the redefined record size for a file

DESCRIPTION

The ISZ option allows any file to be accessed as if it were an Indexed file with the record size specified.

ISZ= is used in conjunction with READ RECORD and WRITE RECORD to handle multiple records or partial records (e.g., the SIT and KEY areas for Sort, Direct or Program files). The FID of a file opened with the ISZ= option reflects the new record size and number of records, but the disc directory is not affected.

The last record in a file OPENed with ISZ is short (less than the ISZ size) if ISZ is not evenly divisible into the file size, but an ERROR 2, END OF FILE, is not generated until there is no data to be read in the file. An ERROR 1, END OF RECORD, is generated when the last record is written if the record to be written is larger than the last record size available.

A file OPENed with ISZ is implicitly LOCKed from use by other tasks.

EXAMPLES

```
>OPEN (1,ISZ=2048)"BOOK"

>READ RECORD(1)A$

>PRINT HTA(A$)
```

FORMAT KEY= string expr

DESCRIPTION This option specifies the key of the record to be accessed by the input/output statement containing the KEY= option.

EXAMPLES

```
0500 READ(1,KEY=A$)X$

0500 WRITE(1,KEY=STR(A:"00000"))A,B$
```

<u>FORMAT</u>

LEN= min,max

where:

min	the minimum length allowable for the variable
max	the maximum length allowable for the variable

<u>DESCRIPTION</u>

The LEN= option specifies the inclusive range for the length of a variable. Min must be less than or equal to Max.

If the length of the variable is beyond the specified range, an ERROR 48, INVALID INPUT, results.

<u>EXAMPLE</u>

```
0100 INPUT (0,ERR=0300)A$:(LEN=2,3)

0300 IF ERR=48 THEN GOTO 8000 ELSE GOTO 7000
```

FORMAT

```
RTY=  x
```

where

an integer between 0 and 255

DESCRIPTION

Used in input/output directives, the RTY= option specifies the number of retries the system is to perform if the attempt to execute the directive is unsuccessful.

If RTY= is not specified, the system performs approximately 19 retries.

EXAMPLE

```
0150 READ (1,ERR=0200,RTY=35)"A$"
```

<u>FORMAT</u>

```
SEQ= fileno
```

<u>DESCRIPTION</u>

The SEQ= option specifies the file number on the track being accessed. This option is only used for magnetic tape cartridge and reel-to-reel units.

<u>EXAMPLE</u>

```
0650 LET N=N+2

0700 OPEN (1,SEQ=N)"C1"
```

FORMAT

```
SIZ= numeric expr
```

DESCRIPTION

This option specifies the maximum number of characters that can be input by the input statement containing the SIZ= option. If the maximum number of characters is entered, input is ended, even if no Carriage Return or Control Bar key is pressed. The CTL variable is set to five (5) if input is terminated due to a SIZ= option.

EXAMPLE

```
0700 INPUT (0,SIZ=1)A$
```

FORMAT	TBL= stno
DESCRIPTION	This option specifies the number of the TABLE statement to be used to translate data. The statement number specified must contain a TABLE statement (see the TABLE directive in this section).
EXAMPLES	0100 READ(1,TBL=2000)A$ 0100 WRITE(2,TBL=5000)A$,B

FORMAT

TIM= numeric expr

DESCRIPTION

This option specifies the number of seconds allowed for completion of input. After that interval has passed, an ERROR 0 is generated. There is no default timeout for keyboard input. The maximum TIM= value is 255 seconds. "TIM=0" returns almost immediately.

EXAMPLE

```
0100 INPUT (0,ERR=0500,TIM=60)"NAME",A$
```

- allows 60 seconds for input; otherwise, control passes to statement 500

FORMAT

```
TRK= trackno
```

DESCRIPTION

On a magnetic tape cartridge, the TRK= option specifies which track is to be used for data transfer. This option is only used with magnetic tape cartridges, and is ignored by magnetic tape reel-to-reel units.

EXAMPLE

```
0700 OPEN (1,TRK=3)"C2"
```

FORMAT

VOL= volume number

DESCRIPTION

The VOL= option was used in early Level 3 systems to specify which volume of magnetic tape was to be used It has since been replaced with the TRK= option. Attempts to use VOL= on later systems result in an ERROR 20, STATEMENT SYNTAX.

EXAMPLE

```
 0200 OPEN (1,VOL=2)"CO"
```

OVERVIEW

System options are used to augment the execution of a directive, and are specified outside the parentheses (input/output options appear within parentheses). The 2 available system options are BNK= and IOL=.

Multiple system options in a statement are separated by commas, and the order in which they appear within the parentheses does not matter.

FORMAT	BNK= (n)
	where
	the number of the bank in which pages assigned to the task are to be located
DESCRIPTION	The BNK= option is used to assign a particular bank in which the pages assigned to the task are to be located. Using BNK= with a START statement, a programmer can control the amount of pages assigned to each bank of memory.
EXAMPLE	0200 START 45,BNK=3>"G2"

FORMAT

```
IOL=  stno
```

DESCRIPTION

The IOL= option specifies the statement number of the IOLIST to be used. The IOLIST contains a list of variables and/or constants.

EXAMPLES

```
0100 IOLIST A$,B,C,IOL=0200

0200 IOLIST D,E

0300 READ (1,KEY=A$)IOL=0100

0400 PRINT (7)IOL=0100
```

Mnemonics are used to prepare devices for the reception or transmittal of data. In some cases, mnemonics return the devices to an idle state upon completion of the data transfer, and flag special action which is device dependent. Some mnemonics merely set flags which are tested during subsequent operations.

Mnemonic constants are subject to TBL= conversion, and are passed as data to the software driver for the device.

Each mnemonic consists of two alphabetic characters enclosed by primes (single quotation marks) and is inserted in a statement at the point where the stated operation is desired. The format and use of the mnemonics is illustrated by the following example:

```
        0100 PRINT @(35,5), A$, 'LF', B$
```

In this example the 'LF' mnemonic is used to perform a line feed on the user terminal after printing the value of A$ at character position 35 on line 5. If the mnemonic is inserted in the statement immediately following the PRINT directive, the line feed occurs prior to printing the value of A$.

The mnemonic constants available for each type of I/O device appear on the following pages. Unless the mnemonic is listed as applicable for a device, an ERROR 29, UNDEFINED MNEMONIC, is generated upon statement execution (ERROR 29's can be turned off for VDT's in Level 4).

After the list is a special section on Mnemonic Hexadecimal sequence.

MNEMONIC	MNEMONIC NAME	APPLICABLE LEVEL	APPLICABLE DEVICES	RESULTANT ACTION
@(x)	Horizontal Position	3,4	VDT, Printer	Display next data at absolute horizontal position defined by x
@(x,y)	Horizontal and Vertical Position	3,4	VDT	Display next data at position x of vertical line y
'BE'	Begin Echo		VDT	Begins the display of input data
'BG'	Begin Generating ERROR 29	4	VDT	Begins the generation of ERROR 29 after execution of 'EG' which ended the generation
'BI'	Begin Input Transparency	4	VDT	Passes input data through the driver with no interference. Prevents the interception (therefore, usefullness) of '<u>ESC</u>' '<u>CAN</u>', and '<u>CTL</u>' X, Y, S, and Q
'BO'	Begin Output Transparency	4	VDT	Causes all data control characters and mnemonic sequences (except 'EO') to be sent to the device, without interference or translation by the driver
'BS'	Backspace	3,4	VDT	Moves the cursor back one space, erasing the previous character
'BT'	Begin Input Buffering (Type-Ahead)	4	VDT	Begins input buffering if the system is so configured
'CE'	Clear Screen to End of Page	4	VDT	Clears the screen from the cursor to the end of the screen

MNEMONIC	MNEMONIC NAME	APPLICABLE LEVEL	APPLICABLE DEVICES	RESULTANT ACTION
'CF'	Clear Fore- ground	3,4	VDT	Replaces all Foreground characters with spaces
'CH'	Cursor Home	3,4	VDT	Positions cursor at home (0,0) and sets Foreground mode
'CI'	Clear Input	3,4	VDT	Clears all data in the input buffer
'CL'	Clear Line	3,4	VDT	Replaces all characters between the cursor and the end of the line with blanks
'CR'	Carriage Return	3,4	VDT, Printer	Cursor drops one line, moves to horizontal posi- tion 0. Varies per type of printer
'CS'	Clear Screen	3,4	VDT	Clears all char- acters from the video screen, posi- tions the cursor at home and sets the mode to Foreground
'DC'	Delete Character	4	VDT	Deletes the char- acter at the cursor and shifts char- acters to the right of the cursor one position to the left. Writes a space in the last position of the line or field. Starts Foreground if Background is in effect
'EE'	End Echo	4	VDT	Ends the display of input data

MNEMONIC	MNEMONIC NAME	APPLICABLE LEVEL	APPLICABLE DEVICES	RESULTANT ACTION
'EG'	End Generation of ERROR 29	4	VDT	Prevents ERROR 29's in Level 4.
'EI'	End Input Transparancy	4	VDT	Restores interception of '<u>ESC</u>', '<u>CAN</u>' and '<u>CTL</u>' X, Y, S, and Q
'EL'	End Load	3,4	Some Printers	Ends the loading of the VFU (vertical format unit)
'EO'	End Output Transparency	4	VDT	Cancels 'BO', causing data, control characters and mnenomic sequences to pass through and be translated by the driver on the way to the device
'EP'	Expanded Print	3,4	Some Printers	Causes all characters in the current line to be printed in expanded print. Printing of null line results in the next line being expanded
'ES'	ESCAPE	3,4	VDT	Sends an ESC character to the device, which treats it as a lead-in code. The next character defines an action code for the VDT

MNEMONIC	MNEMONIC NAME	APPLICABLE LEVEL	APPLICABLE DEVICES	RESULTANT ACTION
'ET'	End Input Buffering (Type-Ahead)	4	Software	Cancels 'BT', ending input buffering
'FF'	Form Feed	3,4	Printers	Causes printers to vertically space to the top of the next page
'IC'	Insert Character	4	VDT	Moves all characters at and to the right of the cursor one space right. The next character output or input occurs in the space at the cursor position. Resets Foreground mode
'LD	Line Delete	3,4	VDT	Removes the line where the cursor is positioned, rolls all lines below it up one line, inserts a blank line at the bottom of the screen and sets the mode to Foreground
'LF'	Line Feed	3,4	VDT, Printer	Outputs a line feed/ carriage return
'LI'	Line Insert	3,4	VDT	Inserts a blank line at the position of the cursor, rolls all lines below it down, deletes the bottom line on the screen and sets Foreground mode

MNEMONIC	MNEMONIC NAME	APPLICABLE LEVEL	APPLICABLE DEVICES	RESULTANT ACTION
'PE'	End Protect	4	VDT	Cancels 'PS', ending the protection mode
'PG'	Page Mode (Printer Port Only)	3,4	7270 VDT with Printer	Sends to the local serial printer, all VDT screen data from the home position (0,0) to the cursor (when the system is so configured)
'PM'	Plot Mode	3,4	Some Printers	Used with each line of Plot Data. In this mode, a 'LF' causes the paper to advance only a single dot row, instead of a normal character line space
'PS'	Start Protect Mode	4	VDT	Begins display protection. Prevents the cursor from entering a previously protected position, and also prevents screen scrolling
'RB'	Ring Bell	3,4	VDT, Printer	Causes beep on VDTs; rings bell on some printers
'RC'	Read Cursor		VDT	Provides current cursor position coordinates. Should be used with or followed by an INPUT directive. Echo is suppressed for the remainder of the INPUT directive and is restored afterward

MNEMONIC	MNEMONIC NAME	APPLICABLE LEVEL	APPLICABLE DEVICES	RESULTANT ACTION
'S2'	Slew 2	3,4	Some Printers	Slew to Channel 2
'S3'	Slew 3	3,4	Some Printers	Slew to Channel 3
'S4'	Slew 4	3,4	Some Printers	Slew to Channel 4
'S5'	Slew 5	3,4	Some Printers	Slew to Channel 5
'S7'	Slew 7	3,4	Some Printers	Slew to Channel 7
'S8'	Slew 8	3,4	Some Printers	Slew to Channel 8
'SB'	Start Back-ground	3,4	VDT	Begins Background mode. Marks Background characters as protectable, though does _not_ begin protection
'SF'	Start Fore-ground	3,4	VDT	Begins Foreground mode.
'SL'	Start Load	3,4	Some Printers	Directs the loading of the electronic VFU (vertical print unit
'TR'	Transmit Screen	3,4	VDT	Sends data from the display screen to the input variable. Unsupported on Level 3
'VT'	Vertical Tab	3,4	VDT, Printers	Provides ability to execute routines on the VDT that are designed for printers

MNEMONIC HEXADECIMAL SEQUENCE

Level 3 uses a BASIC/driver protocol that allows a special X 'FE' lead-in for mnemonics; Hexadecimal codes can be input instead of the two-letter mnemonic name. Using this method, some Level 3 systems can utilize mnemonics normally available to Level 4 only. Use of mnemonics on Level 3, however, is not supported by Basic Four, and can yield unpredictable results. <u>They are listed here (Table 4-1) for informational purposes only, and their use is not recommended</u>.

Level 4 uses a different terminal driver and does not recognize the X 'FE' conversions.

Table 4-1. Level 3 Mnemonic Conversion

Lead-in (Hex)	Convert To
$FE90$	'TR'
$FE91$	@ (X,Y)
$FE92$	'CH'
$FE93$	'LD'
$FE94$	'CL'
$FE95$	'PS'
$FE96$	'PE'
$FE97$	'IC'
$FE98$	'DC'
$FE99$	'SB'
$FE9A$	'LI'
$FE9B$	'RC'
$FE9C$	'CS'
$FE9D$	'CF'
$FE9F$	'SF'

OVERVIEW

The disc is a permanent storage device for programs and data files. This section describes the organization of discs used by the Basic Four operating system.

DISC FORMAT

The disc is divided into segments called "sectors", which are numbered from 0 through the highest user-addressable sector (HSA). A maximum of 1024 bytes of information can be stored in each sector.

Each Disc is catagorized as either a "system dis" or a "user disc". The allocation of space on each type of disc is as follows:

SYSTEM DISC

SECTORS	CONTENTS
0	Bootstrap
1	Header
2-6	Loader
7-n	Directory
n+1-HSA	Files

USER DISK

SECTORS	CONTENTS
0	Bootstrap
1	Header
2-n	Directory
n+1-HSA	Files

where n = sector number

The operating system can be loaded from a system disc only.

DISC COMPONENTS

Each disc is comprised of the following components:

BOOTSTRAP

The bootstrap, stored on sector 0 of all discs, is used to load the operating system into memory. The bootstrap cannot be accessed by the user.

HEADER

The disc header defines certain characteristics of the disc. Created by a utility program when the disc is initialized, the header is stored in sector 1. The header contains the following information:

BYTES	CONTENTS
1-3	Starting sector of directory
4-9	Disc name
10	\$0A\$
11	\$18\$
12-14	Maximum number of directory entries
15-16	\$0000\$
17-19	Ending sector+1 of directory
20	\$01\$
21-24	Reserved for system use

The remainder of sector 1 is reserved for use by utility programs.

LOADER

The loader, stored on sectors 2-6 of the system disc, is used to load the operating system into memory.

The directory is a special form of a Sort file which contains a key for each file stored on the disc. While the size of the directory may vary, its maximum capacity is 32,767 entries.

Each directory entry (key) contains the following information:

BYTES	CONTENTS
1-3	Starting sector of file
4-9	File name
10	File type (low order four bits) 0 - Indexed 1 - Serial 2 - Direct/Sort 4 - Program 8 - Unlinked (Level 4)
11	Key size (00 for Indexed, Program, Serial files)
12-14	Number of records defined in file or actual program size
15-16	Record size <u>or</u> actual program size
17-19	Ending sector+1 of file
20	Reserved
21-22	Pointer to next logical key, sorted on starting sector
23-24	Duplicate Scatter Index Table pointer

NOTE

More detailed information about directory operations can be found in "FILE STRUCTURES AND ACCESS", Section 11.

All nonreserved sectors on the disc can be used to store programs and data files.

SECTION 11 - FILE STRUCTURES AND ACCESS

OVERVIEW

Business BASIC provides the user with several alternative methods of organizing data in a file. Knowledge of the different file structures and access methods aid the user in determining the optimum file type for each application.

This section contains descriptions of the following types of files:

o Indexed

0 Serial

o Program

o Direct

0 Sort

o Unlinked File (Level 4)

In addition to the above file types, this section contains information about the disc directory.

NOTE

This manual is a reference manual rather than a training manual. Explanations in this section are intended for reference purposes only.

INDEXED FILE

OVERVIEW

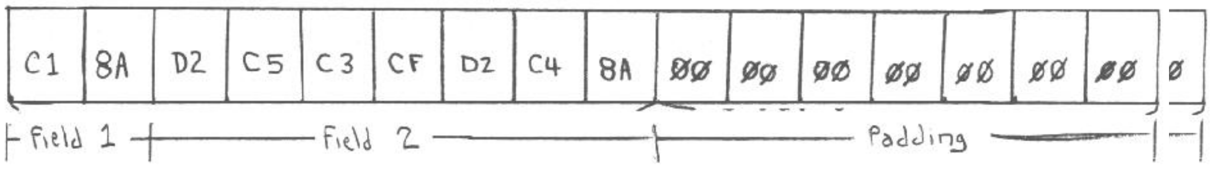
An Indexed file is a collection of fixed length records stored in contiguous sectors on a disc.

RECORD STRUCTURE

Each physical record in an Indexed file has a fixed length specified by the FILE directive. As with all file types, fields within each record are delineated by special characters called "field separators", which are automatically inserted by the operating system when the record is written. Logical records, therefore, may vary in length. The unused portion of the physical record is filled with null characters.

The structure of a record in an Indexed file defined with a record size of 16 bytes, for example, is as follows:

```
WRITE (1)"A","RECORD"
```



ACCESS

Records in an Indexed file can be accessed either randomly by record number, or sequentially, through use of the IND= option. The first record in the file has an index of 0.

PROGRAMMING NOTES

An Indexed file is not cleared when it is defined; the FILE directive inserts the name in the directory, but <u>does not alter the data area of the file itself</u>.

SERIAL FILE

OVERVIEW

A Serial file is a group of variable length records preceded by a header which describes the current contents of the file. The Serial file, which minimizes the use of space, is used for truly sequential operations, such as spooling.

HEADER STRUCTURE

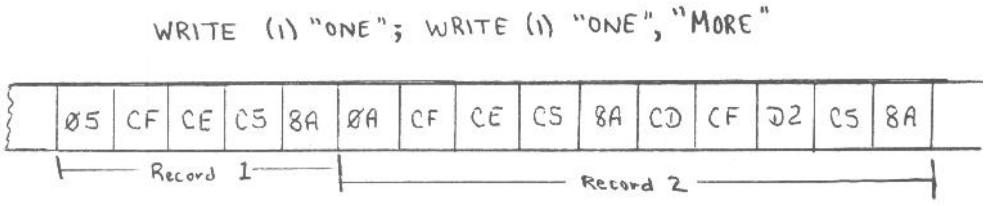
The Serial file header contains the following information:

BYTES	CONTENTS
1-3	Next available file index
4-7	Number of bytes on file, including the header
8-9	0000
10	"S"
11	01

RECORD STRUCTURE

The header is followed by a series of variable length records. The actual record data is preceded by a one or three byte length descriptor, which contains the total number of bytes in the record. If the total length of the record (length descriptor + data) is less than 255 bytes, the size of the descriptor is one byte; otherwise, a three byte descriptor is used (the first byte is null).

The structure of two records in a Serial file is illustrated as follows:



TERMINATOR

Following the final data record in the file is a terminator record of three nulls (00).

ACCESS

Records in a Serial file must be READ or WRITEn sequentially. The IND= option can be used only to position a previously WRITEn file in a forward direction.

PROGRAMMING NOTES

A Serial file must be locked before it is WRITEn. A maximum of 32,764 bytes of data can be stored in each record.

The Serial directive initializes the file header which, in effect, causes the data area to be cleared.

A Serial file to be used for spooling should be created with the PRINT statement to prevent a field separator ($8A$) from insertion between fields. Each PRINT statement causes one record to be WRITEn, with a field separator (line feed) at the end of the record. A comma at the end of the PRINT statement supresses the line feed.

NOTE

A File header is not WRITEn on the disc until the file is CLOSEd. Failure to CLOSE a Serial file after WRITEing it results in a loss of data.

PROGRAM FILE

OVERVIEW

A program consists of four logical sections:

- o Header Structure
- o DEF Table Structure
- o BASIC Statement
- o Terminator

HEADER STRUCTURE

The 11 byte header contains the following information:

BYTES	CONTENTS
1-3	Program length (byte 1 through hash)
4-9	Program name
10-11	Reserved for system use

DEF TABLE STRUCTURE

The DEFined function table begins in byte 12. The length of the table, also stored in byte 12, is computed by multiplying the number of DEF statements in the program by 3 and adding 1. If the program has no DEF statements, byte 12 contains 01.

Each entry in the DEF table contains 3 bytes. The first byte contains the function letter designator, where FNA is represented by 01, FNB by 02, and FNZ by $1A$. The second and third bytes of each entry contain the binary representation of the corresponding DEF statement number.

BASIC STATEMENTS

The program itself begins aft[illegible] the function table; i.e., at DEC (byte 12) + 12 ([illegible]ELLER). Each statement contains three fields: statement length, statement number, and statement body.

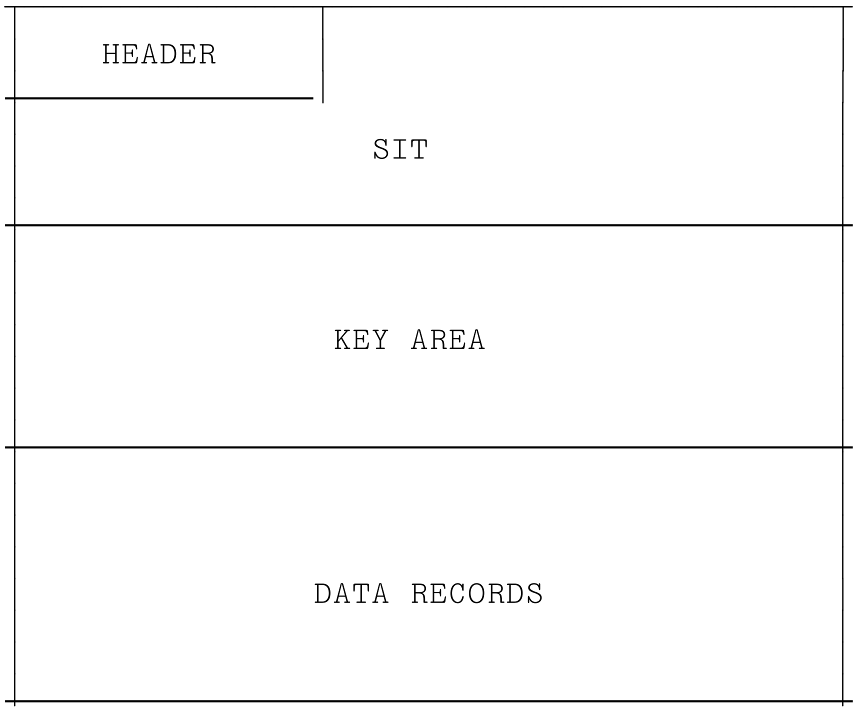
The length field of each statement contains the total length of all three statement fields. If the total length is less than 255 bytes, the length field is one byte; otherwise, the length field is 3 bytes long (the first byte is null values, and the next 2 bytes are the length in binary).

TERMINATOR	The 7 byte terminator follows the final BASIC statement in the program. The first 5 bytes of the terminator are 0427104303. The last 2 bytes of the terminator contain the program checksum. The checksum is computed by taking the HSH of the program from bytes 10 through 5 of the terminator (i.e., the checksum excludes the program length, program name, and the checksum itself).
PROGRAMMING NOTES	The PROGRAM and SAVE directives do not initialize the program area on the disc.

DIRECT FILE

OVERVIEW

A Direct file consists of a Scatter Index Table (SIT), key area, and data records. The general format of the file is illustrated in the following diagram (not drawn to scale):



Associated with each record in a Direct file is a string of characters called a "key". The key provides a convenient method for randomly accessing records.

Keys are linked together in ascending sorted order, which provides the capability to access data sequentially.

SIT STRUCTURE

The SIT (Scatter Index Table) is a collection of 2 or 3 byte values which point to keys in the key area. The size of the pointers is based on the defined size of the file; 3 byte pointers are used if the file is defined with more than 32,767 records.

The first 5 pointers in the SIT comprise the file header, which is used in conjuction with the disc directory entry to fully describe the characteristics of the file.

HEADER STRUCTURE

The header contains the following information

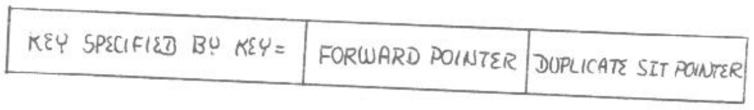
BYTES	CONTENTS
1	Index of the last key removed (00's if none)
2	Index of the next available key slot (FF if none)
3	Index of the lowest logical key in the file
4	Number of active keys in the file
5	Reserved

The index is the physical position, relative to one, of the key slot in the key area. The first key slot has an index of one (IND=0); the second, an index of 2 (IND=1); etc.

KEY AREA STRUCTURE

The key area is divided into segments called "slots". The size of each slot is computed by doubling the pointer size (2 bytes if less than 32,767 records, 3 bytes if more) and adding the defined key size. A key slot cannot cross a sector boundary.

Each active key in the file has 2 pointers appended to it:



The forward pointer is the index of the next logically higher key in the file. The duplicate SIT pointer is the index of the next key with a duplicate scatter index value (SIV).

When an active key is removed from the file, the key and its first pointer are filled with FF's. The second pointer is the index of the next key on the removed chain.

When the file is defined the SIT area and key area are initialized to nulls (00). Therefore, Key slots which have never been used contain nulls (00's).

RECORD STRUCTURE

The data records in a Direct file have the same format as those in an Indexed file. The relative position of the data record corresponds to the relative position of its key in the key area.

ACCESS

Records can be accessed either randomly by key, or sequentially in ascending key sort order. Descriptions of random and sequential access follow a brief description of hashing.

The methods used by the operating system to locate or remove an existing key, or insert a new key are transparent to the user. The information in this subsection is for general information only.

HASHING

A hashing algorithm is used to locate an entry in the SIT which corresponds to the key specified on the key option. The following BASIC program duplicates the hashing algorithm used by the operating system:

```
0010 REM "K$: KEY SPECIFIED IN KEY OPTION"
0020 REM "N:  NO. OF SECTORS IN SIT"
0030 REM "R:  NO. OF RECORDS DEFINED IN FILE"
0040 REM "S:  RELATIVE SIT SECTOR CONTAINING POINTER"
0050 REM "P:  OFFSET WITH SIT SECTOR CONTAINING
     POINTER"
0060 LET C$=CRC(K$),H$=HSH(K$,C$)
0070 LET S=MOD(DEC($00$+H$),N),C=DEC($00$+C$)
0080 IF R>32767 THEN GOTO 0110
0090 IF S=0 THEN LET P=2*MOD(C,507)+11 ELSE
     LET P=2*MOD(C,511)+1
0100 STOP
0110 IF S=0 THEN LET P=3*MOD(C,335)+16 ELSE
     LET P=3*MOD(C,341)+1
0120 STOP
```

Hashing is performed only when the input/output directive includes the KEY= option, i.e., when records are accessed randomly.

RANDOM ACCESS

The key specified by the KEY= option is first padded to the defined key size with nulls. The hashing algorithm described in the preceding subsection is applied to the key, and the calculated slot in the SIT is examined.

Key Not Found

If the calculated SIT slot contains nulls, the specified key is not in the file. The action then taken by the operating system depends on the I/O directive used.

READ/EXTRACT	- The system searches for the next higher logical key in the file, updates the next key pointer to the index of that key, and sets an error condition to notify the user that the specified key was not found.
FIND/REMOVE	- The system immediately sets the error condition to notify the user that the specified key was not found.
WRITE	- The system first determines the logical location of the new key in the sorted chain by searching for the key closest in value to, but less than, the key specified in the KEY= option. The physical location of the new key depends on the contents of the file header contained in the first sector of the SIT. If possible, the new key is placed in the slot occupied by the last removed key, the index of which is in the first field of the header. If no removed slots are available, the system places the key in the slot pointed to by the second field in the header. If n£ slots are available, an error is issued.

After the logical and physical locations of the new key are determined, the operating system updates the file. This requires several disc accesses.

The new key is inserted in the key area, and the appropriate forward pointers changed to insert it in the sorted chain. The index (relative physical position) of the new key is inserted in the SIT.

Finally, the SIT file header is updated to reflect the addition of the new key. The data record is not written until the key has been successfully inserted into the file.

Key Found

If the calculated SIT slot does not contain nulls, the key pointed to by the index in the SIT is examined. If the specified key does not match the key in the key area, the duplicate SIT pointer (the second pointer appended to the key) is examined.

If the duplicate SIT pointer contains nulls, the key is not in the.file. Otherwise, the key pointed to by the dulicate SIT pointer is compared to the specified key. This process is repeated until the end of the duplicate SIT chain is reached (key not found) or the specified key is found.

If the key is found, the operating system takes action based on the I/O directive used.

READ/FIND	- The system updates the next key pointer to the index of the next higher logical key in the file (the value of the forward pointer appended to the specified key). The data record is then read.
EXTRACT	- The data record is read; the next key pointer is not updated.
WRITE	- If the WRITE statement includes the DOM= option, an error condition is set to notify the user that the specified key is already in the file. If the DOM= option is not used, the system overwrites the existing data record.

REMOVE - The system first searches for the key closest in value to, but less than the specified key. The forward pointer in this key is then replaced with the pointer appended to the specified key, so that the specified key is eliminated from the sorted chain. In addition:

- The slot occupied by the specified key is filled with FF's.
- The index of the next removed key is placed in the second pointer area of the key slot, so that all removed entries are chained together.
- The SIT slot, or appropriate duplicate SIT pointer, is filled with nulls.
- The file header is updated to reflect the deletion of the key.

After the key is successfully removed, the associated data record is filled with nulls.

SEQUENTIAL ACCESS The READ directive always causes the next key pointer to be advanced, regardless of whether a key is specified. This feature allows records to be accessed in sorted order.

The system reads the key pointed to by the current file index, replaces the current file index with the forward pointer in that key, and reads the record associated with the key. The SIT is not accessed.

PROGRAMMING NOTES

Execution of the DIRECT statement causes the key area and SIT to be intitialized; the data area is not altered. Therefore, if a Direct file is accidentally erased, it cannot be redefined using a DIRECT statement.

NOTE

It is strongly recommended that the key be included as part of the data record so that the Direct file can, if accidentally erased, be recreated by READing it as an Indexed file and WRITEing a new Direct file.

On Level 4, the key area of large Direct files is initialized as a large removed chain. This organization results in improved performance for files whose keys have been written in acsending collating sequence.

EXAMPLES

READ Sequential

```
0010 OPEN (1)"FILE"
0020 READ (1,END=0040)A$
0030 GOTO 0020
0040 PRINT "END OF FILE"
0050 END
```

Examine Keys

```
0010 OPEN (1)"FILE
0020 LET K$=KEY(1,END=0050)
0030 READ (1)
0040 GOTO 0020
0050 PRINT "END OF FILE"
0060 END
```

READ Random

```
0010 OPEN(1)"FILE"
0020 READ (1,KEY=K$,DOM=0040)A$
0030 STOP
0040 PRINT "KEY NOT FOUND"
0050 REM "PRINT THE NEXT HIGHER KEY"
0060 LET K$=KEY(1,END=0080)
0070 PRINT K$
0080 END
```

WRITE Random

```
0010 OPEN(1)"FILE"
0020 WRITE (1,KEY=K$,DOM=0040)A$
0030 STOP
0040 REM "DON'T OVERWRITE DATA"
0050 PRINT "KEY ALREADY IN FILE"
0060 END
```

SORT FILE

OVERVIEW

A Sort file is a Direct file that has no data records; it consists only of a SIT and key area.

A Sort file can be used to effect different sort sequences for Direct of Indexed files.

Examples:

```
0010 REM "BUILD 'CMAST' SORTED ON CUSTOMER NO."
0020 REM "BUILD 1CNAME' SORTED ON CUSTOMER NAME"
0030 OPEN (1)"CMAST"
0040 OPEN (2)"CNAME"
0050 INPUT (0,ERR=0050)"CUSTOMER NO.: ",N:(999999)
0060 IF N=0 THEN STOP ELSE LET N$=STR(N:"000000")
0070 INPUT (0,ERR=0070)"NAME: ",C$:(LEN=5,20)
0080 LET K$=C$(1,5)+N$
0090 WRITE (1,KEY=N$)N$,C$
0100 WRITE (2,KEY=K$)
0110 GOTO 0050
```

```
0010 REM "READ 'CMAST' IN CUSTOMER NAME SEQUENCE"
0020 OPEN (1)"CMAST"
0030 OPEN (2)"CNAME"
0040 LET K$=KEY(2,END=80)
0050 READ (1,KEY=K$(6,6))N$,C$
0060 READ (2)
0070 GOTO 0040
0080 END
```

PROGRAMMING NOTES

Execution of the SORT statement causes the key area and SIT to be initialized. Therefore, if a Sort file is accidentally erased, it cannot be redefined using another SORT statement.

The KEY function must be used to access previously written keys in a Sort file.

I/O directives must not specify any data fields.

UNLINKED FILE (LEVEL 4)

OVERVIEW

The unlinked file has the same general structure as a Direct or Sort file except that the forward pointer appended to each key is filled with nulls.

ACCESS

Because the keys are not linked together in sorted order, unlinked files can only be accessed randomly by key.

PROGRAMMING NOTES

The overhead of maintaining the sorted keys in Direct or Sort files is substantial. If sequential access is not required by the application, the Unlinked file provides the convenience of random access by key in addition to greater speed.

All I/O directives must specify the KEY= option.

Use of the KEY function with Unlinked files produces unpredictable results.

Unlinked files must be defined via the file definition utility program. An Unlinked file can be converted to a Linked file (and vice-versa) through use of a utility program, (see the Level 4 Utilities User's Guide, BFISD 5084).

DISC DIRECTORY

OVERVIEW

The disc directory is a special form of the Sort file. The analogies between the two file types are summarized as follows:

SORT FILE	DIRECTORY
File name	Disc name (sector 1)
FID	Disc header (sector 1)
Insert new key	Define a file
REMOVE	ERASE
READ randomly	OPEN, LOAD, RUN, CALL, etc.

Each key slot ing the directory is 24 bytes long. The first 20 bytes contain the file ID (FID). Appended to the FID are the forward and duplicate SIT pointers. Because the first 3 bytes of the FID contain the starting sector number of the file, the keys are sorted in order of disc location.

ACCESS

The directory can be accessed both randomly by key and sequentially in sorted order. As with all file types, the user must OPEN the directory (using the disc name) before the contents of the directory can be accessed.

DIRECTORY OPERATION

The major difference between directory operation and Sort file operation occurs when a key is accessed randomly. In a Sort file, the hashing algorithm is applied to the entire key (excluding pointers). When randomly accessing a key in the directory, the system applies the hashing algorithm to the file name only.

OVERVIEW

This section discusses errors and the methods of error handling.

Error conditions are classified into two types: Catastrophic and Non-Catastrophic.

NON-CATASTROPHIC ERRORS

Non-Catastrophic errors are those which do not cause damage to files or to the disc.

Non-Catastrophic errors should be placed under program control through use of the ERR= and/or DOM= options, the ERR variable, the ERR function, or the SETERR directive.

NOTE

The ERR variable always reflects the value of the last error until a new error occurs or a "reset" operation is executed (BEGIN, END, STOP, CLEAR, LOAD OR RESET)

When an error occurs, if the ERR= option has not been used and no SETERR is in effect, an error message is displayed on the user terminal in one of the following forms:

```
Level 3  -  !ERROR=nn

Level 4  -  !ERROR=nn
            error message
```

where:

nn	=	a number identifying the type of error that has occurred
error message	=	a short message describing the error

The statement causing the error is printed directly below the error number and/or message, and the system enters Console Mode.

The proper procedure is to correct the error as necessary, then type "RUN" to continue.

If it is necessary to continue the program at a different statement, enter the following:

```
GOTO n
```

where:

n = the number of the statement to be executed

Then enter RUN.

The ERR (Code 1, Code 2, Code 3,...,Code n) function assists in determining which error occurred. The ERR function generates an integer which can be used in an ON/GOTO statement to construct a multiple branch. See ON/GOTO in Section 4.

CATASTROPHIC ERRORS

Catastrophic errors (Error 100 series; e.g., ERROR 103) occur during execution of certain statements which require more than one disc WRITE to complete. If an unrecoverable disc error occurs before the successful completion of all required WRITEs (except for data records), an appropriate disc error code (103 or 104) is issued. Issuance of an ERROR 100 type diagnostic returns the task to Console Mode. ERR options are not taken.

If an ERROR 100 diagnostic appears, the user must assume that either the disc directory or Direct file Key Area has been written incorrectly, and take the appropriate action. The type of error must be determined and corrected before proceeding (see ERRORS 3 and 4, which correspond to 103 and 104).

In most cases, correction of a Series 100 error requires contact with a Marketing Service Representative.

Following is a list of BASIC statements for which ERROR 100 diagnostics are issued:

SAVE (when defining a file)

INDEXED

SERIAL

DIRECT

PROGRAM

SORT

ERASE

WRITE OR WRITE RECORD (Direct file, and only if a new key is created)

REMOVE

NOTE

SAVE (an existing program) and data record WRITES and PUTs are not classified as potential ERROR 100 candidates

The following subsection defines the causes of error codes generated by the system. The error codes are listed in numerical order. The paragraph title for each code illustrates the format in which the error code (along with the message on Level 4) appears on the terminal.

When an error message displays, turn to the following subsection and locate the error. Then, review the list following that error until the cause of the problem is found. In some cases, correcting action is suggested, while in others, the procedure is obvious; e.g., an ERROR 21, INVALID STATEMENT NUMBER, results from the statement:

```
>LIST 99991
```

Correcting action in this case is the reentering of the statement with the proper statement number, which cannot be greater than 9999.

ERROR CODES

This subsection describes error codes and what they mean.

!ERROR=0 FILE/RECORD/DEVICE BUSY OR INACCESSIBLE

This error occurs (usually after a few seconds delay) when an attempt is made:

1. To access a peripheral device (printer, tape, etc.) that is not in the "ready" state. To correct, ready the device being accessed, e.g., make sure the printer is powered up and on-line.

2. To DISABLE a disc on which there is an open file. To correct, close all OPEN files.

3. to DISABLE a disc which is already DISABLEd by another user. Do an ENABLE of the affected disc from the task that DISABLEd it.

4. To ERASE an open file. Do an END on all active terminals.

   To access a record which has been EXTRACTed by another user. To correct, release record from extract by one of the following:

   a. Perform another operation on the file which has the record extracted (same user).

   b. Enter END on all other active terminals.

6. To OPEN a file that has been LOCKed by another user. To correct, the file must be CLOSEd or UNLOCKed by the user who LOCKed the file.

7. To LOCK a file already OPENed by another user. To correct, the file must be CLOSEd by the user that OPENed the file.

8. By a non-ghost task to write to a ghost task which has not done an INPUT. To correct, synchronize the logic so that complementary FUNCTIONS are always performed together in ghost and non-ghost tasks trying to communicate.

9. A time-out has occurred between terminal entries where the TIM= feature was set to some number of seconds. To correct, either set TIM= to a larger value, or instruct the operator to be more prompt.

10. To START a task which had already been STARTed.

11. To START a terminal or ghost which has been OPENed by another task.

This error occurs when an attempt is made to:

1. READ a record with a missing field terminator. To correct, check the possiblity of attempting to read more fields than have been written.

2. WRITE a record which would cause overflow of the record size defined. The record size must allow for field terminators. For example, if a file is defined with a record size of 40, an attempt to WRITE to the file with a single-field record of size 40 (or greater) causes an ERROR 1 because of the field terminator. To correct, reduce the size of the record being written.

3. Execute any input or output statement which specifies a number of variables greater than the number of field terminators received.

4. PRINT beyond the configured line length (on a printer).

5. WRITE beyond the end of file, when using the ISZ= option, and the last record's size is less than the ISZ= value. (This is the case if the ISZ= value is not an integer divisor into the file size.)

This condition occurs when an attempt is made:

1. To READ/WRITE to a record using an IND value greater than the total number of records defined. To correct, redefine the IND of the READ/WRITE statement or enlarge the file.

2. To WRITE a greater number of records than are defined. To correct, define a new file using a new name and with a number of records greater than the current value. Then transfer the data from the old file to the new one.

3. To sequentially READ past the highest indexed record or the highest key. To.correct, enter an END= option in the READ statement.

4. To READ past the end-of-file mark on Magnetic Tape. To correct, enter an END= option in the READ statement (also see ERROR 72).

5. To use the KEY or IND function when the last record in the file has been read. To correct use an END= option.

6. On SERIAL files, to READ or WRITE a record larger than fits in the remaining file space.

7. To READ or WRITE a file opened with an ISZ= beyond the last record of a file. No error is given when attempting to READ or WRITE the last record of the file, even if it is smaller than the ISZ= value. To correct, adjust the ISZ= option.

8. By a non-ghost task to READ from a ghost task which is not in output mode.

9. To MERGE an Indexed file with no END statement, and a PROGRAM statement is in the last record position of the file.

10. To print to a spool file which is filled.

11. To READ a Serial file when the last access was a WRITE.

!ERROR=3
DISC READ ERROR

This error can indicate damage, drive misalignment, or faulty disc data recording. The error can occur repeatedly when attempts are made to access data from a damaged disc. The error can also result from electronic malfunctions, or from running the disc under extreme temperatures.

NOTE

To aid the programmer, the display of an ERROR 3 includes the following information, in addition to the statement content:

```
DSC=discno   SEC=secno   STS=status
```

where:

discno	=	the number of the disc that was accessed
secno	=	the first of the one or more sectors accessed
status	=	the status of the disc drive as determined by the system. For further information contact a Service Representative

There are essentially three reasons why an ERROR 3 occurs:

a. The record was incorrectly WRITEn on the disc.

b. The record was incorrectly READ from the disc.

c. A data error occurred in the disc controller.

If an ERROR 3 occurs, call a Service Representative.

!ERROR=4
DISC NOT READY

This error occurs when an attempt is made to:

1. Define the use of any disc within a DIRECT,INDEXED, SERIAL, PROGRAM, SORT, SAVE, GET, or PUT statement using a configuration which specifies a greater number of disc drives than are physically included in the system. To correct, DISABLE disc numbers in excess of those available for use.

2. Use a disc drive which is not in a "ready" condition. To correct, turn the drive on.

3. Use an inoperative disc drive unit. To avoid/correct an ERROR 4 occurrence, do not use the inoperative disc drive unit; have it repaired, or DISABLE the drive.

4. WRITE to a disc with the READ ONLY switch ON. To correct, turn the READ ONLY switch OFF.

5. Open a file with no disc in the drive and the drive ENABLEd.

NOTE

To aid the programmer, the display of ERROR 4 includes the following information in addition* to the statement content:

```
DSC=discno     SEC=secno      STS=status
```

where:

discno	=	the number of the disc that was to be accessed
secno	=	the sector number of the first sector being accessed
status	=	the status of the disc drive as determined by the system. For further information, contact a Service Representative.

NOTE

A short delay takes place before the ERROR 4 occurs. This is to handle cases where the drive temporarily drops out of the "ready" state.

!ERROR=5
PERIPHERAL DATA
TRANSFER ERROR

This error occurs when:

1. A parity error occurs upon transmission to or from a terminal. A persistent error is indicative of a device malfunction.

   An invalid character is read from an input-output device. It can result from faulty storage media such as a damaged magnetic tape, or device malfunction.

3. An interrupt from the CPU front panel or a power failure occurs during terminal access.

4. A remote printer has a protocol error, or the ACK/NAK sequence is not correct due to transmission problems.

If an ERROR 5 repeatedly occurs, call a Service Representative.

!ERROR=6
INVALID DISC
DIRECTORY OR
NON-CERTIFIED TAPE
CARTRIDGE

This error occurs when the system detects an invalid directory, or no directory, on an ENABLEd disc in the configured system (when for example, defining a file), or a disc or mag tape being READ, is formatted incorrectly. Use of an uncertified tape cartridge can also cause this error (not applicable to reel-to-reel magnetic tapes).

!ERROR=7 SECTOR POINTER OUT OF RANGE	This error occurs when an attempt is made to: 1. Reference sector zero or a sector number above the highest available sector by means of a GET or PUT statement. The reference to the unavailable sector can be made directly by means of the secno (sector number) field of the statement, or indirectly, by establishing a string length which would require the use of unavailable sectors. To correct, change the secno field, or the length of the field referenced by the statement. 2. Access a keyed file which has an out-of-range data pointer. To correct, the file must be copied to a new file to reestablish the pointers.
!ERROR=8 DISC WRITE ERROR/DATA MISCOMPARE	This error occurs when the system is unable to verify correct recording of data on disc. The verification operation consists of reading the data from disc and comparing it with the original data written to disc (PUT), or reading the data from disc twice and comparing the resultant strings (GET). Verification is available as an option on GET/PUT statements by use of the verify string variable.

The ERROR 8 display includes the following information:

```
DSC=discno           SEC=secno
```

where:

discno	=	the number of the disc to/from which data was WRITEn/READ
secno	=	the first sector on/from which data was WRITEn/READ

NOTE

ERROR 8 is a catastrophic error. Persistence of this error is indicative of disc or memory malfunction. All file definitions and key linkages should be examined for integrity. If the error persists, call a Marketing Service Representative.

!ERROR=9 POWER FAILURE	This error occurs when the system power source is subject to external power fluctuations or surges during an input/output operation. Under such conditions, the system completes disc accesses and then terminates CPU operation. When the line voltage is again normal, all peripherals are brought through a system-controlled power-up sequence, the error indication is generated, and the system goes to Console Mode (unless it was doing a disc READ or WRITE - in which case it simply repeats the disc operation and continues, with no indication of a power failure). If a disc model 2500 key search was in progress at the time of power failure, an ERROR 9 is reported in Console Mode and the statement must be re-executed. No correcting action is required to continue operation.

NOTE

An ERROR=9 displays on the VDT for all active tasks in Console Mode, but can be transparent to tasks in Program Mode.

If an input/output statement is being executed at the time of the power failure, ERR options are taken - or if no ERR option is specified, the error message displays, and the task returns to Console Mode. If no input/output statements are involved, the active task continues to run after the power-up sequence is complete with no apparent indication of the error to the user.

<u>Power Fail Recovery For Task Performing Disc I/O</u>	Any task performing a disc I/O operation is suspended for a maximum of 3 minutes until the disc drives become ready again. If the disc is not ready after 3 minutes, an ERROR 4 is generated.
	If the disc does not become ready before the successful completion of one or more, but not all, WRITES required in a disc output command, an ERROR 104 is generated. This error forces the task into Console Mode, indicating the disc directory or Direct file could have been destroyed.
<u>Power Fail Recovery For Tasks Not Performing Disc I/O</u>	All tasks in Console Mode immediately generate an ERROR 9 on power fail recovery. Other tasks performing I/O can generate an ERROR 5 if a transmission error occurs.

!ERROR=10
ILLEGAL FILE NAME
SIZE OR USAGE/
ILLEGAL OVERLAID CALL

This error occurs when:

1. More than six characters are specified as a file identification field of a INDEXED, SERIAL, DIRECT, PROGRAM, SORT, OPEN, ERASE, SAVE, LOAD, CALL ADD or RUN statement. The file identification field must not contain more than six characters.

2. The argument of a KEY function is not included, or the argument field is longer than the defined key size. To correct, adjust the KEY function argument.

3. On Level 4, an attempt is made to overlay a CALL with no valid CALLing program in memory.

!ERROR=11
MISSING OR DUPLICATE
KEY

This error occurs when an attempt is made to access a record of a Direct file using a KEY whose value is not equal to the key defined for any record of the file.

After taking the DOM= option on a WRITE statement, the ERR variable is set to 11.

!ERROR 12
MISSING OR DUPLICATE FILE NAME/NON-CONFIGURED DEVICE

This error occurs when an attempt is made to:

1. OPEN or ERASE a disc data file using a file identification field that has not been previously defined on an available disc by means of a DIRECT, INDEXED, PROGRAM, SORT, SERIAL, FILE or SAVE statement.

2. OPEN or ERASE a file that resides on a DISABLEd disc.

3. OPEN an input/output device not included in the configuration.

4. Define a disc data file or program by means of a DIRECT, INDEXED, SERIAL, PROGRAM, SORT or SAVE statement when a file of the'same name already exists on an available disc.

5. Define a disc data file or program by means of a DIRECT, INDEXED, SERIAL, PROGRAM, SORT or SAVE statement where the file name is the same as the name reserved for a system device (i.e., LP, P1, P2...P7, TO, T1...TF, M0, M1, G0...G3, SY) or the disc name.

6. ADD or DROP a program that is not found.

!ERROR=13 IMPROPER FILE OR DEVICE ACCESS	This error occurs when an attempt is made to

1. READ or INPUT on an output-only device such as a printer.
2. WRITE or PRINT on an input-only device.
3. WRITE or PRINT to a Direct file when the statement does not include an INDex or KEY option and the subject record is not currently EXTRACTed.
4. READ or INPUT a disc data file using a statement that contains a constant or mnemonic.
5. WRITE to mag tape without write ring; or to specify an illegal I/O.
6. WRITE to Serial file, or WRITE to a file using the BLK= option, if the file is not LOCKed.
7. Access a ghost program from a non-ghost program (or vice versa) when both programs are in the same mode (i.e., input or output) at the same time.
8. Access a lower index than previously accessed on a Serial file (attempting to move backwards in the file).
9. ADD a non-program file to the Public Dictionary.
10. DROP a peripheral device or a nonresident program.
11. Access a key in an Indexed file via the KEY function.
12. WRITE to a WRITE PROTECTed magnetic tape.
13. Use IND= with a WRITE RECORD.
14. RELEASE a disc file or a task-tied (OPENed) ghost.

!ERROR=14 IMPROPER FILE OR DEVICE USAGE	This error occurs when an attempt is made to:

1. OPEN a device that is in use (previously OPENed).
2. OPEN a disc file or device using a file/device number that is currently being used by that user.
3. START with an illegal device ID.
4. Perform an input/output operation using a file/device number that was not previously used in an OPEN statement by the same user.
5. Define a disc data file, or program, by means of a DIRECT, INDEXED, SERIAL,'PROGRAM, SORT or SAVE statement on a disc that was previously DISABLEd.
6. DISABLE a disc that was previously DISABLEd.
7. ENABLE a disc that was previously ENABLEd.
8. LOCK a file that has not been OPENed by the same user.
9. LOCK a file that has already been LOCKed by the same user.
10. UNLOCK a file that has not been LOCKed, or which has been OPENed with an ISZ= option.
11. ADDR a program already in the resident Public Programming Dictionary.
12. START a device (rather than a terminal or ghost task).
13. Perform an input/output operation that is not valid for magnetic tapes.
14. RELEASE a task which has not been STARTed.

!ERROR=15 DISC SPACE OCCUPIED BY ANOTHER FILE	This error occurs when an attempt is made to define a data file or program to the disc directory using a DIRECT, INDEXED, PROGRAM, SORT, FILE, SERIAL or SAVE statement that specifies sectors currently allocated to another file or program. To correct, change the sector specification.
!ERROR=16 DISC OR PUBLIC PROGRAMMING DIRECTORY IS FULL	This error occurs when: 1. An attempt is made to define a disc data file or program using a DIRECT, INDEXED, PROGRAM, SORT, FILE, SERIAL, or SAVE statement when the capacity of the disc directory has been reached. To correct, ERASE unneeded data files and/or programs. 2. There is an overflow of the dictionary (directory cache on Level 4). To correct, DROP unneeded Public programs from the dictionary, if possible, and CLOSE files which are not needed.

!ERROR=17 INVALID PARAMETER/ NON-CONFIGURED DISC	This error occurs when an attempt is made to:

1. Use a GET or PUT when the first variable name is the same as the verify variable name.
2. Use a disc number (1-7) not recognized in the system configuration in a SAVE, PROGRAM, INDEXED, SERIAL, DIRECT, SORT, FILE, ENABLE, DISABLE, or RESERVE statement (disc numbers outside the range of 0-7 generate an ERROR 41).
3. Use a GET or PUT when the length of the first variable name is not the same as the verify variable.
4. SAVE or LOAD a non-program file, or RUN a Direct, Sort, Serial, or an Indexed file.
5. Perform an input or output operation to a PROGRAM file.
6. LIST or MERGE from anything other than an Index or Serial file or device.
7. Use AND, IOR or XOR functions with different length arguments.
8. Specify an invalid value for a magnetic tape I/O option, or specify an undefined string variable.
9. Execute a FILE statement with bad file parameters.
10. SAVE, without parameters, a null program area.
11. Use a KEY= option on an Indexed file.
12. Access a DIRECT file record which has a bad link.

!ERROR=18 ILLEGAL CONTROL OPERATION	This error occurs when an attempt is made to: 1. DROP a program that is busy. 2. SAVE an ADDed program. 3. START the communications driver (OS2780) in a bank other than 1. 4. SAVE to an OPENed file. 5. Access a "protected" program (LIST, SAVE, PGM function, etc.) (Level 4). 6. CALL a program more than 127 times (recursively).
!ERROR=19 PROGRAM SIZE IS LARGER THAN DEFINED SIZE	This error occurs when an attempt is made to: 1. SAVE a program that consists of a greater number of bytes than specified for the program in the length field of the PROGRAM or SAVE statement used to define the program file. 2. LOAD or RUN a program with insufficient data space.

!ERROR=20
STATEMENT SYNTAX

ERROR 20 is a general catch-all error for the compiler. Illegal punctuation, non-existent or misspelled directives and incorrect syntax are just some of the causes of an ERROR 20.

In addition to compiler errors, several other situations can cause an ERROR 20. This error can occur when an attempt is made to:

1. Enter a command which is not in the minimal compiler's instruction set, and the full compiler is not configured.

2. Execute a statement that has a format mask with illegal characters.

3. Execute an EDIT statement that has an illegal parameter option.

4. Execute a command reserved for the control task only; e.g., RELEASE "T3".

5. RELEASE SCT.

6. Enter or execute an I/O statement that contains a key function. For example:

```
>0010 PRINT (1,KEY=K$)
```

7. Use a second argument on a CRC or HSH function whose length is not equal to 2.

8. Execute a user-defined fuction reference (FNx) where the FNx argument list does not match the DEF argument list.

!ERROR=21 INVALID STATEMENT NUMBER	This error occurs when an attempt is made to:

1. Enter, during Console Mode operation, a statement whose directive is preceded by a statement number greater than 9999 or less than 1.

2. LIST or DELETE a statement number greater than 9999 or less than 1.

3. MERGE a statement whose directive is preceded by a statement number greater than 9999 or less than 1.

4. Enter or execute an EDIT, GOSUB, GOTO or ON/GOTO statement with a branch statement number greater than 9999 or less than 1.

5. Enter a statement that contains an IOL=, ERR=, TBL=, DOM=, or END= which specifies a statement number greater than 9999 or less than 1.

6. Execute an EDIT or DELETE statement on a non-existent statement number.

7. Add to a nonexistent statement number by use of the Console Mode editing feature,

8. Execute a CPL function on a text string which has no statement number.

!ERROR=23
MISSING VARIABLE/
NON-DIMENSIONED
STRING

This error occurs when an attempt is made to:

1. Enter or execute a statement whose structure implies the absence of a variable, for example:

```
0010 *ERR 23
0010 FOR5=1TO10

0010 *ERR 23
0010 FORITO
```

2. Use a GET or PUT with a string variable with a length of zero (undefined).

!ERROR=24
DUPLICATE FUNCTION
NAME

This error occurs when an attempt is made to establish a user-defined function by means of a DEF statement, when the function name within the statement has been previously defined.

!ERROR=25
UNDEFINED FUNCTION

This error occurs where an attempt is made to execute a statement containing a user-defined function (FNA through FNZ) that was not previously defined by a DEF statement in the user's program, or which was defined for a different function (e.g., FNA reference to a DEF FNB).

!ERROR=26
INCORRECT VARIABLE
USAGE

This error occurs when an attempt is made to execute a function of any kind where the argument is of an incorrect mode (i.e., where the argument is a string and should be numeric, or where the argument is numeric and should be string). The error occurs when an attempt is made to:

1. Enter more than 14 digits, or enter a non-numeric character at a terminal in response to an INPUT statement whose expression field specifies a numeric value.

2. READ non-numeric data from a file into a numeric variable. The error is usually indicative of a READ statement in which the type and order of variables do not correspond to the type and order of variables in the WRITE statement used to create the file.

3. Enter or execute a statement or function where the type of variable (numeric or string) defined by the argument is in disagreement with the type of variable implied by the statement or function name.

4. Specify a string or string variable as an INDex, or specify a number or numeric variable as a KEY.

5. GET a numeric variable.

6. Take the NUM of a non-numeric character.

7. Convert non-hexadecimal characters via the ATH function.

!ERROR=27
RETURN WITHOUT
GOSUB/DELETE
WITH ACTIVE GOSUB
OR FOR/NEXT

This error occurs when an attempt is made to:

1. Execute a RETURN without a previously executed GOSUB. This is indicative of an error in program logic.

2. Execute a RETRY without an ERROR branch resulting from a SETERR or ERR=.

3. Execute an EXITTO with neither a GOSUB nor a FOR statement previously executed.

4. Execute an EXIT from the main (not Public) program.

5. DELETE or EDIT a statement in a program with an active FOR-NEXT or GOSUB-RETURN routine.

!ERROR=28
NEXT WITHOUT FOR

This error occurs when an attempt is made to execute a NEXT without execution of a previous, corresponding FOR.

!ERROR=29
UNDEFINED MNEMONIC

This error occurs when an attempt is made to enter or execute a statement containing a peripheral device, mnemonic constant or a positioning expression (e.g., (@ 10,10)) on a printer that is not recognized as valid.

!ERROR=30
USER PROGRAM
INCORRECT CHECKSUM

This error occurs when an attempt is made to:

1. LOAD, CALL, LIST, ADDR or RUN a program with an incorrect check field (HSH).

2. Perform a LST function on an invalid string.

3. EXIT back to an overlaid program which has been modified (Level 4).

!ERROR=31
INSUFFICIENT MEMORY
WITHIN TASK

This error occurs when an attempt is made to:

1. ENTER or MERGE a statement which, if added to the program, would make the program too large to fit in the available user area. To correct, see number 2 below.

2. EDIT an existing statement to increase its length to the extent that the additional program area required would make the program too large to fit in the available user area. To correct:

   a. SAVE the program, enlarge the user area, LOAD the program and continue; or

   b. Split the program and add statements to initiate overlay; or

   c. Reduce the size of the existing program to provide space for the coding to be included.

3. Execute a program whose operation has filled the user area. The specific action that caused the error is usually the addition of a new variable or the lengthening of an existing variable. In either case, it is likely that a loss of data has occurred and the program must be reRUN following corrective action. It is possible that the error was due solely to failure to CLEAR the user data area prior to the execution of the program.

4. Execute string manipulations within a program which temporarily require more data area than is available. After the error occurs, the data area is returned to the size remaining prior to the string manipulation.

5. Enter a statement via a terminal keyboard when the user area is almost full (this is a less common cause for the error). In this instance, the error results from the fact that all Console Mode keyboard entries are stored in a buffer within the user area prior to processing of the carriage return terminator. To correct:

   a. Enlarge the size of the user area using the START command.

b. CLEAR the data area (if possible) prior to execution (or revision) of the program. If CLEARing is not appropriate, select one or more unnecessary string variables of sufficient length and set their values to null; or

c. Modify the program so that less data is required (e.g. remove REM statements).

6. LOAD or ADDR a Program file that has non-valid data.

7. Execute a CALL statement with insufficient data space available to store the CALL stack information.

8. OPEN a tape unit (MO, CO, etc.) with insufficient data space to hold the driver (MTR, MTC and Word Processing printer drivers are brought into individual user areas when OPENed).

!ERROR=32 (HARDWARE) STACK OVERFLOW	This error occurs when hardware register storage requirements of multiple peripheral device interrupts exceed the capacity of the hardware "stack". In normal operation, the stack is sufficient to accommodate all interrupts. Therefore, the occurrence of this error indicates a recurring hardware (peripheral device or controller) malfunction. However, it can be caused by an attempt to compile a statement with a large number of parentheses, or with nested functions.

Usually, an ERROR 35 serves as an indicator that stack overlow is imminent (see ERROR 35).

This error can also occur on an IOLIST statement that loops on itself.

For example:

```
0010 PRINT IOL=0020
0020 IOLIST IOL=0030
0030 IOLIST IOL=0020
>RUN
```

!ERROR=33 INSUFFICIENT MEMORY CAPACITY	This error occurs when an attempt is made to execute a START or CALL statement requesting allocation of processor memory when the available (unused) memory is less than that required to satisfy the request. To correct: a. Reduce the size of the user area requested by the START statement; or b. RELEASE any terminals not in use; or c. Reduce the amount of memory reserved for the programs and data of other users. If ERROR 33's appear on a regular basis, it may be advisable to purchase more memory.
!ERROR=34 VDT BUFFER OVERFLOW	This error is caused by the inability of the CPU to keep up with the VDT transfer rate. To correct: 1. Increase input buffer size (Level 4 only), if possible (max 255); 2. Reduce overall system loading, if possible, by temporarily stopping other tasks. 3. Slow down input; or 4. Design an application program such that INPUT statements are executed more frequently.

!ERROR=35 PARENTHETIC EXPRESSION LIMIT	This error occurs when stack overflow is imminent due to complex arithmetic (or logical) expressions being executed (with complexity directly related to the number of parentheses within the expression). To correct repeated occurrences: 1. Simplify the arithmetic (or logical) expression by the elimination of the parentheses; or 2. Divide the arithmetic (or logical) expression into two or more parts and include each part in a separate statement.
!ERROR=36 CALL/ENTER VARIABLE MISMATCH	This error occurs when: 1. The number of variables or the mode of the variables are not consistent between CALL and ENTER statements. 2. ENTER is executed more than once in a CALLed program. 3. An attempt is made to execute ENTER in a main (not public) program.

!ERROR=37 INVALID FUNCTION	This error occurs when an attempt is made to execute an unsupported function.
!ERROR=38 ILLEGAL COMMAND IN PUBLIC PROGRAM	This error occurs when an attempt is made to: 1. Execute one of the following commands in a Public program: EXECUTE LIST RUN ESCAPE DELETE MERGE SAVE 2. Use an undefined variable in a CALL parameter list. 3. DIMension an ENTERed variable. 4. Re-START from within a Public program (Level 4)
!ERROR=39 ESCAPE IN PUBLIC PROGRAM	This error occurs when ESCAPE is pressed in a public program.
!ERROR=40 NUMERIC VALUE OVERFLOW	This error occurs when an attempt is made to execute a statement involving arithmetic operations that result in an absolute numeric value less than $-10^{60}+1$, or greater than $10^{60}-1$. This excessive value can also result from an attempt to divide by zero. When this error occurs, previous arithmetic processes should be checked to determine if a zero value divisor was generated.

!ERROR=41
INVALID INTEGER
RANGE

This error occurs when an attempt is made to:

1. Enter or execute a statement using a negative value, fractional value, or too large a value to identify the following:

   a. A file ID or device ID (maximum 7 in Level 3, 8 in Level 4).

   b. A disc number (maximum 7).

   c. The number of records in a file (maximum 2^{23}-1 records).

   d. The record size (maximum 32767 bytes).

   e. A sector number greater than the HSA.

   f. An INDex or ISZ= value (maximum 2^{23}-1).

   g. At position @ (maximum 255).

   h. A subscript (range = 1 to 32767).

   i. A program size (maximum 32767 bytes),

   j. A PRECISION (maximum 14).

   k. An ON/GOTO statement whose expression field results in a value greater than 32K.

   l. A power (^)(maximum 255).

   m. A key size in a DIRECT or SORT statement (maximum 56).

   n. An increment length in a POS statement (maximum 32K).

   o. A Block size (BLK=) with a value other than 0, or 1024 bytes (sector size) (Level 3).

   p. A BNK=, PUB or TSK specification (max. bank=15).

   q. A START size where size is less than 3 on Level 3 or 4 on Level 4 or greater than 128.

   r. A SELECT value greater than 63 (and not 255)

   s. A BIN function length (max=32767)

2. Execute the CHR code conversion function of a value that is less than zero or greater than 255.

3. DIMension a numeric array that requires greater than 32K of memory (more than 4095 elements).

4. Enter a minimum or maximum LEN specification for input verification which is greater than 32K.

5. Close file 0.

!ERROR=42
NONEXISTENT NUMERIC SUBSCRIPT

This error occurs when an attempt is made to:

1. Execute a statement which contains an expression that references an undefined numeric array or a non-existent element of a DIMensioned numeric array. To correct:

   a. Define the numeric array using a DIMensione. statement that includes the referenced element; or

   b. Revise the coding that causes generation of an unexpectedly large variable that is used as the subscript.

2. Return a POS function with a length field of zero.

!ERROR=43 INVALID FORMAT MASK SIZE	This error occurs when an attempt is made to execute a PRINT or WRITE statement or a STR function that references a formatted numeric variable having more significant digits to the left of the decimal point than have been provided for in the format mask; or when the format mask contains invalid characters. To correct, redefine the format mask allowing sufficient positions to handle the larger number or correct format mask characters. NOTE If this error occurs on a WRITE or PRINT to disc, it results in the WRITEing or PRINTing of a partially complete record. The record is correct up to and including the field prior to the error field.
!ERROR=44 STEP SIZE OF ZERO	This error occurs during execution only and is caused by a STEP value (in either constant or variable form) of zero existing on the first execution of a FOR statement. Changing of a variable STEP value to zero during the execution of a FOR/NEXT loop does not cause an error, since the STEP value is set at the beginning of execution of the loop.

!ERROR=45
INVALID STATEMENT USAGE

This error occurs when an attempt is made to:

1. Enter a statement which is restricted to Console Mode only, including a statement number (indicating Program Mode).

2. Enter a DELETE or LIST command that references descending statement numbers.

3. Execute a statement with a TBL= option that references a statement number which is not a TABLE statement.

4. Enter a statement (EXECUTE, FOR, NEXT, GOSUB, RETURN or RETRY) in Console Mode which is available in Program Mode only.

5. Enter a statement with an IOL= option that references a statement which is not a valid IOLIST statement.

!ERROR=46
INVALID STRING SIZE

This error occurs when an attempt is made to:

1. Execute a statement whose KEY= field defines a key to a Direct data file whose length exceeds the key size inferred by the keysz field of the associated DIRECT statement.

2. Execute the ASC function with a null argument (string length = 0).

3. Enter other than eight characters with the SETDAY statement.

!ERROR=47
SUBSTRING REFERENCE
OUT OF RANGE

This error occurs when an attempt is made to:

1. Reference a string variable using subscript notation that is not within the range of the length of that variable.

   For example:

   ```
   >A$="ABCD"
   >PRINTA$(2,4)

   !ERROR=47
   ```

2. Reference a substring of an undefined string.

!ERROR=48
INVALID INPUT

This error occurs when an attempt is made to:

1. Input into a string variable when the branch list conditions are not met, and/or the LENgth of the data input is outside the range specified in the LEN= specification.

2. Input a numeric value when the number and/or value falls outside the range specified for verification in the input statement, or has too many fractional digits.

!ERROR=49
NON-TRANSLATABLE
STATEMENT

This error occurs when a non-translatable statement is encountered during the translation of a program from one level to another (used only by Basic Four translators and the renumbering utility *P).

!ERROR=50 GENERAL MEMORY ERROR	An ERROR 50 indicates that a problem exists in the operating system. If an ERROR 50 occurs, please call a Marketing Service Representative.
!ERROR=51 (<u>Level 3</u>) COMPILE OR LIST OPERATION WITHOUT COMPILER/LISTER	This error occurs when an attempt is made to compile or list a program while the Compiler/Lister is not resident in memory.
!ERROR=54 OPEN OF SERIAL FILE WITH INVALID HEADER	This error occurs on an attempt to open a Serial file with an invalid header.
!ERROR=55 TAPE CONTROLLER	This error occurs when the MTR/MTC controller returns garbled information on a READ operation. To correct, include RETRY logic in the statement.
!ERROR=72 END OF TRACK ON TAPE/UNEXPECTED ETX	This error occurs when the End of Tape (EOT) is reached on a magnetic tape.

!ERROR=103 CATASTROPHIC READ FAILURE/FILE POINTERS DAMAGED	A file (Direct or Sort) or directory has invalid key pointers due to a critical write operation that could not complete due to a disc error. The task is forced into Console Mode. To correct, identify the operation, but do not proceed until the file or"directory has been rebuilt. A RUN can appear to be successful, but could result in a serious error in the file or directory structure after the appearance of the error.
!ERROR=104 CATASTROPHIC DISC FAILURE/FILE POINTERS DAMAGED	An ERROR 104 occurs when an attempt is made to: 1. WRITE to a file when the 'READ ONLY' switch on the disc drive is on. 2. WRITE to a disc when there is a hardware malfunction.
!ERROR=123 CATASTROPHIC PARITY ERROR/FILE POINTERS DAMAGED	If a parity error occurs after a task begins updating a Direct, Sort or Serial file (or the directory), but before all WRITES are completed, the error is displayed, and the task is placed in Console Mode.

!ERROR=124 PARITY ERROR	If a parity error occurs before a task begins updating a file (or directory), or after the WRITES to the file (or directory) have been completed, the error is displayed and the task is placed in Console Mode.
!ERROR=126 (Level 4) CTL+Y KEY USED	Use of the CTL+Y operation can be captured in Level 4. This is <u>not</u> a catastrophic error. If SETCTL is not in effect, CTL+Y is ignored.
!ERROR=127 ESCAPE	The system variable ERR is set to the value 127 when the ESCAPE key is pressed.

APPENDIX A - FEATURES OF THE BUSINESS BASIC OPERATING SYSTEM

OVERVIEW

This section describes features of the Levels 3 and 4 operating systems as they pertain to the language. This section is not intended as a comprehensive guide to operating systems, and is included only as a convenience to the user.

BUSINESS BASIC OPERATING SYSTEM

The BASIC operating system provides common support functions for application programs, including Execution Scheduling, Peripheral Device Allocation and Control, File Management and Disc Control.

Execution Scheduling is the interpreting and executing of an application program, but with the additional dimension of sharing the computer's resources between other programs at the same time. The process is cyclic, in that control is returned to the Program Executive when the execution of a statement has been completed. The statement next selected for execution by the Executive is from another application program or task. This prevents monopolization of the central processor by any one task.

Peripheral Device Allocation and Control involves the processing of requests for use of peripheral devices by a task. It checks the device's availability, then assigns it to that task. It also handles all input/output operations on the device, providing a simplified, common interface to the application program.

File Management and Disc Control handles all aspects of the system's disc files. It executes the file definition statements, maintains the directory of file names, and handles all OPEN, CLOSE, READ, FIND, WRITE, EXTRACT, KEY, REMOVE, LOCK, UNLOCK, RESERVE, ENABLE and DISABLE statements. It also maintains the Scatter Index tables, and key chains for Direct and Sort files.

TASKS, TERMINALS, AND I/O DEVICES

In Business BASIC terminology, a "task" is a program or other activity, such as program development, that is running under the control of the operating system.

In the case of a program requested by a terminal operator for interactive use (such as data entry), the program can be loaded from the disc into a portion of main memory specifically assigned to that operator's terminal.

Alternatively, any terminal can use a "Public program" that is shared by several other operators. The Public program concept reduces main memory requirements, but places certain restrictions on any program that is used as a Public program. The activation of terminals, assignment of user task areas in memory, and the use of Public programs are described later in this section.

Some programs (such as report printing) do not require any action by an operator, and therefore do not require a video display terminal. Such programs can be activated as "ghost tasks" as described in this section under Ghost Tasks.

For each task, the operating system allows up to 8 (numbered 0-7) I/O devices or files on Level 3, or 9 (numbered 0-8) on Level 4. The operating system manages any conflicts between tasks that are competing for use of the I/O devices.

If the task is not a ghost task, its controlling terminal is automatically assigned file/device number zero; the terminal is automatically readied for use (OPENed); and I/O statements involving the terminal are not required to reference the file/device number (0), unless input/output options are used.

For other I/O devices to be used, the devices (LP for the first printer, Pn for additional printers, Mn for magnetic tape units, Tn for other terminals) must first be readied (OPENed), and must be assigned a file/device number that is used by the program for all communication with the device.

Similarly, to gain access to a file, the file must be OPENed and a file/device number must be assigned for communicating with the file.

The operating system manages conflicts between tasks competing for I/O devices and files by returning error codes when a task attempts to OPEN an I/O device that has been OPENed by another task, and when an attempt is made to READ a record that has been EXTRACTed by another task.

The operating system provides many other error code indications - codes representing operator problems, equipment problems, conflicts between tasks, and routine logical indicators (see ERROR Processing, Section 10).

COMPILER/LISTER

The compiler/lister functions automatically, without intervention by the programmer. The compiler portion compiles program statements into a more efficient form, requiring less storage space and less running time. The lister portion performs the reverse function, retrieving statements in a compiled format and listing them in a format similar to that originally entered by the programmer.

The compiler and lister are permanent parts of the operating system on Level 4, and cannot be DROPed or ADDed.

On Level 3 systems, the compiler and lister normally begin in memory as the first Public programs (this can vary depending on how the system start-up was programmed). Both can be removed by use of the DROP directive, provided all other Public programs that have been added to memory after the compiler and lister have been DROPped first.

If the lister has been DROPped, attempts to perform LIST, EDIT, or the LST function cause an Error 51. Further, with the Lister DROPped, the operating system does not display statements that are in error, nor traced statement, but instead displays the program name and statement number only.

Even when the compiler has been DROPped, a "minimal compiler" remains resident which accepts certain critical directives, such as END, BEGIN, ENABLE, RUN ("prog"), and CALL ("prog"). Thus, the compiler and lister can be returned to memory by running a program which contains the ADDC and ADDL directives.

JOB CONTROL AND MEMORY MANAGEMENT

When the Load Button is pressed, the bootstrap loads the loader, which loads the operating system. The operating system, in turn, checks available memory against the system's configuration, then initiates the operating system monitor program "OSMONR" on the configured system control task (SCT: TO or GO) with 30 pages of task memory on Level 3 and 32 pages on Level 4 (if the SCT is GO, then OSMONR is initiated with 10 pages on Level 3 or 25 pagees on Level 4).

"OSMONR" first loads the compiler/lister, then performs an interactive dialog on the system control task to establish requirements for the Spooling option (see Spooling in this section); and finally places TO in the "ready" state.

An operator at controlling terminal TO can then use the START command to change TO's memory allocation, to allocate memory to other tasks and terminals, and to start programs running on other tasks and terminals. Subsequent control of tasks, terminals, and programs involves the use of the RELEASE and START commands.

If GO has been started as the SCT, individual terminals are activated by striking of the ESCAPE key (GO moniters them surreptitiously).

In Level 3, "OSMONR" can be modified by the user to start task TO in Program Mode instead of Console Mode, and the first task performed by TO can be a program that activates other terminals in Program Mode, running application programs.

On Level 4, use the Start Up Control supplied with the Utility set to modify OSMONR (see the Level 4 Utilities Users Guide, BFISD 5084).

Using the START and RELEASE commands in programs run by TO, and RELEASE commands in programs run as other terminal tasks and ghost tasks, the system designer can create a comprehensive control scheme that schedules jobs and allocates system resources for maximum throughput.

Specialized functions of the SCT include the STARTing and RELEASEing of terminal tasks, RELEASEing of ghost tasks, and the DROPing of Public Programs (see Public Programming in this section) in Level 3.

After the loading sequence during start-up, only the SCT is active, and the START command must be executed from it, to activate the other terminals. For example, the statement:

```
START 30, "*A", »T1"
```

executed in either Console Mode or Program Mode, initiates the utility program "*A" at terminal "T1" in about 30 pages of user memory (there are 2-3 pages of overhead whenever a task is STARTed). But the statement:

```
START 30, "T1"
```

assigns about 30 pages of user memory to "T1", with no program initiated, and T1 is activated in Console Mode.

Once activated by a START command from the SCT, other terminals can use the START command to reassign memory to their task. Terminals other than the SCT can only START their own tasks and ghost tasks.

The activity of tasks and the availability of memory for new tasks can be monitored by using the following system functions:

BSZ	-indicates the number of unassigned bytes available in a memory bank
TSK	-indicates the tasks active in a memory bank, their starting locations, and size.

PUB -indicates the names of Public Programs in a memory bank, their starting locations, and size.

TSK (0) -indicates the current status of all tasks and devices configured on the system.

(See Section 4 for more information on functions)

<u>USER MEMORY</u>

The allocation of memory space to tasks cannot span memory banks (except for banks 0 and 1 on Level 3). This means if a 16 page area exists in bank 1 and a 16 page area exists in bank 2, two 16 page tasks could execute, but a single, 32 page task could not. The fragmented memory space is, however, available for any task to use for Public Program modules (which also cannot span memory banks).

<u>GHOST TASKS</u>

A ghost task is one which is not dependent on a terminal for operation. Examples are print programs and file updating programs. Ghost tasks are started from any other task, or by the operator using a terminal. The START command is used to start a ghost task.

Example:

```
0010 START 20, "PRINT", "GO"
```

where "START 20" indicates 20 pages of user memory, "PRINT" is the name of the program and GO is the name of the ghost task. Up to 8 ghost tasks can be configured on Level 4 (GO through G7), and up to 4 on Level 3 (GO through G3).

When a ghost task is finished, it should execute a RELEASE statement. This RELEASES the ghost task's memory for reassignment to another task.

The following code allows a task to RESEASE itself if it is running as a ghost task:

```
9900 LET F$=FID(0)

9910 IF F$(1,1)="G" THEN RELEASE ELSE END
```

RESTRICTIONS ON GHOST PROGRAMS

The following restrictions apply to ghost programs:

- o The program cannot attempt to communicate with a controlling terminal because none is assigned; and
- o A SETERR should be executed at the beginning of the program to prevent an error which might cause a return to Console Mode (which requires a terminal for output of the error message, the ">", etc.).

COMMUNICATION WITH A GHOST TASK FROM A "STANDARD TERMINAL TASK"

It is possible to communicate with a ghost task from a "standard task" (one with a controlling VDT) through the use of the utility program "*G" on Level 3, or "*GHOST" on Level 4. The ghost task ID (G0-G7) is input to the utility whenever it is RUN or CALLed. If the ghost task specified is active, the utility OPENs it on the first available unit (1-7) and READs and WRITES to the ghost task as though it were an I/O device. Data entered on the VDT keyboard (except ESCape key) is transmitted to the ghost task by *G(or *GHOST), and any output from the ghost task is displayed on the VDT as though it were connected to the ghost. The utility CLOSES the ghost unit number and terminates when the ghost task RELEASES itself, or when the operator presses the ESCape key on the VDT.

SPOOLING

Automatic deferred printing (spooling) allows an application program to proceed with printing a report, even though all printers are busy. The output which would have gone to the printer is intercepted and stored in a Serial file for subsequent printing. The Spooling feature requires a few extra pages in any bank.

No language changes are required to take advantage of this feature, and it can be completely transparent and automatic to the application program.

LEVEL 3

Enabling the Spooling Feature

When the LOAD button has been pressed, the Basic Four proprietary message displays, followed by the question:

```
DO YOU WANT SPOOLING?  (CR/N)
```

A CR response enables the spooling function in the system by activating the program OSSPOL and running the program *.I.   *.I continues the dialog by asking:

```
SPOOL FILES BEING ACTIVATED;
DEFAULT NUMBER OF LINES IS 1500
MAKE ENTRY TO CHANGE OR JUST CR
```

A CR leaves the number of print lines per spooled job at 1500 lines. By entering a number, the number of lines is changed.

*.I then asks:

```
START THE DESPOOLING PRINTER
TASK NOW? (CR/N)
```

This question gives the operator the option to start printing any queued print jobs from a previous period, or to wait until a later time. If CR is entered, the utility program *.P starts as a ghost task. The *.P utility program "despools" the print jobs which have been previously queued.

Defining a Task's Spool Files

Permanent Spool Files - For each task configured into a given system, there is a predefined permanent spool file. These files have the following characteristics: Type = Serial, 132 bytes/record, 1500 records. The name format for these files is "LPTx" for terminal tasks and "LPGx" for ghost tasks, where x is the specified value assigned to the task. These files act as the primary spool file for their associated tasks in that, if not currently in use, the appropriate permanent spool file is opened for the task by the 0SSP0L function. If a task's permanent spool file is already in use, 0SSP0L defines and opens a secondary spool file.

Secondary Spool Files - Secondary spool files are created and opened by the 0SSP0L function to enable a task to process multiple print jobs without waiting for printer availability. After a secondary spool file is printed, it is erased from the disc. These files have the same charactersitics as the permanent spool files, but utilize the format: "__Snnnn", where nnnn is a four digit sequence number taken from the counter in the null key record of file "Queue". This provides reasonable assurance of uniqueness of file names between tasks and print jobs.

Priority - A print file is assigned a print priority form 0 to 9. All poriority 9 print files are printed prior to priority 8, etc., through priority 1. Priority 0 print files are not printed and can be thought of as being in a "Hold" status until "released" by a change to a non-zero priority. The default priority is 5 for automatically spooled output.

Class - A print file is also assigned to a print class (A to Z). All print files of a given class (and of the same priority) are printed as a group. Thus, print class can be used as a means of forms specification and grouping in the print queue to minimize form changes on the printer. For a given priority, the order of printing is based on class.

Adding a Print Job to the Spooling Queue

There are three ways that print jobs can be submitted into the Spooling Queue: Automatically by an application program attempting to OPEN a busy printer; interactively by the operator through the use of the utility program *.S; or directly from a special application program which makes an entry into the queue file.

Automatic Submission - The automatic submission process begins whenever an active task attempts to OPEN(1)"LP". This operation is intercepted by OSSPOL and the permanent spool file for that task is opened instead (e.g., OPEN(1)"LPTx"). If the permanent spool file is already active, OSSPOL creates and opens a secondary spool file for the task. All data transfers made by the task to "Device 1" are automatically routed to the spool file instead.

When the task closes the device CLOSE(1), OSSPOL closes the spool file and makes the appropriate entry in the queue file. If a spool file fills, an ERROR 2 is produced.

Interactive Submission - Any file can be submitted for printing by operator interaction with the utility *.S. By running *.S, the operator is allowed to specify a file name, priority, copies, etc., and *.S then builds a queue entry for the file.

Printing the Queued Print Jobs (Despooling)

Files are printed and their corresponding entries are deleted from the print queue by the utility program *.P. Current implementation only permits spooling print files for the printer called LP. The execution of *.P can be initiated by the operator in either of the two ways: when the system is first started up (see "ENABLING THE SPOOLING FEATURE" above), or by STARTing *.P from a VDT, either as a ghost or terminal task, at a later time.

When a print job is ordered, files are selected for printing from the print queue file in the following order:

1.	Priority	(9 first, 1 last, 0 hold)
2.	Class	(A first, Z last)
3.	Sequence*	(N before 1 +N. N is greater than 0)
4.	File Name	(Alphabetically)

* The sequence number is assigned by spooling to assure the uniqueness of an entry in the queue.

When a spool file is submitted for printing via RUN "*.S", a brief message about that file can be included in the "Instructions" field of the queue modification and display utility *.M. When a spool file is selected for printing by *.P, the message in the instructions field is changed to:

"***PRINTING*** COPY--"

All spool files still logged in the queue (from previous system operating periods) are placed in a holding state by setting their priorities to zero and placing the string "***HOLDING***" in their instruction fields. If such old spool files exist, it is left to the operator to either delete these from the queue or to make them non-zero priority to force them to print. *.I tells the operator at the VDT how many files are still in the queue and asks if the despooling printer task should be started now. If the queue file contains old spool file entries, *.I is terminated by running *.M, the "Modify and Display 'Queue' File" utility.

Changing the Print Queue

The print queue can be modified from any VDT using *.M. This utility allows printer queue entries to be changed in priority, class, and number of copies, and allows reversal of the auto-erase flag and revision of operator instructions. Print requests can also be completely deleted from the queue by blanking the first character of the file name. Any attempt to modify or access a print queue entry while it is being printed causes termination of the job. When a file is deleted from the queue, it is also erased if its "auto-erase" field contains "Y" (for Yes).

The print queue is displayed in key sequence. This yields results in the order to be printed within each priority category. Files currently being printed are displayed with the operator instructions replaced by the string, "***PRINTING COPY***".

SPOOLING (Cont'd)

LEVEL 4

A complete description of Level 4 Spooling can be found in the LEVEL 4 UTILITIES USER'S GUIDE, BFISD 5084.

PUBLIC PROGRAMMING

The main objective of Public programming is to reduce the overall memory requirements of a system. This is done by putting one copy of frequently used programs, utilities, and subroutines into a common, mutually accessible place, and allowing any task to "share" the stored code on a reentrant basis. As an example, an order entry system with 10 VDTs, all doing order entry and using 31 pages of memory per VDT for multiple copies of the necessary programs, would require 310 pages of memory. The same function might be accomplished with Public programming by using just one 22 page copy of the program, plus data storage and overhead for each VDT of 10 pages each, for a total of 100+22 = 122 pages.

DICTIONARY CONSIDERATIONS

Dictionary entries stored in the operating system area of main memory are used to support Public programming and OPENed files and devices. When the system is configured, eight dictionary entries are allocated to Public programming if the feature is requested (Automatic on Level 4). If spooling is selected when the system is loaded during start-up, one of the dictionary entries is dedicated to spooling. However, the total number of Public program dictionary entries available is increased automatically by the operating system to make use of dictionary entries normally assigned to tasks and in Level 3, to the compiler/lister. If the compiler/lister is dropped from the Level 3 dictionary, two entries become available.

Further, any unused task entries in the dictionary are available automatically for Public programs: seven dictionary entries are assigned to each task in Level 3, eight in Level 4, when the system is configured, and a dictionary entry is used for each unique file or device opened by a task. When all available dictionary entries have been used - either by tasks opening files, or by an accumulation of Public program activity - any attempt by a task to open an additional file or to CALL or ADD an additional program fails, and results in an ERROR 16.

PUBLIC PROGRAM COMMANDS

An entry is made in the dictionary whenever a Public program is ADDed. This command does not bring the program into memory, but locates it on the disc and maintains its disc address in the dictionary so that subsequent CALLs to the program access the program from the disc without a normal disc directory search. If an entry for a program is made in the dictionary, any attempt to modify the disc file or the normal disc directory entry causes an ERROR 18.

As an alternative to ADDing a program to the dictionary, the ADDR command can be used to LOAD the program (make it a Resident program) as well as ADD it's directory information to the dictionary. DROP is a command that deletes program entries from the dictionary and memory. The CALL, ENTER and EXIT commands are used to run Public programs.

NOTE

ADD is unnecessary on Level 4. See the "ADD" directive in Section 4.

For programs not in the dictionary, the CALL command automatically ADDRs the program (and DROPs it on EXIT).

The PGM and PSZ functions return information about the CALLing program when executed in a Public program.

OVERLAID CALL (LEVEL 4 ONLY)

If an attempt is made in Level 3 to CALL a program into public memory, and public memory is full, an ERROR 33 results. If the same situation occurs on a Level 4 system, however, the system attempts to write the CALLed program over the CALLing program. If the CALLed program is not too large to fit into the space occupied by the CALLing program, it overwrites the CALLing program, clearing it from memory. At the execution of the EXIT directive, the CALLing program is brought back into memory from the disc, and the CALLed program disappears.

The overlaid CALL can also be forced in Level 4 when room exists in public memory, and when the CALLed program is resident in public memory, by use of the SIZ= parameter in the CALL statement. The SIZ= parameter specifies the space needed to run the CALLed program and may force the system to use the overlay procedure described above, pre-empting the search in public memory. If insufficient space exists to overlay the program, an ERROR 33 results.

RESTRICTIONS ON PUBLIC PROGRAMS

The following statements cannot be executed from a Public program. If an attempt is made to do so, an ERROR 38 results.

EXECUTE	LIST	SAVE	ESCAPE
DELETE	MERGE	RUN	START

The trace flag is not altered by a Public program, so the statements can be traced. Statements that are traced in Public programs are not displayed, however. Each line traced in a Public program displays only the statement number and program name. Tracing is initiated and terminated by the SETTRACE and ENDTRACE commands.

Programs can be removed from public memory with use of the DROP directive. In Level 3, the only Public program which can be DROPed (on a bank-by-bank basis) is the last one that has been ADDRed to a given bank. This is known as the Last In, First Out (LIFO) rule, and does not apply to Level 4 systems.

INPUT BUFFERING

Input buffering allows an operator to enter input data on the VDT keyboard without having to wait for a prompting message or a request for input to appear on the display during the execution of a Business BASIC program. The operator can enter responses required by the program in the sequence in which the data is requested. However, the characters are not displayed until the statement requesting the data is executed by the processor. Up to 26 variable characters can be buffered in Level 3. This number is variable in Level 4, with a minimum of 26, and a maximum of 255 characters.

CLEARING THE INPUT BUFFER - 'CI'

'CI', the "clear input" mnemonic, provides a means to insure that no unprocessed input is used at critical prompt points in a program. The execution of 'CI' in a statement clears all data in the input buffer. A statement such as:

```
     INPUT 'CI', "PLEASE REENTER DATE:   ", A$
```

clears any data in the input buffer, prints the character string, and waits for the operator to enter the field. Subsequent inputs are then buffered as they were before the execution of this mnemonic.
On Level 4 systems, the input buffer feature can be turned off on any task by use of the 'ET' mnemonic, and can be reinitiated with the 'BT' mnemonic (see "MNEMONICS" in Section 8).
On systems which do not support input buffering, the 'CI' mnemonic is ignored.

ESCAPE PROCESSING

The operator can correct an error after a field terminator has been buffered and before the field has been processed (displayed) through use of the ESCAPE key. When the ESC key is pressed, the input buffer is cleared and the terminal is returned immediately to Console Mode, unless fielded by SETESC.

If the ESCAPE occurred during the processing of the input buffer, that portion of the input field which has been moved to the program area is lost. When the RUN statement is entered, processing begins at the beginning of the statement which was interrupted by the ESCAPE. If the program has a SETESC in effect, the buffer is cleared before executing the SETESC routine.

TBL = PROCESSING

If a TBL= is in effect in an input statement, input buffering is not supported for that statement. The input buffer is cleared in the initial execution of the statement, and again at the end.

ERROR PROCESSING

Any error which returns the terminal to Console Mode clears the input buffer. Buffering is not in effect during Console Mode. In Program Mode, only ERRORs 5, 34, and 9 clear the input buffer when errors are fielded using ERR= or SETERR.

Buffer overflow (ERROR 34) is flagged whenever one more character is put into the input buffer than the buffer can hold. The error is issued on the next I/O directive to the terminal and is processed as other errors described.

PROGRAMMING

When operator verification of system output is required, the 'CI' mnemonic should be used on the input statement. This forces the operator to wait for the system prompt before keyboard input is accepted.

Example:

```
   0090  PRINT (0,ERR=1010)"BALANCE=", A
   0100  INPUT (0,ERR=100) "CORRECT? (YES/NO)", 'CI',
   0100: D$:("YES"=650,"NO"=725)
```

To avoid confusion, input buffering should not be used with the TBL= option. The input buffer is cleared upon execution of the I/O statement containing the TBL= function. Any data in the input buffer is then lost.

Input buffering can be disabled in Level 4 by use of the 'ET' mnemonic.

EXAMPLE OF INPUT BUFFERING

```
0010 BEGIN
0020 SETERR 0500
0030 FOR X = 1 TO 20000
0040 REM "THIS LOOP IS TO SIMULATE PROCESSING TIME
0050 NEXT X
0060 INPUT "ENTER A:",A
0070 INPUT "ENTER B:",B
0080 INPUT "ENTER C:",C
0090 PRINT 'CI',
0100 INPUT "ENTER D:",D
0110 INPUT "ENTER E:",E
0120 PRINT "HERE ARE THE RESULTS:",A,B,C,D,E,
0200 STOP
0490 PRINT 'CI'
0500 ON ERR (26, 34) GOTO 0510, 0530, 0550
0510 PRINT "PROGRAM TERMINATED BECAUSE OF ERROR",
0510:ERR; STOP
0530 PRINT "ENTER ONLY NUMERIC DATA,"; WAIT 2; RETRY
0550:PRINT "YOU HAVE EXCEEDED THE INPUT BUFFER AREA.
0550:PLEASE REKEY DATA"; WAIT 2; RETRY
1000 END
```

The preceding program can be used as a sample method of handling input buffer overflows and other errors that affect the state of the input buffer. The loop beginning at statement 30 is used as a timing loop to allow the filling of the input buffer. To overflow the buffer, key in more characters within the time of the loop. When statement 60 is executed (the first I/O statement encountered after the buffer overflow), an error branch occurs at statement number 0550 and the overflow error message is printed. The input buffer is cleared automatically, and all input accumulated in the buffer is cleared.

An example of the 'CI' mnemonic appears in statement 90. This means that the buffer*area is cleared at this point and the next input line, "ENTER D:", always waits for a response.

In the example, an ERROR 26 occurs if an alpha character is entered. An error branch takes the program to statement 530 and the error message is printed. Since error processing does not clear the input buffer, input statements after an error condition takes their data from the input buffer. Consequently, the 'CI' mnemonic should be used in the statements processing the error (see Examples 2 and 3).

The following data tests the example:

<u>Data Test 1</u>

<u>Input</u>	<u>Result</u>
1 (CR) 2 (CR) 3 (CR)	ENTER A: 1 ENTER B: 2 ENTER C: 3 ENTER D:

Data Test 2

```
Input                              Result

1 (CR) W (CR) 3 (CR)               ENTER A: 1
4(CR) 5(CR)                        ENTER B: W
                                   ENTER ONLY NUMERIC
                                   DATA
                                   ENTER B: 3
                                   ENTER C: 4
                                   ENTER D:
```

The preceding example shows why it is important to clear the buffer area during error processing. If statement 20 is changed to SETERR 490, the following occurs:

Data Test 3

```
Input                              Result

1 (CR) W (CR) 3 (CR)               ENTER A: 1
4 (CR) 5 (CR)                      ENTER B: W
                                   ENTER ONLY NUMERIC
                                    DATA
                                   ENTER B:
```

BRANCHING

Some directives cause program control to transfer to another statement number when certain conditions exist, as a method of program control. These directives are:

```
GOTO           ON/GOTO
GOSUB          SETCTL (Level 4)
EXITTO         SETESC
               SETERR
```

Some I/O options also transfer program control. These include:

```
DOM=      ERR=
END=
```

The Level 4 terminal driver supports mnemonics which protect display fields from being overwritten. Protected fields are written in Backgound Mode, and once written and protected, cannot be overwritten unless Protected Mode is discontinued.

Protection is a two step process: First, Background Mode must be started ('SB') prior to dislay of any line or partial line to be protected; then Protect Mode must be initiated('PS').

The following mnemonics are associated with Field Protection, and are fully described in Section 4 under MNEMONICS:

'SB' - Start Background Mode; Start Write Protect

'SF' - Start Foreground Mode; End Write Protect

'PS' - Start Protect Mode

'PE' - End Protect Mode

Default resets regarding Field Protection and use of other mnemonics include:

1. @(X,Y) allows the cursor to overwrite a protected position. Input or output at that point overwrites the X,Y position, but not other positions following it. (The cursor and data are placed in the first unprotected display position to the right and below the protected positions).

2. Use of any of the following mnemonics resets the VDT from Background ('SB') to Foreground mode:

```
'CE'         'DC'
'CF'         'IC'
'CL'         'LD'
'CS'         'SF'
```

   Start of screen scroll

3. Use of the following mnemonics when 'PS' (protect mode on) is in effect are ignored by the VDT:

```
'LD'
'LI'
'CL'
```

Use of the following mnemonics reset protect mode:

'CS'
'CF'
'PE'

5. Following execution of 'PS', the cursor is at home position (0,0).

<u>PRINTER PORT OPTION</u>

The printer port option (when configured) allows a serial printer to be connected to a VDT without using another physical I/O channel. The VDT and printer share the same channel.

<u>NULL OUTPUT CHARACTERS</u>

Level 3 systems count NULL characters (00 or 80) as printable characters on the display screen. Level 4 does not. Systems converting from Level 3 to Level 4 should review applications with NULLs in horizontal positioning routines.

00 does not move the cursor.

SPECIAL KEY CONTROLS

Level 4 provides additional keyboard controls accessed by use of the CONTROL key in conjunction with X, Y, S or Q. Use of one of these combinations results in the following action:

CTRL + X	- used to generate an ERROR 5
CTRL + Y	- used with SETERR to shift program control to a specified statement
CTRL + S	- causes task to stop processing. The task can be restarted from where it left off by use of CTRL + Q. It is often used with the LIST directive
CTRL + Q	- used to begin processing at the point where processing was stopped by use of CTRL + S

Character Codes

Characters are represented in Business BASIC in 8-bit ASCII code with b8 = 1 (high order bit set). The chart shown below provides the coding for the 128 characters in the ASCII code set. Business BASIC does not require the use of all the codes available.

ASCII Code Set

B4-B1		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
B8-B5		0	1	2	3	4	5	6	7	8	9	10 (A)	11 (B)	12 (C)	13 (D)	14 (E)	15 (F)
1000	8	NUL [128]	SOH [129]	STX [130]	ETX [131]	EOT [132]	ENQ [133]	ACK [134]	BEL [135]	BS [136]	HT [137]	LF [138]	VT [139]	FF [140]	CR [141]	SO [142]	SI [143]
1001	9	DLE [144]	DC1 [145]	DC2 [146]	DC3 [147]	DC4 [148]	NAK [149]	SYN [150]	ETB [151]	CAN [152]	EM [153]	SUB [154]	ESC [155]	FS [156]	GS [157]	RS [158]	US [159]
1010	10 (A)	SPACE [160]	! [161]	" [162]	# [163]	$ [164]	% [165]	& [166]	' [167]	([168]	) [169]	* [170]	+ [171]	, [172]	- [173]	. [174]	/ [175]
1011	11 (B)	0 [176]	1 [177]	2 [178]	3 [179]	4 [180]	5 [181]	6 [182]	7 [183]	8 [184]	9 [185]	: [186]	; [187]	< [188]	= [189]	> [190]	? [191]
1100	12 (C)	@ [192]	A [193]	B [194]	C [195]	D [196]	E [197]	F [198]	G [199]	H [200]	I [201]	J [202]	K [203]	L [204]	M [205]	N [206]	O [207]
1101	13 (D)	P [208]	Q [209]	R [210]	S [211]	T [212]	U [213]	V [214]	W [215]	X [216]	Y [217]	Z [218]	[[219]	\ [220]	] [221]	↑ [222]	← [223]
1110	14 (E)	‘ [224]	a [225]	b [226]	c [227]	d [228]	e [229]	f [230]	g [231]	h [232]	i [233]	j [234]	k [235]	l [236]	m [237]	n [238]	o [239]
1111	15 (F)	p [240]	q [241]	r [242]	s [243]	t [244]	u [245]	v [246]	w [247]	x [248]	y [249]	z [250]	{ [251]	| [252]	} [253]	~ [254]	DEL [255]

Explanation of Codes

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

Sort Sequence

The ranking of each character in a list of sorted characters depends upon the character's ASCII value (e.g., the lowest ranked printable character is a space, ASCII value 10100000; and the highest ranked printable character is a lower-case z, ASCII value 11111010). Key values are compared character by character, left to right, and the key with the first higher ranking character is ranked higher; but if the ASCII values of two keys are equal up to the length of the shortest key, then the longer key is ranked higher. When accessing keys in sorted order, lowest ranked keys are accessed first.

APPENDIX C - ALPHABETICAL SUMMARY OF DIRECTIVES, FUNCTIONS, VARIABLES I/O OPTIONS, AND SYSTEM OPTIONS

* = Level 3 only
** = Level 4 only

TASK	TYPE	DESCRIPTION
ABS	Function	Return absolute value
ADD	Directive	Add program's file ID to Public program directory
ADDC*	Directive	Add compiler to resident memory
ADDE	Directive	Add error-handling to resident memory
ADDL*	Directive	Add Lister to resident memory
ADDR	Directive	Add program to resident memory
ADDS**	Directive	Add SORTSTEP module to resident memory
AND	Function	Combine the bits of two strings
ASC	Function	Convert string to decimal
ATH	Function	Convert hexadecimal to ASCII
BEGIN	Directive	Reset system
BIN	Function	Return binary value
BLK=*	I/O Option	Assign user memory for buffer
BNK=	System Option	Assign bank number
BSZ	Function	Return bytes available in a bank
CALL	Directive	Transfer program control to another program
CHR	Function	Convert numeric expression to ASCII
CLEAR	Directive	Reset system
CLOSE	Directive	Release file or device
CPL	Function	Compile string expression
CRC	Function	Check for data integrity
CTL	Variable	Return field terminator last used
DAY	Variable	Return system date
DEC	Function	Convert binary to signed decimal
DEF FNx	Directive	Define arithmetic operation or string expression
DELETE	Directive	Remove statement(s) from a program
DIM	Directive	Define an array
DIRECT	Directive	Define a Direct file
DISABLE	Directive	Place a disc drive off-line
DOM=	I/O Option	Transfer program control if duplicate or missing key
DROP	Directive	Remove a program from Public Program Directory
DSZ	Variable	Return unused bytes in user task memory
EDIT	Directive	Add, replace or delete characters in a statement
ENABLE	Directive	Place a disc drive on-line

TASK	TYPE	DESCRIPTION
END	Directive	Terminate a program
END=	I/O Option	Branch at end of file
ENDTRACE	Directive	Terminate SETTRACE listing
ENTER	Directive	Pass values from CALLing to CALLed program and back
EPT	Function	Return exponent of expression
ERASE	Directive	Delete entry from Disc Directory
ERR	Function	Return last occuring error
ERR=	I/O Option	Branch on error
ESCAPE	Directive	Interrupt program
EXECUTE	Directive	Generate or modify statements from within a program
EXIT	Directive	Return control to CALLing program
EXITERR	Directive	Return control to CALLing program when an error occurs
EXITTO	Directive	Transfer program control to specified statement
EXTRACT	Directive	Read data field from a file into variable field in a statement
EXTRACT RECORD	Directive	Read a full record from a file or device
FID	Function	Return file or device information
FILE	Directive	Define file type or restore ERASEd file
FIND	Directive	Read data from a file into a variable
FIND RECORD	Directive	Read a full record from a file or device
FLOATING POINT		Initiate Floating Point Mode
FNx	Function	Define function
FOR/NEXT	Directive	Begin looping
FPT	Function	Return fractional part of expression
GAP	Function	Generate odd-parity, byte-for-byte
GET	Directive	Transfer data from a sector to a variable
GOSUB	Directive	Transfer program control to internal statement
GOTO	Directive	Transfer program control to a subroutine
HSA	Variable	Return highest available sector
HSH	Function	HASH; check for data integrity
HTA	Function	Convert ASCII to hexadecimal
IF	Directive	Conditionally execute a statement
IND	Function	Return index of next record
IND=	I/O Option	Specify index of record to be access
INDEXED	Directive	Define Indexed file
INPUT	Directive	Used for communication between oper-ator and program

TASK	TYPE	DESCRIPTION
INPUT RECORD	Directive	Read a full record from a file
INT	Function	Return integer of expression
IOL=	System Option	Branch to IOLIST statement
IOLIST	Directive	Define list of variables
IOR	Function	Combine the bits of two strings
ISZ=	I/O Option	Define record size for a file
KEY	Function	Return key of next record
KEY=	I/O Option	Specify key to be accessed
LEN	Function	Return length of string expression
LEN=	I/O Option	Specify length range of variable
LET	Directive	Assign value to a variable
LIST	Directive	Print statement(s)
LOAD	Directive	Bring a program into memory
LOCK	Directive	Protects a file from access by other users
LRC	Function	Check for data integrity
LST	Function	Convert compiled BASIC to LIST format
MERGE	Directive	Combine two programs
MOD	Function	Divide integers, return the remainder
NEXT	Directive	Used with FOR to create looping
NOT	Function	Return inverse of string
NUM	Function	Return numeric value of characters in a string
ON/GOTO	Directive	Transfer program control to a statement
OPEN	Directive	Access a file or reserve a device
PGM	Function	Return compiled format of a statement
POS	Function	Return character position
PRECISION	Directive	Set number of places of rounding
PRINT	Directive	Print to a file or device
PRINT RECORD	Directive	Write a full record to a file
PROGRAM	Directive	Define a program file
PSZ	Variable	Return bytes used by a program, not including data
PUB	Function	Return information about Public programs in a bank
PUT	Directive	Write data in a string to a sector (NOT RECOMMENDED IN APPLICATIONS PROGRAMS)
READ	Directive	Read data into a variable
READ RECORD	Directive	Read a full record into a variable
RELEASE	Directive	Close files and release task
REM	Directive	Insert a comment
REMOVE	Directive	Delete the key of an existing record in a keyed file

TASK	TYPE	DESCRIPTION
RESERVE	Directive	Reserve a disc for exclusive use
RESET	Directive	Reset system
RETRY	Directive	Transfer program control to the statement where the last error occurred
RETURN	Directive	Transfer program control to the statement following the OOSUB 1
RTY=	I/O Option	Specify number of retries if operation fails
RUN	Directive	Execute a program
SAVE	Directive	Copy program from user memory to program file on disc
SEQ=	1/0 Option	Specify file number on the tape track being accessed
SERIAL	Directive	Define a Serial file
SETCTL**	Directive	Branch when operator enters CTL+Y
SETDAY	Directive	Set value of DAY variable
SETERR	Directive	Branch to error routine
SETESC	Directive	Branch when ESCAPE is pressed
SETTIME	Directive	Set value of TIM (time) variable
SETTRACE	Directive	List statements as they execute
SGN	Function	Return sign of numeric expression
SIZ=	I/O Option	Set maximum allowable characters for input
SORT	Directive	Define a Sort file
SORTSTEP**	Directive	Convert a batch of input strings into sorted sequences of strings
SSN	Varlable	Return the system serial number
SSZ	Variable	Return bytes in a sector
START	Directive	Reset system, start tasks
STOP	Directive	Terminate program before the physical end of the program
STR	Function	Convert numeric expression to string
SYS**	Variable	Return operating system level
TABLE	Directive	Define values to translate characters to another code during I/O operation
TBL=	I/O Option	Specify number of TABLE statement to be used
TCB	Variable	Return task information
TIM	Variable	Return current system time
TIM=	I/O Option	Specify seconds allowed for input
TRK=	I/O Option	Specify tape track to be used for data transfer
TSK(O)	Variable	List configured devices (except discs)
TSK(1-9)	Variable	Return string for all tasks in the bank