Business BASIC 86
Reference Manual

April, 1987
006262-001

M6262A

MAI BasicFour®

PAGE STATUS

		Effective Date
Page Status	iii/iv	
Table of Contents	v through x	April, 1987
Preface	xi/xii	April, 1987
Section 1	1-1 through 1-6	April, 1987
Section 2	2-1 through 2-6	April, 1987
Section 3	3-1 through 3-14	April, 1987
Section 4	4-1 through 4-116	April, 1987
Section 5	5-1 through 5-42	April, 1987
Section 6	6-1 through 6-22	April, 1987
Section 7	7-1 through 7-16	April, 1987
Section 8	8-1 through 8-18	April, 1987
Section 9	9-1 through 9-28	April, 1987
Section 10	10-1 through 10-34	April, 1987
Section 11	11-1 through 11-38	April, 1987
Appendix A	A-1 through A-10	April, 1987
Appendix B	B-1 through B-50	April, 1987
Appendix C	C-1 through C-2	April, 1987
Appendix D	D-1 through D-4	April, 1987
Appendix E	E-1 through E-2	April, 1987
Appendix F	F-1 through F-8	April, 1987
Appendix G	G-1 through G-12	April, 1987
Index	1-1 through 1-18	April, 1987

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

PREFACE

The Business BASIC 86 Reference Manual describes the MAI Basic Four Business BASIC 86 Language used on BOSS/VS and BOSS/IX systems. The information includes the new features provided in Business BASIC 86 and also additional features that take advantage of the special features of the BOSS/VS and BOSS/IX operating systems.
The major topics covered in this user guide are:

WARNING

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructons manual, may cause interference to radio communications, as temporarily permitted by regulation. It has not been tested for compliance with the limits for Class A Computing Devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference, in which case the User at his own expense will be required to take whatever measures that may be required to correct the interference.

OVERVIEW

This reference manual describes MAI Basic Four's Business BASIC 86 (BB86) programming language. BB86 is a new version of Business BASIC introduced with the BOSS/VS Level 8.6 and BOSS/IX Level 7.3 operating systems.

BB86 provides a new level of program compatibility between BOSS/IX and BOSS/VS systems by providing and documenting a language core that will behave compatibly between the two operating systems. In this way, programs that are restricted to use of the BB86 core syntax will be directly portable between systems running BOSS/IX (levels 7.3 and higher) and BOSS/VS (levels 8.6 and higher).

While the BB86 standard maintains this core, additional features are included in the implementation of BB86 on BOSS/IX and BOSS/VS to take advantage of the special features of these operating systems. However, programs using these additional features may not be portable between systems without some, possibly manual, conversion.

SCOPE

This reference manual is written as a tool for programmers in the everyday use of the MAI Basic Four systems. The explanations in this manual are presented in a condensed manner. All sections are structured to help the user find the answers to common questions, such as format or parameter selection, as quickly as possible. Some aspects of Business BASIC are given expanded discussion in the appendices. The Business BASIC 86 Quick Reference Card, (M0018) has been prepared to complement this quick reference function.

The manual is specifically directed toward users of Basic Four systems who develop, program and support business applications. It is not a tutorial, although a programmer already knowledgable in some other version of BASIC will be able to learn Business BASIC from it.

COMPATIBILITY

Business BASIC 86 is introduced for the first time with BOSS/VS Level 8.6 and BOSS/IX Level 7.3. BB86 is a standard, defining a set of directives, functions, and system variables as a common core to be supported and compatible on future releases of BOSS/VS and BOSS/IX Business BASIC. The core is based on compatibilities that already existed between BOSS/IX and BOSS/VS. The standard is a superset of that commonality. As a result of the evolution of Business BASIC, some compatibility with Level 3/4 Business BASIC is also maintained.

Accordingly, new programs written using only the syntax of the BB86 core are guaranteed to be compatible between systems running BOSS/VS Level 8.6 or higher and BOSS/IX Level 7.3 or higher.

The implementations of BB86 on BOSS/VS and BOSS/IX, however, are themselves supersets of the BB86 standards. These implementations include additional directives, functions and system variables that take advantage of facilities provided by the BOSS/VS and BOSS/IX operating systems. These additions continue to provide upward compatibility for programs written under BOSS/VS levels 8.5 and lower, and for programs written under BOSS/IX levels 7.2 and lower. However, programs written using these additional facilities may not be compatible across operating systems. For instance, a program written using special BOSS/IX features will probably require conversion to run on BOSS/VS systems.

CONTENTS

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, and defines style conventions.
- Section 2: Features of Business BASIC - describes variables, constants, expressions, logical operations and output data formatting.
- Section 3: Statement Formats - explains each component of a statement and defined parameters, common parameter abbreviations, and input/output (I/O) options. Included also here are symbols, compound statements and input terminators.
- Section 4: Directives - lists and describes each directive in alphabetical order.
- Section 5: Functions - lists and describes each directive 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 option in alphabetical order.
- Section 8: Mnemonics - lists and describes each mnemonic and its devices by category.

- Section 9: Error Processing - Lists each error, describes what the error number and message mean, and outlines the procedures to follow for correction.
- Section 10: BOSS/IX Specific Instructions
- Section 11: BOSS/VS Specific Instructions
- Appendix A describes features of the Business BASIC programming environment.
- Appendix B introduces the use of Multi-keyed files.
- Appendix C describes the variable tables for the BOSS/IX implementation of BASIC, provided for use with the CPL and LST functions.
- Appendix D contains tables of ASCII character codes; included are a high-bit-on table for BOSS/VS and a high-bit-off table for BOSS/IX.
- Appendix E is an alphabetical listing of the keywords available in Business BASIC 86.
- Appendix F is an alphabetical summary of the features available in Business BASIC 86.
- Appendix G is a quick reference summary of the system variables, I/O options, operators, functions, error messages, mnemonics, and directives available in Business BASIC 86. This information is also included in the Business BASIC Quick Reference Card, M0018.

CONVENTIONS

This manual uses certain conventions for indicating language syntax. Symbols used are defined as follows.

Symbols

Symbols used in the formats of language elements include the following:

{} Parameters enclosed in braces 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 braces are required by the system. Do not enter the braces themselves, only what they contain.

() Parameters enclosed in parentheses are required. Parentheses are to be entered with the parameters they surround.

[] Brackets are to be entered with the parameters that appear within them. Brackets are vised only in the EDIT statement.

"" Parameters enclosed in quotation marks are required. Quotation marks are to be entered with the parameters they surround.

| A series of parameters separated by the vertical bar are alternatives. Generally, at most one of the alternatives is to be used.

$ String.

$$ These can be used in place of "" for null. They also identify hexadecimal characters.

NOTE

All the above parameters are optional when enclosed in {} braces. 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:

add_alloc additional number of records to be allocated to a growing file when its initial allocation is consumed. As space is needed the file will grow in increments determined by add alloc until it reaches its defined maximum number of records.

arg-list a list of one or more variables, constants or expressions.

current working directory

first directory to be searched if a full path name is not part of the file name. This is the first directory in the prefix list, if the prefix list is specified. If the prefix list is not changed in BASIC, the user's prefix list consists of his working directory plus alternate directories upon entering BASIC. The current working directory is alternately referred to as the primary prefix.

directory	a string expression that specifies a directory name.
diskno	can be either a number from 0 to 7 or 255. This parameter is ignored by both BOSS/IX and BOSS/VS BASIC.
file-ID/dev-ID	a string expression that identifies a file or a device. The string's content is system dependent. Refer to the BOSS/VS User Guide, M5098F, and the BOSS/IX User Reference, M6210 for more detailed information.
fileno	the channel (logical unit) number that identifies a file or device, such as data files, terminals and printers. Assigned numbers may be from 0 to 63 only; each must be an "int-expr"(see below).
init-alloc	the number of records initially allocated to a file when the file is defined.
int-expr	a number, numeric variable or arithmetic expression with an integer value.
keysz	the size of a key in a direct or sort file; for direct files: minimum=1, maximum=56; for MULTI files: minimum=1, maximum=80.
log-expr	an expression containing a logical relation (>, =, <, >=, or <=) or connectives (AND, OR).
num-expr	a numeric variable or constant or an arithmetic expression with a real value.
prefix-list	the list of directories to be searched when looking for a file for which the name, but not the full path name, is given. If the file is not found in one of the directories specified in the prefix list, the search terminates. When the user first enters BASIC, the prefix list is set to his current working directory plus alternate directories.
prog-ID	the name of a program.
recno	the number of records in a file.

reesz — the size of each record of a file, in bytes.

secno — denotes the sector number on which a file is to be defined for the Business BASIC Level 3/4 system. It is retained here for compatibility only. This parameter is ignored by both BOSS/IX and BOSS/VS BASIC.

stno — statement number.

str-expr — a string variable or substring, such as A$(5,3); a literal; or an expression containing a combination of them (with a "+" for concatenation). String literals are enclosed in quotation marks ("). Hexadecimal strings are enclosed in dollar signs ($).

var-list — a list of variables separated by commas (",").

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, sometimes called Motor Bars, <CTL-I>, <CTL-II>, <CTL-III> and <CTL-IV>, the LF (line feed) and NULL. All operations in this manual are to be entered using the <RETURN> key. More information on input terminators can be found in the description of the CTL function in Section 5.

OVERVIEW

This section introduces many of the features of Business BASIC 86. A few of these features are new to BB86; most have been established in earlier versions of Business BASIC.

OPERATING MODES

The Business BASIC environment has two operating modes: console mode and program mode.

Most directives are permitted in both console and program mode, but some are restricted. Table 2-1 shows the restrictions.

Console Mode

In console mode, statements are entered without statement numbers at the BASIC prompt, > or]. Statements entered in console mode are executed immediately, upon pressing the <RETURN> or <ENTER> key. These statements are not added to any program , and are not stored in user program memory.

Program Mode

In program mode, statements are entered with statement numbers. Each statement is checked for syntactical correctness and is then added to the program currently in user memory. It is not executed until the program is run.

TABLE 2-1. Restricted Use Directives

DIRECTIVE	PROGRAM MODE	CONSOLE MODE
DEF FN	X	
EDIT		X
EXECUTE	X	
EXIT	X	X*
EXITTO	X	
EXTEND		X
GOSUB	X	
IOLIST	X	
LOAD		X
NO EXTEND		X
RETURN	X	
FOR/NEXT	X	X**
RETRY	X	
RUN w/o arguments		X

* Can be both if you type in a called mode.

** BOSS/VS BASIC allows FOR in console mode if its NEXT is in the same line.

Once a program has been created in program mode, it can be executed from console mode by use of the START or RUN directive. In addition to the optional execution of a program, START is used in BOSS/IX to assign a specified amount of memory to the user area to run the program; RUN is used when enough memory already exists in the user area.

OPERATING SYSTEM ACCESS

Since BB86 operates under the control of an independent operating system, either BOSS/VS or BOSS/IX, facilities are provided for accessing special functions provided by the operating system without leaving the BASIC environment.

An operating system command can be preceded by an exclamation point (!). In this case the command string is not placed in quotation marks and is not preceded by the word, "EXECUTE." The exclamation point is described as a special directive for each system in sections 10 and 11. An alternate form is the SYSTEM directive, which uses a string expression. In either case the operating system command format is system dependent.

INPUT/OUTPUT DEVICES

Input and Output (I/O) devices include terminals, printers, files, and other peripherals with which a program can communicate.

To use an I/O device, you must first prepare the device by using the OPEN directive. In the directive you assign the device a file number (also called a channel or logical unit number). The program uses this number for all communication with the device.

Up to 64 channels can be opened simultaneously. The user terminal is automatically assigned a file number of zero (0) and is always opened to that channel, leaving 63 channels available for accessing files and other devices. I/O statements involving the user terminal are not required to specify its file number unless I/O options are used.

BB86 uses error codes to indicate problems resulting from improper device access (refer to section 9 for a complete explanation of the error codes.)

I/O Directives

Table 2-2 lists the input and output directives available to the programmer, and the files and/or devices which can be specified for each.

TABLE 2-2. Input/Output Directives and Applicable Files/Devices

DIRECTIVE	Serial/ Indexed	Direct/ Sort	Multi- Keyed	Terminal	Printer	Tape*
CLOSE	YES	YES	YES	YES	YES	YES
EXTRACT	YES	YES	YES	NO	NO	**
FIND	YES	YES	YES	NO	NO	**
INPUT	YES	YES	YES	YES	NO	**
LIST	YES	NO	NO	YES	YES	NO
LOCK	YES	YES	YES	NO	NO	NO
MERGE	YES	NO	NO	YES	NO	NO
OPEN	YES	YES	YES	YES	YES	YES
OPEN INPUT	YES	YES	YES	YES	NO	YES
PACK	YES	YES	YES	YES	YES	**
PRINT	YES	YES	YES	YES	YES	**
READ	YES	YES	YES	YES	NO	**
REMOVE	NO	YES	YES	NO	NO	NO
RETAIN	YES	YES	YES	NO	NO	
UNLOCK	YES	YES	YES	NO	NO	NO
UNPACK	YES	YES	YES	YES	NO	**
WRITE	YES	YES	YES	YES	YES	**

Certain tape devices (e.g., cartridge streamer) are not supported for BASIC programs.

Tape supports these directives in RECORD mode only (e.g., READ RECORD, but not READ); otherwise an ERROR 13 is generated.

PUBLIC PROGRAMMING Public Programming allows libraries of commonly used programs to be accessed by many different applications. This helps build large, structured systems of application programs.

For example, if you were building a system of 37 application programs and each program verified a particular item of data the same way, placing the verification code (which we assume to be 1000 bytes in length) into a single Public Program offers the following advantages:

1. The overall size of the application system decreases by roughly 36,000 bytes.

2. You modify only one body of code to change the verification, rather than 37 different versions.
3. It helps in documenting how the system functions.

Public Programs are executed by the CALL directive. Normally, when a CALL is executed, BB86 issues commands to the operating system to find the program on the disk and load it into memory.

On BOSS/IX systems, performance is improved if frequently used public programs are placed in a reserved memory area by the ADDR directive. When a program is ADDR'ed, it is loaded into memory and remains there until DROP'ed . The DROP directive deletes the program address and removes it from memory. (BOSS/VS systems automatically do the equivalent of ADDR'ing and DROP'ing.)

Other directives that are used within Public Programs are ENTER and EXIT. ENTER is used for passing arguments between the CALLing program and the CALL'ed program . EXIT returns control to the CALL'ing program .

The following directives cannot be used in a Public Program and still be BB86 compatible. An attempt to use these on the BOSS/IX results in an error 38.

EXECUTE	MERGE	ESCAPE
DELETE	SAVE	
LIST	RUN	

INPUT BUFFERING During the execution of a Business BASIC program, input buffering lets you enter data without waiting for a prompt. You can enter required responses in the sequence the data is requested. However, the characters are not echoed until the processor executes the statement requesting the data.

In BASIC, any error that returns the terminal to Console Mode also clears the input buffer. In Program Mode, only errors 5, 9, and 34 clear the input buffer when errors are trapped using ERR= or SETERR.

Buffer overflow occurs when too many characters are put into the input buffer. The buffer overflow is handled by the operating system, which issues an error 34.

At critical prompt points in a program, it may be desirable not to use unprocessed data in the buffer. The mnemonic 'CI' clears the input buffer, assuring that the prompt is displayed before input is accepted. For

example:

```
1240 INPUT 'CI',"PLEASE REENTER DATA:  ",AS
```

Subsequent inputs are then buffered as they were prior to execution of the mnemonic.

When operator verification of system output is required, you should use the 'CI' menmonic on the input statement. This forces the operator to wait for the system prompt.

If the <ESCAPE> is pressed during the processing of the input buffer, that portion of the input field moved to the program area is lost. When you resume execution with RUN, processing starts where it left off, even within a statement interrupted by ESCAPE. If the program has a SETESC in effect, the buffer is cleared before executing the SETESC routine.

RETAIN BUFFERING

BB86 introduces RETAIN buffering, a new I/O option, particularly as an aid in handling multi-keyed files. A retain buffer is kept for each channel opened to a a file or device The buffer holds an entire record, which can then be moiified by the program used to load values into variables and written. Two additional new directives, PACK and UNPACK, have been introduced to manage the retain buffer.

Examples of how to use the retain buffer with multi-keyed files are given in Appendix F. The buffer is also available for use with other files and devices. Refer to the descriptions of the PRINT, WRITE, READ, and INPUT directives for additional information.

CONTROL BRANCHING

Branching occurs when program control is transferred to a statement other than the next statement in sequence. Program control may be transferred by use of certain directives and I/O options.

Directives that cause program control to transfer to another statement when certain conditions are met include the following:

```
GOTO
GOSUB
EXIT
EXITTO
NEXT
ON/GOSUB
ON/GOTO
RETRY
RETURN
SETCTL
SETESC
SETERR
```

The I/O options that transfer program control include:

```
DOM=
END=
ERR=
```

See the descriptions of these directives and options in Chapters 4 and 6, respectively.

OVERVIEW

This section describes the syntactical elements that contribute to a BASIC statement.

STATEMENT FORMAT

Every BASIC statement contains two elements: a directive and a list of parameters. In addition, statements in Program mode begin with a statement number. BASIC statements are given as in the following example:

```
    500 PRINT "EXPRESSION"
```

```
  500
                    PRINT            'EXPRESSION"

 Statement
  Number:          Directive:         Parameter(s)
```

A number that uniquely identifies the statement within the program	The operation to be performed	Required and/or optional values used by the directive to further specify the action

Each element is separated from the preceding and following elements by a space. When several parameters are entered, they are separated by commas.

A BASIC program may contain more than one BASIC directive. A simple statement contains only one directive. A compound statement contains several directives, each separated by semicolon (;). The statement numbers specify the order in which the statements will be performed.

Statement Numbers

Each statement in a Business BASIC program begins with a statement number. Statement numbers can be any integer between 1 and 16000, inclusive.

Statement numbers are normally assigned according to a step sequence great enough to allow insertion of additional statements, if any are needed. Statements may be entered in any order; they are then automatically sorted into ascending order by statement number.

If a statement is entered without a statement number, it is executed immediately (in console mode) and does not become part of the program.

If a new statement is entered that uses an existing number, it will replace the existing statement.
r

If the number of an existing statement is entered alone, 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.

Directives

Because it instructs the system to perform specific operations, such as PRINT or READ, the directive is the key element of the BASIC statement. Most directives can be executed in both Console and program modes. Exceptions are noted in the description of the directive.

LET and THEN are optional parts of the LET and IF/THEN directives. If they are omitted, the BOSS/IX compiler/lister supplies them before listing the program .

Parameters

Parameters condition the exact operation of the directive. Parameters specify such items as the values on which the directive is to operate (such as an expression to PRINT), how the directive is to handle its values (should it access a file by index or by key) and how the directive is to handle special conditions (such as errors).

The required and optional parameters vary for each directive. Some directives do not require any parameters. Parameters are described in the discussion of each directive.

Parameters can generally be specified by constants, variables, functions, arithmetic operations or string concatenations, allowing flexibility within a program .

Compound Statements

Statements can be compounded on a single numbered statement line. A semicolon is used between simple statements to form compound statements. For example:

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

The following rules apply to compound statements:

1. Both console and program mode statements may be compound.

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

3. REM statement can appear only as the last part of a compound statement. A remark cannot be followed by a continuation; otherwise the continuation is treated as part of the remark.

4. The following commands, because they transfer control, cannot be immediately followed by any command other than REM or ELSE:

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

5. NEXT might return to the next statement in a compound sequence, either after the FOR or after the NEXT.

6. RETURN causes a return to the next statement in the compound sequence following a GOSUB routine or a SETCTL routine.

7. RETRY re-executes the appropriate statement within a compound sequence after an error has occurred.

Variables, Constants and Expressions

Business BASIC provides for the use of numbers, strings, variables and expressions composed of these . A new type of variable, the field variable, is introduced in BB86 for working with multi-keyed files. 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 provided 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.

The following are both valid ways to represent the same

number:

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

The (floating point) numbers can range in magnitude from -.99999999999999E63 (14 nines) through -0.1-E63 (largest number less than zero) and from 0.1-E63 (smallest number greater than zero) through 0.99999999999999E63 (also 14 nines); zero is included. Numbers outside this range result in an ERROR 40. The system retains up to 14 significant digits.

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

Variable Names

BB86 allows variable names of up to eight characters, plus "$" for string variables or "#" for field variables. The first character must be a letter and may be followed by up to seven letters and digits. Upper- and lower-case have no significance.

User defined variable names cannot be keywords, but they can contain keywords. BASIC keywords are all commands, functions, system variables, and:

ALL, ELSE, EXCEPT, FI, FN, RECORD, STEP, THEN, TO

The only special restriction is that a user defined variable cannot begin with FN, which causes the system to expect a function definition.

A user defined function can be a keyword, since the FN prefix eliminates any ambiguity.

Field Variables

A new category of variables is introduced in EB86 for handling fields in Multi-Keyed files. A field variable

name consists of a letter, followed by up to seven letters or digits, followed by a pound sign (#). The pound sign may not be separated from the rest of the name. Acceptable variable names are: SALARY#, F#, H9#.

The name without the pound sign may not be a BASIC keyword, but it may contain a keyword.

The rules for field variable names are exactly the same as for field names (refer to the MULTI directive), and are the same as for string and numeric variables except for the ending #.

Field variables are assigned values only by the FIELD ALIAS directive, and are assigned on a channel basis. Refer to the FIELD ALIAS directive and also to Appendix B and Appendix F for additional information.

Simple Numeric Variables

A simple numeric variable is denoted by between 1 and 8 letters or digits; the first character must be a letter. ACCOUNT2 and EZ4ME are examples of names for simple numeric variables. 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); otherwise, Business BASIC returns an error 42, nonexistent subscript.

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, DEC and ASC functions in section 5).

Arithmetic expressions are evaluated according to a predefined priority:

Order	Symbol	Meaning	BASIC	Math
			2^2	2^2
1		Exponentiation		
2	* and /	Multiply & Divide	2*2, 2/2	2x2, 2/2
3	+ and -	Add and Subtract	2+2, 2-2	2+2, 2-2
		(also negation)	-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. For example:

Math	BASIC	Result
10+20	10+20	30
10+20x10	10+20*10	210
(10+20)xl0	(10+20)*10	300
$\frac{10+20}{10}$	(10+20)/10	3
2^2x3	2^2*3	12
$\frac{2+6}{4} \times \frac{2+3}{5}$	(2+6)/4*(2+3)/5	2
$-(2^2)$	-2^2	-4

Note that constants can be replaced by variables.

String Constants

A string constant is a string of characters, and can be any length up to 2K bytes long.

Character strings can be represented in two ways in Business BASIC: either as the literal characters or by their hexadecimal code representation.

With one exception, any character that can be typed at the keyboard can be entered as a literal character. The characters are typed in as they should appear, and the entire string is enclosed in quotation marks (e.g., "string"). The exception is the double quotation mark; use two adjacent ones to represent a single one in a literal string:

```
0100 A$="""R.P. McMurphy's the name,"" he said with a grin."
```

Characters can also be entered by their two-digit hexadecimal ASCII code. Characters that cannot be generated from the keyboard must be represented in this way. Each character is represented by two hexadecimal digits (0 - 9, A - F). The entire hexadecimal string is enclosed in dollar signs (e.g., $737472696E67$). (Refer to Appendix D for hexadecimal character values.) If an odd number of characters are specified, the string is padded on the right with a zero (i.e., 1 is treated as 10).

String Variables

A string variable is identified by one to eight letters or digits followed by a dollar sign ($); the first character must be a letter. Examples of valid string variable names are: A$, SEARCH$, and USER7$. There is no limit (other than computer memory size) to the number of characters that can be stored in a string variable. For example:

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

Subscripted String Variables (DIM)

The Dim statement is used to assign a length and, optionally, a filler character 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, then the fill character is a blank.

For example:

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

B$ is 5 characters in length.

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

B$ is five characters in length and is filled with asterisks (*****).

String Expressions

Business BASIC uses the plus sign (+) with string variables and string constants to form string expressions. The plus sign represents concatenation. For example:

```
00010 LET AS="HEAD"
00020 LET B$="ACHES"
00030 LET C$ =A$+B$
00040 PRINT C$

>RUN
HEADACHES
```

The amount of data area overhead required when concatenating string expressions varies based upon the strings involved and the existence of the result variable. For example, if the statement "A$=A$+B$" is used, a temporary storage area is created and the strings in A$ and B$ are placed there. The space reserved for A$ is then expanded to hold the result. The contents of the temporary storage area are copied into A$ and the storage area becomes free.

String expressions can include mnemonics. For example:

```
00010 SCREEN$='cs' + @(30,10) + "Operator Id________" +
00010:@(32,12) + "Password________"
```

can be used to store a login display screen in a BASIC string variable.

String Comparison

Strings are compared character by character. The character values are compared until the value of a character in one string exceeds the value of a character in the same position of the second string. The string "RED", for example is greater than the string "BLUE" because the first positions evaluated show "R" to be greater than "B".

If two strings are equal for the length of the shortest string, then the longer string is considered greater in value. For example:

```
00100 LET A$="SOME"
00110 LET B$="SOMEMORE"
```

B$ is greater in value. Note that "a" is greater than "ZZZ" because all lower case letters are greater than upper case letters.

Logical Expressions

Logical expressions provide a method of testing the relationship between two numeric or string expressions. Conditional branching is commonly determined by the results of such comparisons.

Logical expressions are used with the IF/THEN/ELSE directive to specify the conditions for branching. For example:

```
00100 IF A=l AND B=2 or C=3 THEN GOTO 0200
```

(Refer to the description of the IF directive in section 4.)

Test criteria are established by the following relational operators:

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

Simple, or atomic logical expressions are composed of two numeric or two string expressions, separated by one of the above relational operators. For example:

```
X - 2.35
Y + 5.4 <> 8 * (Z +2.3)
AS <= "A STRING"
```

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

```
A=l AND B=2
X$="HEAD" OR Y$="ACHE"
```

Logical operations, like arithmetic expressions, are evaluated according to a priority. If used together, arithmetic expressions are evaluated before logical operations.

Compound logical expressions are evaluated left to right. AND and OR have the same strength, so neither takes precedence. Parentheses may be used to change the order of evaluation.

Using AND or OR, the second conditional expression will not be evaluated unless it's necessary. For example:

```
IF A>1 AND X$(A)="B" THEN...
```

will not evaluate X$(A)="B" if A is one or less.

OUTPUT DATA FORMATTING

Business BASIC includes expressions for formatting the output of data. The following paragraphs describe the syntax for vertical and horizontal positioning of the output display, and for numeric display editing.

Positioning Data Display

Vertical and horizontal positioning are provided by the positioning expression within the PRINT, INPUT, READ and WRITE directives.
Both a vertical and a horizontal position can be specified for terminal input and output; horizontal position can be specified for printer output. The positions can be given by any valid numeric expression; non-integer values are rounded to the nearest integer.

The positioning expression immediately precedes the output expression, if any, or the input variable. Multiple positioning expressions can occur in a single statement.

For example:

```
00020 PRINT (0,ERR=1000) @(0,5),"Customer Data"
00030 INPUT (0,ERR=1200) @(10,8),"Customer Name: ",A$

Note: @(column,row)
```

Line 20, above, positions the beginning point for the screen heading. Line 30 positions the beginning of the description of the data to be input, followed by the input variable. The input variable can also have its own positioning expression:

```
00030 INPUT (0,ERR=1200) @(10,8),"Customer
00030:Name:",@(25),A$
```

Only the horizontal position is given for the variable since the desired vertical position is already determined by the position of the comment. Both vertical and horizontal positions could be given, and the vertical position can be either the same as, or different from, the preceding position. Note: if the row X position is not specified, the space between the first @ position and the next @ position is blanked out. In the following example, the space between 0 and 10 is cleared.

```
PRINT @(0,10),@(10)
```

The following program shows valid positioning expressions:

```
0010 LET A$="STAGGER"

0020 PRINT @(5,10),A$,@(20),A$
0030 LET R=2,B=5
0040 PRINT @(R*5,B+10),A$
>RUN


    STAGGER       STAGGER


          STAGGER
```

Terminals in normal mode provide 80 (0-79) horizontal character positions (columns); in wide mode they provide 132. Printers provide either 80 or 132 (0-131) horizontal

character positions, depending on the width of the printer. Only the horizontal position can be specified on a printer.

Terminals usually have 24 (0-23) vertical character positions (lines). If the vertical position is greater than or equal to 24, and the terminal supports only 24 lines, the display appears at (0,0), line 0, column 0 (top left of scan).

Numeric Editing

Formatted display of numeric values is provided by format expressions.

Editing of numeric values to be printed or displayed is provided by a form expression which includes a form operator (:) and a format mask. The format mask may be a string constant or string variable. The form expression follows a numeric expression as follows:

```
PRINT numeric expr:"###,##0.00+"
B$=STR(num-expr:"###,##0.00+")
```

or

```
A$="###,##0.00+"
PRINT num-expr:A$
B$=STR(num-expr:A$)
```

Following are numeric editing options:

num-expr	The numeric expression that specifies the value to be printed or displayed.
:	Indicates the beginning of the format mask.
0	Zero forces the printing of a digit or a zero in the position indicated.
#	Pound sign indicates a position that is to be filled by a digit of the expression, but suppresses the printing of a leading or trailing zero when there is no digit.
*	The asterisk is used as a "fill" character in lieu of the first # to cause printing of an asterisk in each leading zero position following the printed data (e.g., "*##,##0.00").

$	Dollar sign is a "floating" character used in front of the first # or 0 to cause the printing of a dollar sign in place of the rightmost suppressed leading zero.
,	Comma is placed at the point where a comma is inserted, if required.
	Period is placed at the point where a decimal point is inserted, if required.

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.

For example:

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

or

```
>PRINT -1:"-##0.00"
>      -1.00
>
```

If you choose to omit all sign editing elements, the value to be output will be an absolute value. Optional elements include:

(mask)	outputs the value masked as specified; enclosed in parentheses if negative, no parentheses if positive. The left parenthesis "floats", like the dollar sign, to the left of the numeric expression.
+	outputs "+" if the value is positive and "-" if the value is negative. This will be "floating" if specified at the beginning of the mask.
-	outputs a blank if the value is positive and "-" if the value is negative. This will be floating if specified at the beginning of the mask.

DB	outputs DB 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.
B	outputs a blank in this position.

If the value of the number to be printed to the left of the decimal point exceeds the mask size, an error 43 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.

For example:

```
>AS="+##,##0.00M
>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

>A=5.0555
>PRINT A:A$
>+5.06
```

NOT E

When using more than one floating element, only one will float. Some non-standard combinations of mask elements yield unusual results.

Non-Formatted Printing of Numeric Values

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, otherwise a blank.

If the program is in floating point mode, the value is printed as a floating point number, consisting of the sign 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).

For example:

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

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

NOTES

SECTION 4 – DIRECTIVES

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

Directives can be executed in both console and program modes, unless otherwise noted for a directive.

This section describes the BB86 standard directives and is

presented in alphabetical order. BOSS/IX and BOSS/VS specific directives are described in later sections.

Format

BEGIN {{EXCEPT} var-list}

where variable-list is a list of up to 63 variable and array names, separated by commas. Numeric arrays are represented by their names followed by "(ALL)".

Description

The BEGIN directive without a variable list resets the system by performing the following functions:

1. Resets the ERR and CTL system variables to zero.
2. Resets incomplete GOSUB and FOR/NEXT loops.
3. Resets precision to 2.
4. Clears the user data area, eliminating all variable definitions.
5. Closes all OPEN files and devices.
6. Deactivates SETESC, SETCTL and SETERR.

The BEGIN directive with a variable list and BEGIN followed by EXCEPT and a variable list does all the items above except item 4.

If BEGIN occurs in a CALL'ed program, but before the ENTER statement, it has no effect on the entered variables. If BEGIN occurs after the ENTER, then the variables are cleared in the CALL'ed program only; the connections between the variables in the CALL'ed and CALL'ing programs are severed.

Example 1

```
>ONE=1, TWO=2, THREE=3, FOUR=4
>PRINT ONE, TWO, THREE, FOUR
1 2 3 4

>BEGIN TWO

>PRINT ONE, TWO, THREE, FOUR
1 0 3 4

>BEGIN EXCEPT THREE
>PRINT ONE, TWO, THREE, FOUR
0 0 3 0
```

BEGIN (cont'd)

Exanple 2

Program 1; CALLER

```
0010 CALLER1$="ABC"
0020 CALLER2=7
0030 CALL "CALLED", CALLER1$, CALLER2
0040 PRINT CALLER1$, CALLER2
```

Program 2; CALLED

```
0010 ENTER CALLED1$, CALLED2
0020 CALLED1$=CALLED1$+CALLED1$
0030 CALLED2=CALLED2+CALLED2
0040 BEGIN
0050 CALLED1$="NEW"
0060 CALLED2=12
0070 PRINT CALLED1$,CALLED2
0080 EXIT
```

```
>RUN "CALLER"
NEW 12
ABCABC 14
```

In this example, CALLER passes values for CALLED1$ and CALLED2 to CALLED. These are modified by CALLED, updating the values of CALLER1$ and CALLER2 in CALLER before the link is severed by BEGIN. Once the link is severed, further changes to CALLED1$ and CALLED2 do not affect CALLER1$ and CALLER2.

Format

CALL "prog-ID" {,ERR=stno} {,arg-list}

where argument-list is a list of one or more variables or expressions, separated by commas.

Description

The CALL directive is used to transfer control and pass arguments to another program .

Each argument in the argument list is referenced in the CALL'ed program by a variable or array name in the cor-

responding ENTER statement.
When a CALL'ed program ends, control is returned to the

command following the CALL statement in the program originally issuing the CALL.

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" indicates that the passed variable or array value is subject to change upon return-ing from a CALL'ed program (call by reference), and "N" indicates that the passed value is used locally by the CALL'ed program and does not change the value in the CALL'ing program when control is returned (call by value).

An attempt to pass the same variable twice in the same CALL results in an ERROR 38. For example, CALL "PROGRAM", A$, B, C, B+0 is correct, but CALL "PROGRAM", A$, B, C, B results in a ERROR 38.

Examples

```
1000 CALL "MEACAB"

1010 CALL "MEABUS", ERR=12000,A$,B
```

Table 4-1. CALL/ENTER DIRECTIVES

CALL Argument	ENTER Argument	CHANGE	ACTION/RESULT
		Y	A in the calling program is used/modified by reference to A in the called program.
	B	Y	A in the calling program is used/modified by reference to B in the called program.
A+n (n=numeric expression or constant)	A	N	A in called program is set to value of the calling program 's A plus the value of n. The value of A in the calling program is unchanged.
A$	B$	Y	A$ in caller is used/modified by reference to B$ in called program. Original A$ of caller can be changed.
A$<5,1)	B$	N	The passed sub-strings are not modified.
"XYZ"	C$	N	C$ in called program is set to "XYZ".
D(l)	E	N	E in the called program is set to value of the caller's D(l). The caller's D(l) is not changed.
D(ALL)	E(ALL)	Y	E(...) in called program is set to value of each element of caller's D(...). The caller's D(...) changes each time E(...) changes. This is a special case to make an entire array common.

Format

CLEAR {{EXCEPT} var-list}

where variable-list is a list of up to 63 variable and array names, separated by commas. Numeric arrays are represented by their names followed by "(ALL)".

Description

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

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

CLEAR performs the following functions:

- Resets the ERR and CTL system variables to zero.
- Resets incomplete GOSUB and FOR/NEXT loops, clears the stack.
- Resets precision to 2.
- Resets active statement numbers to 0 for SETESC, SETCTL, and SETERR.
- Clears the user data area eliminating existing variable definitions.

The CLEAR directive used with a variable list resets only the variables and arrays in the list. CLEAR followed by EXCEPT and a variable list resets only variables not in the variable list.

If CLEAR occurs in a CALL'ed program, but before the ENTER statement, it has no effect on the entered variables. If CLEAR occurs after the ENTER, then the variables are reset in the CALL'ed program only.; the connections between the

variables in the CALL'ed and CALL'ing programs are severed (refer to the BEGIN directive, example 2, for an example).

Examples

```
>0NE=1, TWO=2, THREE=3, FOUR=4
>PRINT ONE, TWO, THREE, FOUR
 1 2 3 4
>CLEAR TWO
>PRINT ONE, TWO, THREE, FOUR
 1 0 3 4
>CLEAR EXCEPT THREE
>PRINT ONE, TWO, THREE, FOUR
 0 0 3 0
```

Format

CLOSE (fileno {,ERR=stno} {,IND=num-expr})

IND= is used only for 1/2-inch magnetic tape access (see the description below for permitted values).

Description

The CLOSE directive releases use of a file or device. CLOSE also unlocks files that were locked using the LOCK directive.

Files and devices are also closed when a STOP, END or BEGIN directive is executed, except when the END is in a CALL'ed program.

No error is returned if CLOSE is issued for a channel that is not open. If a channel is open but there is a problem closing the device, an error is returned. Such problems can be caused by an I/O error on a write to a disk file prior to closing the channel, or an attempt to close a printer which is offline (on an 8000 BOSS/VS system , no error will be returned).

Tape

When closing a file on 1/2-inch tape, the results differ depending on what actions were performed.

If the tape was opened and either closed immediately or

opened and only accessed by READRECORD's, the file is closed and no filemarks are written to tape.

If one or more WRITERECORD's were performed, then at least one filemark is written, at the point writing was stopped. The exception is in the case that the previous action was an unsuccessful WRITERECORD, in which case no file marks are written, regardless of the IND= specified, and an ERROR 5 is reported.

The IND= option affects the tape CLOSE as follows:

IND = 0 or 2 - rewinds tape to load point

IND = 1 - rewinds tape to load point and takes tape offline

IND = 9 - writes 2 file marks on tape, then rewinds tape

If IND= is not specified when closing the tape, nothing is done except that the tape is closed.

CLOSE (cont'd)

In addition, if the preceding action was a (successful) WRITERECORD, one file mark is written if IND= is not specified, two file marks are written if IND= is specified.

An ERROR 13 is reported if an IND= is specified other than 0,1,2, or 9, but two file marks are still written.

Examples

```
1200 CLOSE (1)

1200 CLOSE (1,ERR=0150)          The ERR= is ignored.
```

Format

CONSOLE LOCK {str-expr {,MSG=str-expr}}

where the two string expressions are, respectively, a password and a message.

Description

The CONSOLE LOCK directive prevents BASIC from automatically dropping out of a program into console mode.

BASIC attempts to enter console mode when the following conditions occur:

- an ESCAPE without an active SETESC
- an ERROR occurs without an active SETERR, ERR=, END= or DOM=
- completion of a program run from BASIC console mode
- completion of a program with "-nr" specified on the command line (BB7)

CONSOLE LOCK is ignored if:

- a SETESC is active when the ESCAPE key is pressed
- a SETERR is active or an ERR=, DOM= or END= is specified in the directive causing the error

In these cases, the specified branches are taken.

When BASIC attempts to enter console mode with CONSOLE LOCK active, the user is prompted with the message string. The user must then enter the password. If the password is entered incorrectly, BASIC attempts to continue execution of the program.

WARNING

If there is the directive STOP, ESCAPE or END in the program, CONSOLE LOCK will allow the user to enter the password three times. If the user cannot enter the correct password in three tries, BASIC will log the user off the terminal. Any other way, CONSOLE LOCK will allow the user to enter the password continually.

CONSOLE LOCK (cont'd)

Examples

```
0010 CONSOLE LOCK
```

Turns off the console lock function, clearing the password and message strings.

```
0100 CONSOLE LOCK "secret",msg="Enter password: "
```

Turns on the console lock function, with "Enter password:" displayed as the prompt, and "secret" defined as the password.

```
1000 CONSOLE LOCK "NEW"
```

Activates the console lock feature using the password "NEW", keeping the prompt message unchanged.

```
10000 CONSOLE LOCK ""
```

Turns off the console lock feature by clearing the password. The message is preserved for subsequent use.

Format

CREATE ATTR= str-expr {,EM T=str-expr} {,ERR=stno}

Description

The CREATE directive creates a new file with the attributes described in the first string expression. The attribute can be built by the program or returned by the ATTR function.

CREATE will not create any type of program file (organizations BAS, COB or PAS).

The FMT= option is used to specify the format of a multi-keyed file. Refer to the MULTI directive for the string details.

The CREATE directive is similar to the FILE directive, but is more system independent and provides more parameters.

CREATE permits the creation of remote files.

Attribute String Format

The attribute string has the same format as the string returned by the ATTR function using the LONG form. All of the ATTR function attributes may be specified, although the RECORDS USED clause is ignored. Any alphabetic characters in the string may be either upper or lower case with no change in effect except for those in the NAME attribute under either of two conditions:

1. the system is BOSS/IX or

2. the total number of characters in the name is six or less.

In either of these cases, changing a letter in the NAME changes the name of the file.

If an attribute is specified more than once in the string, the last specification takes effect.

Spaces are ignored on either side of the equal sign (=) in an attribute clause and at either end of an attribute clause. Spaces may not occur within keywords, numbers or value strings.

If an attribute clause is omitted , CREATE will assign a default value, except for NAME clause and the KEY SIZE clause for a direct file, which are required. The following default values are assigned:

```
    ORGANIZATION = IND
    RECORD SIZE = 80
    REOORDS_ALL0WED = 1000
    REOORDS USED = (ignored)
    KEY_SIZE = 0 (required for direct file)
    INITIAL = (system dependent value)
    GROWTH = (system dependent value)
    OWNER = (current account)

    USAGE RIGHTS = (the default for the system or account)
```

If the EM T= option is used, then ORGANIZATION=MUL is required.

Assigning explicit OWNER and USAGE RIGHTS will cause difficulties in writing interfamily-compatible programs.

For further details, refer to the ATTR function.

Examples

```
1000 A$=ATTR(1,"ALL");CREATE ATTR= A$+"NAME=XYZ
1000:RECORDS_ALLOWED=10"

1010 CREATE ATTR= "NAME=MYFILE REOORD_SIZE=10
1010:RECORDS ALLOWED=10"
```

Format

```
DEF FNx(var-list) = arithmetic-expr

DEF FNx$(var-list) = str-expr
```

where x is the function name, following the same syntax rules as for variables.

Description

DEF is used to create user defined functions. There can be a maximum of 63 functions in a program . These functions are in addition to the predefined functions which are part of the Business BASIC 86 language (see "FUNCTIONS" in section 5).

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

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. However, the output (expression) is limited to strings (DEF FNx$) or numbers (DEF FNx).

The formal 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; however, caution should be exercised because the values of the variables can change.

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
```

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

DEF FNx
DEF FNx$
(cont'd)

Examples

```
0010 DEF FNA(A,B) = (A+B)/A
0020 LET C=FNA(2,6)
```

Statement 20 assign A=2, B=6 and C=(2+6)/2=4

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


>RUN
DOE-SIDO
```

Format

```
DELETE stno-a
DELETE stno-a,
DELETE ,stno-b
DELETE stno-a,stno-b
```

where, if both stno-a and stno-b are specified, stno-a < stno?b.

Description

The DELETE directive is used to remove one or more statements from the program in memory. The BB86 standard and BOSS/IX do not allow DELETE to be used in a CALL'ed program (although BOSS/VS does allow it in a CALL'ed program).

The combination of statement numbers and the comma have different effects. Refer to the examples below.

Attempting to DELETE a nonexistent line number has no effect and does not generate an ERROR 21. Attempting to delete a nonexistent statement number by typing the number followed by a carriage return does generate an ERROR 21.

Examples

```
DELETE
```

Deletes all statements in memory

```
DELETE 100
```

Only statement 100 is deleted

```
DELETE 100,
```

All statements from 100 to the end of the program are deleted

```
DELETE ,100
```

All statements from the beginning of the program through statement 100 are deleted

```
DELETE 100,200
```

All and only statements from 100 through 200 are deleted.

Format

DIM array-name (range1 {,range2 {,range3}})

where:

array-name = the name of the numeric array, using the same syntax as for numeric variables

range1 ... range3 = integers giving the size of each array

Description

The DIM array statement is used to define a numeric array. An array is a 1-, 2- or 3-dimensional grouping of numeric values, referenced by a common name and the appropriate dimensions as follows:

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

A two-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(l,1)  A(1,2)  A(1,3)
A(2,0)  A(2,1)  A(2,2)  A(2,3)
A(3,0)  A(3,1)  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 dimensioning the array to zero:

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

BB86 supports long array names following the same syntax as for numeric variables; the name may be up to eight characters and digits long, but must begin with a letter.

A simple numeric variable and an array can both have the same name without conflict.

DIM is allowed on ENTER'ed variables.

DIM array (cont'd)

Examples

```
00010 DIM A(0)
```

Releases most of the dimension space.

```
00100 DIM A<2,2,2)
```

Defines a three-dimensioned, 27-element array.

```
0200 DIM A(5)
0210 FOR 1=0 TO 5
0215 LET A=37
0220 LET A(I)=I*10; NEXT I
0225 PRINT A(5),A(4),A(3),A(2),A(1),A(0),A
>RUN
50 40 30 20 10 0 37
```

Note that DIM A(3,0) is okay; but DIM A(3) will result in an error 42.

Format

DIM variable-name (int-expr {,str-expr})

where:

variable-name = the name of the string variable to be dimensioned

string-expr = is a character used to fill the dimensioned

variable.

Description

The DIM string directive creates the named string variable with the specified length. The string variable is initialized with blanks or with the fill character specified. If the fill character expression is more than one character long, only the first is used.

Dimensioning a string variable filled with underlines, for example, may be useful in programming input routines in which the user is prompted by showing the allowable length of the input. For each field you need only display the required number of underlines by printing the subscripted string.

Examples

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

1300 DIM B$(5,"A")          - assigns "AAAAA" TO B$
```

Format

DIRECT "file-id", keysz, recno, recsz {,ERR=stno}

where:

keysz = the key size (min=l, r nax=56)

recno = the maximum number of records in the file
(max=8,388,608)

recsz = the size, in bytes, of each record in the file
(max=32,767).

Description

The DIRECT directive creates a Direct type file. A Direct file is a single-keyed file.

The key, which provides access for both reading and writing a record, is usually 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.

Records of the file can also be accessed sequentially, by using the IND= I/O option, or in logically ascending order of the keys.

Examples

DIRECT "HIT", 10,100,50

This defines a Direct file named "HIT" with a key size of 10 bytes, and a maximum of 100 records of 50 bytes each. The file is created in the user's primary prefix. The filename may have specified an absolute filename indicating a specific directory.

Format

ENCRYPT "source prog-id","destination prog-id" {,ERR=stno}

where:

source prog-id = the string expression representing the name of the PROGRAM file to encrypt

destination prog-id = the string expression representing the name of the PROGRAM file which will contain the encrypted program.

Description

The ENCRYPT command encrypts the BASIC program contained in the source file, and leaves the result in the destination file.

If the destination file does not exist, it will be created. If the destination file specified already exists, its file type must be PROGRAM. Specifying a null string as a file name results in an error 10.

Specifying the same file name for both source and destination will cause the source program to be encrypted over itself.

WARNING

The source cannot be recovered if a program is encrypted over itself.

The ENCRYPTed program will RUN the same as the original program . Any attempt to LIST, EDIT, SAVE, INSERT, DELETE, or MERGE statements will generate an error 18.

ENCRYPT will not operate on a remote program file.

Example

```
>ENCRYPT  "ORDINARY", "SECRET"
```

This takes the BASIC program called "ORDINARY", encrypts it, and then stores the encrypted program as "SECRET".

Format END

Description The END directive is used to terminate a program .

END performs the following operations:

- Resets the program execution counter to the first statement of the program
- CLOSE'S all open files and devices, except when used to exit a CALL'ed program
- Performs a RESET operation
- Returns the terminal to Console Mode
- If executed from a CALL'ed program, END performs EXIT

The termination point established by the END directive is also used to discontinue MERGE (but not VMERGE) 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 MAI 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 (but not VMERGE) is used for Index files; if you use serial files, the MERGE stops with an error 2.

Example

```
9999  END
```

Format ENDTRACE

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

Example

```
>ENDTRACE

0200 ENDTRACE
```

Format ENDTRACE

Description The ENDTRACE directive stops string expression translation.

Refer to the SETTRANS directive for further explanation of string translation.

Example

```
0100 ENDTRACE
```

Format

ENTER arg-list

where argument-list is one or more variable names, separated by commas. It 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 kind (numeric, string or dimensioned array).

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 CALL'ing program to the CALL'ed 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. Refer to table 4-1 (under the CALL directive) for a summary of the conditions that do and do not change argument values.

Each time a public program is called, it can execute only one ENTER directive.

Examples

Refer to the CALL directive for a more extensive example.

```
0010 ENTER A$,B,C
```

Passes parameters A$, B and C between the CALL'ed and the CALL'ing programs.

```
0010 ENTER A(ALL)
```

Passes the entire numeric array of parameters between the CALL'ed and the CALL'ing programs.

Format ERASE "file-id" {,ERR=stno}

Description The ERASE directive deletes a file entry from the disk directory. It also deallocates all the sectors and blocks in use by the specified file. Since removal of the directory entry cancels all the references to the space occupied by the file, an erased file cannot be reclaimed.

The ERASE directive will erase a specific file if the file-id is a full path name (i.e., with directory names). If the path name is partial, then it will erase the matching file name contained in the first directory in your list in which the file is found.

Examples

```
01000 ERASE "AGOOF"
```

Deletes the file "AGOOF" from the user's current working directory.

Format ESCAPE

Description When executed in a program, 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.

When in console mode, typing the directive ESCAPE causes the system to list the next line to be executed in the currently running program. (However, there is no indication of which statements in that line remain to be executed.)

Strategic placement of ESCAPE within a program permits periodic examination of data, thereby simplifying program debugging. (However, it may not be used in a called program on BOSS/IX, but may be used on BOSS/VS.)

Example 2000 ESCAPE

Format

EXECUTE str-expr

where the string expression is either a console mode command or a line of program code.

Description

EXECUTE executes the command contained in the string expression.

The EXECUTE directive can only be used in program mode. The BB86 standard and BOSS/IX BASIC do not permit EXECUTE in CALL'ed programs; however, BOSS/VS does allow it.

EXECUTE permits the use of directives and commands in program mode which are normally available only in console mode.

EXECUTE can be used to build statements, and so provides an ability to generate and modify program statements.

Examples

EXECUTE can be used to print the values in the variables A0$,...,A9$:

```
 0010 FOR X=0 TO 9
 0020 EXECUTE "PRINT(1)A"+STR(X)+"$"
 0030 NEXT X
```

EXECUTE can be used to execute system commands from a BASIC program:

```
 11000 INPUT "ENTER SYSTEM COMMAND TO EXECUTE: ",COMMAND$
 11010 EXECUTE COMMAND$
```

Format

EXIT {int-expr}

where the integer expression is a value from 0 to 255 representing an error code.

Description

The EXIT directive is used to exit a CALL'ed program , returning control to the CALL'ing program .

The first statement executed after an EXIT directive is the statement following the CALL statement in the CALL'ing

program . If the CALL was made from console mode, EXIT returns control to console mode.

If an integer expression is specified, the value is returned to the CALL'ing program as an error code. If the system variable ERR is used as EXIT'S numeric expression, the error status of the called program becomes the error status of the calling program.

Examples

```
16000 EXIT

16000 EXIT ERR

16000 EXIT A + B
```

Format EXITIO 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 exit a subroutine (called by GOSUB) without executing a RETURN.

The statement number referenced by the EXITTO statement must be a constant, not a variable. If the specified statement number does not appear within the program, program control transfers 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 0060


0050 NEXT I
0060 REM "COMPARE ROUTINE IS COMPLETE"
```

In this example, when A(I)=B, control branches to statement 60.

Format

```
EXTRACT  (fileno {,RETAIN} {,ERR=stno} {,END=stno }
    {,DOM=stno} {,IND=int-expr} {,KEY=string-expr}
    {,TBL=stno} {,SIZ=int-expr} {,TIM=num-expr}) {arg-list}
    {,IOL=stno}
```

where the argument list is a list of string or numeric variables, separated by commas.

NOTE

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

Description

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

EXTRACT differs from READ in two ways:

1. It prevents other users from accessing the record until another operation is performed on the file by that user;

2. 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 then 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.

NOTE

Refer to section 7, "Input/Output Options."

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,RETAIN  (,ERR=stno} {,END=stno}
    {,DOM=stno} {,IND=int-expr} {,KEY=str-expr}
    {,TBL=stno} {,SIZ=int-expr} {,TIM=int-expr})
    {string-variable}
```

where string-variable is a string variable into which the record is 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:

1. It prevents other users from accessing the record until another operation is performed on the file by that user;

2. 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 then 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. When a new WRITE RECORD directive is executed to modify the file, the pointer is already in

position to write data to the correct record.

Format

```
FIELD ALIAS ( logical_unit [, ERR=stmt_num  ] )
                    field_var# [ = ] name_string$
                    [, field_yar# t = ] name_string$ ...]
```

Description

The FIELD ALIAS directive defines a link between one or more field variables and actual field names in a multi-keyed file. Fields aliases apply only to the logical unit(s) for which they have been defined.

The string expression contains the actual name of the field as it is defined in the multi-keyed file format, including the #.

The use of field alias assignments allows for using multi-keyed files without having to hard code field names into each program. Aliases can be used throughout the program, and the actual field names used only in the FIELD ALIAS statement.

Example

```
0210 FIELD ALIAS (1) X#="DEPTNAME#"


0550 READ (1,KEY=X#=G$) A$,B$,C$
```

Format

```
FIND  (fileno {,RETAIN} {,ERR=stno} {,END=stno} {,DOM=stno}
     {,KEY=str-expr} {,TBL=stno} {,SIZ=int-expr})
     {arg-list} {,IOL=stno}
```

where the argument list is a list of variables into which fields of the record are to be read, separated by commas.

NOTE

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

Description

The FIND directive is used to read data from a file into variables. FIND differs from READ and EXTRACT in that, if the specified key is not in the file, FIND does not update the key pointer position to the next highest key following the unfound key. 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.

NOTE

Refer to section 7, "Input/Output Options."

Example

```
0200 FIND <1,KEY=K4,ERR=0500)A,*,B$
```

Format

```
FIND RECORD (fileno,RETAIN  {,ERR=stno} {,END=stno}
     {,DOM=stno} {,KEY=str-expr} {,TBL=stno{ {,SIZ=size})
     {string-variable}
```

where string-variable is the variable into which the entire record is read.

Description

The FIND RECORD directive is used to read a full record from a Direct file into a variable 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. The 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 turns off automatic rounding and thus maintains maximum accuracy (14 digits) while permitting the generation of very large or very small values by using "E" to indicate a power of 10.

BB86 allows numbers in the range 1E-62 to 1E62 or .1E-63 to .1E63.

Numbers are output in floating point notation unless a mask is specified.

Example

```
0010 FLOATINGPOINT
0020 FOR 1=0 TO 5

0030 PRINT 2^1;NEXT I

 >RUN
 .1E+01
 .2E+01
 .4E+01

 .8E+01
 .16+02
 .32E+02
```

Format

```
FOR numeric-variable = num-expr TO num-expr
     {STEP num-expr}
```

where the numeric expressions represent, in order, the start value, the end value, and the increment value for the loop.

Description

The FOR/NEXT loop is used to repeat a series of statements in a program , with the beginning and ending conditions given as an interval.

When a FOR statement is first executed , the numeric variable is set equal to the start value. The statements following the FOR statement are executed in sequential order

until the NEXT statement is reached . The value of the numeric variable is then incremented by the step value (the step value is one if the value is not specified) and compared to the end value. The step value must not be zero.

If the variable value is less than the end value (greater than, in the case of a negative STEP), control passes to the statement following the FOR STATEMENT . This sequence is repeated until the variable value is greater than (less than, for negative STEP) the end value. Execution then continues with the statement following the NEXT statement.

FOR/NEXT loops can be "nested" to a maximum of 256 FOR/NEXT/GOSUBs (above that, error 31 is returned). However, POR/NEXT loops cannot cross. For example:

Correct

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

Incorrect

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

Attempting to delete an active FOR statement, either by

escaping to console mode and executing DELETE or by using EXECUTE in program mode, generates an error 27. FOR/NEXT loops can terminate normally, as described above, or by an EXITTO statement executed before the specified number of iterations is complete.

FOR/NEXT (cont'd)

Note that a FOR...NEXT loop will always be executed at least once, so that the following will print "1."

```
100 FOR 1=1 to 0
110 PRINT I,
120 NEXT I
```

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=l TO 3
0030 PRINT 10*I+J,
0040 NEXT J
0050 PRINT ' LF'
0060 NEXT I

>RUN
 11 12 13
 21 22 23
```

The following example is a loop which terminates before its normal number of executions. Note the use of EXITTO rather than GOTO to escape the loop.

FOR/NEXT loop

```
0010 BEGIN
0020 INPUT "NUMERIC? -",A$
0030 IF A$="END" THEN GOTO 00120
0040 IF A$=" " THEN LET A$="0"
0050 LET F$="Y"
0060 FOR 1=1 TO LEN(A$)
0070 REM "THE FOLLOWING LINE EXITS TO 100
0080 J=POS(A$(1,1)="0123456789+- ");IF J=0 OR J>10 AND
0080:10 1 THEN F$="N";EXIT TO 00100
0090 NEXT I
0100 IF F$="N" THEN PRINT  "INVALID"
0110 GOTO 0030
0120 END
```

Format GOSUB stno

Description The GOSUB directive 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.

GOSUB can be executed only in program mode.

Every subroutine referenced by a GOSUB directive must be ended by a RETURN or EXITTO statement. RETURN resumes program execution at the statement immediately following the GOSUB. EXITTO resumes execution at a specific statement number and clears the top level entry from the RETURN

address stack.

Attempting to delete an active GOSUB statement, by either escaping into console mode and using the DELETE statement or using the EXECUTE directive, generates an ERROR 27.

Example

```
0010 REM "EXAMPLE OF REPORT PROGRAM USING GOSUB"
0020 BEGIN
0030 OPEN  (7)"LP"
0040 OPEN (1) "INVENT"
0050 LET P$="###0.00"
0060 GOSUB 1000
0070 READ (1,END=00500,ERR=00600)A,B,C,D
0080 LET L=L+1
0090 IF L>50 THEN GOSUB 01000
0100 PRINT  (7)A:P$,@(10),B:P$,@(20),C:P$,@(30),D:P$
0110 GOTO 00070
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
```

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.

Example

```
0100 OPEN(7) "LP"
0110 LET X=L+1
0120 GOTO 00500
0130 PRINT "THIS"

>GOTO 130
>RUN
THIS
```

NOTE

In BOSS/VS, "GOTO n" will automatically start executing from the specified statement number (no need to RUN).

Format IF log-expr {THEN} stno-a {ELSE stno-b} {ENDIF}

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, or a compound of such logical expressions. The relational operators are;

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

Some examples of logical expressions are:

```
IF A = B
IF Len(X$) <= 16
IF C >= B
IF 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 AND's and OR's can be used in an IF

statement, and they have equal precedence; the system by default evaluates them from left to right. Parentheses can be used to change the order of evaluation. For example:

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

>RUN
30 IS TRUE
```

The action taken by the IF statement is determined by whether the logical expression is true or false. If the logical expression evaluated as true, the THEN clause is executed. If the expression evaluates 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 DEF, IOLIST, and TABLE.

IF/THEN/ELSE/ENDIF (cont'd)

IF/THEN/ELSE commands can be fully nested into a single statement in BB86. The nested IF can occur in either the THEN or the ELSE clause of the preceding IF. For example:

```
IF A = 1 THEN
  IF B = 2 THEN PRINT "HERE"
      ELSE PRINT "THERE"
  ELSE PRINT "DOWN UNDER"
```

The ELSE clause always terminates the most recent unterminated IF. Above, the first ELSE terminates the second IF, and the second ELSE terminates the first IF.

The ENDIF terminator is used in nested IF statements to terminate an IF clause that is not terminated with an

ELSE. For example:

```
IF A ? 1 THEN
  IF B = 2 THEN PRINT "HERE"
      ENDIF
  ELSE PRINT "DOWN UNDER"
```

In this scheme, the ENDIF terminates the second IF and the ELSE terminates the first IF. Without the ENDIF, the ELSE would terminate the second IF and the first IF would terminate without an ELSE clause.

BOSS/VS allows ENDIF to be preceded immediately by a semicolon; BOSS/IX does not.

The ENDIF clause is also used to delimit the scope of an IF statement, making other statements following it unconditional:

```
0030 IF A=20 THEN GOSUB 09000; C=3
0040 IF A=20 THEN GOSUB 09000 ENDIF; C=3
```

Line 30 sets C=3 ONLY IF A=20, but 40 sets C=3 in either case.

Examples

```
10 IF A=B 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.):

```
>LIST ;REM BOSS/IX form
0010 IF A=B THEN GOSUB 6000 ELSE GOTO 9999

>LIST ;REM BOSS/VS form
0010 IF A=B GOSUB 6000 ELSE GOTO 9999
```

Format

INDEXED "file-ID", recno, recsz {,ERR=stno}

where:

recno = the maximum number of records for the file (maximum = 8,388,608)

recsz = is the size, in bytes, of each record in the file (maximum = 32,767).

Description

The INDEXED directive defines a file comprising records that can be read from, or written to, 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 by special characters called field marks, which are inserted by the system.

Example

```
00130 INDEXED "FINGER", 100, 50
```

Format. INITFILE "file-ID" {,ERR=stno}

Description The INITFILE directive initializes a file. The name, type, structure, usage rights and access modes of the file are preserved, but the contents are deleted.

In the case of multi-keyed files, the format string is preserved. (The format string includes, among other things, the primary key and secondary key definitions.)

An attempt to initialize an opened file generates an ERROR 0.

INITFILE can initialize a remote file.

Example

```
>INITFILE "STARTOVER"
```

Format

```
INPUT {(fileno {,RETAIN} {,ERR=stno} {,END=stno}
    {,DOM=stno{ {,IND=int-expr} {,KEY=str-expr} {,TBL=stno}
    {,TIM=time-expr} {,SIZ=int-expr})} {,mnemonic}
    {,string-const} {,variable} {,IOL=stno}
```

where:

mnemonic = a terminal mnemonic

string-const = the input prompt message

variable = the variable to receive the input data.

NOTE

A comma is inserted before IOL= only when IO] > is used with a variable.

Description

The INPUT directive is identical to the READ directive in all respects. Because READ is commonly used for files and INPUT is commonly used for the terminal, only terminal INPUT is described here. Refer to the READ directive for file input .

The INPUT directive provides for two-way communication between the operator and the program. For each variable in the statement, INPUT prompts for one keyboard entry.

An INPUT may contain messages to be displayed on the terminal prompting the operator for specific information. The operator's response is read into the variable following the message.

Several messages and variables can be included in a single INPUT statement, one following the other. Mnemonics can be used with the messages to perform standard terminal functions, such as locating the cursor, clearing the screen, and protecting fields.

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.

INPUT
(cont'd)

The system then waits for the operator to respond. The operator enters the response, then presses a field terminator key (usually <RETURN>). Following input, the CTL (control) system variable is set to a value determined by the terminator key used. The following list identifies the available field terminators and the resulting CTL VALUES:

Keys	ASCII Character	Control (CTL) Value
(no key)	NULL (hex 00)	0
CR	Carriage Return (with a line feed)	0
LF	Line Feed	0
CTL-I	FS (field separator)	1
CTL-II	GS (group separator)	2
CTL-III	RS (record separator)	3
CTL-IV	US (unit separator)	4

An INPUT ended by the SIZ= option sets the CTL value to 5.

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 be thus be given the ability to determine the cause of processing that ensues.

When logical unit 0 is used as the input device, either by explicit specification or by default, all entries typed at the terminal keyboard are received by the system, and are then immediately returned to the terminal for display or printing.

Immediate display of data can be inhibited (e.g., to mask the entries before display) by specifying a logical unit other than 0. Display of the entries is then achieved by a PRINT statement. The logical unit number used must have been previously assigned to the terminal by means of an OPEN statement. For example:

INPUT (cont'd)

```
0010 LET F$=FID(0)
0020 OPEN (2)F$
0030 INPUT (2,ERR=00030)@(0,10),"ENTER QUANTITY SOLD-",B
0040 PRINT (0,ERR=00030)@ 0,11),B:"00000"
>RUN

ENTER QUANTITY SOLD-
ENTER "123"
000123
```

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

file parameters are as given in the general format.

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.

INPUT
(cont'd)

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

Examples:

```
0010 INPUT (0, ERR=00010)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:

file parameters are as given in the general format.

branchlist = one or more items whose syntax is:

```
string-literal=stno (e.g.,"END"=9999).
```

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 the input will result in an error 48.

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

INPUT
(cont'd)

Examples:

```
0010 INPUT (0,ERR=00010)"L/N/C",A$("L"=0200,
0010:"N"=0300,"C"=0400)
```

If A$ ="L", program control is transferred to statement 200.

If A$ = "N", program control is transferred to statement

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.

```
0100 INPUT (0,ERR=00100)"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 characters but a CTL key or CR has been entered, program control is transferred to statement 1000.

If the length of A$ is less than 8 or greater than 10, an ERROR 48 occurs.

Format

```
INPUT RECORD  (fileno {,ERR=stno} {,END=stno}
        {,DOM=stno} {,IND=int-expr} {,KEY=str-expr}
        {,SIZ=int-expr}) {string-var}
```

where string-var is the string variable into which the record is to be input.

Description

The INPUT RECORD and READ RECORD directives are identical in all respects. INPUT RECORD is usually used for input from the terminal and READ RECORD is usually used for input from files.

The INPUT RECORD directive places 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 terminal, since a RETURN or CONTROL bar key is treated as part of the data , rather than as a terminator.

Example

```
0010 INPUT RECORD(0,ERR=00100,SIZ=5)A$
```

Format

IOLIST arg-list {,IOL=stno}

where the argument list is a list of data items to be input or output in I/O statements. The list can contain variables, constants, expressions, and mnemonics.

Description

The IOLIST directive, available in program mode only, is used to define a set of variables or data items 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.

IOLISTS also may be used for printing of screens or report information. The IOLIST must not be terminated with **a** comma (","); for example:

```
0100 IOLIST @(0,10),"THIS IS A TEST"
0110 PRINT IOL=100,
```

An asterisk ("*") may be used in an IOLIST, provided that IOLIST is not used for write or write record operations.

Example

```
0050 OPEN  (1) "AFILE"
0100 IOLIST AS,B,C$,D$,IOL=00010
0110 IOLIST E,F$,G$
0120 IOLIST AS,B:"###","ABC","05678",IOL= 00110
0200 READ (2,KEY=A$)IOL=00100
0250 WRITE (1,KEY=A$)IOL=00120
0260 PRINT ,SB,,@(0,1),IOL=00120
```

NOTE

IOLIST takes the same type of variable list as that allowed on PRINT and WRITE statements. An error 20 will be generated if input verification items are used (see INPUT).

Format

{LET} assignment-list

where each item in the assignment list is in the form:

num-var = num-expr

or

str-var = str-expr

Items are separated by a comma.

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, i.e., 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,A$="VALUE"

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

Format

```
LIST {(fileno {,ERR=stno} {,IND=recno} {,TBL=stno})
     {stno-a} {,} {stno-b}
```

where stno-a and stno-b are the first and last statement numbers to be listed, respectively.

Description

The LIST directive is used to output a statement or series of statements.
The selected statements(s) are read 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 BB86 standard and BOSS/IX BASIC allow the LIST directive to be used as a statement in any program except a public program; however, BOSS/VS BASIC allows it within a CALL'ed 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 disk file, the file must be an INDEXED or SERIAL file with at least as many records as there are lines in the program that are to be listed. SERIAL files must be locked. An INDEXED file roust have a minimum record size of 80 bytes.

When listing to the terminal, output pauses at the end of the screen, and the user is prompted to continue. <CTL-II> causes listing to continue without pause. <CTL-IV> terminates the listing. Anything else displays the next screenful.

Examples

```
>LIST            - lists the whole program

>LIST 10         - lists statement 10 only

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

>LIST 10,100     - lists statements 10 through 100.

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

Format

LIST PROGRAM "prog-ID" ,"file-ID" {,ERR=stno}

where file-ID is the name of the Serial file which will contain the program listing of a printer.

Description

The LIST PROGRAM directive lists a BASIC program to a Serial file.

If the serial file does not exist, LIST PROGRAM creates it.

If the program is encrypted, an error 61 is generated.

If the name given for the program file is not a BASIC Program file, an error 17 is generated.

If the file named exists but is not a Serial file, an ERROR 17 is generated.

BOSS/IX requires that the existing Serial file be large enough to hold the listing, or an ERROR 2 is generated .

LIST PROGRAM cannot list from a remote Program file, but can list to a remote Serial file.

Example

```
LIST PROGRAM "CORNFLAKES", "CEREAL"
LIST PROGRAM "MYPROGRAM", "LP"
```

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 following is done:

- the current program in the user area is deleted.
- all FOR/NEXT/GOSUB/SETERR/SETESC return addresses are cleared.
- precision is set to 2.
- ERR is reset to 0.
- the specified program is READ into the user area.

The LQAD'ed program can then be executed or modified. The execution of a LOAD command has no effect on the user data area.

On BOSS/IX, if insufficient program area is available, an ERROR 19 (PROGRAM SIZE) is generated. The old program is

not removed until it has been determined that the new program can be loaded. This is not a problem on BOSS/VS.

If a corrupted or miscoded program is loaded and the BASIC runtime detects the error, the user program is cleared (as with START), and an ERROR is returned.

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

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

If program "PGM" uses the A variable, its value is still 129, unless a BEGIN or CLEAR is executed, or "PGM" resets it to another value.

Example >LOAD "INZONE"

Loads the program INZONE from the user's current working directory into the user's program area.

Format LOCK (fileno {,ERR=stno})

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

A LOCK'ed file is released by an UNLOCK, CLOSE or BEGIN statement.

A SERIAL file must be locked prior to writing to it.

If other users are accessing (attempting to open) the file when a lock is issued, an error 0 will result.

Example

```
0100 LOCK (1,ERR=00200)
```

Format

MAKE PROGRAM "file-ID", "prog-ID" {,ERR=stno}

where file-ID is the Serial file containing a program listing.

Description

The MAKE PROGRAM directive creates a BASIC program using the Serial file as the source and saves it in the program file specified.

The Serial file must be in LIST'ed or LIST PROGRAM format,

with each record having a line number.
If the Serial file does not exist, an ERROR 12 is generated. If the specified file exists but is not of type Serial, an ERROR 17 is generated.

If the program file does not exist, MAKE PROGRAM creates it. If the specified file exists but is not of type BASIC Program, an error 17 is generated. The program file specified may be an encrypted file, but it will lose its encrypted status when the new program is saved.

If a syntax error is found in the Serial file, MAKE PROGRAM continues and does not report the error. MAKE PROGRAM will report an ERROR 21 on lines with no line number, a zero line number, or a line number with no text. With the exception of syntax errors, MAKE PROGRAM will stop processing the Serial file when an error is encountered, and nothing will be saved.

MAKE PROGRAM may reference a remote Serial file but may

not reference a remote BASIC Program file.

Example

```
MAKE PROGRAM "COMPUTER", "EASY"
```

Format MERGE (fileno {,ERR=stno} {,IND=int-expr} {,TBL=stno})

Description The MERGE directive retrieves a program in LIST format from an Indexed or Serial file and adds that program to the program currently existing in a user memory. The Indexed or Serial file may be read from disk or any other input device, except tape devices.

The statements of the two programs are merged together.

If both programs have a statement with the same statement number, the one in the merging 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 merging of an END statement. If no END statement is present in the program being read, either an error 2 (END OF FILE) or an error 21 (STATEMENT NUMBER ERROR) will be encourntered.

With one exception, MERGE cannot be used in a public program . The exception is that BOSS/VS allows MERGE within CALL'ed programs; but this usage is not part of the BB86 standard.

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 00020
1000 END
```

2. OPEN an Indexed file ("TRUNK"), and temporarily store the program to be MERGED in it in LIST'ed format:

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

MERGE (cont'd)

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(AS$),3 THEN GOTO 0150
0040 READ (1,ERR=0150,KEY=A$)*
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 "LOADING 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 MERGE'ing program survives.

Format

MULTI "file-ID", recno {,recsz}, EMT=str-expr {,ERR=stno}

where:

recsz is the maximum size of the record

str-expr defines the record format, as described below

Description

The MULTI directive creates a multi-keyed file.

If the maximum record size is not specified, the size is calculated from the format string. If the size is specified, it must be at least as large as is required by the Format string, or an ERROR 17 is generated.

Refer to Appendix B for an extended discussion of multi-keyed files.

Format String

The format string defines the record structure of the file. Each field definition is given in the form:

field-id = istart-posJ field-fmt {keyset-info}

The field-id is the name of the field. The format is the

same as for a field variable and follows the standard BB86 rules (add "#") for variable names.

The field format argument gives information about the structure and format of the field. Fields can be fixed-length string or numeric, variable-length string or numeric, or composite .

A fixed-length string field is given in the format:

{LEFT ! RIGHT} pad-chr int-expr

The first argument is optional, and specifies whether the field is left or right justified. The string is left justified unless specified otherwise.

The second argument (pad-chr) specifies the padding as follows:

S - null padded, nulls are deleted on a read
C - space padded, spaces are deleted on a read
X - null padded, nulls are retained on a read

The integer expression "int-expr" specifies the length of the field in bytes.

The padding character and the integer should not be separated by a space. The length of a fixed-length string field is always "int-expr" bytes.

A fixed-length numeric field format provides the output Format mask, and is given in the format:

```
{ UNSIGNED  | SIGNED  | + | - } "N" int-expr {"." {int-expr}}
{ UNSIGNED  | SIGNED  | + | - } "N" "." int-expr
```

The first argument is optional and specifies whether the numeric data in the field is in signed or unsigned format. Signed format (SIGNED, "+", "-") precedes the value with its sign ("+" or "-") on output. If the value is negative and the field was unsigned, the value written into the field is unsigned, i.e., no negative sign. The default is unsigned.

"N" indicates only that the field is numeric.

The two integer expressions specify, respectively, the number of digits preceding the decimal point and the number of digits following the decimal point. Either can be left unspecified. The decimal point is always a period, even in European format. The length of a fixed-length field is, in bytes, the sum of the two "int-expr's" plus one (if signed)

plus one (if "." is used).

A variable-length string field is given in the format:

```
"S" "*" int-expr
```

The integer expression gives the estimated length of the string, and is only used for calculating the record's maximum size. The field is terminated with a linefeed character. This additional byte should not be counted in estimating the string length, since the system automatically includes it.

A variable-length numeric field is given in the format:

```
"N" "*" int-expr
```

The integer expression gives the estimated length of the string returned by STR(n), where n is the numeric value entered into the field. The field is terminated with a linefeed character. This additional byte should not be counted in estimating the string length, since the system automatically includes it.

A composite field is given in the format:

```
comp-subpart {+ comp-subpart}
```

MULTI
(cont'd)

Each composition subpart consists of a field name and an optional byte offset and length. For example,

```
FIXEDFLD#(4):3
```

describes a subpart as bytes 4, 5, and 6 of the field FIXEDFLD#. Even the field name is optional, but describing the subpart by offset and length alone is not recommended. The rules are similar to those for the start-position argument.

The optional start position (start-pos) specifies the starting location of the field, overriding the default rule that a field begins at the byte following the last byte of the previous field (exceptt composites).

The start position specification is given either as a field designator (name or variable) alone, a field designator followed by an integer offset, or by an integer offset alone. The integer offset specifies the first byte, and optionally the number of succeeding bytes, to be used for the field. For example, if a field FIXEDFLD# is previously defined, the start position can be specified by any of these expressions:

```
FIXEDFLD#:
FIXEDFLDt(7):
65:
```

The offset and byte count are allowed to extend beyond the named field. In this case, bytes are taken from the succeeding field.

The optional keyset information field can be any of these four values:

PRIMARY	This is the major keyset for the file. Duplicate entries are not allowed. A primary keyset is required, and there may be only one primary specified.
ALTKEY	This is a secondary keyset. Duplicate key values are not allowed.
DUPKEY	This is a secondary keyset. Duplicate key values are allowed.
NOKEY	This is a non-keyed field.

If no keyset information is specified, the field is defined as NOKEY.

A special field name, FILLER, is provided for specifying an unnamed field to reserve space in the data record. The format is:

```
FILLER = {start-pos} length
```

Example

```
0100 DEPFMT$ =  "DEPTNAME# = S12 ALTKEY
0100:            DEPTNUM#  = N5 PRIMARY
0100             DRCTRNUM# = N6
0100             BUDGET#   = N8.4
0100             EXPENSE#  = N8.4"
0110 MULTI "DEPTFILE",100,FMT=DEPTFMT$
```

An extended description of this example and others is contained in Appendix B.

Format NEXT num-variable

where num-variable is the variable to be incremented or decremented.

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

Refer to the FOR/NEXT directive in this section.

Format

ON int-expr GOSUB stno-list

where:
int-expr is a numeric variable or expression that evaluates as an integer to determine the next statement number to be executed.

stno-list is a list of statement numbers, separated by commas, specifying the line number to execute for each value of integer-expr.

Description

The ON/GOSUB directive functions the same as the ON/GOTO directive but performs a GOSUB (refer to GOSUB for additional information).

The ON/GOSUB directive is used to transfer program control to a specified statement number beginning execution of a subroutine. The statement number selected depends upon the value of the integer expression and the relative positions of the statement numbers after the GOSUB.

The first statement number is executed when the value of the expression is less than or equal to 0. The second

statement number is executed when the value of the expression is 1, the third when the value is 2, the fourth when the value is 3, and so on. The last statement number is used for the next value of the expression and all values greater than it. For instance, if the fourth statement number is the last in the list, then it is executed when the integer expression evaluates to 3 or greater.

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

Example

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

If X<=0, the next statement executed is 200.

If X=1, the next statement executed is 300.

If X=2, the next statement executed is 400.

If X>=3, the next statement executed is 500.

The following example shows how ON/GOSUB nay be used in conjunction with the SETERR directive and the ERR system variable. The SETERR directive transfers program control to a specified statement when an otherwise untrapped error occurs. That statement then determines where the program branches based on which error occurred. The particular error is identified by the ERR variable, which is set at each occurrence.

```
0200 SETERR 9000


9000 ON ERR(12,14,17) GOSUB 09100, 09200, 09300, 09400
```

When an error causes the program to branch to statement 9000, statement 9000 transfers control as follows:

ERR	NEXT STATEMENT
12	9200
14	9300
17	9400
OTHER	9100

Format

ON int-expr GOTO stno-list

where:
int-expr = a numeric variable or expression that evaluates

as an integer to determine the next statement number to be **executed.**

stno-list = a list of statement numbers, separated by commas, specifying the line number to execute for each value of integer-expr

Description

The ON/GOTO directive functions the same as the ON/GOSUB directive but performs a GOTO (refer to GOTO for additional information).

The ON/GOTO directive is used to transfer program control to a specified statement number. The statement number selected depends upon the value of the integer expression and the relative positions of the statement numbers after the GOTO.

The first statement number is executed next when the value of the expression is less than or equal to 0. The second statement number is executed next when the value of the expression is 1, the third when the value is 2, the fourth if the value is 3, and so on. The last statement number is used for the next value of the expression and all values greater than it. For instance, if the fourth statement number is the last in the list, it is executed when the integer expression evaluates to 3 or greater.

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

Example

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

    If X<=0, the next statement executed is 200.

    If X=1, the next statement executed is 300.


    If X=2, the next statement executed is 400.

    If X>=3, the next statement executed is 500.
```

The following example shows how ON/GOTO nay be used in conjunction with the SETERR directive and the ERR system variable.

ON/GOTO
(cont'd)

The SETERR directive transfers program control to a specified statement when an otherwise untrapped error occurs. That statement then determines where the program branches based on which error occurred. The particular error is identified by the ERR variable, which is set at each occurrence.

```
0200 SETERR 9000
```

```
9000 ON ERR(12,14,17) GOTO 09100, 09200, 09300, 09400
```

When an error causes the program to branch to statement 9000, statement 9000 transfers control as follows:

ERR	NEXT STATEMENT
12	9200
14	9300
17	9400
OTHER	9100

Format

```
OPEN {INPUT} (fileno f,ERR=stno} {,SEQ=int-expr{)
    "file/device-ID"
```

where the integer expression following SEQ= specifies the sequence number of the file on tape.

Description

The OPEN statement is used for two purposes:

1. To permit a user to access a specified disk data file for subsequent input/output operations;

2. To allow a user to reserve a specified input/output device for his/her exclusive use.

Each user may access (OPEN) a maximum of 64 files and/or devices (numbered 0 - 63) at any given time. The terminal running the user program is always available for opening. Logical unit 0 is not available for opening.

Additional files/devices can be opened by closing those files/devices that are no longer needed.

If the INPUT option is used, the file or device is opened for input only, and cannot be written to . An attempt to

write to a file/device opened with OPEN INPUT generates an ERROR 18.

The SEQ= option allows opening files on tape by specifying the sequence number of the desired file on the tape.

The "file/device-id" string expression will automatically be translated to the actual file/device-id when file name translation is in effect (refer to SETTRANS).

Examples

```
0010 OPEN (1) "ADOOR"

0020 OPEN INPUT (2,ERR=00050) "DONTCHANGE"

0030 OPEN  (7)"LP"

0040 OPEN (3,SEQ=4) "R0"
```

Format

PACK U fileno {,RETAIN} {,ERR=stno}) {var-list}

Description

The PACK directive modifies the contents of the retain buffer with the data in the variable list.

If the RETAIN clause is used, the record currently in the retain buffer is modified.

If the RETAIN clause is not used, the record currently in the retain buffer is replaced. Any fields in the record not specified are filled with nulls.

WRITE with the RETAIN option is used to write the contents of the retain buffer.

Examples

```
1000 PACK(1)
```

Clears the retain buffer for logical unit 1.

```
1500 PACK(1,RETAIN)A$,C
```

Merges A$ and C with the current contents of the retain buffer.

Format

PRECISION int-expr

where the int-expr has an integer value between 0 and 14.

Description

The PRECISION directive is used to change the number of places to the right of the decimal point that will be used in calculations and for display.

PRECISION is always reset to 2 when a BEGIN, CLEAR, RESET, END, STOP, RUN or LOAD statement is executed.

Examples

```
0010 BEGIN
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
```

Format

```
PRINT {(fileno {,RETAIN} {,END=stno} {,ERR=stno}
    {,IND=int-expr} {,KEY=str-expr} {,DOM=stno}
    {,TBL=stno})} {,mnemonic} {,var-list} {,IOL=stno} {,}
```

where:

mnemonic = a cursor/print head positioning mnemonic

var-list = one or more numeric or string expressions.

NOTE

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

Description

The PRINT directive is used to PRINT to a file or device. PRINT does not add a character to either the beginning or end of string fields. PRINT precedes a numeric field with a blank. PRINT appends a linefeed CHR(IO) to the last field.

This distinguishes PRINT from WRITE, which does not begin a numeric field with a blank and which appends a linefeed after every field (as a field marker) but does not append a final linefeed after all fields have been written.

A comma (,) at the end of all items suppresses the terminating line feed 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 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.

The RETAIN clause used with a print to a file that is not a multi-keyed file causes the retain buffer, (stripped of a trailing linefeed, if any) to be attached to the normal results of the PRINT. The RETAIN clause used with a print to a multi-keyed file is treated as a WRITE.

Example

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

Format

```
PRINT RECORD (fileno f,END=stno} {,ERR=stno}
     {,SIZ=int-expr} {,DOM=stno} {,IND=int-expr}
     {,KEY=str-expr} {,TBL=stno}) {str-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 one 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.

PRINT RECORD works like a WRITE RECORD, except that PRINT RECORD appends a single linefeed character.

Example

```
0130 PRINT RECORD(3,ERR=00340)A$
```

Format

PSAVE "prog-ID" {,ERR=stno}

Description

The PSAVE directive performs a protected SAVE, that is, it saves the program in user memory in encrypted form to the specified "program-id" on disk.

A PSAVE'd program will run the same as an unprotected program . Any attempt to LIST, SAVE, EDIT, DELETE or MERGE statements generates an ERROR 18.

The program name must be supplied to PSAVE. If the file already exists and is large enough, the system saves the program. If the program exists but is not large enough, PSAVE automatically resizes it. In this case the original program is not deleted until it has been successfully saved.

The BB86 standard and BOSS/IX specify that PSAVE cannot be used in public programs; however, it is allowed there on BOSS/VS systems? CAUTION: PSAVE on BOSS/VS actually removes any reference to the original source (listed version of the program). Once PSAVE'ed, a program's souurce cannot be recovered. Hence care should be taken to ensure that a source copy of the program is retained.

PSAVE cannot create or write a remote file.

Example

```
>LOAD "ORDINARY"
>PSAVE "ENCRYPTED"
```

Format QUIT

Description The QUIT directive closes files and then releases the

task's memory. In most cases, QUIT'ing returns you to the system command level.

QUIT will take you back one level. If you came from the menu system, you go back there. If you came from another BASIC, you go back to that BASIC: in BASIC, type !BASIC and then, at the prompt, type QUIT; this takes you back to the other BASIC.

Examples

```
0010 QUIT
>QUIT

!BASIC
>!BASIC
>QUIT
>QUIT
!
```

Format

```
READ  f(fileno {,RETAIN } {,END=stno} {,ERR=stno}
    {,IND=int-expr} {,KEY=str-expr} {,TBL=stno}
    {,SIZ=int-expr} {,DOM=stno} {,TIM=time-expr})}
    {,mnemonic} {variable-list} {,IOL=stno}
```

NOTE

A comma is inserted before IOL= only when both IOL= and a variable list, or multiple IOL= entries, or positioning mnemonics are used.

Description

The READ directive is used to read data from a file or device. It is identical to the INPUT directive, except that READ is usually used for input from files while INPUT is used for input from the terminal.

The RETAIN clause places the data just read into the retain buffer for the logical unit. This may be used in later UNPACK, PACK and WRITE directives.

The fields are read into their respective variables in the READ statement. If a field contains non-numeric information, and the corresponding variable is numeric, an ERROR 26 is generated. However, a numeric field can be read into a string 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, string constants and mnemonic constants are not allowed.

Specific information for different file types is given below .

Direct File

A Direct file can be READ either with or without the KEY= option. If a key is not specified, the directive reads the record with the next higher key value. When the READ operation is complete, the "next key" pointer is updated , 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 a key pointer to be updated, FIND or EXTRACT should be used instead of READ. (Refer to Appendix B for how READ may be used with MULTI-KEY files.)

Examples

Reading and writing a Direct file.

```
0010 REM "PROGRAM 1 -- UPDATE PRICES"
0020 BEGIN
0030 OPEN (1)"AA"
0040 INPUT (0,ERR=00040)"PRODUCT  NUMBER OR END:",AS:<"
0040:END"=01000,LEN=1,5)
0050 EXTRACT  (1,ERR=00500,KEY=AS)*,A,B
0060 PRINT "OLD PRICE IS ",B
0070 INPUT (0,ERR=00070)"ENTER NEW PRICE ",B:(99999.99)
0080 WRITE (1)AS,A,B
0090 GOTO 00040
0500 IF ERR<>ll THEN GOTO 00600
0510 PRINT "INVALID PRODUCT  ENTERED.  PLEASE RE-ENTER"
0520 GOTO 00040
0600 IF ERR<>0 THEN GOTO 00700
0610 PRINT "RECORD FOR THIS PRODUCT  IN USE. WAITING"
0620 GOTO 00050
0700 PRINT "ERROR: ",ERR:"00","OCCURRED ON READ"
0710 STOP
1000 PRINT "END OF JOB"
1010 STOP
16000 END
```

```
0010 REM "PROGRAM 2 ?  READ DIRECT  FILE IN SEQUENCE AND
0010:PRINT PRICE"
0020 BEGIN
0030 OPEN (1)"AA"
0040 READ (1,END=01000)AS,*,B
0050 PRINT "PRODUCT-",AS,"   PRICE: ",B
0060 GOTO 00040
1000 PRINT "ALL PRODUCTS AND PRICES PRINTED"
1010 STOP
9999 END
>
```

Sort File

The READ statement for Sort files cannot specify any data to be read, since it is a key-only file.

Examples

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:

READ (cont'd)

```
0010 REM "CREATE SORT FILE"
0020 SORT "SORT", 10,50,0,0
0030 OPEN (1)"SORT"
0040 FOR 1=1 TO 50
0050 WRITE (1,KEY=STR(I:"00000")+"AAAAA")
0060 NEXT I
0070 REM "READ SORT FILE SEQUENTIALLY AND PRINT KEYS"
0080 READ (1,KEY="",DOM=120)
0090 LET K$=KEY(1,END=00200)
0100 REM "    K$ CONTAINS THE KEY OF SORT FILE"
0120 PRINT "KEY=",K$
0130 READ (1)
0140 REM "READ IS NECESSARY TO ADVANCE TO NEXT KEY"
0150 GOTO 00120
0160 REM "END OF FILE"
0170 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 "MASTER" 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.

This sample program prints an alphabetic listing of all the customers in the master file which have a non-zero amount due:

```
0010 OPEN (1)"MASTER"
0020 OPEN (2)"NAME"
0030 OPEN  (7)"LP"
0040 REM "  K$ CONTAINS THE FIRST 5 CHARACTERS OF CUST NAME
0050 REM "  PLUS THE CUSTOMER CODE IN POSITION 6-10
0060 LET K$=KEY(2,END=01000)
0070 REM "  CUSTOMER CODE IS THE KEY TO FILE "MASTER"
0080 READ (1,KEY=K$(6,5)>A$,B$,*,D,*
0090 REM "  THE VARIABLE D CONTAINS THE AMOUNT DUE
0100 REM "  IF NOT ZERO, THE CUSTOMER WILL BE LISTED
0110 IF DO 0 THEN PRINT (7)"CUST CODE",A$,"NAME: ",B$,"AMT:",D
0120 READ (2)
0130 GOTO 00040
9999 END
```

Indexed and Serial Files and Peripheral Devices

READ statements for Indexed or Serial files 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)"ADDRESS"
0040 OPEN  (7)"LP"
0050 READ (1,END=00100)A$,BS,C$,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 00050
0110 CLOSE (1)
0120 CLOSE (7)
0130 END
```

Format

```
READ RECORD  (fileno {,DOM=stno} {,END=stno} {,ERR=stno}
     {,IND=int-expr} {,KEY=str-expr} {,TBL=stno} {,TIM=time}
     {,SIZ=int-expr}) str-variable
```

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.

When READ RECORD is done from the half-inch tape device R0, the string variable must be previously allocated to be at least as big as the record to be read from the tape. The reason for this is that the READ RECORD is done directly from the tape into the string variable. This improves the performance of tape reads and makes it possible for the tape to stream.

Refer to INPUT RECORD for additional information.

Example

```
0100  DIM A$(4096)
0110  READ RECORD(1,END=00900)A$
```

Format

RELEASE {"task-id"}

where task-id is the terminal or ghost task identifier.

Description

The RELEASE directive CLOSE'S all files opened by a task, and releases the task's memory. RELEASE can release any Ghost task or the process executing the RELEASE (i.e., itself). RELEASE without a task ID logs the user off of the system.

When a task releases itself, any job in process is terminated and the screen is cleared. The terminal is then inactive until ESCAPE is pressed, or it is opened by another task.

Examples

```
>RELEASE

>RELEASE "G1"

>!RELEASE *

>RELEASE "T2"
```

Format

REM {{"}str-expr{"}}

where the string expression is 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, for EOSS/IX, to ensure that any blanks within a remark are retained.

Example

```
0010 REM "PROGRAM TO GENERATE PAPERWORK"
```

Format

```
REMOVE (fileno {,KEY=atr-expr} {,DOM=stno} {,ERR=stno}
     {,END=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 KEY= parameter is optional when an EXTRACT is performed on the record to be removed. If present, the parameter refers to the primary keyset of a multi-keyed file, or the key of a direct file. If absent, the record that was previously EXTRACT'ed is removed (but only if no other I/O operations occur on that logical unit between the EXTRACT and the REMOVE).

Example

```
0100 REMOVE (1,KEY=K$,DOM=9000)
```

Format RENAME "old-file-ID", "new-file-ID" i,ERR=stno{

Description The RENAME command changes the name of a file.

If the file to be renamed does not exist, an ERROR 12 is generated.

If the new file name is already the name of a file, an ERROR 12 is generated.

After RENAME has successfully executed, the old file name no longer exists.

RENAME cannot be used to rename files across filesystem (BOSS/IX) or family (BOSS/VS) boundaries. Attempting to do so generates an error.

RENAME cannot reference remote files.

Example

```
>RENAME "OLDFILE","NEWFILE"
```

Format RESET

Description The RESET directive performs a BASIC system reset that affects only the task that issued the statement.

RESET resets the ERR and CTL system variables (to zero) and any GOSUB and FOR/NEXT loops that have not been fully executed. The RESET statement also re-establishes the

arithmetic mode at PRECISION 2 and any statement numbers active for SETESC or SETERR are cleared.

Execution of the RESET statement does not clear the user data area, close and open file or devices, or reset the program execution pointer, which identifies the next statement to be executed.

Example

```
>RESET
>0100 RESET
```

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 directives.

When a SETERR statement is branched into, it normally will be reset to 0 (to keep errors within a program 's error

handling from causing infinite loops); RETRY restores SETERR to its pre-error value.

Example

```
0010 REM "PROGRAM TO INPUT NEW CUSTOMERS"
0020 BEGIN
0030 OPEN  (1)"MASTER"
0040 LET P$="00000"
0050 INPUT (0,ERR=0210)'CS'," CUSTOMER NUMBER (CR TO END)
0050:",N:(99999)
0060 IF N=0 THEN STOP
0070 LET N$=STR (N:P$)
0080 FIND (1,DOM=0120,KEY=N$)
0090 INPUT (0,ERR=0090)@(0,22),'RB',"CUSTOMER ON FILE
0090:(DEL TO DELETE , CR TO CONTINUE :,T$:("DEL" =0100,""
0090:=0050)
0100 REMOVE (1,KEY=N$)
0110 GOTO 0050
0120 SETERR 0210
0130 INPUT @(0,1),"ADDRESS",AS:(LEN=0,30)
0140 INPUT @(0,2),"CITY",C$:(LEN=0,15)
0150 INPUT @(0,3),"STATE", S$:("CA"=00160,"AZ"=00160,
0150:"OR"=00160)
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$,Z,B
0200 GOTO 0050
0210 INPUT (0,ERR=0210)@(0,22),'RB',"INVALID (CR TO
0210:CONTINUE) ",T$:(""=0220)
0220 RETRY


 8000 REM "ERROR HANDLING ROUTINE"
```

Format RETURN

Description The RETURN directive is used to terminate a GOSUB, SETESC, or SETCTL routine. It returns program control to the statement following the GOSUB and the SETCTL or, in the case of SETESC, to where it left off.

RETURN can be used only in program mode.

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 . If no program ID is specified, the program currently in user memory is run. If a program ID is specified, the program is loaded and then run.

When the program ID is specified, RUN will LOAD the program, clear FOR/NEXT, GOSUB, SETERR and SETESC addresses, and reset PRECISION to 2. Program execution begins at the lowest line number.

In general, RUN begins program execution at the line number currently selected by the program line pointer. For a newly loaded program, this is the lowest line number. If a program is interrupted or in some other way the program line pointer is pointing at some other program line, RUN begins execution at this point. This is usually caused by execution of a GOTO in console mode or previous interruption of program execution by an ESCAPE or some other interruption other than an END or STOP.

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 .

The program area is not cleared until it has been determined that the specified program can be LOAD'ed.

Examples

```
>RUN

0400 RUN "AMOK"
```

Format

SAVE {prog-ID} {,int-expr}

where the integer expression specifies the program size in bytes <maximum= 32,767 on BOSS/IX and 65,536 on BOSS/VS).

Description

The SAVE directive is used to copy a program from user memory to a Program file on disk.

Generally, SAVE is used with no arguments, or with only the file ID argument. When no program ID is specified, the program is SAVE'd into the currently LOAD'ed program file.

If SAVE is executed with only the file ID and the program file does not exist already, the file is created.

If the program file exists but is not large enough, the file is enlarged. The original file is not deleted until the enlarged program has been successfully saved.

BOSS/IX and the BB86 standard do not permit SAVE to be used in a public program (BOSS/VS does).

SAVE cannot create or write a remote file.

Examples

```
>SAVE "STAMPS"
```

SERIAL

Format

SERIAL "file-ID", av-recno, av-recsz {,ERR=stno}

where:

av-recno = the average number of records in the file

av-recsz = the average size, in bytes, of each record in the file

Description

SERIAL creates a Serial file.

The average record size and average number of records are multiplied to obtain the number of bytes required for the file.

Part of the space allocated is used for system overhead. The amount of overhead is system dependent, and should be allowed for when defining the file.

Rules for using Serial files are as follows:

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. Record-to-record movement of the index can be forward or backward (though backward movement might be slightly less efficient).

The SERIAL directive can also be used with a complete access list, but the two cannot be mixed.

Examples

```
>SERIAL "NAME",100,80
```

Defines a serial file called "NAME" with an average number of records equal to 100 and average record size equal to 80, in the user's working directory.

Note that since space is determined by multiplying the average number of records and the average record size, the same amount of space would be allocated by:

```
>SERIAL "NAME",50,160
```

Format SETCTL stno

Description The SETCTL directive is used to cause branching which the operator enters <CTL>+<Y> (holding down the CTL key while pressing then releasing Y). When this combination is entered, control is transferred to the statement number set by the currently active SETCTL statement. If there is no SETCTL in effect, the sequence is treated like other keystroke combinations, and no special action is taken.

The branch defined by SETCTL is like a GOSUB. Processing continues in the subroutine until a RETURN statement is encountered. Control is then transferred to the statement following the point of execution when <CTRL>+<Y> was pressed.

SETCTL has an error-stacking feature which preserves the ERR value immediately before <CTRL>+<Y> was pressed. Even though the value of ERR may change during the SETCTL sub-routine, the value is restored when control RETURN'S to the main program.

Example

```
0010 SETERR 0100;SETCTL 0200
0020 PRINT 1/0
0100 PRINT "AFTER TAKING SETERR BRANCH, ERR: ",ERR
0110 INPUT "HIT <CTRL>+<Y> THE FIRST TIME, HIT RETURN THE
0110:SECOND TIME ",A$
0120 PRINT "AFTER RETURN FROM SETCTL, ERR: ",ERR
0130 STOP
0200 PRINT 'LF',"IN SETCTL, ERR: ",ERR
0210 OPEN(1,ERR=0220)"DKFLJ"
0220 PRINT "AFTER OPEN, ERR: ",ERR
0230 RETURN


>RUN
AFTER TAKING SETERR BRANCH, ERR: 40
HIT <CTRL>+<Y> THE FIRST TIME, HIT RETURN THE SECOND TIME
IN SETCTL , ERR:126
AFTER OPEN, ERR: 12
HIT <CTRL>+<Y> THE FIRST TIME, HIT RETURN THE SECOND TIME
AFTER RETURN FROM SETCTL, ERR: 40

READY
>
```

Format

SETDAY str-expr

where the string expression is an 8-character string specifying the date.

Description

The SETDAY directive is used to set the value returned by the system variable DAY.

The argument must be a string, eight characters in length. The format most commonly used is "MM/DD/YY", though alternate formats may be specified by the system administrator.

An improper length results in an error 46, and an improper date returns an error 17.

Example

```
SETDAY "03/31/87"
```

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 re-execution. 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=, DOM=, END= clauses take 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 statement where the error occurred.
- 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, unless a RETRY is performed. 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 or to allow normal termination of a program to occur.

SETESC causes the program to branch if ESCAPE is pressed. The system executes a GOSUB to the line number specified in the SETESC statement. Following a RETURN, the system resumes processing at the point.from which the SETESC branch was taken.

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

SETESC has an error-stacking feature which preserves the ERR value immediately before <ESCAPE> was pressed. Even though the value of ERR may change during the SETESC sub-routine, the value is restored when control RETURN'S to the main program. (Refer to SETCTL for a closely related example.)

Example

```
0010 BEGIN
0050 SETESC 9000
0059 REM "ESCAPE KEY WILL BE PRESSED DURING EXECUTION
0059:OF 60 OR 70
0060 LET A=A+1,B=B+1,C=C+1
0065 IF A>100 STOP
0070 GOTO 0060
9000 REM
9001 REM "ESCAPE ROUTINE"
9002 REM
9003 PRINT "YOU CANNOT ESCAPE"
9004 RETURN
```

Format

```
SETFIELD file-ID, FMT=str-expr f,MSG=str-expr}
    {,ERR=stno}
```

Description

The SETFIELD directive changes the keyset type for a field in a multi-keyed file.

The file cannot be open when the change is made.

The PRIMARY field may not be changed, and no other field may be declared PRIMARY by SETFIELD .

The FMT clause specifies the field name, followed by "=", and then by the new keyset type for the field, either ALTKEY, DUPKEY, or NOKEY.

There is a possibility of generating an error when changing a field from DUPKEY to ALTKEY, in the case that there was a duplicate key in the field.

The optional MSG= clause displays a message at the cursor position followed by a running percentage complete value. The final percentage displayed is 100%. The message may contain positioning mnemonics. If the MSG= clause is not specified, no percentage complete is displayed.

Example

```
1300 SETFIELD "DEPTFILE", FMT="DEPTNAME#=DUPKEY",
1300:MSG=@(65,20)+"Progress: "
```

See Appendix B for further examples of the use of SETFIELD and multi-key files.

Format

SETTIME num-expr

where numeric-expr has a value between 0.00 and 24.99 representing the time (e.g., 13.50 = 1:30 p.m.). The following formula can be used to determine the proper format:

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

where H = Hours, M = Minutes and 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
0120: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)O 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),
0200:S=S-M*60
0210 PRINT "DATE IS",DAY
0220 PRINT "TIME IS",H:"00",":",M:"00",":" ,S:"00"
0230 STOP
```

Format SETTRACE {(fileno)}

Description

The SETTRACE directive initiates the listing of statements as they are executed. SETTRACE is especially useful when debugging a program that appears to be branching in an unforeseen or undesirable manner. The resulting listing shows the exact sequence in which program statements are being executed.

The SETTRACE listing can be sent to a file or printer by opening a channel to the desired device and specifying the channel number. In all cases, the SETTRACE listing is in LIST format.

SETTRACE can be used as a statement within the program at selected points until the program is debugged. SETTRACE can also be entered in console mode to begin the listing of executed statements. In either 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 is not ready, 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. The statement listed may not be in error.

In BB86, there is no difference between the display of traced statements which come from a main program and statements which come from a CALL'ed program .

Example

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

>RUN
0010 FOR 1=1 TO 3
0020 LET A=I+1; NEXT I
0020 LET A=I+1; NEXT I
0020 LET A=I+1; NEXT I
END

READY
>
```

Format

SETTRANS file-ID {,ERR=stno!

where file-id is the name of the file containing the translation instructions.

Description

The SETTRANS directive initiates the translation of string expressions in BASIC statements.

The string expressions representing file IDs in the following BASIC directives' arguments are translated when translation is active:

CALL	INITFILE	OPEN	SERIAL
CREATE	LISTPROGRAM	PSAVE	SETFIELD
DIRECT	LOAD	RENAME	SETTRANS
ENCRYPT	MAKE PROGRAM	RUN	SORT
ERASE	MULTI	SAVE	START
INDEXED			

Note: in the preceding list, the underlined directives cannot reference a remote program file.

The string expressions representing task IDs in the following directives are affected when translation is active:

OPEN
RELEASE
START

The string expressions used as arguments in the following directives are affected when translation is active:

!
SYSTEM

The string expressions representing file IDs in the following BOSS/IX directives are affected when translation is active:

ADDx	FILE	STRING
DROP	PROGRAM	VMERGE

Note: in the preceding list, the underlined directives cannot reference a remote program file.

The following BOSS/VS directives also are affected:

FILE
PROGRAM

Translation File Format

The translation file is created using the system text editor. On BOSS/IX systems, the file is a String file; on BOSS/VS systems, the file is a Serial file.

The file may contain blank lines, comments, translation lines, wild-card characters and continuation lines.

Leading blanks on a line are ignored.

Comments

Blank lines are treated as comments.

If the first two characters (following leading blanks) are the same, the line is treated as a comment. E.G.:

```
!! a comment
%% a comment
```

Wild-cards

A wild-card character is provided for pattern matching. It is specified by typing a single character, the character to be the wild-card, on a line. The default wild-card character is the asterisk (*). A wild-card character matches 0 to any number of characters.

Translation Lines

Translation lines consist of a delimiter, a "left part", a delimiter, and a "right part." The first character of a non-comment line becomes the delimiter for that line. (Each translation line defines its own delimiter.) The second occurrence of the delimiter separates the left part from the right part. E.g.:

```
:leftpart:rightpart
/leftpart/rightpart
```

The maximum number of characters in a translation line is 256. This limit includes the left part, the right part,

and the two delimiters.

Continuation Lines

Continuation lines are necessary for any line over 80 characters. To indicate that the translation line has one or more continuation lines, end the translation line with the delimiter. Continuation lines must begin in column one (no leading blanks). There is no limit to the number of continuation lines, but no translation line can be longer than 256 characters.

Duplicate Entries

If the translation file contains more than one line with the same left part, only the first translation line is used; all subsequent lines with the same left parts are ignored.

Translation Process

The string expression subject to translation is taken as the comparison string. This comparison string is then compared to the left parts in the translation file. Comparisons are first made to left parts that do not use wild-card characters, and then, if no match is found, against left parts with wild-card characters.

When a match is found, the search is ended and the right part of that translation line is used instead of the comparison string. If no match is found, no translation takes place.

Translation remains in effect until an ENDTRACE directive is encountered. Translation of string expressions occurs

before execution of the directive. Ghost tasks inherit the translation file of the initiator.

Examples

A Translation File:

```
++ The default wild-card is *
!! These first two lines are comments

:ABC:(DISK ).FMF.ABC
\DEF\/sta/titanic/include/defines

:a long:
left part:
with a:
:right part

:: Comment; the next line changes the wild-card
#


:#pre:gran-o1e-opry
:i#n:ION
*
:post*:post hoc ergo propter hoc
```

Given the following lines in "transfile" on a BOSS/IX system :

```
    :error:/include/bfs/error.h
    :DATA:/sta/titanic/src/oms/DATA
```

SETTRANS (cont'd)

and given the following lines in "transfile" on a BOSS/VS system :

```
:error:(DISK).FMT.ERR.X.ERROR
:DATA:(DISK).LAN.MST.DATA
```

the following program can run on each system and open the appropriate files:

```
0010 SETTRANS "transfile"
0020 OPEN  (1) "DATA"
0030 OPEN  (2) "error"

0040 ENDTRANS
```

BASIC programs can make use of system dependent features and still remain transportable with the use of the translation facility.

For example, given the following in a line in a BOSS/IX translation file:

```
:dir:ls -l |p
```

and this line in a BASIC program:

```
0010 !dir
```

the program will give a directory listing on either system. Note, however, that wildcards are applied to each translation of a string, not just to file-id's. This means that the translation line

```
:d*:.dfiles.d*
```

will turn 0010!dir into 0010!.dfiles.dir

Format

```
SORT "file-id", keysz, recno {,ERR=stno}
```

where:
keysz = the size of the key, in characters (nunimum=l, maximum=56)

recno = the maximum number of records in the file (maximum=8,388,608)

Description

The SORT statement is used to define a Sort file.

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

SORT can create a remote file.

Example

```
SORT "ACUTE",15,100
```

Format

START fpages} {,ERR=stno} {,prog-ID} {,task-id}

where:

pages = an integer expression specifying the number of pages (256 bytes per page) assigned to the task (BOSS/IX: min=10, max=65535; no effect on BOSS/VS).

prog-id = the name of the program to be run, enclosed in quotation marks. A full path name is required on BOSS/IX, but is not required on BOSS/VS. It may not reference a remote file.

task-id ? the string expression representing the ghost task being started, enclosed in quotation marks (e.g., "GO", "Gl", ...).

Description

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

START can be used in both program mode and console mode.

The START directive will only affect the currently executing task (i.e., the one executing the START directive) or

a ghost task, depending upon the statement.

START with no arguments restores user memory from a previous START. A START with no arguments clears the program variable tables and data areas and closes any open files.

For BOSS/IX, if no number of pages is assigned, the amount of memory assigned to the task is not changed.

Although BB86 will accept partial file path names, full path names are recommended in BOSS/VS to guarantee transportability.

START cannot reference a remote program file.

Examples

BOSS/IX

```
START 30, "/bin/isys/PROG", "G0"
```

BOSS/VS

```
START "(DISK).lan.prog", "G0"
```

Format STOP

Description The STOP directive is used to terminate a program at any point other than the end of that program.

STOP resets the program execution pointer to the first statement of the program, closes all open files and devices, resets ERR and CTL variables, clears the RETURN/NEXT stack, and leaves the task in console mode.

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

STO P is identical in function to END, except that END is used to terminate program loading during a MERGE operation.

Example

```
6510 STOP
```

Format

SYNTAX str-expr {,ERR=stno}

where string-expr contains a BASIC statement.

Description

The SYNTAX directive is used to check the string expression for syntactical correctness.

If the string expression passed to SYNTAX represents a directive that is permitted only in program mode, then the string expression must contain a line number.

If the syntax is incorrect, an error is generated; otherwise no additional action is taken.

Examples

```
>LET AS = "IF A=0 THEN B=0"
>SYNTAX AS
>RUN

READY
>
```

```
0010 LET AS = "IF A=0 THEN B"
0020 SYNTAX AS, ERR=100

0100 PRINT "Syntax error found"

>RUN
Syntax error found
```

Format

SYSTEM str-expr

where string-expr is a system directive to be executed at the system level.

Description

The SYSTEM directive allows the user to execute a system directive and remain in the BASIC environment. It is similar to the "i" directive, except that is followed by an unquoted literal while SYSTEM is followed by a proper string expression. This syntactical difference allows SYSTEM (unlike "!") to appear at the beginning or in the middle of compound statements, since the scope of the directive is unambiguous.

Example

```
12000 REM SUBROUTINE TO EXECUTE SYSTEM COMMANDS
12010 INPUT "ENTER SYSTEM COMMAND TO EXECUTE: ", COMMAND$;
12010:SYSTEM COMMANDS;RETURN
```

Note that "string-expr" is subject to automatic translation (refer to SETTRANS).

Format TABLE hexadecimal-string

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

Any input/output instruction that 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, conversion is performed after field terminators are supplied by the system.

The first two digits of the hexadecimal string are used as a mask byte which filters (by the AND function) each input byte. The remainder of the hex string is the code comparison table and can be 256 or fewer bytes.

An AND function is done with the mask byte and each input byte to form a temporary result byte. The AND operation operates at the 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 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 resulting 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.

NOTE

Proper selection of the mask byte reduces the size of the table. If the mask byte is 0111 1111 (7F), as in the examples above, the result byte never exceeds 0111 1111 (7F), 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.

TABLE (cont'd)

Example

The following paragraphs provide an 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, a second table can be built showing all possible masked decimal values and their corresponding hexadecimal values.

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, so 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 4-2. Table Statement Table

Column 1	Column 2	Column 3	Column 4	Column 5
EBCDIC CHAR	EBCDIC BINARY VALUE	MASKED DECIMAL VALUE	ASCII CHAR EQUIV	OUTPUT HEX VALUE BB7 BB8
A	1100 0001	1	A	41 C1
B	1100 0010	2	B	42 C2
C	1100 0011	3	C	43 C3
D	1100 0100	4	D	44 C4
E	1100 0101	5	E	45 C5
F	1100 0110	6	F	46 C6
G	1100 0111	7	G	47 C7
H	1100 1000	8	H	48 C8
I	1100 1001	9	I	49 C9
J	1101 0001	17	J	4A CA
K	1101 0010	18	K	4B CB
L	1101 0011	19	L	4C CC
M	1101 0100	20	M	4D CD
N	1101 0101	21	N	4E CE
0	1101 0110	22	O	4F CF
P	1101 0111	23	P	50 DO
Q	1101 1000	24	Q	51 D1
R	1101 1001	25	R	52 D2
S	1110 0010	34	S	53 D3
T	1110 0011	35	T	54 D4
U	1110 0100	36	U	55 D5
V	1110 0101	37	V	56 D6
W	1110 0110	38	W	57 D7
X	1110 0111	39	X	58 D8
Y	1110 1000	40	Y	59 D9
Z	1110 1001	41	Z	5A DA

BOSS/IX (low-order ASCII)

```
Masked Decimal Value    0  1  2  3  4  5  6  7  8  9 10*11*12*13*14*
Output Hex Value       20 41 42 43 44 45 46 47 48 49 20 20 20 20 20

Masked Decimal Value   15*16*17*18 19 20 21 22 23 24 25 26 27 28 29
Output Hex Value       20 20 4A 4B 4C 4D 4E 4F 50 51 52 20*20*20*20*

Masked Decimal Value   30 31 32 33 34 35 36 37 38 39 40 41 42 43 .. 63
Output Hex Value       20*20*20*20*53 54 55 56 57 58 59 5A 20 20 .. 20

0100 TABLE 3F4142434445464748492020202020
           204A4B4D4E4F505152202020202020202020
           535455565758595A2020202020202020
           20202020202020202020202020202020
```

BOSS/VS (high-order ASCII)

```
Masked Decimal Value    0  1  2  3  4  5  6  7  8  9 10*11*12*13*14*
Output Hex Value       AO CI C2 C3 C4 C5 C6 C7 C8 C9 AO AO AO AO AO

Masked Decimal Value   15*16*17*18 19 20 21 22 23 24 25 26 27 28 29
Output Hex Value       A0 A0 CA CB CC CD CE CF DO D1 D2 A0*A0*A0*A0*

Masked Decimal Value   30 31 32 33 34 35 36 37 38 39 40 41 42 43 .. 63
Output Hex Value       A0*A0*A0*A0*D3 D4 D5 D6 D7 D8 D9 DA AO AO .. AO
```

```
0100 TABLE 3FA0C1C2C3C4C5C6C7C8C9A0A0A0A0A0A0
           A0CACBCDCECFD0D1D2A0A0A0A0A0A0A0
           D3D4D5D6D7D8D9DAA0A0A0A0A0A0A0
           A0A0A0A0A0A0A0A0A0A0A0A0A0A0A0
```

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. It has also been used to convert between Basic Four character sets and different languages such as English and German.

Format UNLOCK (fileno {,ERR=stno})

Description The UNLOCK directive unlocks files and devices locked by the LOCK directive, allowing access to the file by other users.

A locked file automatically becomes unlocked when the file is CLOSE'd.

Example

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

Format UNPACK (fileno {,ERR=stno}) var-list

Description The UNPACK directive reads values from the retain buffer for the specified logical unit, and assigns those values to the variables. This is analogous to the way READ takes values from the I/O buffer.

Parts of the retain buffer that have not been filled are considered to be filled by nulls. No error is generated.

Example

```
1500 UNPACK (1)A$,C
```

See Appendix B for further examples (though UNPACK is not limited to just multi-key files).

Format

WAIT seconds

where "seconds" is a numeric expression specifying the number of seconds 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 128,000 seconds.

The number of seconds can be given in tenths of seconds. No ESCAPE can occur during the wait period. Caution should be taken not to enter too large a wait amount. If ESCAPE key interruption is desirable, a small wait period and a counter could be used.

Example

```
0200 WAIT 2

0200 WAIT 59.6
```

Format

```
WRITE {(filno {,RETAIN} {,DOM=stno! {,END=stno}
    {,ERR=stno} {,IND=int-expr} {,KEY=str-expr}
    {,SIZ=int-expr} {,TBL=stno})} {,mnemonic}
    {,variable-list} {,IOL=stno}
```

NOTE

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

Description

The WRITE directive is similar to the PRINT directive except that the system automatically appends a one-byte line feed CHR(10), or CHR(138) on EOSS/VS, as a record termitor after every record field it outputs. Also unlike the PRINT directive, WRITE does not precede a numeric field with a blank.

Since keys are contained among the records of a multi-keyed file, the KEY= clause is not allowed in writes to a multi-keyed file. This generates a run-time error.

Multi-keyed file composite fields must not be specified in the variable list.

The RETAIN clause causes the data in the retain buffer to be written, updated with any fields specified in the variable list. The retain buffer itself is not modified by this operation.

Mnemonic constants and positioning expressions, if included as parameters, are output as data to devices other than terminals and printers.

Direct Files

Unless an EXTRACT preceded the WRITE operation (see EXTRACT), a Direct or Sort file WRITE statement must include a key. The system searches the key area to see if the key already exists in the file. If the (primary) key already exists, the new record is written over the old record (unless the DOM= clause was specified). The operation is then complete. If the key does not exist, the system must find space for the key and data.

Example

```
1000 WRITE (2,KEY="JSMITH") NAME$,FIRST$,ACCT
```

Format

```
WRITE RECORD (fileno {,DOM=stno} {,END=stno}
    {,ERR=stno} },IND=int-expr} {,SIZ=int-expr}
    {,KEY=str-expr} {,TBL=stno}) {string-variable}
```

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.

The KEY= clause may not be used for writing to a multi-keyed file. An attempt to do so generates a run-time error.

Example

```
0100 WRITE REC0RD(1)A$
```

See Appendix B for the use of WRITE's in multi-key files or with the RETAIN clause.

NOTES

NOTES

INTRODUCTION

The functions described in this section are commands built into the system that are used to manipulate data for a variety of reasons.

One category of functions contains the binary conversions used in Boolean algebra. These functions are used primarily in testing whether relationships are true or false, on or off, open or closed. The functions included here are AND, IOR (logical or), NOT (inverse string), and XOR (exclusive or).

A second category of functions contains the various conversions based on the ASCII table. The functions included here are ATH (ASCII characters to hexadecimal value), HTA (hexadecimal value to ASCII characters), ASC (ASCII character to decimal number), CHR (decimal number to ASCII character), BIN (decimal number to binary), and DEC (binary to decimal number). These functions are used to "pack" and "unpack" data in limited memory space, to convert data to recognizable or useful representations, to affect speed of processing, etc.

There are a variety of functions besides the types mentioned above. Some are used in data transmission to check data integrity, some for requesting file identification information, some for performing arithmetic functions such as modulo and absolute value, and some are used for compiling statements.

Besides these defined functions, the system allows you to define 63 other functions using the FNx directive.

Format ABS (nuneric-expr)

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<-6.23)   - assigns the value 6.23 to X
```

Format AND (str-expr, str-expr)

Description The AND function takes two string expressions as arguments and returns a single string expression as its result. The resulting string is obtained as the value of the bit-wise logical conjunction (Boolean product) of the argument strings, according to the following rules:

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

In other words, a bit in the result string is set to 1 if and only if the corresponding bits in both argument strings are set to 1; otherwise, the bit in the result string is set to 0.

```
LET X$=AND($0F$,$DC$)
   PRINT HTA(X$)
   0C
```

This result is obtained as follows:

```
       $0F$ = 0000 1111
AND    $DC$ = 1101 1100

       $0C$ = 0000 1100
```

Format ASC(str-expr {ERR=stno{)

Description The ASC function returns the numeric ASCII value of a single character. If the string expression is longer than one character, the value returned is the ASC of the first character in the string.

The value returned is system dependent, depending on whether the system uses low-order (7-bit) or high-order (8-bit) ASCII for BOSS/IX and BOSS/VS, respectively.

Examples BOSS/IX

```
0500 LET X=ASC("A")
```

Returns a value of 65 to X.

```
0500 LET X=ASC("ASCII")
```

Returns a value of 65 to X, the value of the first character only.

```
0500 LET X=ASC($41$)
```

Returns a value of 65 to X, the character "A" is given as a hexadecimal string.

BOSS/VS

```
0500 LET X=ASC("A")
```

Returns a value of 193 to X.

```
0500 LET X=ASC("ASCII")
```

Returns a value of 193 to X, the value of the first character only.

```
0500 LET X=ASC($C1$)
```

Returns a value of 193 to X, the character "A" is given as a hexadecimal string.

For a complete form of the ASC function, see "ASCII."

Format

ASCII(str-expr {,ERR=stno})

Description

ASCII takes a string expression as an argum ent and returns the industry standard numeric ASCII code. If the string expression is longer than one character, only the code for the first character is returned; all others are ignored.

This function provides a single, system independent method for producing industry standard ASCII codes. Whereas BOSS/IX systems use the standard ASCII codes (decimal

codes 0 to 127), BOSS/VS systems use high-order equi-valents of ASCII codes (decimal codes 128 to 255).

Refer to the CHAR function for a table describing the ASCII, BOSS/IX and BOSS/VS character codes.

Examples

```
>PRINT ASCII("A")
 65
```

BOSS/IX

```
10 PRINT ASCII($0A$);PRINT ASCII($8A$)

 >RUN
 10
 138
```

BOSS/VS

```
10 PRINT ASCII($8A$);PRINT ASCII($0A$)

>RUN
 10
 138
```

Format ATH (str-expr {,ERR=stno})

Description The ATH function takes pairs of ASCII characters in the argument string, representing numbers in hexadecimal notation, and returns a string of character codes having the represented numeric values.

The argument string can only contain characters 0 through 9 and A through F, since they are representing hexadecimal digits. If the string has an odd number of characters, a 0 is added to the left.

Each pair of characters is taken as a hexadecimal representation of numbers in the range 0 to 255 (hexadecimal 00 to FF). The pair of characters (requiring two bytes) in the argument string is replaced by a single byte in the

result string having the same numeric value.
For example, "Al" is a string consisting of two ASCII

characters, but represents a number in hexadecimal notation. This is replaced in the result string by a single byte with the numeric value hexadecimal Al, (=161, decimal). Accordingly, ATHC' Al") is equivalent to $A1$.

Examples BOSS/IX (low-order)

```
 >PRINT ATH("303132")
012
```

BOSS/VS (high-order)

```
 >PRINT ATH("B0B1B2")
012
```

Format

```
ATTR (fileno {,"{ALL} ( NAME} { OWNER } i USAGE_RIGHTS}
    { ORGANIZATION} { REOORD_SIZE} { REOORDS_ALL0WED}
    { REOORDS_USED} { KEY_SIZE} { INITIAL} { GROWTH}

    { LONG} { SHORT} { WRITE-THRU}"} {,ERR=stno})
```

Description

The ATTR function returns a string containing information of an open file. The specific information returned depends on the file attributes requested.

The file number parameter is required, and must specify an already open channel.

The file attributes returned are determined by the list of attribute names, as shown in the format. In the list of attribute names to be enclosed within parentheses, it doesn't matter whether any letter is upper or lower case. Where underscores are shown, they are required.

When more than one attribute is returned, the attributes are separated by two (2) spaces, A program can search for the two spaces to separate attributes.

The attribute names and the information returned are as follows:

LONG	Returns all attributes following the option in the format "attribute = value".
SHORT	Returns the value only for all attributes following the option in the list. This is the default, unless ALL is specified.

Note: the preceding two items, LONG and SHORT, can be intermixed in the attribute list; for example:

```
PRINT ATTR(1,"LONG OWNER SHORT NAME LONG GROWTH SHORT
RECORDS USED")
```

Thus LONG and SHORT can modify all the following attributes.

ALL	Returns information on all specifiable attributes. Information is returned in LONG format, unless SHORT is specified.
NAME	Returns the full path name of the file opened on the specified channel, beginning with the family or root.

REOORD_SIZE Returns a number for the defined record size. Sort files return a size of zero (0).

RECORDS_ALLOWED Returns the maximum number of records. Devices return zero (0).

REOORDS USED Returns the number of records written into the file. Devices return zero (0).

ORGANIZATION Returns a 3-byte string describing the file organization as follows:

BAS = BASIC Program
COB = COBOL Program
DEV = Device, i.e., not a file
DIR = Direct or Sort file
IND = Indexed file
MUL = Multi-keyed file
PAS = Pascal Program (BOSS/VS only)
SER = Serial file
STR = String file (BOSS/IX only)

KEY SIZE Returns a number for the key size in Direct and Sort files. Returns zero (0) for other file types.

INITIAL Returns the number of records initially allocated. Devices return zero (0).

GROWTH Returns the number of records added to a file each time it needs to be enlarged. Devices return zero (0).

OWNER Returns the account that owns the file.

USAGE-RIGHTS Returns the usage rights of users other than the owner.

WRITE-THRU Returns T or F if it is for that file (T=true, on; F=false, off).

The information returned by ATTR is similar to the in-**Format**ion returned by FID, except that:

- ATTR returns a variable length ASCII string while FID returns a fixed length binary string.
- ATTR can return all the attribute information that FID can, plus initial extent, growth extent, owner and usage rights.

- ATTR allows specification of the attributes to be returned; FID returns everything.

- ATTR is sytem indipendend whereas FID is not. Only the specific, system dependend nformation returned varies, e.g. .file path name format and usage rights format.

Examples

The following examples assume the following file open on channel 1:

```
NAME (BOSS/IX)     /usr/barry/src/myfile
     (BQSS/VS)     (DISK).ABC.MYFILE
ORGANIZATION        IND
RECORD SIZE         120
RECORDS ALLOWED     1000
RECORDS USED        352
KEY SIZE            0
INITIAL             334
GROWTH              150
OWNER (BOSS/IX)     barry
      (BOSS/VS)     MY.ACCT
USAGE_RIGHTS
      (BOSS/IX)     rw. rw.
      (BOSS/VS)     W(SR.*);R(*.*)
WRITE THRU          T
```

```
>A$=ATTR(1, "ORGANIZATION")
>PRINT A$
IND

>LIST$="Long key_size"
>A$=ATTR(1,LIST$)
>PRINT A$
KEYSIZE=0

>A$=ATTR(1,"SHORT KEY_SIZE")
>PRINT AS
>0

>PRINT ATTR(1,"ALL")
```

ATTR (cont'd)

BOSS/IX returns:

```
NAME=/usr/barry/src/rayfile  ORGANIZATION=IND
RECORD_SIZE=1000  RECORDS_ALLOWED=1000  RECORDS_USED=352
KEY_SIZE=0  INITIAL=334  GROWTH=150  OWNER=barry
USAGE_RIGHTS=rw. rw.  WRITE_THRU=F
```

BOSS/VS returns:

```
NAME=(DISK).ABC.MYFILE  ORGANIZATION=IND  REOORD_SIZE=1000
RECORDS_ALLOWED=1000  RECORDS_USED=352  KEY_SIZE=0
INITIAL=334  GROWTH=150  OWNER=MY.ACCT
USAGE_RIGHTS=W(SR.*);R(*.*)  WRITE_THRU=T
```

Format

BIN (num-expr, int-expr)

where the integer expression is the length of the 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, an ERROR 40 results.

The leftmost bit is considered the "sign" bit that tells the system to interpret the number as positive or negative. If the leftmost bit is zero ("off") the value is positive. If it is one ("on") the value is negative. Negative numbers are stored in "two's complement," an inversion of the bit structure.

The Binary to Hexadecimal Conversion Table is as follows:

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

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 value of X$ in hexadecimal format, enter:

```
>PRINT HTA(X$)
C1
```

Format CHAR(num-expr {,ERR=stno})

Description CHAR takes an industry standard (low-order) ASCII code and returns the charcter from the BOSS/IX or BOSS/VS character set.

The numeric-expression must have an integer value between 0 and 255. The hexadecimal string value returned is system dependent (low-order on BOSS/IX, high-order on BOSS/VS).

The use of CHAR (rather than the CHR function) is recommended whenever a literal hexadecimal string is required to ensure the transportability of the BASIC program between BOSS/IX and BOSS/VS systems.

Examples

```
>10 X = POS(CHAR(10)=STRING$)

>A$ = CHAR(10)
>B$ = CHAR(66)
```

BOSS/IX

```
>PRINT HTA(AS)
OA
>PRINT B$

B
```

BOSS/VS

```
>PRINT HTA(A$)
8A
>PRINT BS
B
```

Table 5-1. Character Code Conversions

Character	ASCII Decimal	BOSS/IX Hex	BOSS/VS Hex	Character	ASCII Decimal	BOSS/IX Hex	BOSS/VS Hex
NUL	0	00	80	0	48	30	B0
SOH	1	01	81	1	49	31	B1
STX	2	02	82	2	50	32	B2
ETX	3	03	83	3	51	33	B3
EOT	4	04	84	4	52	34	B4
ENQ	5	05	85	5	53	35	B5
ACK	6	06	86	6	54	36	B6
BEL	7	07	87	7	55	37	B7
BS	8	08	88	8	56	38	B8
HT	9	09	89	9	57	39	B9
LF	10	OA	8A	:	58	3A	BA
BT	11	0B	8B	;	59	3B	BB
FF	12	OC	8C	<	60	3C	BC
CR	13	0D	8D	=	61	3D	BD
SO	14	0E	8E	>	62	3E	BE
SI	15	OF	8F	?	63	3F	BF
DLE	16	10	90	@	64	40	CO
DC1	17	11	91	A	65	41	CI
DC2	18	12	92	B	66	42	C2
DC3	19	13	93	C	67	43	C3
DC4	20	14	94	D	68	44	C4
NAK	21	15	95	E	69	45	C5
SYN	22	16	96	F	70	46	C6
ETB	23	17	97	G	71	47	C7
CAN	24	18	98	H	72	48	C8
EM	25	19	99	I	73	49	C9
SUB	26	1A	9A	J	74	4A	CA
ESC	27	1B	9B	K	75	4B	CB
FS	28	1C	9C	L	76	4C	CC
GS	29	1D	9D	M	77	4D	CD
RS	30	1E	9E	N	78	4E	CE
US	31	1F	9F	O	79	4F	CF
space	32	20	A0	P	80	50	DO

Table 5-1. Character Code Conversions (Cont'd)

Character	ASCII Decimal	BOSS/IX Hex	BOSS/VS Hex	Character	ASCII Decimal	BOSS/IX Hex	BOSS/VS Hex
!	33	21	Al	Q	81	51	Dl
"	34	22	A2	R	82	52	D2
#	35	23	A3	S	83	53	D3
$	36	24	A4	T	84	54	D4
%	37	25	A5	U	85	55	D5
&	38	26	A6	V	86	56	D6
?	39	27	A7	W	87	57	D7
(	40	28	A8	X	88	58	D8
)	41	29	A9	Y	89	59	D9
*	42	2A	AA	Z	90	5A	DA
+	43	2B	AB	[	91	5B	DB
,	44	2C	AC	\	92	5C	DC
-	45	2D	AD	[	93	5D	DD
?	46	2E	AE	^	94	5E	DE
/	47	2F	AF	_	95	5F	DF
?	96	60	E0	p	112	70	F0
a	97	61	El	q	113	71	Fl
b	98	62	E2	r	114	72	F2
c	99	63	E3	s	115	73	F3
d	100	64	E4	t	116	74	F4
e	101	65	E5	u	117	75	F5
f	102	66	E6	v	118	76	F6
g	103	67	E7	w	119	77	F7
h	104	68	E8	x	120	78	F8
i	105	69	E9	y	121	79	F9
j	106	6A	EA	z	122	7A	FA
k	107	6B	EB	{	123	7B	FB
l	108	6C	EC	|	124	7C	FC
m	109	6D	ED	}	125	7D	FD
n	110	6E	EE	~	126	7E	FE
o	111	6F	EF	DEL	127	7F	FF

Format CHR (num-expr t,ERR=stno})

Description The CHR function converts the numeric expression to an ASCII character. The number passed is system dependent, based on the character set of the system (refer to the CHAR function for a table of values).

The value must be in the range 0 - 255; otherwise an ERROR 41 is generated.

Examples

BOSS/IX

```
0100 LET X$=CHR(65)          stores "A" in X$
0100 LET X$=CHR(49)          stores "1" in X$
```

BOSS/VS

```
0100 LET X$=CHR(193)         stores "A" in XS
0100 LET X$=CHR(177)
                             stores "1" in X$
```

See the CHAR function for a system-independent version of CHR.

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

where the 2-byte string is the seed or start 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 in A$.

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 (str-expr)

Description The DEC function converts a binary string expression into a signed decimal number (either positive or negative). 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.

The DEC and BIN functions are complements.

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($FF$)
                           -  X is -1

LET X=DEC($00$+$FF$)       -  X is 255

PRINT DEC($00FFCE$)        -  65486


>PRINT DEC($0032$)
 50


BOSS/IX

LET X=DEC("A")             -  X is 65

LET X=DEC($00$+"A")        -  X is 65

BOSS/VS

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

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

Format EPT (num-expr)

Description The EPT function returns the exponent of the numeric expression.

Examples

```
LET X=EPT(55)                          then: X=2
                        .55*10^2
LET X=EPT(5.23)                        then: X=1
                        .523*10^1
LET X=EPT(-500>                        then: X=3
                       -.5*10^3
LET X=EPT(0)             0*10^0        then: X=0


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


>PRINT EPT(55)
 2
```

Format

FMTINFO (fileno {,field-selector {,info-selector}})

where:

field-selector = a field variable or an integer expression giving the sequential location of the field

info-selector = 0 or 1, specifying the information to be returned (see below)

Description

The FMTINPO function returns multi-keyed file format in-Formation.

FMTINPO returns an empty string when the channel is open to anything other than a multi-keyed file.

FMTINPO also returns the correct field information, using the field alias name.

The field or fields for which information is to be re-turned is specified either by a field variable which has been set to a field name, or by an integer indicating the

sequential position of the field in the record. If no fields are specified, or the selector is set to 0, the entire format string is returned, with each field sepa-rated by two spaces.

The information selector can be either 0 or 1, and speci-fies what information is to be returned for a particular field, as follows:

0 (or omitted)	Returns the part of the format string for the field, including the field name and characteristics.
1	Returns a two-byte string with the follow-ing interpretation ("x" can be 4 bits of anything):

Byte 1 - field type

$1x$	"N" fixed length
20	"S" fixed length
21	"C" fixed length
22	"X" fixed length (not composite)
$5x$	"N*" variable length
$6x$	"S*" variable length
Fx	Composite field

FMTINFO (FORMAT INFORMATION) (cont'd)

Byte 2 - key type

$0x$	NOKEY
$1x$	PRIMARY
$2x$	ALTKEY
$3x$	DUPKEY

Example

```
>A$ = FMTINFO(1,NAME#)
```

Returns the format string for the field associated with NAME# from the multi-keyed file on channel 1.

Format

FNx {$} (arg-list)

where:

x is the function name, following the same rules as for variable names.

$ specifies a string function.

argument-list is the list of arguments provided for by the DEF statement.

Description

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

Example

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

Format FPT (num-expr)

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 (str-expr)

Description This function generates a string that is identical to the specified string expression except that the high-order bit of each byte is set so that the byte has odd parity (that is, an odd number of bits in the byte are "on").

Example

```
0200 LET A$=GAP($0FDC$)   -   A$ is equal to $8FDC$
0300 LET B$=GAP($8FDC$)   -   B$ is equal to $8FDC$
```

Format

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

where the 2-byte string is the seed or start string.

Description

The HSH function computes a "hash string value" from the system 's algorithm.

If the seed string is specified but is not two bytes long, an error 20 is generated.

Examples

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

Computes the hash algorithm on B$ and stores the two-byte result in AS.

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

If C$ is 0000, this returns the same result as A$=HSH(B$). Otherwise, C$ is used by the hash algorithm to calculate the two-byte result.

Format HTA (str-expr)

Description The HTA function converts the hexadecimal value of a string expression to pairs of ASCII characters representing that hexadecimal value. Accordingly, the string returned by HTA is twice as long as the string passed to The HTA function is the converse of the ATH function and is used to print the stored value in a form recognizable as a hexadecimal number.

Examples

BOSS/IX

```
LET X$=HTA("ABC")          - X$ is "414243"

LET X$=HTA("123")          - X$ is "313233"
```

BOSS/VS

```
LET X$=RTA("ABC")          - X$ is "C1C2C3"

LET X$=RTA("123")          - X$ is "B1B2B3"
```

Format IND (fileno {,END=stno} {,ERR=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.

The IND function is not supported for multi-keyed files.

An attempt to use it generates a run-time error.

IND will not move the current record position to the next record.

Example

```
LET A=IND(1,ERR=0500)
```

Format INT (num-expr)

Description The INT function returns the integer part of the numeric expression. Any fractional digits are removed, and rounding does not occur.

```
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 (str-expr, str-expr)

Description The IOR function takes two string expressions as arguments and returns a single string expression as its result. The resulting string is obtained as the value of the bit-wise logical disjunction (Boolean sum or logical OR) of the argument strings, according to the following rules:

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

Example

```
LET X$=IOR($OF$,$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 record to be accessed from the file.

For multi-keyed files, the key returned is the next key in the current key set. The string is not converted to the type of its underlying field, but is treated as an "S" type field, with the trailing nulls removed.

KEY strips trailing nulls from the key it returns.

KEY will not update the current record position to the next record.

Example

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

Format LEN (atr-expr)

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

Exanples

```
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(str-expr)

Description Used to perform a data integrity check, this function computes a longitudinal redundancy check based on the string expression.

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

Example

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

Format MOD (num-expr-a, num-expr-b)

Description The MOD function returns the result of the modulo function. MOD can be thought of as returning the remainder of the division of the first numeric expression by the second, except that the result is always positive.

Precisely, MOD(X,Y) is defined as follows:

Case 1: If Y = 0, MOD(X,Y) = X

Case 2: If Y <> 0, MOD(X,Y) = X - <Y*FL0OR<X/Y)), where,FLOOR(Z) is the largest integer less than, or equal to, Z.

Examples

MOD(26,7) is 5

MOD(22,11) is 0

MOD(-5,3) is 2

MOD(7,-4) is 3

MOD(-8,-5) is 3

Format NOT (str-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$)

     $DC$  = 1101 1100
NOT ($DC$) = 0010 0011 = $23$
```

Format NUM (str-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, and an error 26 results.

Format

POS (scan-str relational-op target-str {,step})

where:

scan-string is the string (in constant or variable form) being searched for in the target string

relational-op is one of the valid relation symbols:

=	<> or X
<	<= or =<
>	>= or =>

target-str is the string (in constant or variable form) to be searched for an occurrence of the scan string
step value is the increment defining the intervals at which the target string is examined for each subsequent comparison (default value is 1)

Description

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.

Examples

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

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=P0S("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"<AS,3)     - X  is 7
```

Format SGN (num-expr)

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.

Exanples

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

LET X=SGN(6)      -  X=1

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

Format

STR (num-expr {:mask})

where mask is a format mask (refer to "NUMERIC EDITING" in Section 2)

Description

The STR function converts the numeric expression to a string of characters. The length and format of the string are specified by a format mask. The mask can be expressed as a string constant surrounded by 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 TBL(str-expr,atno)

Description The TBL function performs table translation. It translates the string in its first argument using the TABLE statement referenced by the second argument.

This function performs the same operation as the TBL= option on an input (READ, INPUT , FIND) or output (WRITE, PRINT) directive. TBL is the only way to do table translation independent of an I/O directive.

Examples

```
0010 INPUT "ENTER ASCII STRING ",ASCII$
0020 EBCDIC$ = TBL(ASCII$,14000)
```

where statement 14000 has the ASCII to EBCDIC translation table.

```
0100 ASCII7$=TBL(ASCII8$,12000)
12000 TABLE 7F
```

This example assigns into ASCII7$ the contents of ASCI18$, with the high bit turned off. For example, if ASCII8$ is $C1C2C3$, ASCII7$ will be 414243.

Format

TRANS (str-expr)

DESCRIPTION

The TRANS function returns a string which is the result of translating the string expression argument. The translation rules specified in the SETTRANS directive are followed. The string expression argument can be thought of as the "left part" and the string result as the "right part."

The TRANS function always attempts to translate the string even if translation has been turned off using the ENDTRANS directive.

If a SETTRANS directive has never been issued, and therefore no translation file has been named, the input string will be returned unchanged. Otherwise, the most recently referenced translation file will be used to perform the translation.

Examples

```
0010 A$ = TRANS("BOSS/IX id")

0100 EXECUTE "copy "+TRANS(A$)+ " "+TRANS(B$)
```

Format XOR (str-expr, str-expr)

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$=XOR($OF$,$DCS)

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

             $D3$ = 1101 0011
```

NOTES

NOTES

SECTION 6 - SYSTEM VARIABLES

A system variable is a variable whose value is established by the operating system.

System variable names are often mnemonic, suggesting their values, e.g., the time (TIM) and date (DAY).

Format CSW

Description The CSW system variable tells whether the program currently in use is a CALL'ed program or a RUN program . CSW has a value of 1 if the program is CALL'ed; otherwise its value is 0.

Examples

```
>PRINT CSW

00100 IF CSW=1 THEN ENTER AS ELSE BEGIN
```

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 table 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.

CTL is set only by INPUT and READ statements. INPUTRECORD and READRECORD leave CTL unchanged.

Table 6-1. TERMINATOR KEY CONTROL VALUES

KEY	BOSS/IX VALUE	BOSS/VS VALUE	ASCII CHARACTER	CTL VALUE
(None)	00	00	NULL	0
LINEFEED	$0A$	$8A$	LF (linefeed)	0
RETURN	$0D$	$8D$	CR (carriage return)	0
CTL-I	ic	$9C$	FS (field separator)	1
CTL-II	$1D$	$9D$	GS (group separator)	2
CTL-III (or CTRL+'N*)	$1E$	$9E$	RS (record separator)	3
CTL-IV (or CTRL+'O')	$1F$	$9F$	US (unit separator)	4
(SIZ=satisfied)	(none)	(none)		5

Examples

```
00100 PRINT CTL

00100 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 format, mm/dd/yy, on BOSS/IX systems. On BOSS/VS systems, it is returned in the current system date format.

Examples

```
>PRINT DAY

00100 LET X$=DAY
```

Format DEVINFO

Description The DEVINFO system variable contains a string containing information about each of the system's configured devices.

The string is composed of ten-byte substrings, one substring per device, in the following format:

Bytes	Description
1-5	Device name, padded with trailing blanks
6	Shared memory controller number and IMLC line number (see SMC ID codes below)
7	Device type code (see table below)
8	Device status code
9	ISDC line number
10	Not used, always zero

The information returned for the SMC ID code (byte 6), device type code (byte 7), device status code (byte 8), and ISDC line number (byte 9) apply to BOSS/VS systems only. These values are set to zero (0) on BOSS/IX systems.

The lower 3 bits of the byte contain the line number (0-7) of the devive if the device resides on an ISDC controller. On a 4-way ISDC (MCS) controller, this is a 2-bit line number and bit 3 is zero. The 16-way ISDC controller is treated as two consecutively addressed 8-way ISDC controllers. For any other type of device, this field is zero. The other 5 bits are reserved for future use.

Table 6-2. SMC ID CODES

Bits	Description
0	Line number - A is 0, B is 1 on IMLC
1-6	Shared memory controller number, range is 0-63
7	Not used

If the device is on an ISDC controller, the shared memory controller number field of this entry is valid and the line number is zero. If the device is neither an IMLC nor an ISCD, the entire SMC ID code is zero.

Table 6-3. Device Type Codes

CODE		DESCRIPTION
hex	dec	
00	0	no device
01	1	high speed VDT
02	2	Dataword II MDT in WP mode
03	3	Dataword II MOT in VDT emulation mode
04	4	Ghost terminal
05	5	7250 terminal
06	6	Transportable Batch Communications (TBC)
07	7	TBC autodial unit
08	8	3270 running on IMLC
09	9	X.25 running on IMLC
OA	10	Basic Four interface system serial printer
0B	11	asynchronous driver
OC	12	asynchronous modem driver
0D	13	7270 terminal
0E	14	EVDT terminal
OF	15	
10	16	Basic Four interface slave printer
11	17	Parallel matrix printer
12	18	Parallel band printer
13	19	MTR 1/2" Reel-to-reel tape drive
14	20	MTS 1/2" Streaming tape drive
15	21	COT terminal
16	22	S/10 terminal
17	23	Special VOT device
18	24	Letter quality serial system printer
19	25	Reserved for DMP serial system printer with IGP
1A	26	DMP serial system printer
IB	27	DMP parallel system printer
1C	28	Industry Standard slave printer (S/10 slave printer
ID	29	Reserved for Industry Standard system serial printer
IE	30	Reserved for letter quality slave printer
IF	31	Reserved for future GCR tape device
20	32	MCS 1/4" cartridge streamer tape drive

Table 6-3. Device Type Codes (cont'd)

CODE hex	CODE dec	DESCRIPTION
21	33	Reserved for tape devices
*	*	*
*	*	*
2C	44	Reserved for tape devices
2D	45	EOT terminal
2E	46	Reserved for EOT with monochrome graphics
2F	47	Reserved for IMLC diagnostic port
30	48	MAGNET socket
31	49	VDT/B
32	50	14" intelligent terminal
33	51	available
*	*	*
*	*	*
FF	255	available

Table 6-4. Device Status Codes

Bit	Description if bit is "ON"
0	ESCAPE entered on terminal device
1	Device is open or in use
2	Device is not configured
3	Printer is dedicated
4	Terminal has a slave printer
5-7	unused

Examples

```
00100 A$ = DEVINFO
00110 B$ = POS(FID(0)=A$,10)
00120 IF B$(7,1) = $2D$ PRINT "THIS IS AN EOT TERMINAL"
```

Format ERR {(code-1, code-2,...,code-n)}

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 printed 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.

Examples

```
00100 PRINT "ERROR CODE = ", ERR
00999 EXIT ERR
```

```
00050 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=ll
    branch to 300 if error=12
    branch to 400 if error=47
```

The same operation can be written using a LET statement:

```
00050 LET E=ERR (11,12,47)
00060 ON E GOTO 100,200,300,400
```

Format PNM

Description PNM returns the name of the program currently in main memory. When used in a Public Program , PNM returns the name of the CALL'ed program.

The format of the string returned is that of a full path name, including the family on on BOSS/VS systems and the root directory on BOSS/IX systems.

Example

```
>LOAD "BOXCARS"
```

BOSS/IX

```
>PRINT PNM
/usr/trainset/BOXCARS
>
```

BOSS/VS

```
>PRINT PNM
(DISK).TRAINSET.BOXCARS
>
```

Format PRC

Description The PRC system variable returns a numeric value which is the current arithmetic precision of the user's BASIC task. This numeric value is between 0 and 14, inclusive, matching the range of arguments for the PRECISION directive, or -1 (-.1E+01) when BASIC is in floatingpoint. PRC can only be changed by the PRECISION and FLOATINGPOINT directives.

Examples

```
>CLEAR
>PRINT PRC
 2
>PRECISION 14
>PRINT PRC
 14
>FLOATINGPOINT
>PRINT PRC
 -.1E+01
>
```

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 CALL'ed program, the value is the size of the CALL'ed program.

In BOSS/IX, PSZ includes the user program area overhead. Therefore, PSZ always equals at least 34. Part of this overhead is environment-dependent and may vary (generally not more than 20 bytes). Therefore, PSZ may return slightly different values for the same program.

In BOSS/VS, PSZ is the size of the program segment and does not include the size of tables or source. This number can vary considerably from the value of PSZ returned for the same program on a BOSS/IX system .

Example

```
>PRINT PSZ
```

Format SSN

Description The SSN variable contains the system serial number, returned in a 10-byte string.

Example

BOSS/IX

```
>PRINT SSN
2000-90034
```

BOSS/VS

```
>PRINT SSN
810-30000
```

Format SYS

Description The SYS variable contains a string identifying the BASIC Language release and version levels. The format of the string is:

```
    name release*version
```

BOSS/IX

```
>PRINT SYS
BB86 07.03A
```

BOSS/VS

```
>PRINT SYS
BB86 08.06A
```

Format

TCB(num-expr)

where numeric-expr has a value ranging from 0 to 12.

Description

The TCB variable contains information that pertains to a particular task.

Some TCB1s must be converted into decimal or hexadecimal Format to be useful. Table 6-5 shows the contents of each TCB .

Some TBC cells, in particular, cells 4, 5, 6, 7, and 12, return the same kind of information on all systems running BB86, though the format may differ. Other cells return specific information depending on whether the operating system is BOSS/IX or BOSS/VS, and are not defined in BB86.

TCB (TASK CONTROL BLOCK) (cont'd)

Table 6-5. TCB VARIABLE FORMAT

TCB(n)	BYTE LENGTH	DESCRIPTION
0-2		undefined
3	2	BOSS/IX, communication device status
4	2	current statement number, if any; or 0
5	2	statement number of last error, if any; or 0
6	2	statement number SETESC references, if any; or 0 - if SETESC references a non-existing statement number, TCB(6) will be |600|
7	2	statement number SETERR references, if any; or 0 - if SETERR references a non-existing statement number, TCB(7) will be |600|
8-9		undefined
10	2	BOSS/VS - 0 = EXTEND mode; 1 = NO EXTEND mode
		BOSS/IX - logical unit of most recent I/O error
12	2	BOSS/IX - most recent system error code, a negative number
	4	BOSS/VS - most recent system error code, in fourtuple format
11	1	BOSS/IX - the last logical unit number accessed; it is always zero following a successful START, BEGIN, or END
13		undefined
14	2	format string error

TCB (TASK CONTROL BLOCK) (cont'd)

Examples

```
0008 REM TCB(12) returns an integer
0010  A=TCB(12)
0012 REM Convert it first to a string
0020  AS=BIN(A,4)
0022 REM decompose the string
0030  S1=ASC(A$(1)>
0040  S2=ASC(A$(2))
0050  S3=ASC(A$(3))
0060  S4=ASC(A$<4))
0070 REM print the results
0100  PRINT "This is the number returned by TCB(12): ", A
0200  PRINT "The string looks like this: ", HTA(A$)
0300  PRINT "in final ""fourtuple"" form : ",
         S1,",",S2,",",S3,",",S4
```

The following example displays the line number where the error occurred.

```
01000 INPUT (0,ERR=8000)@(5,10)'CL', A
  .
  .
  .
08000 PRINT @<0,21),'CL',"YOU GOOFED.  ERR = ",
      ERR, "AT LINE:",TCB(5); INPUT *; RETRY
```

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

```
00100 LET T=TIM
00200 LET H=INT(T)
00300 LET S1=INT(FPT(T)*3600)
00400 LET M=INT(Sl/60)
00500 LET S=S1-M*60

    - where H=hours, M=minutes, S=seconds

>PRINT TIM
```

Format TRX

Description The TRX system variable contains the full path name of the translation file currently in use. If the translation facility has not been started by the SETTRANS directive, or if translation has been turned off with ENDTRANS, then TRX will return the NULL string.

Example

```
>PRINT TRX
TRANSFILE.ID

>100 A$ = TRX
```

Format UNT

Description The UNT variable returns the lowest logical unit number, or I/O channel number, that is available.

A CALL'ed program can use UNT to open a device, without knowing which devices have been opened by the CALL'ing program. For example, OPEN(UNT) "name".

Examples

```
>END
>PRINT UNT
1
>OPEN  (1)"P1"
>PRINT UNT
2
>OPEN (3)"P3"
>PRINT UNT
2
>OPEN (2)"P2"
>PRINT UNT
4
>CLOSE (1)
>PRINT UNT
1
```

The following example opens a device without knowing which devices are already open.

```
OPEN(UNT) "name"
```

Format WHO

Description The WHO system variable contains the task's account name. This is the account name of the user who logged on to the terminal on which the variable is used.

BOSS/IX

```
>PRINT WHO
franz
```

BOSS/VS

```
>PRINT WHO
LAN.MST
```

NOTES

NOTES

OVERVIEW

Input/Output options are used to modify 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 the range of the permitted length of a variable, and more.

Multiple I/O options in a statement are separated by commas.

Format DOM= stno

Description The DOM= option transfers control to the specified statement if the key specified in an INPUT, READ, or REMOVE operation is not found in the file, or if the key specified m a PRINT or a WRITE operation is already in the file.

If a DOM= option is not used in an INPUT, READ, or REMOVE statement, an ERROR 11, MISSING OR DUPLICATE KEY, is generated when the specified key is not found.

If a DOM= option is not used in a PRINT or WRITE statement, the record in the file which corresponds to the specified key is replaced with the new record.

Examples

```
00100 READ (2,KEY=A$)R$
```

If the KEY is not in the file, an ERROR 11 occurs.

```
00100 READ (2,KEY=A$,DOM=500)R$
```

If the KEY is not in the file, the DOM= branch is taken, and ERR=11 is set.

```
00100 WRITE (2,KEY=A$)R$
```

If the KEY is in the file, old data is overwritten.

```
00100 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 overwritten.

```
00100 WRITE (2,KEY=A$,DOM=500,ERR=400)R$
```

If the KEY is not in the file, control passes to statement 500, but any other error causes branching to statement 400.

Format END= stno

Description The END= option transfers program control to the specified statement number when the end of file is reached. If an END= option is not used, an ERROR 2, END OF FILE, is generated.

End-of-file is reached when the program tries to read beyond the last record in the file. It is also reached when the program tries to write to a record number that is higher than those specified for the file (See IND=), for example, if the program tries to write record 7 to a SERIAL file that contains only three records, or if the program tries to write record 7 to an INDEXED file that was defined to contain only three records.

Examples

```
00200 READ (1,END=0500)AS

00200 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.

With the exception of ERROR 126 (CTRL+Y KEY USED) and ERROR 127 (ESCAPE), errors greater than 99 are not processed by an ERR= option; rather, they cause an immediate exit to console mode, due to the nature of these errors.

Use of DOM= is recommended in statements performing INPUT, READ, REMOVE, PRINT, or WRITE directives when the KEY= option is also used. When DOM= appears in the syntax before ERR=, special branching occurs in cases of missing or duplicate keys.

BB86 does not support multiple ERR= clauses in a single BASIC statement. However, both BOSS/IX and BOSS/VS do support this. If more than one ERR= option appears in a statement, the final statement number is used .

Example

```
00200 READ (1,ERR=0500)AS
```

Format

IND= num-expr

where numeric-expr 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 and Serial files, and with String files on BOSS/IX. Use of IND= when reading Direct or Sort files speeds record access by using the relative (to 0) record number. However, files are not returned in key-sorted order when this method is used, and records which have been deleted may be read, with no indication that they are no longer valid.

IND= is not supported for multi-keyed files, and an attempt to use it generates a run-time error.

If IND= is used in a CLOSE directive, it can have the

values 0, 1, 2 or 9. These are used when closing a unit to the half-inch tape device. These numbers cause the following actions:

0 - Rewinds tape to load point.

1 - Rewinds tape to load point and takes tape off-line.

2 - Rewinds tape to load point. If CLOSE is preceded by a WRITE RECORD, 2 file marks are written to tape.

9 - Writes 2 file marks to tape, then rewinds the tape.

Example

```
00200 READ(1,IND=10)
```

Format IOL= stno

Description The IOL= option specifies the statement number of the IOLIST to be used. Refer to the IOLIST directive for details.

Examples

```
00100 IOLIST AS,B,C,IOL=0200

00200 IOLIST D,E

00300 READ (1,KEY=A$)IOL=0100

00400 PRINT  (7)IOL=0100
```

Format KEY= str-expr

Description This option specifies the key of the record to be accessed by the input/output statement containing the KEY= option.

The KEY= clause is only allowed on DIRECT files and on input operations to multi-key files. Using KEY= on any

other file types or a multi-key WRITE/PRINT will cause an ERR0R=13.

When reading a Multi-Keyed file using the KEY= clause, one may specify the searching of any field which is either PRIMARY, ALTKEY, or DUPKEY (that is, anything but NOKEY). For example, one may say, "read the record whose key in keyset Fl# is 'Jones'" by using the following clause:

```
KEY=Fl#="Jones"
```

Of course, it is also permissible to have the key value in a string variable and use this clause:

```
KEY=F1#=STRING$
```

If the variable name matches the field name (except for the # at the end of the field name), certain short cuts

may be taken. The following examples are equivalent:

```
KEY=F1#=F1$
KEY=#=F1$
KEY=#F1$
```

These five examples will only work if field Fl# is of type S, C, or X, but not N (numeric). If the field is of type N, then the following examples will work, with the last three being identical in effect:

```
KEY=Fl#=-987.33
KEY=Fl#=PAYMENT
KEY=F1#=F1
KEY=#=F1
KEY=#F1
```

It is also acceptable not to specify the keyset, in which case the PRIMARY keyset is used. Here are two examples:

```
KEY="Jones"
KEY=-987.33
```

KEY (ACCESS KEY IN FILE) (cont'd)

If one does not use the KEY= clause, then the "next" record is read. Since different keysets place different ordering on the records, the keyset which is used to find the "next" record is the last keyset which was previously used in a KEY= clause, whether that previous KEY= clause was in a REMOVE statement or in a READ statement (or variant such as EXTRACT or READ RECORD). If no KEY= clause has been used for this logical unit since it was opened, then the PRIMARY keyset is used for ordering purposes on sequential reads.

For more information, refer to Appendix B.

Examples

```
00500 READ(1,KEY=AS)X$

00500 WRITE(1,KEY=STR(A:"00000"))A,B$
```

Format

LEN= min,max

where min and max are, respectively, the minimum and maximum allowable lengths of the variable.

Description

The LEN= option specifies the inclusive range for the length of a variable. The minimum length must be less than or equal to maximum length.

If the length of the variable is beyond the specified range, an ERROR 48, INVALID INPUT, results.

The LEN= option is only allowed on input operations. It may not be used in an IOLIST directive, and it may not appear on output operations.

Example

```
00100 INPUT (0,ERR=0300)A$:(LEN=2,3)

00300 IF ERR=48 THEN GOTO 8000 ELSE GOTO 7000
```

r

Format RETAIN

Description A RETAIN clause is used with the PACK, PRINT, READ and WRITE directives to specify how the retain buffer is to be used.

A retain buffer is associated with each channel opened to a file or device. The buffer can hold one record. EXTRACT, FIND, INPUT, READ and PACK directives with the RETAIN option, and the PACK directive without the option place data into or modify the data currently in the retain buffer. WRITE and PRINT with the RETAIN option and UNPACK retrieve data from the retain buffer.

Example

```
0100 WRITE (1,RETAIN)
```

See appendix B for further examples of RETAIN. Note that it may be used on any type of file, not just multi-key files.

Format SEQ= int-expr

Description The SEQ= option is used only for OPEN'ing the 1/2-inch tape device. The numeric expression specifies where the

tape is to be positioned upon opening, as follows:

SEQ= 0 positions the tape at Beginning of Tape
SEQ= 1 positions the tape just after the first filemark

SEQ= n positions the tape just after the nth filemark

Examples

```
OPEN(1,SEQ=3)"R0"
```

Positions the tape just after the third filemark.

Format SIZ= int-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 neither the <RETURN> nor any Control Bar key (<CTL-I> through <CTL-IV>) 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. Refer to the TABLE directive for more details.

Examples

```
00100 READ(1,TBL=2000)A$

00100 WRITE(2,TBL=5000)A$,B
```

Format TIM= num-expr

Description The TIM= 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 time out for keyboard input, i.e., if TIM= is not specified, the operation never times out. "TIM=0" returns almost immediately.

Both BOSS/IX and BOSS/VS systems wait in tenths of seconds.

Example

```
00100 INPUT (0,ERR=0500,TIM=60.4)"NAME",A$
```

Allows 60.4 seconds for input; otherwise, control passes to statement 500.

NOTES

NOTES

OVERVIEW

Mnemonics are easily remembered names for standard operations. These operations are generally the equivalents of ESCAPE sequences, used to access special features of input and output devices. Since the escape sequences required to access these features are device specific, Business BASIC provides mnemonics to give a uniform interface to them. System drivers handle the translation of the mnemonics to the sequences required by each device.

Most mnemonics specify characteristics of the output produced by printers and terminals. For instance, mnemonics are provided for changing the character pitch on printers, switching between bold and normal display intensity on terminals, and for locating the cursor or print head on terminals and printers. A few mnemonics condition the way the operating system intervenes on input and output operations. For instance, mnemonics are provided to tell the operating system to stop processing mnemonics, or to discard buffered keystrokes.

Mnemonics are transmitted as data, and so are subject to TBL= translation.

Mnemonic Format

Most mnemonics consist of two alphabetic characters enclosed by single quotation marks. A few mnemonics have longer names and cursor/print head positioning mnemonics are not enclosed in quotation marks.

In general, the mnemonic is inserted at the point where the stated operation is desired. For example:

```
0100 PRINT @(35,5), AS, 'LF', BS
```

In this example the 'LF' mnemonic is used to perform a line feed on the user's terminal after printing the value of AS 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$.

Mnemonics can be used as string expressions. When so used, they are evaluated to their internal codes, then passed to the terminal driver. This feature allows mnemonics to be assigned to BASIC variables. For example, the contents of an application display screen can be assigned to a variable. For example:

```
00200 SCREEN$= 'CS'+'SB'+@U 2,5)+"1. CUSTOMER NUMBER OR
00200:END: " + 'CF'+ @(3,5)+,'CL',+"_______"
00210 PRINT SCREEN$
00220 INPUT @(37,5),A
```

VFU DEFINITION

A Vertical Forms Unit (VFU) definition describes the page length and vertical locations on a form. Specific mnemonics are provided to define and use the VFU.

The 'SL' and 'EL' mnemonics are used to start and end loading a VFU definition. The VFU definition is contained in a string expression. The format for defining a VFU is:

```
'SL', <VFU-def.-string>, 'EL'
```

The length of the form (number of lines) is specified by the length of the VFU definition string, with each character in the string representing one line. If the length of the string is 66 characters, then the form is 66 lines long.

The character at each position in the VFU specification string defines which, if any, slew channel is defined there. If the character is "0", then no slew channel is defined there. If the character is "1", then this is the top of the form. The characters "2", "3", "4", "5", "7", "8" define slew channels 2, 3, 4, 5, 7, and 8, respectively. The character "6" defines the line for a vertical tab. Several lines can be specified for vertical tabs, or channel 6.
For example:

```
'SL',"1002003060040600",'EL'
```

defines a VFU of 16 lines with the top of the form as the first line, slew channel 2 on the fourth line, slew channel 3 on the seventh line, slew channel 4 on the twelfth line, and vertical tabs on the ninth and fourteenth lines.

The mnemonics 'S2' through 'S8' can be used to slew to the specified lines. 'S61 and 'VT' can both be used to slew to the vertical tab lines.

SLEW performs a paper throw. Each SLEW mnemonic specifies how far the paper will be fed before the next print is executed. A SLEW feeds the paper faster than a normal line feed.

MNEMONICS DESCRIPTIONS

Table 8-1 lists the mnemonics supported by BB86, including an abbreviated description. Following the table, full descriptions are given for each mnemonic. The mnemonics are grouped as Terminal Control, Printer Control and Operating System Control mnemonics, and are described in alphabetic order in those groups.

Table 8-1. ALPHABETICAL LISTING OF MNEMONICS

MNEMONIC	DESCRIPTION
@(x)	Horizontal Position
@(x,y)	Horizontal and Vertical Position
'10'	10 Pitch
'16'	16 Pitch
'6L'	Six Lines Per Inch
'8L'	Eight Lines Per Inch
'B1'	Sheet Feeder Bin 1 (BOSS/IX only)
'B2'	Sheet Feeder Bin 1 (BOSS/IX only)
'BB'	Begin Blink
'BE'	Begin Echo
'BG'	Begin Generating ERROR 29
'BI'	Begin Input Transparency
'BO'	Begin Output Transparency
'BR'	Begin Reverse Video
'BS'	Backspace
'BT'	Begin Input Buffering
'BU'	Begin Underline
'CE'	Clear Screen to End of Page
'CF'	Clear Foreground
'CH'	Cursor Home
'CI'	Clear Input Buffer
'CL'	Clear Line
'CR'	Carriage Return
'CS'	Clear Screen
'DC'	Delete Character
DN	Down Cursor (BOSS/IX only)
'DACS'	Disable Alternate Character Set
'DBLH'	Double Height Print (BOSS/IX only)
'DBLW	Double Width Print (BOSS/IX only)
'DPM'	Reset to Default Character Printing Mode (BOSS/IX only)
'EB'	End Blink
'EE'	End Echo
'EG'	End Generating ERROR 29
'EI'	End Input Transparency
'EL'	End Load
'EO'	End Output Transparency
'EP'	Expanded Print
'EPM'	Even Dot Plot Mode
'ER'	End Reverse Video
'ES'	Escape
'ET'	End Input Buffering

Table 8-1. ALPHABETICAL LISTING OF MNEMONICS (cont'd)

MNEMONIC	DESCRIPTION
'EU'	End Underline
'FF'	Form Feed
'IC'	Insert Character
'KL'	Keyboard Lock
'KU'	Keyboard Unlock
'LD'	Line Delete
'LF'	Line Feed
'LI'	Line Insert
'LT'	Cursor Left (BOSS/IX only)
'NL'	New Line
'OP'	Overprint
'OUT(n)'	Output (n) characters without translation (BOSS/IX only)
'PE'	End Protect
'PG'	Print Screen
'PM'	Plot Mode
'PS'	Start Protect Mode
'RB'	Ring Bell
'RC	Read Cursor
'RT'	Cursor Right (BOSS/IX only)
'S2'	Slew 2
'S3'	Slew 3
'S4'	Slew 4
'S5'	Slew 5
'S6'	Slew 6
'S7'	Slew 7
'S8'	Slew 8
'SACS'	Start Alternate Character Set
'SB'	Start Background
'SET6'	Six LPI (BOSS/IX only)
'SET8'	Eight LPI (BOSS/IX only)
'SF'	Start Foreground
'SL'	Start Load
'SN'	Screen narrow
'SP'	Superscript
'SPMl'	Set Print Mode 1
'SPM2'	Set Print Mode 2
'SPM3'	Set Print Mode 3
'SPM4'	Set Print Mode 4
'SPM5'	Set Print Mode 5
'SS'	Subscript
'SW '	Screen Wide
'TL'	Transmit Line
'TP'	Transmit Line Protected
'TR'	Transmit Screen
'TS'	Transmit Screen Protected
'UP'	Cursor Up (BOSS/IX only)
'VT'	Vertical Tab
'WPM'	Letter Quality Emulation Mode (BOSS/IX only)

Terminal Control

@(X) - Horizontal Position

The cursor is positioned at column X on the current line. For terminals, all characters on the screen between the current horizontal position and the new X position are blanked out.

@(X,Y) - Horizontal and Vertical Position

The cursor is positioned at column X and line Y.

'BB' - Begin Blink

The text following this mnemonic is displayed in blink mode.

'BR' - Begin Reverse Video

The text appears with a light background and dark print.

'BS' - Backspace

A destructive backspace is performed. The cursor moves back one space and replaces the character with a space.

The remainder of the string is not shifted left.

'BU' - Begin Underline

All characters following the mnemonic are underlined for that line or until 'EU', whichever comes first.

'CE' - Clear Screen to End of Page

The screen is cleared from the current cursor position to the bottom of the screen. Background mode is cancelled, if it is in effect.

'CF' - Clear Foreground

All the characters printed in Foreground Mode are replaced with spaces. Background Mode is cancelled, if it is in effect.

'CH' - Cursor Home

The cursor is positioned at "home" (0,0).

'CL' - Clear Line

All the characters from the cursor to the right-hand end of the line are replaced with spaces. Background mode is cancelled, if it is in effect.

'CR' - Carriage Return

The cursor moves down one line and back to column zero. Note that this operation code is not the same as the ASCII

character. (This is also a printer operation, for which refer to the description below.)

'CS' - Clear Screen

All characters are cleared from the screen; the cursor is positioned at home (0,0); protect mode is reset to OFF; and background mode is cancelled, if it is in effect.

'DC' - Delete Character

The character at the cursor is deleted, and all characters to the right of the cursor are moved one position to the left. A space is written in the last position of the field.

'DN' - Cursor Down (BOSS/IX only)

The cursor moves down one line, retaining the same horizontal position.

'EB' - End Blink

Text blinking begun by 'BB' is cancelled. All following text appears on the screen in a normal manner.

'ER' - End Reverse Video

Text is no longer printed with light background - dark print (begun by 'BR').

'EU' - End Underline

Text following is no longer printed with underlining (begun by 'BU').

'IC' - Insert Character

All the characters at and to the right of the cursor move one space to the right. The next I/O character appears in the space at the cursor.

'KL' - Keyboard Lock

Transmission of data from the keyboard is halted. To reset, use 'KU' or turn the terminal OFF/ON.

'KU' - Keyboard Unlock

Transmission of data from the keyboard is resumed.

'LD' - Line Delete

The line where the cursor is positioned is removed, all lines below it scroll up one line, and a line of spaces is printed at the bottom.

'LF' - Line Feed

A line feed (cursor moves down one line) and a carriage return are executed.

'LI' - Line Insert

A blank line is inserted at the cursor position, all following lines scroll down one line, and the bottom line is deleted.

'LT' - Left Cursor (BOSS/IX only)

The cursor moves one space to the left. This is a non-destructive backspace that leaves the existing character intact.

'PE' - Protect End

Protection Mode ('PS') is cancelled.

'PG' - Print Screen

Prints the contents of the screen, from the home position up to but not including the cursor position, to the terminal printer port. Trailing spaces and lines are not transmitted.

'PS' - Protect Start

Display protection is initiated. The cursor is prevented from entering a protected field (a field in background mode), and screen scrolling is prohibited.

'RB' - Ring Bell

The terminal buzzer sounds.

'RC - Read Cursor

Used with the INPUT directive, this mnemonic returns the current cursor position coordinates as a two byte string. The two bytes returned are hexadecimal values. Position 0,0 would be 2020 (20 hex = 32 decimal). The first byte contains the vertical position plus 32 (dec), and the second byte contains the horizontal position plus 32 (dec). This mnemonic is useful when a screen must be restored following some subroutine, such as a help text display.

```
>10 PRINT @(10,12),
>20 INPUT 'RC', A$
>30 PRINT ASC(A$(l))-32,ASC(AS(2))-32
>RUN
 12 10
```

WARNING

The 'RC' mnemonic also sends 'LF' to the screen. If the cursor is on line 23, the screen will scroll and the old line 0 is lost.

'RT' - Cursor Right (BOSS/IX only)

The cursor moves one space to the right. This is a non-destructive control, leaving existing characters intact.

*SB' - Start Background

Background Mode is initiated. All subsequent text is distinguished from Foreground text, and may be controlled separately. Background text is marked as protectable, but protection does not begin.

'SF' - Start Foreground

Foreground Mode is initiated. This is the normal (default) I/O mode.

'SN' - Screen Normal

Sets the screen to normal, 80-column display mode.

'SW' - Screen Wide

Sets the screen to wide, 132-column display mode.

*TL' - Transmit Line

Transmits all unprotected data from the beginning of the line through the cursor position.

'TP' - Transmit Line Protected

Transmits all data, including protected fields, from the beginning of the line through the cursor position.

'TR* - Transmit Screen

All data displayed on the screen is placed into the specified input variable. Example:

```
00120 INPUT 'TR', A$
```

AS contains everything that was displayed on the screen from the home position to the cursor position.

'TS' - Transmit Screen Protected

Transmits all data on the screen, including protected fields, from the home position through the cursor position.

'UP* - Cursor Up (BOSS/IX only)

The cursor moves up one space, retaining the same horizontal position.

Printer Control The following mnemonics instruct the printer driver to perform specific functions on printers.

@(X) - Horizontal Position

The next data is printed at the horizontal position specified by X. Printers will not accept a second vertical coordinate.

'6L' - Six Lines Per Inch

The printer's lines per inch setting is overridden and set to 6 lpi. This mnemonic stays in effect until the end of the line. Lines per inch mnemonics cannot be mixed on a line because each mnemonic causes the print line to be flushed and the upcoming characters to be printed on the next line.

'8L' - Eight Lines Per Inch

The printer's lines per inch setting is overridden and set to 8 lpi. This mnemonic stays in effect until the end of the line. Lines per inch mnemonics cannot be mixed on a line because each mnemonic causes the print line to be flushed and the upcoming characters to be printed on the next line.

?10' - 10 Pitch

The printer's characters per inch setting is overridden and set to 10 cpi. This mnemonic stays in effect until the end of the line. The current line is printed before the mnemonic takes effect . Characters per inch mnemonics cannot be mixed on a line.

'16' - 16 Pitch

The printer's characters per inch setting is overridden and set to 16 cpi. This mnemonic stays in effect until the end of the line. The current line is printed before the mnemonic takes effect. Characters per inch mnemonics cannot be mixed on a line.

'Bl' - Bin 1 (BOSS/IX
'B2' - Bin 2 only)

Paper from bin 1 (front bin) or bin 2 (back bin) of the cut sheet feeder is fed into the printer. The application must issue a form feed at the end of every page to cause the paper to be moved into the out-tray.

'BU' - Begin Underline

The printer is set in underline mode. All data is underlined until 'EU' (end underline) mnemonic is encountered.

'CR' - Carriage Return

The printer head returns to the beginning of the line (without a line feed) in order to perform underlines, etc. Some printers do perform a line feed and carriage return. This mnemonic depends upon how the printer is set up.

'DPM' - Reset to Default Character Printing Mode (BOSS/IX only)

This mnemonic causes the printer to be reset to its default character printing mode. The printer completes printing the current line before returning to data processing mode.

'BACS' - Disable Alternate Character Set

This mnemonic selects the standard character set for the printer. It is the complement of 'SACS'.

DBLH' - Double Height Print (BOSS/IX only)

This mnemonic has the effect of printing the characters, for the current line only, at twice their normal height. Note that this causes the page to have half the normal number of lines.

'DBLW' - Double Width Print (BOSS/IX only)

This mnemonic has the effect of printing all characters, for that line only, at twice their normal width. Note that this causes lines to have half their normal number of characters. @(x) positioning also positions the print head at twice the column specified.

'EL' - End Load

On some types of printer, this mnemonic ends the loading of the VFU (vertical format unit). See the instructions for each printer.

'EP' - Expanded Print

The characters in the current line will be printed in expanded print. For the 150-300 line printers 'EP' expands the height to two spaces. For 120 line printers 'EP' expands horizontally to two spaces per character.

'EPM' - Even Dot Plot Mode

This mnemonic prints dots on the even dot positions on a print line. A line feed character terminates an even dot plot line, but does not advance the paper. The 'PM' mnemnonic prints on the odd dot positions on a print line. 'PM' must be used to advance the.paper.

'EU' - End Underline

This mnemonic ends underlining of text (refer to *BU' above). The underlines are printed with the rest of the line whenever the buffer is flushed.

'FF' - Form Feed

This mnemonic advances the paper to the top of the next page. The 'CS' (clear screen) terminal mnemonic also causes a form feed.

'LF* - Line Feed
'NL' - New Line

These mnemonics are identical. Both cause the line to be printed with the carriage return and line feed characters.

'OP' - Overprint

The previously printed line is followed by a carriage return without advancing the paper. Thus the next line may overprint the previous line. This feature is not supported on some printers.

'OUT'(n) - Output (BOSS/IX only)

This mnemonic causes the printer driver to output the next n characters without translation. The next n characters are simply placed into a buffer without translation. Note that (n) must be specified. Form control is not disabled by this mnemonic.

'PM' - Plot Mode (with line feed)

The 'PM' mnemonic enables use of the graphics capability of the printer. The 'PM' must precede the graphics data and the data must end with a return. Refer to the printer manual for each printer for the actual hex codes for the graphics.

When using the plot mode of mnemonics to print a bit pattern, output transparency should be set. For example:

```
10 OPEN(1)"LP"
20 A$=$7F$
30 PRINT (1) 'PM','BO',A$,'EO'
40 CLOSE(1)
```

'RB' - Ring Bell

This mnemonic causes the buffers to be flushed and the "rimg bell" code to be sent to the printer. A repeat factor may be specified for this mnemonic.

'Sn' - Slew n

Where n is a character 2 through 8, this mnemonic performs a paper feed as defined by the VFU settings. Refer to the description of VFU above.

'SACS' - Start Alternate Character Set

This mnemonic selects the printer's alternate character set. All characters after this mnemonic are printed with the alternate character set.

'SB' - Start Background (normal print)

This mnemonic ends bold print mode (foreground, 'SF'). Foreground mode is automatically ended after each line.

'SET6' - Set Printer to 6 Lines Per Inch (BOSS/IX
'SET8' - Set Printer to 8 Lines Per Inch only)

These mnemonics cause the printer to output its data at 6 or 8 lines per inch. This remains in effect until changed by the 'SET6' or 'SET8' mnemonic respectively. The current line is printed before the change takes effect. Lines cannot mix the number of lines per inch.

These mnemonics differ from '6L' and '8L* in that these latter mnemonics are effective for only one line.

'SF' - Start Foreground (bold print)

This mnemonic causes printing to be in bold (darker) print. This continues until the 'SB' mnemonic is encountered or the end of the line, whichever comes first.

?SL' - Start Load

The electronic VFU (vertical print unit) is loaded. This is only available on some printers. Refer to the discussion of VFU at the beginning of this section.

'SP' - Superscript Print

This mnemonic causes the printer to print the following characters superscripted, up 1/4 of a line. It is also used for terminating subscripting. For example:

```
"x",'SP',"2",'SS'," + y",'SP',"2",'SS'," = z",'SP',"2"
```

will produce:

$x^2 + y^2 = z^2$

Note that:

```
" 2    2    2",'SS','CR'"x  + y  = z"
```

will produce the same result except that the latter example may use an extra 1/4 line.

'SPMx' - Select Print Mode x

There are five (5) 'SPMx' mnemonics, 'SPMl' through 'SPM5'. The print buffer is flushed before the print mode is changed. The selected print mode remains in effect until another 'SPMx' mnemonic, or the 'WPM' or 'DPM' mnemonic is sent.

'SS' - Subscript

This mnemonic causes the printer to print the following character subscripted, down 1/4 of a line. It is also used to end subscripting. See the example above for the 'SP' mnemonic.

'VT* - Vertical Tab

Slew to channel 6. A repeat factor may be specified for this mnemonic.

'WPM' - Word Processing Mode (BOSS/IX only)

This mnemonic causes the printer to enter letter quality emulation mode. For most printers this causes a reduction of print speed. The buffer is flushed before word processing mode is entered.

OS Control

The mnemonics in this section direct the operating system to perform certain functions which control how or whether devices receive data. Specifically, most of these mnemonics affect the processing of other mnemonics.

'BE' - Begin Echo

(Corresponds to 'EE'.) 'BE* requires the operating system to begin transmitting input/output data to the VDT screen.

'BI' - Begin Input Transparency

(Corresponds to 'EI', End Input Transparency.) 'BI' will start transparent input. Transparent input means that no terminal services interpretation of any characters input from the keyboard will be performed (i.e., certain terminal commands, such as ESC, and CTL+X,Y,S, or Q, will not be intercepted and executed).

WARNING

There is NO terminator on inputs after execution of this mnemonic. All input is treated as data including all terminators, the Xon and Xoff characters, and the ESC key. This mnemonic remains in effect on the terminal until an 'EI' mnemonic is output to it or until the terminal process is logged off.

'EG' - Begin Generating ERROR 29

The generation of ERROR 29 (Undefined Mnemonic) is restored (necessary to end the suppression begun by 'EG')

'BO* - Begin Output Transparency

(Corresponds to 'EO '.) 'BO' is a data Transparency transfer mnemonic that disables all data control characters and mnemonic sequences (except for 'EO '), causing all data to be passed to the device without interference or

translation by the driver.

'BT' - Begin Input Buffering

(Corresponds to 'ET'.) 'BT' initiates the buffering process, which allows an operator to input data without waiting for prompts or input requests to appear on the VDT screen. The buffer accumulates the data in sequence and assigns it to each field as it appears.

'CI' - Clear Input Buffer

All data accumulated in the input buffer (see 'BT') is cleared. This prevents data from being entered in the wrong field after an interrupt has occurred. 'CI' should be used with error traps and verification routines.

'EE' - End Echo

(Corresponds to 'BE'.) The operating system is instructed to stop sending I/O data to the VDT screen. All data input or output after 'EE' and before 'BE' will not appear on the screen.

'EG' - End Generation of ERROR 29

The generation of ERROR 29 (Undefined Error 29 Mnemonic) is terminated.

'EO' - End Output Transparency

The 'BO' mnemonic is cancelled, restoring the effectiveness of control characters and mnemonic sequences. The driver now intercepts and translates such commands being passed to communication devices.

'ES' - Escape

An escape character is sent to the device, which treats it as a lead-in code. The next character defines an action for the terminal.

ET - End Input Buffering

Input buffering begun by 'BT' is cancelled, requiring the operator to input data only as each prompt or request appears.

CAUTION

The mnemonics affecting input and output transparency must be used with caution. The mnemonics involved are 'BI', 'EI', 'BO',and 'EO '. The only mnemonics that are serviced during input or output transparency are those that end the process: 'EI' and 'EO '. All others are simply passed as data without checking validity. Accidentally executing input or output transparency may "lock up" the terminal so that only the resetting of the terminal will establish proper operation.

NOTES

INTRODUCTION

This section discusses errors and error messages that occasionally appear on the terminal screen. Also described are the methods of error handling required to correct or avoid an error situation.

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 disk.

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 and a DOM= or END= branch is not taken, an error message is displayed on the user terminal in the following form:

```
!ERROR=nn : ERROR MESSAGE
```

where:

nn is a number identifying the type of error that has occurred.

ERROR MESSAGE is 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.

Error Processing

If it is necessary to continue the program at a different statement, enter the following:

```
GOTO n
```

where n is 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 occured. The ERR function generates an integer which can be used in an ON/GO statement to construct a multiple branch. Refer to the ON/GOTO directive in Section 4.

Catastrophic Errors

Errors between 100 and 199 (excluding 126 and 127) and ERROR 254 indicate a serious problem with either the system itself, or with what the user is doing. In most cases, correction of a 100 series error requires the intervention of a service representative.

Following is a list of BASIC statements for which ERROR 100 diagnostics are issued:

SAVE (when defining a program file)
INDEXED
SERIAL
DIRECT
MULTI
PROGRAM

SORT
ERASE
WRITE OR WRITE RECORD (Direct file, and only if a new key is created)
REMOVE

ERROR CODES

This subsection describes 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 appears on the terminal.

When an error message displays, locate the error and 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. For example, 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 16000.

!ERROR=0 FILE/REOORD/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 logical disk on which there is an open file. To correct, close all OPEN files.

3. To DISABLE a directory which is already disabled. First ENABLE the affected directory.

4. To ERASE an open file. Do an END on all active terminals.

5. 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 ghost task which had already been started.

11. To START a terminal or ghost which has been opened by another task.

!ERR0R=1 END OF RECORD

This error occurs when an attempt is made to :

1. READ a record with a missing field terminator. To correct, check the possibility 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. 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.

5. The user attempts a READ RECORD from the half-inch tape device, R0, and the string variable is not big enough to accommodate the size of the tape record.

NOTE

BB4 generated an ERROR 1 when an attempt was made to print a string longer than the configured line length of the printer. BB86 does not generate an ERROR 1 in this case, but carries the rest of the string over to the next line(s).

lERROR=2 END OF FILE

This error occurs when an attempt is made:

1. To WRITE a record to a SERIAL file using an IND value greater than the number of records already in the file.

2. To READ/WRITE to a record using an IND value greater than the total number of records defined. (This does not apply to SERIAL files.) To correct, redefine the IND of the READ/WRITE statement or enlarge the file.

3. To WRITE a greater number of records than are defined. (This does not apply to SERIAL files.) 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.

4. To READ sequentially past the highest indexed record or the highest key. To trap this error, use the END= option in the READ statement.

5. To use the KEY or IND function when the last record in the file has been read. Use an END= option to trap this error.

6. On SERIAL files, to READ or WRITE a record too large to fit in the remaining file space.

7. To READ or WRITE a file opened with an ISZ= (BOSS/IX only) 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 print to a spool file which is filled.

10. To READ a Serial file when the last access was a WRITE.

!ERROR=3 DISK READ ERROR

This error can indicate damage, drive misalignment, or faulty disk data recording. The error can occur repeatedly when attempts are made to access data from a damaged disk. The error can also result from electronic malfunctions, or from running the disk under extreme temperatures.

There are essentially three reasons why an ERROR 3 occurs:

a. The record was incorrectly written on the disk.

b. The record was incorrectly READ from the disk.

c. A data error occurred in the disk controller.

If an ERROR 3 occurs, call a Service Representative.

1ERR0R=4 DISK NOT READY

This error occurs when an attempt is made to:

1. ENABLE a disk device when the device name to be enabled is not defined for the system.

2. Use an inoperative disk drive unit. To avoid/correct an ERROR 4 occurrence, do not use the inoperative disk drive unit; have it repaired, or DISABLE the drive.

3. WRITE to a disk when the media or drive is in the READ

   ONLY state (e.g., a floppy disk with the write protect tab on).

4. Perform operations on a drive when the media are "not ready" (e.g., a floppy disk is improperly installed or missing).

!ERR0R=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.

2. An invalid character is read from an input-output device. It can result from faulty storage media such as a damaged diskette or device malfunction.

3. 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.

!ERR0R=6 INVALID DISK DIRECTORY

This error occurs when the system detects an invalid directory, or no directory, on an enabled disk or when a disk or diskette is formatted incorrectly.

!ERROR=7 CORRUPTED FILE

(BOSS/IX only) This error occurs when an attempt is made to access a corrupted file. It is usually caused by a corrupted keyed file, although it may also be caused by a corrupted non-keyed file. The File Repair Utility is provided to repair corrupted files.

1ERR0R=9 POWER FAILURE

(BOSS/IX only) Programs running on BOSS/IX systems will not be managed during a power fail and cannot be restarted from a power-fail condition. The system will not be able to preserve the state of programs or open files. The system does, however, attempt to preserve filesystem integrity by flushing all outstanding operations from the buffer area to the disk prior to power-fail shutdown.

Nevertheless, there is a chance of file damage. Under these conditions the system provides no power-fail error messages.

!ERROR=10 ILLEGAL FILE NAME SIZE OR USAGE/ILLEGAL OVERLAID CALL

This error occurs when:

1. More than 128 characters on BOSS/IX or 52 characters on BOSS/VS are specified as a file identification field of a INDEXED, SERIAL, DIRECT, MULTI, PROGRAM, SORT, OPEN, ERASE, SAVE, LOAD, CALL ADD or RUN statement. (Each file directive specifies the exact file name syntax to be followed.)

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 clause's argument.

3. The file name, combined with the prefix or selected directory name strings, exceeds the total allowable length, which is 128 characters on BOSS/IX or 52 characters on BOSS/VS.

4. The file name size is zero ("") or greater than the maximum file name length of 20 characters. An example would be as follows: LOAD "" or LOAD "MORETHANTWENTYCHARACTERS".

5. An attempt is made to overlay a CALL with no valid calling program in memory.

!ERROR=ll 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 DCM= option on a PRINT or WRITE statement, the ERR variable is set to 11.

!ERR0R=12 MISSING OR DUPLICATE FILE NAME/NON-CONFIGURED DEVICE

This error occurs when an attempt is made to:

1. OPEN a disk data file using a file identification field that has not been previously defined on one of the selected directories by means of a DIRECT, INDEXED, MULTI, PROGRAM, SORT, SERIAL, FILE or SAVE statement.

2. ERASE a file that does not exist on the first directory in the current prefix list.

3. OPEN an input/output device not included in the configuration.

4. Define a disk data file or program by means of a DIRECT, INDEXED, MULTI, SERIAL, PROGRAM, SORT or SAVE statement when a file of the same name already exists in the working directory.

5. Define a disk data file or program by means of a DIRECT, INDEXED, MULTI, SERIAL, PROGRAM, SORT or SAVE statement where the file name is the same as the name reserved for a system device (i.e., LP, Pi, P2...P7, TO, T1...TF, MO, Ml, G0...G3, SY).

6. ADD or DROP a program that is not found.

7. Perform a BOSS/IX LIB directive with an entry point which is not found in the currently loaded BASIC library.

!ERR0R=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 IND or KEY option and the subject record has not been extracted.

4. READ or INPUT from a disk data file using a statement that contains a constant or mnemonic. E.g.:

```
OPEN(1) "FILE"
READ(1) "ENTER NAME: ",A$
```

5. WRITE to Serial file, if the file is not locked.

6. 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.

7. ADD a non-program file to the public memory.

8. DROP a peripheral device or a nonresident program.

9. Attempt a KEY function or KEY= clause on a non-keyed file (that is, other than a Direct, Sort or Multi-keyed file).

10. RELEASE a task-tied (OPEN) ghost task or a task not currently started.

11. Access tape with a BASIC input-output instruction which is not in RECORD mode (e.g., WRITE rather than WRITE RECORD).

12. Attempt an IND function of IND= clause on a multi-key file.

!ERROR=14 IMPROPER FILE OR DEVICE USAGE

This error occurs when an attempt is made to:

1. OPEN a device that is in use (already OPEN).

2. OPEN a disk file or device using a file/device number that is currently being used by that user.

3. START with a non-ghost task 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 disk data file, or program , by means of a DIRECT, INDEXED, MULTI, SERIAL, PROGRAM, SORT or SAVE statement on a disk directory that was previously disabled.

6. LOCK a file that has not been opened by the same user.

7. LOCK a file that has already been locked by the same user.

8. UNLOCK a file that has not been locked, or which has been opened with an ISZ= option on a BOSS/IX system.

9. ADDR a program already resident in the public programming memory area on a BOSS/IX system.

10. START a device (rather than a terminal or ghost task).

11. RELEASE a task that has not been started.

!ERROR=15 DISK SPACE ALLOCATION ERRORS

This error occurs when:

1. No additional disk space is available for file growth or creation.

2. On a BOSS/IX system, a START of a ghost task is attempted, and there is no disk space available for the pipe required by the ghost task.

3. An attempt is made to remove a record from a Direct file, and there is not enough space on disk to add to the file which keeps track of removed keys.

!ERROR=16 DISK OR PUBLIC PROGRAMMING DIRECTORY IS FULL

This error occurs on a BOSS/IX system when:

1. An attempt is made to define a disk data file or program using a DIRECT, INDEXED, MULTI, PROGRAM, SORT, FILE, SERIAL, or SAVE statement when the capacity of the disk directory has been reached. To correct,

   ERASE unneeded data files and/or programs.

2. There is an overflow in the public programming memory. To correct, DROP unneeded public programs.

!ERROR=17 INVALID PARAMETER/NON-CONFIGURED DISK

This error occurs when an attempt is made to:

1. Use a nonexistent directory in a SAVE , PROGRAM, INDEXED, SERIAL, MULTI, DIRECT, SORT, FILE, ENABLE, or DISABLE statement.

2. ADD, SAVE, LOAD or RUN a non-program file.

3. LIST to anything other than an Indexed or Serial file or a device.

4. MERGE from anything other than an Indexed or Serial file on a BOSS/IX system.

5. VMERGE from anything other than a String file.

6. Use AND, IOR or XOR functions with different length arguments.

7. Execute a FILE statement with bad file parameters.

8. SAVE, without parameters, a null program area.

9. Specify ten or more directories for the PREFIX directive on a BOSS/IX system.

10. Illegal field usage on a multi-key file, such as writing to a composite field or reading a numeric value from a string field.

!ERROR=18 ILLEGAL CONTROL OPERATION

This error occurs when an attempt is made to :

1. DROP a program that is busy or has not been ADDR'ed on a BOSS/IX system.

2. SAVE a program that is in public memory on a BOSS/IX system .

3. SAVE to an OPEN file on a BOSS/IX system.

4. Access an encrypted program (LIST, SAVE, PGM function, etc.) on a BOSS/IX system.

5. READ from a file for which the user has no read permission.

6. WRITE to a file for which the user has no write permission.

!ERROR=19 INVALID PROGRAM SIZE

This error occurs when an attempt is made to:

1. On a BOSS/IX system, 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.

3. LOAD, RUN, CALL, ADDR, or ADDE an empty program file.

!ERROR=20 STATEMENT SYNTAX

ERROR 20 is a general catch-all error for the compiler. Illegal punctuation, nonexistent 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. Execute a statement that has a format mask with illegal characters.

2. Execute an EDIT statement that has an illegal parameter option.

3. Enter or execute an I/O statement that contains a key function. For example:

```
>0010 PRINT (1,KEY=K$)
```

4. Use a second argument on a CRC or HSH function whose length is not equal to 2.

5. Execute a user-defined function reference (FNx) where the FNx argument list does not match the DEF argument list.

6. Enter a number which is out of range, such as DIM AS(-1) or DIM A$(33000).

   A distinction is made between entering a number that is out of range and executing a number that is out of range. The following enters a number that is out of range:

```
>0010 DIM A$(33000)
```

   The following attempts to execute a number that is out of range:

```
>0010 LET A=33000
>0020 DIM AS(A)
>RUN
```

   The first example results in an ERROR 20, whereas the second results in an ERROR 41.

7. Enter an exponent that is equal to or greater than 63 or less than or equal to -63. For example,

```
>0100 LET A= .0004E63
```

   results in an ERROR 20. Note that the number can be re-entered as .004E62 without generating an error.

8. The format string (FMT= clause) of a multi-key file is not properly formed. See the TCB(14) for the position of the error within FMT='s string.

!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 16000 or less than 1.

2. LIST or DELETE a statement number greater than 16000 or less than 1.

3. MERGE a statement whose directive is preceded by a statement number greater than 16000 or less than 1.

4. Enter or execute an EDIT, GOSUB , GOTO or ON/GOTO statement with a branch statement number greater than 16000 or less than 1.

5. Enter a statement that contains an IOL=, ERR=, TBL=, DOM=, or END= which specifies a statement number greater than 16000 or less than 1.

6. Execute an EDIT or DELETE statement on a non-existent statement number.

7. Attempt to delete a non-existent statement by entering its statement number in console mode.

8. Add to a nonexistent statement number by use of the Console Mode editing feature,

9. On a BOSS/IX system , execute a CPL function on a text string which has no statement number.

1ERR0R=23 MISSING VARIABLE/NON-DIMENSIONED STRING

This error occurs when an attempt is made to enter or ex-

ecute a statement whose structure implies the absence of a variable, for example:

```
00010 *ERR 23
00010 FOR5=1T010

00010 *ERR 23
00010 FORITO
```

!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 using a function name that 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, for example, 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 usually indicates 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. For compatibility with Level 4, entering a parameter greater than 32767 generates an ERROR 26 if:

   a. LEN= field is greater than 32767 in input verification in an INPUT or READ

   b. Increment to the POS function is greater than 32767

   c. ON-GOTO numeric expression is greater than 32767

   d. Size in PROGRAM statement is greater than 32767

6. 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. DELETE or modify 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.

IERR0R=29 INVALID MNEMONIC

This error occurs when an attempt is made to :

1. Enter or execute a statement containing a mnemonic to an inappropriate device; for example, PRINT 'CS' on a printer.

2. Execute an invalid positioning mnemonic, such as "@(100,100)".

3. Execute an unrecognized mnemonic, such as 'ZZ'.

!ERROR=30 USER PROGRAM INCORRECT CHECKSUM

This error occurs when an attempt is made to:

1. LOAD, CALL, LIST, ADDR or RUN a corrupted BASIC program .

2. Perform a LST function on an invalid string (BOSS/IX only).

3. Run a program in which the interpreter detects invalid or corrupt code (BOSS/IX only).

NOTE

On a BOSS/IX system, ERROR 30 is similar to the ERROR 254, which occurs when an attempt is made to SAVE a corrupted BASIC program . The ERROR 254, however, may indicate that the BOSS/IX interpreter is corrupted , and so it was made a separate error.

!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. On a BOSS/IX system , SAVE the program , enlarge the user area by using the START statement, 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 length-ending of an existing variable. In either case, 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. On a BOSS/IX system , 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. On a BOSS/IX system , 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. On a BOSS/VS system nest more than 256 FORs and GOSUBs.

!ERROR=32 STACK OVERFLOW

This error occurs when internal storage for BASIC program management has been exhausted . This seldom occurs, if ever, and is due to a number of conditions, such as extreme expression complexity, an attempt to compile a statement with a large number of parentheses or nested functions, etc. It may also be caused by an IOLIST statement that loops on itself. For example:

```
00010 PRINT IOL=0020
00020 IOLIST IOL=0030
00030 IOLIST IOL=0020
>RUN

!ERROR=32: STACK OVERFLOW
```

!ERROR=33 INSUFFICIENT MEMORY CAPACITY

On a BOSS/IX system , this error occurs when an attempt is made to use memory that is not available. To correct:

1. Reduce the size of the user area requested by the START statement.

2. RELEASE any terminals not in use.

3. Reduce the amount of memory reserved for the programs and data of other users.

4. Reconfigure with a greater number of logical units.

If ERROR 33's appear on a regular basis, it is 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. Reduce overall system loading, if possible, by temporarily stopping other tasks.

2. Slow down input.

3. Design an application program such that INPUT statements are executed more frequently.

I

!ERROR=35 COMPILER OUT OF MEMORY

On a BOSS/IX system, this error occurs when a statement is entered and there is not enough memory to compile it. It often occurs when a complex arithmetic or logical expression is entered. To correct repeated occurrences, use a larger START size, or simplify the arithmetic or logical expression.

!ERR0R=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=38 ILLEGAL COMMAND IN A PUBLIC PROGRAM

This error occurs when an attempt is made to:

1. Execute one of the following commands in a public program on a BOSS/IX system or in NO EXTEND mode on a BOSS/VS system .

EXECUTE	LIST	RUN	ESCAPE
DELETE	MERGE	SAVE	VMERGE

On a BOSS/VS system , EXTEND mode will flag only RUN.

2. Pass the same single variable more than once as an argument to CALL; e.g., CALL "FILE", AS, AS.

3. Execute a START command in a public program on a BOSS/IX system , when the START is for anything except STARTING a ghost task.

!ERROR=39 ESCAPE IN PUBLIC PROGRAM

This error occurs when ESCAPE is pressed in a public program on a BOSS/IX system .

1ERROR=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 $-10^{63}+1$, or greater than $10^{63}-1$. 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 is 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 = 63)

   b. The number of records in a file (maximum $2^{33}-1$ records).

   c. The record size (maximum 32767 bytes).

   d. An IND or ISZ= value (maximum $2^{33}-1$).

   e. At position @ (maximum 255).

   f. A subscript (range = 1 to 32767).

   g. A program size (maximum 32767 bytes).

   h. A PRECISION (maximum 14).

   i. An ON/GOTO statement whose expression field results in a value greater than 32K.

   j. On a BOSS/IX system, a power (^) (maximum 255).

   k. A key size in a DIRECT or SORT statement (maximum 56).

   1. An increment length in a PCS statement (maximum 32K).

   m. A START size where size is less than 10 or greater than 65535.

   n. 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 4080 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 dimensioned 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. Use the 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 statement that uses a format mask and the number being passed through the mask has more significant digits to the left of the decimal point than been provided for in the format mask.

To correct, redefine the format mask allowing sufficient positions to handle the larger number of digits.

!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.

1ERR0R=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. (BOSS/VS allows FOR if it is followed by NEXT on the same line.)

5. Enter a statement with an IOL= option that references a statement which is not a valid IOLIST statement.

6. Execute a TBL function that references a statement which is not a TABLE statement.

!ERROR=46 INVALID STRING SIZE

This error occurs when an attempt is made to :

1. Execute the ASC function with a null argument (string length = 0).

2. Enter other than eight characters with the SETDAY statement.

3. Enter a SETDAY string expression in a format other than mm/dd/yy.

!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$="ABCD"
>PRINT A$(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

On a BOSS/IX system , this error occurs when a non-translatable statement is encountered during the translation of a program from one level to another.

!ERROR=50 GENERAL MEMORY ERROR

On a BOSS/IX system, this error indicates that a problem exits in the operating system . If an ERROR 50 occurs, call your service representative.

!ERROR=54 OPEN OF SERIAL FILE WITH INVALID HEADER

This error occurs on an attempt to open Serial file with an invalid header.

!ERROR=68 BAD SECOND ARGUMENT TO CPL OR LST

(BOSS/IX only) This error occurs on an attempt to use either the CPL or LST function with a second string argument which is not in the correct format. Refer to the description of the CPL function in Section 10 for the correct argument format.

!ERROR 69 MISSING VARIABLE ID

(BOSS/IX only) This error occurs upon use of the CPL function when the first argument contains one or more variable or user-defined functions whose ids are not described by the eight tables which comprise the second argument. Unlike ERROR 68, the second argument has the correct Format and contains the required variable ID tables.

When an ERROR 69 occurs, the third argument to the CPL function, which is a string argument, is assigned to the first variable or function ID in the first argument that is not described in the second argument. This ID is null-terminated (CHR(O)). If the id is that of a numeric array, the first byte has hexadecimal 80 added to it.

If the third argument does not exist as a string variable, or exists but is too short to accommodate the ID, either an ERROR 70 or ERROR 71 is returned instead of ERROR 69.

!ERROR=70 THIRD ARGUMENT TO CPL FUNCTION IS NOT AN ACTIVE VARIABLE

(BOSS/IX only) This error occurs when the third argument to the CPL function, which should be a string variable, has a length of 0. (Refer to the description of ERROR 69.)

!ERROR=71 THIRD ARGUMENT TO CPL FUNCTION IS NOT LONG ENOUGH

(BOSS/IX only) This error occurs when the third argument to the CPL function is defined, but is not long enough to accommodate the ID. For example, if the ID is 3 characters long, the third argument must have a length of at least 4: 3 for the ID and 1 for the null terminator. The maximum length required for the third argument is 12 bytes. (Refer to the description of ERROR 69.)

!ERROR=95 LAN ERROR

This error indicates that an error has occurred in the LAN (Local Area Network) process. The TCB(12) variable contains more information.

!ERROR=98 SPOOLER ERROR

On a BOSS/IX system, this error indicates a spooler error. The TCB (12) variable contains more information.

!ERROR=99 COMM ERROR

This error indicates a communications error. The TCB(12) variable contains more information.

!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 be completed due to a disk error. The task is forced into console mode.

!ERROR=104 CATASTROPHIC DISK FAILURE/FILE POINTERS DAMAGED

An ERROR 104 occurs when an attempt is made to:

1. WRITE to a file when the disk drive is write protected.

2. WRITE to a disk 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 WRITE operations 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 WRITE operations to the file (or directory) have been completed, the error is displayed and the task is placed in console mode.

!ERROR=126 CTRL+Y KEY USED

Use of the <CTRL>+<Y> key sequence can be captured. This is not a catastrophic error. If SETCTL is not in effect , <CTRL>+<Y> is ignored.

!ERROR=127 ESCAPE

The system variable ERR is set to the value 127 when the ESCAPE key is pressed.

!ERROR=254 PROGRAM SAVE ERROR

On a BOSS/IX system , the ERROR 254 occurs only during a SAVE operation on a BASIC program, indicating that the program cannot be converted to its SAVE'd format.

An ERROR 254 is usually caused by a corrupted BASIC program, as is ERROR 30. However, ERROR 254 may indicate that the basic interpreter itself has become corrupted, and so is listed as a separate error.

!ERROR=255 UNKNOWN ERROR

An ERROR 255 indicates that an error has occurred that the system does not recognize.

NOTES

SECTION 10 - BOSS/IX SPECIFIC INSTRUCTIONS

This chapter describes special features of the BOSS/IX implementation of Business BASIC. The first section describes the options available when accessing BASIC from the command interpreter prompt. Following that, the directives, functions, system variables and I/O options

specific to the BOSS/IX implementation of BB86 are described. The instructions are all listed in alphabetic order.

To enter BASIC console mode from the command interpreter mode, the user types "basic" following the prompt.

The command line format is:

```
usrname> basic {options} {command string}
```

The command string must be the last argument on the command line. The order of the options is unimportant, except that they must precede the command string.

-help or -h	This field displays the command line options for the BASIC command.
-nr	(no release) If the -nr option is used, the user remains in the BASIC environment following completion of the specified program and/or directive. If it is not used, the system exits BASIC after running the specified program or directives.
pgm=	This option takes a quoted string argument containing the name of a BASIC program. The program is loaded and run immediately. (The -nr option determines whether the user exits BASIC following execution of the program.)
size= or s=	This option requires an integer argument specifying the task start size. Refer to the START directive (Section 4) for further explanation.

-e This parameter invokes the European format mask. Format masks are described in section 2. The European mask is identical except that commas and decimal points are interchanged.

-q This parameter indicates “quit mode”. The text of error messages is not displayed, nor is the initial BASIC release level displayed upon entering BASIC. If a syntax error occurs, for example, the system will display only:

```
!ERROR 20
```

The usual explanatory text is suppressed.

lib= This parameter takes a string argument specifying a library of executable subroutines required by the program being run. The full directory path of the library should be specified.

trans= This parameter takes a string argument specifying a translation file. The translation file provides for translation of file names. Refer to the SETTRANS directive in Section 4 for a description of the translation file and process.

-x This option causes BASIC to ignore the <ESCAPE> key, unless a SETESC is active. This prevents the user from interrupting execution of the program.

Commnand String The command field is made up of a single string expression. The string, which may be within quotation marks (and must be if it includes any spaces), contains any BASIC directive permissible in console mode. Immediately upon entering the BASIC console mode, the directive is executed. For example:

```
usrname>basic 'AS="HELLO";PRINT AS'
```

The first thing that occurs upon entering BASIC, even before BASIC displays the "READY" message, is that "HELLO" is assigned to AS and then is printed. Semicolons are considered part of the string.

The command string must be the last option on the command line.

Examples

1. usrname>basic -help

   This option displays the menu of options.

2. usrname>basic pgm="PROGRAMNAME"

   This option brings up the BASIC environment, loads the specified program, runs the program, exits BASIC and returns to the command interpreter.

3. usrname>basic pgm="ANOTHERPGM",'A$="STRING EXPRESSION",X=12;PRINT "ABOUT TO RUN ANOTHERPGM"'

   This option list brings up BASIC, initializes the variables A$ and X as specified, prints the message "ABOUT TO RUN ANOTHERPGM", loads and runs the program, and exits BASIC returning to the command interpreter.

4. usrname>basic pgm="YETANOTHER" -q -e -nr

   This option line also brings up BASIC and runs the named program. It also initiates quiet mode <-q) so that error messages display only the error number, invokes the European mask format (-e), and enters BASIC console mode when the program ends (-nr). It does not return to the command interpreter.

INSTRUCTIONS

The remainder of this section describes the directives, functions, system variables and I/O options specific to the BOSS/IX implementation of BASIC.

Format ! {unquoted BOSS/IX command line}

Description "!" is a directive to execute a BOSS/IX command while remaining in BASIC. It can be used in either console or program mode.

```
00010 PRINT "YOUR WORKING DIRECTORY IS",; Ipwd
00020 PRINT "THESE ARE THE FILES IN YOUR DIRECTORY"
00030 !ls | p
```

```
>OPEN(1)"NEWNAME"
!ERROR12
>!copy OLDNAME NEWNAME
>OPEN(1)"NEWNAME"
>
```

! returns BOSS/IX errors to BASIC. For example, the following BOSS/IX command will generate a BOSS/IX error -05, for missing file. BASIC translates this error to the BASIC ERROR 12.

```
>!xyz
Can't execute 'xyz'.  File does not exist.
!ERROR=12  :  MISSING OR DUPLICATE NAME/NON-CONFIGURED
DEVICE
>
```

Format

ADDR "prog-ID" {,ERR=stno}

ADDE "prog-ID" {,ERR=stno}

Description

The ADDR and ADDE directives are used to add a program to the shared area of main memory. The shared memory segment identification is then passed to the BASIC interpreter for use during a CALL, RUN or LOAD. This procedure insures that only one copy of a program is resident in main memory at one time and reduces the search time required for accessing the called programs from disk.

ADDR Adds the specified program into memory. If the program is to be called more than once, adding it is recommended to reduce the time for each CALL. The added program remains in memory until it is DROP'ped by all tasks that ADDR'ed it.

All programs using such frequently called programs should ADDR the program. Otherwise a directory search will be executed even though the program then found to be already in memory.

An attempt to ADDR a program that has already been added by the same task generates an ERROR . 14.

ADDE Adds an error-handling program into memory. This program is then called wherever an error occurs that would cause BASIC to fall into console mode and display an lERROR=nn message.

NOTE

An error handling program that has been ADDE'ed runs automatically whenever an error occurs that would normally cause the system to drop into console mode. This could cause an infinite loop if an error occurs in the error handling program, itself. The only way to end the loop is to kill the process fro, another terminal.

Examples

```
00100 ADDE "YOUGOOFED"

00110 ADDR "OFTENUSED"
```

CLASS=
(SPECIFY PRINT JOB ATTRIBUTES)

Format

CLASS= "str-expr"

where "str-expr" is the name of a print job class.

Description

The CLASS= option is used with the OPEN directive when opening a printer to specify a class entry in the "/etc/class" system file.

The "/etc/class" file contains print job class descriptions, sets of commonly used print job parameters. Refer to the BOSS/IX User Reference Manual for a complete description of Spooling and the class file.

The OPTS= option allows print job parameters to be specified individually.

Example

```
00110 OPEN (1, CLASS="QUICK")"LP"
```

This statement opens a channel to the printer "LP" and invokes the print class "QUICK" for determining print job parameters.

Format

CPL (str-expr {,str-expr} {,str-var} {,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.

In BOSS/IX BASIC, the compiled code for the same BASIC

variables can be different from one BASIC program to another. If one BASIC program is taking the CPL of statements from another BASIC program , the CPL will ordinarily reflect the environment of the first program.

However, it is sometimes desirable to have the CPL reflect the environment of the second, or target program . To achieve this, CPL may take a second string argument. This string contains information about the target program. The information is in the format of the following eight tables: the numeric id table, the numeric sort table, the numeric offset table, the string id table, the string sort table, the string offset table, the numeric location table, and the string location table. These tables are described in Appendix C.

Examples

```
>AS=CPL("1 END")
>PRINT HTA(A$)
 0005000143
 |   |   |
 |   |   |
 |   |   |-compiled "END"
 |   |
 |   |-statement number in
 |   | binary
 |
 |-length of compiled statement
 | binary
```

CPL
(COMPILE)
(cont'd)

In the following program , ARG$ contains information from the 8 tables described in Appendix C. If ARG$ is not specified , the CPL will use the tables currently in memory, i.e, the tables from the last program that was loaded and run. MISSING$ is the variable used to return the id of the missing variable of user-defined function. MISSING$ should be 12 characters long to assure enough room for ids (11 characters plus the null terminator, for a user-defined function).

```
0005 DIM MISSING$(12)
0010 LET ARG$=$0000000F46495253545345434F4E440100000008000
0010:1000200000000A00040009000F000000040000000400000006000
0010:40004000A00080600153455448$
0020 LET B$=CPL("10 FIRST=SEOOND",ARG$,MISSING$)
0030 LET C$=CPL("10 PRINT FNSETH(7)",ARG$,MISSING$)
0060 ARG$(5,1)="G"
0070 PRINT LST (B$,ARG$)
0080 ARG$(LEN (ARG$),1)="X"
0090 PRINT LST (C$,ARG$)

>RUN
0010 LET GIRST=SEO0ND
0010 PRINT FNSETX(7)
```

Format DROP "prog-ID" {,ERR=stno}

Description The DROP directive is the complement of the ADDR/ADDE directive. It informs the system software that a particular program no longer needs to be kept in the public memory. The program is actually removed only when all tasks that have added the program have also dropped it.

Programs that are in use cannot be dropped .

Example

```
1200 DROP "ALINE"?
```

Removes "ALINE" from memory.

DSZ
(AVAILABLE USER MEMORY)

Format DSZ

Description The DSZ variable contains the number of unused bytes remaining in the user memory area .

Example

```
>PRINT DSZ
```

EDIT
(LINE EDITOR)

Format

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

where:

copy-through-value specifies the text in the original statement that is to be kept unchanged.

delete-through-value specifies the text in the original statement that is to be deleted.

replace-value is the new text replacing the existing text on a character-by-character basis.

insert-value specifies the text to be inserted into the original statement without replacing any of the existing characters.

Description

The EDIT directive provides two methods of editing a line of code: an on-screen editor, and a command directed editor. The command directed editor is compatible with the Level 4 BASIC editor. Both editing methods are available only in console mode, except that they may be used in an EXECUTE statement.

The statement number is counted as part of the statement, and so can be edited. In this case, a new statement is added and the original statement remains unchanged .

On-Screen Editing

When EDIT is executed followed only by a statement number, the statement is displayed at the bottom of the screen for editing. The cursor can be moved in the line using the control sequences listed below . Editing can be done in either overstrike or insert mode. The control sequences are:

<CTRL>+<A>	Moves the cursor to the beginning of the current line.
<CTRL>+<C>	Exit the editor and abandon edits.
<CTRL>+<D>	Deletes the character at the cursor.
<CTRL>+<E>	Moves the cursor to the end of the current line.

EDIT
(LINE EDITOR)
(cont'd)

<CTRL>+<F>	Moves the cursor one character right.
<CTRL>+<H>	Moves the cursor one character left.
<CTRL>+<I>	Insert mode is activated.
<CTRL>+<N>	Moves the cursor down one line.
<CTRL>+<0>	Overstrike mode is activated.
<CTRL>+<P>	Moves the cursor up one line.
<CTRL>+<T>	Tab forward, moves the cursor eight spaces to the right.
<CTRL>+<V>	Exit editor and abandon changes.
<CTRL>+<W>	Tab backward, moves the cursor eight spaces to the left.
<DELETE>	Deletes one character left.
<RETURN>	Exit editor and write changes to memory.
<ESCAPE>	Exit editor and abandon changes.

Changes made to the text are made in memory only, and must be SAVE'd to become permanent.

Command Directed Editor

In command directed editing, the changes to a line are specified in the EDIT statement. The editor cursor is understood to begin at the beginning of the line. Copy, Delete, Replace and Insert commands are given to locate the cursor and modify the text.

The Copy command specifies the text to be kept unchanged. The "copy-through" string specifies a pattern of characters to be matched in the statement being edited . The cursor is then positioned at the first character following the first matching sequence in the text, so this corr mand acts as the forward cursor move. All characters are taken literally in finding the match.

EDIT
(LINE EDITOR)
(cont'd)

The Delete command specifies the text to be deleted from the statement. All characters from the cursor position through the first text matching the "delete-through" string are deleted. Text location by matching is done as for the copy command.

The Replace command specifies characters entered over the

existing text, and so is equivalent to editing in over-strike mode. Characters are overwritten one by one as the cursor moves left to right.

The Insert command specifies characters to be entered at the cursor position, and so corresponds to entering text in insert mode. No text is deleted .

All characters following the last character to be deleted , inserted or replaced are automatically copied even without use of the copy option.

Examples

```
LIST 200
00200 REM  "THE ARK IS FULL.  PLEASE LEAVE"

>EDIT 0200 C[E]C[E]C[E]C[E]R[ USE TH][E SKIS"]
LIST 200
0200 REM "THE ARK IS FULL.  PLEASE USE THE SKIS"
```

An alternate method is:

```
LIST 200
00200 REM "THE ARK IS FULL.  PLEASE LEAVE"
>EDIT 0200 C[PLEASE]D["][ USE THE SKIS"]
LIST 200
0200 REM  "THE ARK IS FULL.  PLEASE USE THE SKIS"
```

The series of "ClE]" characters locates the cursor past the fourth "E". The "R[...]" command replaces the succeeding characters. Finally, the insert command appends the last seven characters to the line.

```
>LIST 1200
01200 PRINT(1)"CHANGER"
>EDIT 1200 C["] D[H]
LIST 1200
1200 PRINT(1)"ANGER"
```

Format

ERROR

Description

The ERROR directive displays the BASIC and BOSS/IX error codes for the most recent error. If the error involved only BASIC, and not BOSS/IX, only the BASIC error code is displayed.

The ERR system variable contains the same information on the BASIC error code, and the TCB(12) system variable contains the same information on the BOSS/IX error code.

The error code field is cleared when you enter BASIC or execute a BEGIN or CLEAR. The message is displayed but no code numbers.

Examples

```
>LQAD
!ERROR=20:STATEMENT SYNTAX ERROR
>ERROR
Basic error code: 20
System error code:
>OPEN(1)"MISSING"
!ERROR=12:MISSING OR DUPLICATE NAME/NON-CONFIGURED DEVICE
>ERROR
Basic error code: 12
System error code: -5
```

FID
(FILE INFORMATION)

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 or task, a two-, three- or four-byte name is returned, e.g., "TO", "T12" or "T123".

If the number refers to a disk file, information about the file is returned as described in table 10-1. If the number is 0 and this is a batch job, "Tx" is returned .

Example

```
>OPEN  (2)"AFILE"
>LET A$=FID(2)
>PRINT DEC  (A$(12,3))
```

Displays the maximum number of records for the file opened on unit 2.

FID
(FILE INFORMATION)
(cont'd)

Table 10-1. FID FORMAT

BYTES	LENGTH	DESCRIPTION
1-3	3	000000 (for compatibility with other levels only)
4-9	6	File name (if six or fewer characters; otherwise filled with FF)
10	1	File type: 00 = Indexed file 01 = Serial file 02 = Direct or Sort File 03 = Multi-keyed File 04 = Program file 07 = String file 14 = Protected BASIC Program 17 = Directory
11	1	Defined key size (always 0 for multi-keyed files
12-14	3	Defined maximum number of records
15-16	2	Record size (=1 for STRING)
17-19	2	000000 (for compatibility with other levels only)
20	1	Disk number
21-24	4	Unused
25-28	4	Current number of records
29-31	3	Number of records in the initial extent
32-34	3	Number of records in the growth extent
35-162	6-127	Fully specified filename, e.g., /usr/name/XYZABC

Format

FILE str-expr

where string-expr is a 162-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 string which has the same format as the FID function (see FID function in this section).

FILE may not reference a remote file.

Examples

```
00010 OPEN (1) "ADOOR"
00020 LET F$=FID(1)
00030 CLOSE (1)
00040 ERASE "ADOOR"
00050 FILE F$
```

The above program first erases a file and then re-defines it using the FILE statement on line 50. The file "ADOOR"

will be re-created as a new file having attributes identical to the erased "ADOOR" file. The newly created file is empty, and does not necessarily occupy the same disk location as did the original file.

Format

```
IF log-expr {THEN} statement-a {ELSE statement-b}
     {FI}
```

where:

logical-expr = a relation between variables and/or values using a relational operator sign (=, <, >, <=, >=, <>) or a logical compound of relations (AND, OR)

statement-a and statement-b are BASIC statements.

Description

The BB7 IF directive is exactly like the BB86 directive, with the exception that BB7 accepts FI as the conditional terminator in addition to the BB86 ENDIF terminator. FI is accepted for compatibility with the 7.2 version of BB7.

Refer to the description of the IF directive in Section 4 for a complete description.

Example

```
IF A = 1 THEN
    IF B = 2 THEN PRINT "HERE"
           FI
    ELSE PRINT "DOWN UNDER"
```

ISZ=
(ACCESS FILE AS IF INDEXED)

ISZ=
(ACCESS FILE AS IF INDEXED)

Format

ISZ=recsz

where recsz is 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., KEY the tree 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 disk 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)"BCOK"
>READ RECORD(l)AS
>PRINT HTA(A$)
```

LST
(LIST)

Format

LST (str-expr {,str-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.

For BOSS/IX, the listed code for the same BASIC variables can be different from one BASIC program to another. If one BASIC program is taking the LST of statements from another BASIC program , the LST will ordinarily reflect the environment of the first program .

However, it is sometimes desirable to have the LST reflect the environment of the second, or target program . To achieve this goal, LST may take a second string parameter. This string contains information about the target program. The information is in the format of the numeric id and non-numeric id tables described in Appendix E.

Example

Refer to the description of the CPL function in this section for an extended example, including an example with two string arguments.

```
>10 PRINT AS
>PRINT LST(PGM(10))
>00010 PRINT AS
```

LVL
(RELEASE LEVEL)

Format LVL (num-expr)

Description The LVL function returns information in the form of a string expression containing the system software release level. LVL(O) returns the BASIC release level; LVL(l) returns the BOSS/IX release level.

LVL accepts any numeric argument valued from 0 to 15, but only LVL(O) and LVL(l) are assigned values at this time.

```
>PRINT LVL(O)
EBS7308

>PRINT LVL(l)
EOS7310
```

OPTS=
(SPECIFY PRINTER ATTRIBUTES)

Format

OPTS= str-expr

where str-expr is a list of print job attributes.

Description

The OPTS= I/O option is used with the OPEN statement to specify print job parameters.

The argument string has the same format as the parameter list for the BOSS/IX "lpr" conmand, as described in the

BOSS/IX User Reference Manual, except that the parameter name and value may be either upper or lower case.

The OPTS= parameter allows you to specify such spooling features as priority, number of copies, and whether or not to notify the user upon completion.

Example

```
00100 LET AS="alias=report copies=2 -notify"
00110 OPEN (7, OPTS=A$)"LP"
```

The OPEN statement specifies via the A$ variable that two copies will be printed, and you will be notified when the "report" job is finished.

PFX
(PREFIX LIST)

Format PFX

Description The PFX variable returns a string containing the most recent prefix list. The prefix list is set by the PREFIX directive, described in this section. It is a list of directories to be searched in the event that a file is requested without a full path name.

Example

```
>PRINT PFX
/usr/username/work/memos/scheds
```

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.

If the program is encrypted, an ERROR 18 is generated.

If the statement number is zero, PGM returns the name of the program currently being RUN. If PGM(O) occurs in a

CALLed program, it still supplies the name of the program being RUN.

Examples

```
00100 LET A$=PGM(10)
```

A$ contains the compiled form of statement 10

```
00100 LET A$=LST(PGM(10))
```

A$ contains the listed format of statement 10

Format

PREFIX "prefix_list"

where prefix_list is a list of directories, separated by spaces.

Description

The PREFIX directive defines a set of directories called a prefix list. When searching for files, the system restricts itself to the specific set of directories.

The first directory in this list becomes the user's working directory. If a file is referenced without a full path name, it is first searched for in the working directory.

If the PREFIX list is the null string, the PREFIX list is cleared.

When the user first enters BASIC, an implicit PREFIX is executed; the user's prefix list is automatically set to the current working directory.

Directory names in the prefix list are separated by spaces. Up to 9 directories may be specified.

File creation and deletion operations are attempted only in the first directory in the prefix list, the working directory.

Examples

```
>PREFIX "/usr/fin/acct/rec"
```

Only the working directory is specified.

```
>A$=PFX + " /bin"
>PREFIX AS
```

Adds "/bin" to the current prefix list.

Format

```
PROGRAM "file ID", prog-size {,diskno}{,sectno}
    {,init_alloc}{,add_alloc}{,ERR=stno}
```

where:

prog-size is the maximum size of the program in bytes (cannot exceed 32,767 bytes)

diskno and sectno are ignored

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.

Examples

```
>20 PROGRAM "NOVA", 2000,0,0,ERR=0100
```

Defines program "NOVA", with a maximum size of 2000 bytes in the user's working directory. The disk and sector specifications are ignored.

```
>PROGRAM "DALLAS",3000
```

Defines a program named "DALLAS", with a maximum size of 3000 bytes, in the user's working directory.

PUB
(PUBLIC PROGRAMS)

Format

PUB (int-expr)

Description

The PUB function returns a string representing all of the programs in shared memory.

For compatibility with BB4, PUB accepts integers in the range 0-15. Any number other than zero (0) returns a null string.

The string contains 136 bytes for each public program. If there are none, the string contains no bytes.

The 136 bytes returned for each public program have the Format shown in Table 10-2.

Table 10-2. PUB(0) FORMAT

BYTE	CONTENTS
1	ownership indicator: 0 = someone else 1 = current user
2	type of public program: 0 = called without use of ADDR or ADDE 1 = called with use of ADDR or ADDE
3-6	length of the program in bytes
7-136	fully qualified path name of the program

Example

```
>X$=PUB(0)
>FOR I = 7 TO LEN(X$) STEP 136; PRINT X$(I,128); NEXT I
```

This returns the full path name of each program in shared memory.

Format STRING "file-ID" {,diskno}{,ERR=stno}

Description The STRING directive defines a string file. The disk number parameter is allowed but ignored.

Example

```
STRING "ALONG", ERR=0100
```

TSK
(DISPLAY CONFIGURED DEVICES)

Format

TSK (int-expr)

Description

TSK(O) returns information about each configured device except disk. TSK(l) returns information about each terminal that is logged on (whether it's running BASIC or not), and about each ghost task that was started through BASIC. Integer values 2 - 15 are also accepted for BB4 compatibility, but return the null string.

The string returned by TSK(O) and TSK(l) is a concatenation of six-byte entries. Each entry is in the following Format:

BYTES	DESCRIPTION
1,2	Device name in ASCII (e.g., T0,G1)
3	Device status in ASCII: 0 = available 2 = in use (the status is always 2 for TSK(D)
4,5	Reserved for future use
6	Always the null byte (00)

Example

```
00100 LET A$=TSK(0)
00110 FOR 1=1 TO LEN (A$) STEP 6
00120 LET B$=A$(I,6)
00130 PRINT "DEVICE NAME IS: ",B$(1,2)
00140 IF B$(3,l)="0" THEN PRINT "AVAILABLE"
00150 IF B$(3,l)="2" THEN PRINT "IN USE"
00160 NEXT I
```

Format VMERGE "file-ID"

Description The VMERGE command is used to retrieve a program in LIST Format from a STRING file, and to add that program to the program currently in user program memory.
The statements of the two programs are merged together.

If both programs have a statement with the same statement number, the statement in the string file replaces the one in memory.

The addition of a statement having a statement number that does not exist in the current user program, causes that new statement to be inserted in numerical order according to its statement number.

VMERGE cannot be used in a public program.

Example The following steps show the use of the VMERGE command:

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

```
>LQAD "PGM1"
>LIST
>00010 REM "PGM1, LINE 10"
>00020 REM "PGM1, LINE 20"
>00030 REM "PGM1, LINE 30"
```

2. OPEN a String file ("VED") and temporarily store the program to be merged in it in LISTed format:

```
>STRING "VED", 1
>OPEN (1) "VED"
>LIST (1)
>END
```

3. LOAD, then LIST the program into which PGMl is to be merged (PGM2):

```
>LQAD "PGM2"
>LIST
>00010 REM "PGM2, LINE 10"
>00020 REM "PGM2, LINE 30"
>00030 REM "PGM2, LINE 40"
```

VMERGE
(cont'd)

4. Enter the VMERGE command:

```
>VMERGE "VED"
```

5. LIST the result:

```
>LIST
>00010 REM "PGM2, LINE 10"
>00020 REM "PGM1, LINE 20"
>00030 REM "PGM2, LINE 40"
>00040 REM "PGM2, LINE 40"
```

Statements 10 and 30 appear in both PGM1 and PGM2; the VMERGEd program (PGM2) supersedes the program in user memory (PGM1), so statements 10 and 30 are retained from PGM2.

Statement 20 was in PGM1 only, so it remains; statement 40 was from PGM2 only, so it is merged into user memory.

NOTES

NOTES

OVERVIEW

This chapter describes directives, functions, system variables and I/O options specific to the BOSS/VS implementation of Business BASIC. The instructions are all listed in alphabetic order.

The following single character synonyms are recognized on BOSS/VS systems:

```
'  =  EDIT
?  =  PRINT
/  =  LIST
```

BOSS/VS also allows "$" to be optionally appended to any system string function or variable. These are:

AND, ATH, BIN, CHR, CRC, DAY, FID, GAP, HSH, HTA, IOR, KEY, LRC, NOT, PRX, PNM, SCR, SPX, STR, SYS, WHO, XOR

For example, CHR(13) and CHR$(13) are both acceptable and are equivalent.

BOSS/VS has an EXTEND/NOEXTEND mode. The EXTEND mode allows all BOSS/VS BASIC syntax while NOEXTEND restricts the syntax within programs (but not console mode commands) to a subset which is compatible with BB3 AND BB4 versions of Business BASIC. EXTEND mode is the default.

MAGNET and NS Subroutines

The following NS subroutine is supported through MAGNET in order to maintain compatibility with previous releases for applications using FTF to transfer files. It is used to determine whether a specified system name is defined within the network.

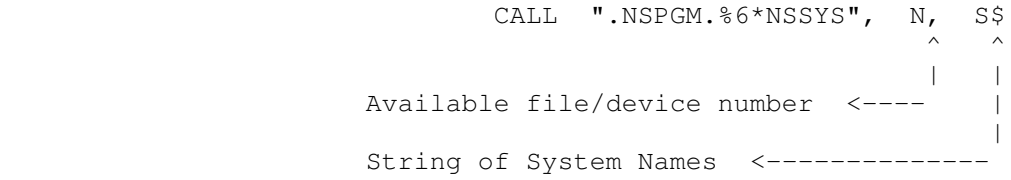
Find System

```
CALL  ".NSPGM.%6*NSFSY" , N, S$, D$
                            ^  ^   ^
                            |  |   |
File/device number <--------   |   |
                               |   |
System Name <------------------    |
                                   |
System Attributes <----------------
```

The file/device number is any legal one which is not in use. The system name is the name of a system to determine whether it is in the network. If the system name is 01, it is changed to the name of the local system . This can be used to determine the name of the system on which the application is running.

A null string returned in system attributes indicates the system does not exist. If the system name does exist, the returned string will not be null but may differ depending on the system release level. On release 8.6A, it is a 10-byte hexadecimal string defining the subnet and station number of the specified system. The first four bytes of the string are the subnet number, and the last six bytes are the station number.

Get System Names



This routing returns a string of all system names in the network separated by a null (00).

Format ! {unquoted BOSS/VS command line}

Description "!" is a directive to execute a BOSS/VS command while remaining in BASIC. It can be used in either console or program mode.

"!" is available in EXTEND mode only.

Examples

```
0100 !RELEASE G31

!DATE
02/12/87
```

ATN
(RADIAN ARCTANGENT)

Format ATN(num-expr)

Description The ATN function returns the arctangent, in radians, of the value specified. That is, ATN returns the size of the angle (in radians) whose tangent is the value specified.

ATN is available in EXTEND mode only.

Note that:

```
2 pi radians = 360 degrees
1 radian     = 75.296 degrees

tan(x) = sin(x)
         -----
         cos(x)
```

Examples

```
>PRINT ATN(30)
 1.54
```

Format

ATTR= str-expr

Description

The BOSS/VS ATTR= I/O option (not to be confused with the BB86 ATTR= option) is used with the OPEN directive to specify Spooler (print job) attributes.

"ATTR" is available in EXTEND mode only.

The string expression is a list of attributes each consisting of the attribute name followed by "=" and the attribute value. The format rules are the same as for the BB86 ATTR= option (described in section 7), except that the attribute names are different. Attribute names and values may be entered in either upper or lower case.

The attributes that take a string expression as their value are:

CLASS=, DATE=, TIME=, FORM=, ALIAS=

The attributes that take an integer expression as their value are:

PRIORITY=, COPIES=, START=, END-, LINES=, SEPPAGES=

The attributes that take a T or F (true or false) value are:

DELETE=, L0ADFORMS=, SP00LON=, NOTIFY=, REQUEUE=, FORMFEED^, RAWMODE=, HIBIT=, PAGING=, NUMBERS=,

HEADINGS=, AUTOFF=, WAIT=

The attributes that take a file name as their value are:

PRINTDEF=, SPOOLNAME=

Refer to the BOSS/VS Spooler documentation for detailed descriptions of these parameters.

If an attribute appears more than once in the list, the last specification is used.

Examples

```
0200 OPEN(l,ATTR="CLASS=X ALIAS=ABC OOPIES=10") "Pi"
0210 OPEN(LPRT,CLASS="X",ATTR="FORM=BIGPAPER'  ,) "LP"
0220 OPEN(3,ATTR="DELETE=F",CLASS="SYSTEM") "P3"
```

Format CIASS=str-expr

Description The CLASS= option is used in an OPEN statement for a printer. It specifies the spooler form class to be used for the print job output to the printer. BASIC allows the name to be up to eight (8) characters.

"CLASS" is available in EXTEND mode only.

Aside from the ERR= option, the ordering of any of the new spooler options or any of the old options (BLK=, SEQ=, and TRK=) is immaterial.

The ERR= clause may be declared more than once and each subsequent ERR= clause immediately overrides the previous one. This is useful only when checking for expression evaluation errors, e.g., CLASS=CHR<500) or COPIES=1/0. The last ERR= clause is the one used if the OPEN fails for most other reasons.

Example

```
0210 OPEN(LPRT,CLASS="X",ATTR="PORM=BIGPAPER") "LP"
```

Format COPIES=int-expr

Description The COPIES= option is used with the OPEN directive to specify the number of copies to be printed. The number of copies can be set to any number from 0 to 99.

"COPIES" is available in EXTEND mode only.

Example

```
0010 OPEN(1, CLASS="X",COPIES=3)"LP"
```

Format

COS(num-expr)

Description

The COS function returns the cosine of the value specified. The value specified must be an angle expressed in radians.

Note that 1 radian = 57.296 degrees.

"COS" is available in EXTEND mode only.

Example

```
>PRINT COS(1.1)
 .45
```

DSZ
(AVAILABLE USER MEMORY)

Format DSZ

Description The DSZ variable always returns 32,767. It is provided for compatibility with other levels of Business BASIC (e.g., BOSS/IX).

Example

```
>PRINT DSZ
 32767
```

EDIT
(LINE EDITOR)

Format

EDIT stno {C[copy through value]} {D[delete through value]}
{R[replace value]} {[insert value]}

where:

copy-through-value specifies the text in the original statement that is to be kept unchanged.

delete-through-value specifies the text in the original statement that is to be deleted.

replace-value is the new text replacing the existing text on a character-by-character basis.

insert-value specifies the text to be inserted into the original statement without replacing any of the existing characters.

Description

The EDIT directive provides two methods for editing a line of code: an on-screen editor, and a command directed editor. The command directed editor is compatible with the Level 4 BASIC editor. Both editing methods are available only in console mode, except that they may be used in an EXECUTE statement.

The statement number is counted as part of the statement, and so can be edited. In this case, a new statement is added and the original statement remains unchanged.

On-Screen Editing

When EDIT is executed followed only by a statement number, the statement is displayed at the bottom of the screen for editing. The cursor can be moved in the line using the key sequences listed below. The key sequences are a subset of those used by the BOSS/VS EDITOR.

On a High-Speed VDT, use the following keys. The NUM LOCK must be turned OFF.

<BACKSPACE>	Destructive backspace, moves the cursor one character left and deletes the character.
ARROW KEYS	The arrow keys can be used to move the cursor anywhere within the line being edited. The -> arrow will append spaces to the end of the statement.

<CTRL>+ ->	Moves the cursor 5 characters right. "->" is the right arrow on the keypad.
<CTRL>+ <-	Moves the cursor 5 characters left. "<-" is the left arrow on the keypad.
<TAB>	Moves the cursor 10 characters right.
<DEL CHAR>	Deletes the character at the cursor.
<SENT>	Deletes all characters from the cursor position to the end of the line.
<INSERT LINE>	Toggles between Insert and Overstrike text entry modes.
<INSERT SPACE>	Moves cursor to the beginning of the line.
<INSERT>	Moves cursor to the end of the line.

On a serial terminal or a high-speed VDT, use the following keys:

<CTRL>+<L> or <CTRL>+<X>	Moves the cursor right one space.
<CTRL>+<N> or <CTRL>+<Z>	Moves the cursor left one space.
<CTRL>+<K>	Moves the cursor up one line.
<CTRL>+<J>	Moves the cursor down one line.
<CTRL>+<W>	Moves the cursor to the beginning of the statement.
<CTRL>+<V>	Moves the cursor to the end of the statement.
<CTRL>+<X>	Deletes the character at the cursor position.
<CTRL>+<R>	Deletes all characters from the cursor position through the end of the statement.

EDIT
(LINE EDITOR)
(conf d)

<BACKSPACE> or <CTRL>+<H>	Destructive backspace, moves cursor one character left and deletes the character.
<TAB>	Moves the cursor right 10 characters.
<CTL-I>	Moves the cursor to the end of the line.
<CTL-II>	Moves the cursor to the beginning of the line.
<ESCAPE>	Ends editing and abandons changes.
<RETURN>	Ends editing and compiles changes.

Changes made to the text are made in memory only, and must be SAVE'd to become permanent.

Command Directed Editor

In command directed editing, the changes to a line are specified in the EDIT statement. The editor cursor is understood to begin at the beginning of the line. Copy, Delete, Replace and Insert commands are given to locate the cursor and modify the text.

The Copy command specifies the text to be kept unchanged. The "copy-through" string specifies a pattern of characters to be matched in the statement being edited. The cursor is then positioned at the first character following the first matching sequence in the text, so this command acts as the forward cursor move. All characters are taken literally in finding the match.

The Delete command specifies the text to be deleted from the statement. All characters from the cursor position

through the first text matching the "delete-through" string are deleted. Text location by matching is done as for the copy command.

The Replace command specifies characters entered over the existing text, and so is equivalent to editing in overstrike mode. Characters are overwritten one by one as the cursor moves left to right.

EDIT
(LINE EDITOR)
(cont'd)

The Insert command specifies characters to be entered at the cursor position, and so corresponds to entering text in insert mode. No text is deleted.

All characters following the last character to be deleted, inserted or replaced are automatically copied without use of the copy option.

Examples

```
LIST 200
00200 REM "THE ARK IS FULL.  PLEASE LEAVE"
 >EDIT 0200 C[E]C[E]C[E]C[E]R[ USE TH][E SKIS"]
LIST 200
0200 REM "THE ARK IS FULL.  PLEASE USE THE SKIS"
```

An alternate method is:

```
LIST 200
00200 REM "THE ARK IS FULL.  PLEASE LEAVE"
>EDIT 0200 C[PLEASE]D["][ USE THE SKIS"]
LIST 200
0200 REM "THE ARK IS FULL.  PLEASE USE THE SKIS"
```

The series of "C[E]" characters locates the cursor past the fourth "E" The "R[...]" command replaces the succeeding characters. Finally, the insert command appends the last seven characters to the line.

```
>LIST 1200
01200 PRINT(1)"CHANGER"

>EDIT 1200 C["] D[H]
LIST 1200
1200 PRINT(1)"ANGER"
```

Format ERROR

Description The ERROR directive displays a description of the most recent error. It is only available from console mode. The description os a four-part numeric message, called a "four-tuple", under which is displayed a descriptive text message:

```
!ERROR=P (Nl, N2, N3, N4)
MESSAGE
```

where:

Nl = a number indicating the OS module detecting the error.

N2 = a number indicating the function involved in the error.

N3 = the BASIC error number that most closely describes the error.

N4 = an internal number indicating the exact error condition.

P = a number, usually equal to N3.

Four-tuples with Nl=28 (system dump condition) are not generated by ERROR.

Example

```
>ERROR

!ERROR=20 (15,0,20,0)

 STATEMENT STRUCTURE (SYNTAX)
```

Format EXP(num-expr)

Description Available in EXTEND mode only, the EXP function returns the value of the natural logarithm, base e raised to the specified power.

Note that e is approximately 2.718, but is 14 digits when used with EXP.

Example

```
EXP(10)
22026.47
```

Format EXTEND

Description The EXTEND directive sets EXTEND mode from NO EXTEND mode, allowing longer (and more meaningful) variable names and some additional functions. It also removes some testic-tions which are imposed on NO EXTEND mode in order to maintain compatibility with other Basic Four systems.

EXTEND can only be used in console mode.

Programs SAVE'd in EXTEND mode cause BASIC to enter EXTEND mode when they are LOAD'ed or RUN.

Example

```
]EXTEND

>
```

Note that the console mode prompt changes from "]" to ">" when in EXTEND mode.

FID
(FILE INFORMATION)

Format

FID (fileno)

Description

The FID function returns information associated with the specified file number.

If the file number refers to a device or task, a two-, three- or four-byte name is returned, e.g., "TO", "T12" or "T123".

If the number refers to a disk file, information about the file is returned as described in Table 11-1.

If the number is 0 and this is a batch job, information about the input file for the job is returned as described in Table 11-1.

Example

```
>OPEN (2)"AFILE"
>LET A$=FID(2)
>PRINT A$(29,52)
```

Displays the full file name of the file opened on channel 2.

FID
(FILE INFORMATION)
(cont'd)

Table 11-1. FID FORMAT

BYTES	LENGTH	DESCRIPTION
1-3	3	000000 (For BB4 conpatibility)
4-9	6	File name (if six or fewer characters; otherwise filled with (FF)
10	1	File type: 00 = Indexed File 01 = Serial File 02 = Direct or Sort File 03 = Multi-keyed File 04 = BASIC Program file 07 = STRING File 14 = BASIC Protected Program File 17 = Directory
11	1	Defined key size of the primary key
12-14	3	Defined maximum number of records
15-16	2	Bytes per record (=1 for STRING)
17-19	3	000000 (fOR bb4 compatibility)
20	1	Always 255
21-24	4	Unused
25-28	4	Current number of records
29-31	3	Number of records in initial extent
32-34	3	Number of records in growth extent
35-162	128	File name if greater than six characters (including installation and family name)

Format

FILE str-expr {,ERR=stno}

where str-expr is an 80 byte string with the same format as returned by the FID function.

Description

The FILE directive can be used to define any file type by placing the parameters of the file into an 80-byte string. This string has the same format as the FID function.

FILE cannot be used to recover a file, as it could on previous systems.

Example

```
10 OPEN(1)"AD00R"
20 F$=FID(1)
30 CLOSE(1)
40 ERASE "ADOOR"
60 FILE F$
```

The following statement can be added to the above program to change "ADOOR" to an indexed file:

```
50 LET F$(10,1)=$00$
```

Format CALL "GETDEVINFO", str-var

Description GETDEVINFO is a routine that returns a string containing information about each of the systems's configured devices. The string is composed of ten-byte substrings, one sub-string per device, in the following format:

BYTES	DESCRIPTION
1-5	Device name, padded with trailing blanks
6	Shared memory controller number and IMLC line number (see SMC ID code below)
7	Device type code (see below)
8	Device status code
9	ISDC line number
10	Always 0

The lower 3 bits of the ISDC line number byte contain the line number of the device if the device resides on an ISDC controller. On a 4-way ISDC (MCS) controller, this is a 2 bit line number and bit 3 is zero. The 16-way ISDC controller is created as two consecutively addressed 8-way ISDC controllers. For any other type of device, this field is zero. The other 5 bits are reserved for future use.

SMC ID Codes

Bits	Description
0	Line number - A=0, B=1 on IMLC
1-6	Shared memory controller number, 0-63
7	Undefined

This is the same format returned by the DEVINFO task variable.

If the device is on an ISDC controller, the shared memory controller number field of this entry is valid and the line number is zero. If the device is neither an IMLC nor an ISDC, the entire SMC ID code is zero.

GETDEVINFO
(cont'd)

Device Type Codes

Code hex	Code dec	Description
00	0	No device
01	1	High speed vdt
02	2	Dataword II MDT in WP mode
03	3	Dataword II MDT in VDT emulation mode
04	4	Ghost terminal
05	5	7250 terminal
06	6	Transportable Batch (Communications (TBC)
07	7	TBC autodial unit
08	8	3270 running on IMLC
09	9	X.25 running on IMLC
OA	10	Basic Four Interface system serial printer
0B	11	Asynchronous driver
OC	12	Asynchronous modem driver
0D	13	7270 terminal
0E	14	EVDT terminal
OF	15	Unused
10	16	Basic Four Interface slave printer
11	17	Parallel matrix printer
12	18	Parallel back printer
13	19	MTR 1/2" Reel-to-reel tape drive
14	20	MTS 1/2" Streamer tape drive
15	21	ODT terminal
16	22	S/10 terminal
17	23	Special VDT device
18	24	Letter quality system serial printer
19	25	Reserved for DMP serial system printer with IGP
1A	26	DMP serial system printer
1B	27	DMP parallel system printer
1C	28	Industry Standard slave printer
1D	29	Reserved for industry standard system serial printer
1E	30	Reserved for Letter quality slave printer
1F	31	Reserved for future GCR tape drive
20	32	MCS 1/4" cartridge streamer tape drive
21	33	Reserved for tape devices
*	*	*
*	*	*
2C	44	Reserved for tape devices
2D	45	EDT terminal
2E	46	Reserved for EDT terminal with monochrome graphics

Device Type Codes (cont'd)

Code hex	Code dec	Desciption
2F	47	Reserved for IMLC diagnostic port
30	48	MAGNET socket
31	49	VDT/B
32	50	14" intelligent terminal
33	51	available
*	*	
*	*	
FF	255	available

Device Status Codes

Bit	Description if bit is ON
0	Escape entered on terminal device
1	Device is open or in use
2	Device is not configured
3	Printer is dedicated
4	Terminal has a slave printer
5-7	Undefined

Example

```
CALL "GETDEVTNFO",AS
```

Format

HELP {str-expr}

where str-expr is a partial or complete BASIC keyword or a BASIC error number.

Description

The HELP directive provides on-line information for BASIC directives, functions, and errors. It is only available from console mode.

HELP without a string expression generates a list of the commonly used BASIC directives and functions.

HELP with a (partial) unique BASIC keyword displays the syntax for that keyword, and information concerning functionality and parameters.

HELP with a non-unique partial BASIC keyword displays a list of all BASIC keywords that match the expression.

HELP with an error number displays a message describing the error.

Examples

```
>HELP SE
SERIAL SETTRACE SETERR SETESC SETCTL

>HELP 20
STATEMENT SYNTAX

>HELP ATN
Arctangent function (in radians):
ATN (<num arg>)
```

LOG
(NATURAL LOGARITHM)

Format LOG(num-expr)

Description Available in EXTEND mode only, the LOG function returns the natural logarithm (base e) of the number specified. This is the power to which e (= 2.718, approximately) must be raised to yield the specified value.

Example

```
>PRINT LOG(10)
 2.3
```

That is, $10 = e^{2.3}$

MAX
(**MAXIMUM ARGUMENT VALUE**)

Format MAX (num-exprl, num-expr2 {,...,num-exprn})

Description Available in EXTEND mode only, the MAX function returns the value of the numeric expressions with the greatest value. The argument list must contain at least 2 arguments.

Examples

```
>LET A=10,B=20,C=30
>PRINT MAX(A,B,C)
 30
```

MIN
(MINIMUM ARGUMENT VALDE)

Format MIN (num-exprl, num-expr2 {,...,num-exprn})

Description Available in EXTEND mode only, the MIN function returns the value of the numeric expressions with the least value. The argument list must contain at least 2 arguments.

Examples

```
>LET A=10,B=20,C=30
>PRINT MIN(A,B,C)
 10
```

Format

NOEXTEND
NO EXTEND

Description

The NO EXTEND directive puts the system into NO EXTEND mode from EXTEND mode. In NO EXTEND mode, the system is compatible with the BB3 and BB4 systems.

In NO EXTEND mode, variable names are restricted to one letter, optionally followed by a single digit (in some cases, as in DEF FNx, only the letter is allowed). Also, the set of directives is restricted to only what BB3 and BB4 allowed, and certain operations, such as using EXECUTE while in a CALL'ed program, are not allowed .

The NO EXTEND directive can only be specified in console mode. When NO EXTEND mode is specified, EXTEND mode variables and functions can be used in console mode, but they cannot be used in a program .

Example

```
>N0 EXTEND

 ]
```

Note that the console mode prompt changes from ">" to "]" when in NO EXTEND mode.

PFX
(PREFIX LIST)

Format

PFX

Description

The PFX system variable contains the user's current directory prefix list.

BOSS/VS prefix names are separated by commas, always begin and end with periods (.), and begin with optional installation qualifiers [enclosed in square brackets] and optional family qualifiers (enclosed in brackets or parentheses).

PFX is available in EXTEND mode only.

Example

```
>PRINT PFX
(family).ACCTG.PETE.
```

Format PRIORITY=int-expr

Description The PRIORITY= I/O option is used when opening a channel to a printer to specify the spooler priority of the print job. The priority is given as an integer value in the range of 0 (low, hold) to 9 (high).

"PRIORITY" is available in EXTEND mode only.

Example

```
0100 OPEN(7,CLASS="ROOM",PRIORITY=5)"LP"
```

Format RANDOMIZE {num-expr}

Description RANDOMIZE is used to establish the seed for the BASIC

pseudo-random function RND.
"RANDOMIZE" is available in EXTEND mode only.

If RANDOMIZE is used without a numeric expression argument, the system sets the random seed to a random initial value. If RANDOMIZE is followed by a numeric expression, the random seed will be set to that value, allowing a previous "random" sequence to be repeated.

Examples

```
>PRINT RND
.9
>RANDOMIZE
>PRINT RND
.>RANDOMIZE 4
>PRINT RND,RND
 0  .73
>RANDOMIZE 4
>PRINT RND,RND
 0  .73
```

Format RND (num-expr)

Description The RND function returns a pseudo-random number.

"RND" is available in EXTEND mode only.

If RND is used alone, the value returned is equal to or greater than 0 but less than 1. If RND is followed by a numeric expression, in parentheses, the random number will be generated and then multiplied by the numeric expression before being returned.

The RANDOMIZE directive may be used before the RND function to establish a seed value, to guarantee a particular

random number sequence, or to insure its unpredictability.

Examples

```
>PRINT RND
 .14
 >RANDOMIZE
 >PRINT RND(10)
 5.08
```

Format

SEQ{UENCE} {stno}{,integer}

Description

The SEQUENCE directive automatically supplies statement numbers when entering a BASIC program.

If a statement number is specified, statement numbers are supplied starting with that number. If no statement number is specified, line numbers begin at 100.

If an increment is specified, statement numbers are stepped by that amount each time <RETURN> is pressed. If no increment is specified, the step is 10.

To terminate automatic statement number generation, press <RETURN> alone following the statement number.

SEQUENCE is only available in console mode.

Examples

```
>SEQUENCE
100 REM "SEQ DEMO"
110
>
```

Format SIN(num-expr)

Description The SIN function returns the sine of the value specified. The value specified is taken as an angle expressed in radians.

"SIN" is available in EXTEND mode only.

Note that 1 radian is approximately 57.296 degrees (i.e., pi / 3).

Example

```
>PRINT SIN(1.1)

 .89
```

SPX
(SYTEM PREFIX)

Format

SPX

Description

The SPX variable returns the current system prefix list. This prefix list is established for all users when the operating system is installed, and provides the mechanism for all users to access system files, such as the utilities.

The format of the returned prefix string is identical to that of the PFX system variable.

SPX is available in EXTEND mode only.

Example

```
>PRINT SPX
```

results in an answer similar to:

```
 ().R6A55.SYS.,.R6A55.INST.
```

Format SQR(num-expr)

Description The SQR function returns the (positive) square root of the non-negative number specified.

"SQR" is available in EXTEND mode only.

Example

```
PRINT SQR(16)
 4
```

SSZ
(SECTOR SIZE)

Format SSZ (<disk number>)

Description The SSZ variable contains the number of bytes in a sector on the specified disk. This value is always 1024.

Example

```
>PRINT SSZ (3)
  1024
```

NOTES

NOTES

APPENDIX A – FEATURES OF THE BUSINESS BASIC PROGRAMMING ENVIRONMENT

OVERVIEW

This appendix provides a more detailed description of some of the programming features of the Business BASIC environment than that given in Chapter 2.

Described here are the following:

- Ghost Tasking
- Public Programming
- Input Buffering
- Field Protection

Programming considerations on Multi-keyed files are covered in a later appendix.

GHOST TASKS

A ghost task is a BASIC task which is not dependent on a terminal for operation. It runs in the background, allowing all terminals to be used for other programs.

Ghost tasks are started by another task, or by the operator using a terminal. The START command is used to start a ghost task. For example:

```
00010 START 32, "PRINT", "GO"
```

where "START 32" indicates 32 pages of user memory, "PRINT" is the name of the program and "GO" is the name of the ghost task that "PRINT" is to run as. The "32" need not be specified on a BOSS/VS system, and will be ignored if it is specified. Up to eight ghost tasks can be configured on a BOSS/IX system (GO through G7), and up to 32 can be configured on a BOSS/VS system (GO through G31). 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 segment causes a task to RELEASE 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:

- The program ordinarily should not attempt to communicate with a controlling terminal because none is assigned; and

- 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 .).

Connunication With a Ghost Task

It is possible to communicate with a ghost task from another task, such as one controlling a VDT. The procedures for reading from and writing to a running ghost task are as follows:

1. Open the ghost

   In order to successfully open a ghost task, it must first have been started and cannot be opened by another

   user at the same time. It must be opened on an alternate channel (logical unit number).

2. READ/WRITE to the ghost

   Once the ghost has been successfully opened, the user can satisfy the task's input/output requests.

   a. If the ghost task is trying to read from a keyboard, the controlling task may only WRITE to the ghost. An attempt to READ results in an ERROR.

   b. If the ghost task is trying to write to a VDT screen, the controlling task may only READ from the ghost. An attempt to write to the ghost results in an ERROR. The ghost task halts on every attempt to perform an input/output routine to a VDT. It resumes when the I/O request is satisfied. If the ghost task is not attempting any terminal I/O, the BASIC user hangs on every attempt to access the ghost. The terminal hangs last only ten seconds. If the ghost performs the complementary I/O operation prior to the end of the timeout, the terminal hang ends and the I/O operation continues.

3. CLOSE the ghost

   Communication with the ghost task ends when the BASIC user closes the channel to that ghost. Any other user may now open the ghost.

The following is a running description of a ghost task that halts on I/O attempts and a BASIC procedure for satisfying the requests.

1. The ghost task halts waiting to print to the terminal.

2. The BASIC user enters:

```
>OPEN (1) "GO"
>READRECORD (1)A$
>PRINT AS
```

3. The terminal then displays:

```
ENTER DATA HERE:
```

which satisfies the output directive.

4. The ghost task then halts on the statement, INPUT AS

5. The user then enters the following:

```
>WRITERECORD (1) "HI THERE!"
```

which satisfies the input directive.

6. The ghost task then halts on the next statement, PRINT A$.

7. The user then enters:

```
>READRECORD (1)A$
>PRINT A$
```

8. The terminal displays:

```
HI THERE!
```

9. The user then closes the channel to the ghost task:

```
>CLOSE
```

The task continues running until it releases itself or is reopened by another user. Ordinarily, the user will not communicate directly with the ghost taks but will write a program to do so.

PUBLIC PROGRAMMING

Public programming is a feature available on the BOSS/IX implementation of BB86, and is supported by a few specific directives. It is also supported on BOSS/VS. It functions in a similar manner on both systems except that BOSS/VS performs ADDR and DROP automatically, and does not support ADDE or PUB. This discussion pertains mainly to BOSS/IX systems.

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.

For 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.

The BOSS/IX operating system manages an area of main memory for maintaining the public (shared) programs and

all open files or devices. The number of control block entries available for this purpose is configurable, and one entry is made for each active public program and each open file or device. When all available control block entries are used, any further attempts to open a file or device or to call or add a public program results in an ERROR 16. This problem does not occur with BOSS/VS.

The ADDR command can be used in BOSS/IX to LOAD the program (make it a resident program). The DROP directive deletes program entries from memory. The CALL, ENTER and EXIT commands are used to run and terminate public programs.

For programs not entered and always in BOSS/VS, the CALL command automatically executes an ADDR directive and adds the program to the shared memory area (and drops it on EXIT).

The PGM and PSZ functions return information about the calling program when executed in a public program .

Restrictions on Public Pro gramming

The following statements cannot be executed from a public program in BOSS/IX. If an attempt is made to do so, an ERROR 38 results. In BOSS/VS, only RUN is restricted.

EXECUTE	MERGE	ESCAPE
DELETE	SAVE	START
LIST	RUN	VMERGE

The trace flag is not altered by a public program , so the statements can be traced. Statements that are traced in public programs are displayed in the same manner as the statements of the calling program .

Programs can be removed from public memory with use of the DROP directive.

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.

The input buffer feature can be turned off for any task by use of the 'ET' mnemonic, and can be reinitiated with the 'BT' mnemonic (see "MNEMONICS" in Chapter 8).

Clearing the **Input Buffer**

The "clear input" mnemonic, 'CI', 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:   ", AS
```

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.

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 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 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 in this section.

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. For example:

```
00090  PRINT <0,ERR=L010)"BALANCE=", A
00100  INPUT <0,ERR=100) "CORRECT? (YES/NO)", 'CI',
00100: D$:("YES"=650,"NO"=725)
```

Input buffering can be disabled by use of the 'ET' mnemonic.

```
00010 BEGIN
00020 SETERR 0500
00030 FOR X = 1 TO 20000
00040 REM "THIS LOOP IS TO SIMULATE PROCESSING TIME
00050 NEXT X
00060 INPUT "ENTER A:",A
00070 INPUT "ENTER B:",B
00080 INPUT "ENTER C:",C
00090 PRINT 'CI',
00100 INPUT "ENTER D:",D
00110 INPUT "ENTER E:",E
00120 PRINT "HERE ARE THE RESULTS:",A,B,C,D,E,
00200
00490 STOP
      PRINT  'CI'
00500 ON ERR (26, 34) GOTO 0510, 0530, 0550
00510 PRINT "PROGRAM TERMINATED BECAUSE OF ERROR",
00510:ERR; STOP
00530 PRINT "ENTER ONLY NUMERIC DATA,"; WAIT 2; RETRY
00550:PRINT "YOU HAVE EXCEEDED THE INPUT BUFFER AREA.
00550:PLEASE REKEY DATA"; WAIT 2; RETRY
01000 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 550 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 take 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:

Data Test 1

Input	Result
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) 4(CR) 5(CR)	ENTER A: 1 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) 4 (CR) 5 (CR)	ENTER A: 1 ENTER B: W ENTER ONLY NUMERIC DATA ENTER B:

The terminal driver supports mnemonics which protect display fields from being overwritten. Protected fields are written in Background Mode, and conce written and protected, cannot be overwritten unless Protect Mode is discontinued.

Protection is a two step process. First, Background Mode must be started ('SB') prior to display of any line or partial line to be protected. Second, Protect Mode must be initiated('PS').

The following mnemonics are associated with Field Protection, and are fully described in Chapter 8 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' 'CS*
   'CF' 'SF'
   'CL'

3. Use of the following mnemonics when 'PS' (protect mode on) is in effect is ignored by the VDT:

   'LD'
   'LI'
   'CL'

4. Use of the following mnemonics resets protect mode:

   'CS'
   'CF'
   'PE'

5. Following execution of 'PS', the cursor is at home position (0,0).

NOTES

NOTES

INTRODUCTION

A Multi-Keyed file is a file of variable-length records which keys contained within the records. While the Direct file can only have a single key associated with each record, and that key must be unique within the file, a Multi-Keyed file may have a large number of keys associated with each record, and duplicate keys are allowed.

The information in this appendix is intended to be an introduction to the use of Multi-Keyed files. It contains guidelines to assist the BASIC programmer to determine when it is appropriate to use Multi-Keyed files, examples of useful techniques to be used when accessing Multi-Keyed files and how to convert an existing application program from the use of Direct and/or Sort files to the use of Multi-Keyed files.

The term "key" is used to refer to a specific entry in a keyed field in an individual record; a "keyset" is the collection of keys corresponding to a single keyed field in a file. One keyset, designated the primary keyset, must always be defined, and is not allowed to have duplicate entries. The remaining keysets may or may not allow duplicates, depending on its definition. Keysets, except the primary keyset, may be added or removed dynamically.

The enhancements to Business BASIC to handle Multi-Keyed files include five new BASIC directives, one new string function, one new I/O clause and a new class of variables. A discussion of these Business BASIC language enhancements is included in this appendix. For a detailed description, refer to the appropriate section in this manual. The new language elements are:

Directives:

- MULTI creates a Multi-Keyed file. This statement uses a "format string" to describe the individual fields within Multi-Keyed file records.
- PACK puts value into a retain buffer in the same way WRITE outputs values to a logical unit.
- UNPACK extracts values from a retain buffer in the same way READ fetchs values from a logical unit. This allows the rereading and reformatting of an I/O record.
- FIELD ALIAS allows dynamic associations of BASIC'S field variables with field names in the file.
- SET FIELD allows a user to add and drop keysets in an existing file.

Functions:

- FMTINFO returns information about the fields within an opened Multi-Keyed file.

I/O Options:

- RETAIN within READ saves the "raw" I/O record so that it may be used by later UNPACKS, PACKs and WRITE RETAINS.

Variables:

- Field variables allow efficient associations to be created between BASIC'S regular variables and fields within a record. These field variables may be used within READ, EXTRACT, FIND, INPUT, WRITE, PRINT, PACK, UNPACK and IOLIST directives.

There is only one difference between BOSS/VS and BOSS/IX implementations of Multi-Keyed files. The BOSS/VS system uses actual variable-length records, so that differing amounts of disk space are used depending on the record's size. The BOSS/IX system, on the other hand, uses physically fixed-length records which are "logically" variable-length. That is, the disk space required for each record is the same, but the record's "size" is retained with each record.

The primary focus of the syntax changes incorporated into BB86 is the addition of specifying the structure of records within a data file at the time the file is created. The method for creating the logical record structure of the data file is to include a "Format" string in the statement that creates the file. This Format string consists of a series of field definitions. Each field defined in the Format string corresponds to a field or item within the data record.

Once the logical structure of records within a file has been described to the system, the BASIC programmer can be freed from much of the complexity of keeping track of the physical position or structure of a field within a data record. This freedom is often referred to as data independence. The syntax of BB86 allows programmers to concentrate on the logic flow of their programs without becoming mired in details about exact field order or location. Instead, the program simply references the name of the appropriate field and the system performs the necessary work to locate the field within the data record. Other benefits of using BB86 syntax and Multi-Keyed files are detailed in this appendix.

Some of the material presented here is specific to the BOSS/VS and BOSS/IX operating systems. When this is the case, the applicable operating system is mentioned.

APPLICATIONS FOR MULTI-KEYED FILES

Multi-Keyed files offer the capability to reference a set of records (a file) by a number of different access paths (keysets). This capability has been possible in the past only through the use of combinations of files. For example, an employee data base might be implemented by storing all of the employee data records in a Direct file with the employee number being the key. Random access to the data records based on other sorts (e.g., employee name or department number) could be implemented by creating one Sort file for each required access path. The keys for these Sort files would contain the information needed (like employee name) as well as some unique identifier that would allow reference to the Direct "master" file. This unique identifier should be the key value associated with the record.

With the use of Multi-Keyed files, it is possible to maintain multiple access paths in a single file. Each access path provides a means to reference the data records both randomly and sequentially according to the order of the alternate key values. Following are some examples of specific cases where the use of Multi-Keyed files would be recommended.

Existing Applications That use Sets of Files

Existing applications that use "sets" of Direct and Sort files to emulate the behavior of Multi-Keyed files may be modified to use the Multi-Keyed file type. The key of the Direct file becomes the primary key of the new Multi-Keyed file. Each Sort file becomes an alternate (or duplicate) keyset on the Multi-Keyed file. Specific instructions for converting programs are given below .

Existing Applications That use the Sort Utility

Some applications do not need an alternate access path to be maintained during normal operations, but only on special occasions such as end-of-month processing. In the past, this requirement might be met by using the Sort utility to generate a new file by sorting the Direct file on the desired field. With the advent of Multi-Keyed files, this requirement can be satisfied by using SETFIELD statements in the BASIC program that requires the alternate access path . The SETFIELD directive can be used to generate a new keyset (access path) on a defined field. The same SETFIELD directive can be used to remove the keyset when there is no further need for that access path . Note that SETFIELD locks the file during the creation of the keyset.

Enhancement of Existing Applications

Existing applications may be enhanced by the use of multiple access paths available with Multi-Keyed files. The easy-to-use nature of the Multi-Keyed syntax will make development of these enhancements to programs much simpler and will reduce development time.

Rewriting Old Or Writing New Applications

When applications need to be rewritten for other reasons (for example, for improved maintainability or added functionality) or when new applications are planned, the

added functionality of Multi-Keyed files and the improved readability of BB86 syntax should certainly be considered.

THE BENEFITS OF USING MULTI-KEYED FILES

When a single Multi-Keyed file is used in place of the techniques previously available, such as using sets of Direct and Sort files, system administrators and operators will be able to take advantage of several benefits. These benefits will take the form of reduced complexity and improved productivity, both in program development and in day-to-day operations.

Reduced File Maintenance

One of the benefits that system administrators and operators will receive is a reduction in file maintenance tasks. When a set of files is used, care must be taken to save or restore the entire set when making or using backup copies. Obviously, with a single file, there are no concerns about missing part of the data base on a backup.

Improved Data Integrity

It is easier to maintain data integrity within the data base by using Multi-Keyed files because file maintenance is reduced. If an error is made saving, restoring, or copying a set of files, the result may be a mismatched set. Problems of this type would cause unpredictable results that would be very difficult to diagnose.

Moreover, when a record is added to a set of files, it is implemented as a WRITE to the Direct file and a WRITE of the corresponding key to each of the related Sort files. If this "logical write" operation takes place in more than one place in the application, care must be taken to make sure that all files are correctly updated. When operations for rewriting or deleting records are taken into account, it becomes apparent that there is significant potential for programming errors.

Furthermore, files are not 'self describing and the correct updating sequence is "hidden" in the application program. The problem of maintaining correct record updates is compounded when a new access path (ans Soft file) is to be maintained. None of these problems exist with Multi-Keyed files because the file system is responsible for maintaining all keysets and all other bytes associated with each of the fields in the file.

Another potential cause for data integrity problems is a system failure. While each file in a set of files may be separately recovered using the file repair utility (depending on system type), problems may still exist if an update operation was in progress at the time of the system failure. For example, if a record was being rewritten, it might require updates to several of the associated Sort files. If the system failed between two of the Sort file updates, the set of files would contain inconsistent in-Formation. This kind of error is undetectable by the file repair utilities. Therefore, Multi-Keyed files eliminate many of the typical problems associated with DIRECT and SORT files becoming inconsistent.

Improved Performance

There are several areas where the use of Multi-Keyed files will provide better performance than an equivalent implementation using a set of files. One area of particular interest is the reading of a record based on an alternate keyset. When accessing a set of files, this function requires a READ of the appropriate Sort file, extraction of the Direct file key from the Sort file key, then a READ of the Direct file. The corresponding operation on a Multi-Keyed file requires only a single READ. The READ of the Multi-Keyed file is significantly faster than the sequence required on a set of files.

Reduced Disk Space Requirements

Multi-Keyed files use disk space more efficiently than do sets of Direct and Sort files. The actual amount of savings depends on how many Sort files are involved and the relative sizes of the keys vs. the data records. As the number of Sort files that are replaced increases and as the key size increases, the percentage of saved disk space will increase. In addition, on BOSS/VS, some applications can make use of the variable length record feature of Multi-Keyed files for more disk space savings.

Reduced Complexity of Applications

As mentioned previously, the use of Multi-Keyed files and the new syntax available in BB86 will simplify record access to the user data base. This will lead to fewer programming errors that result in inconsistencies in the user data base. In addition, it may reduce program development and maintenance time and costs.

THE BB86 SYNTAX FOR MULTI-KEYED FILES

This section introduces the BB86 syntax for operations on Multi-Keyed files. It conveys the more important aspects of most operations by use of examples. There are a few more complex, less frequently used aspects which the reader may at first skip.

Creating a Multi-Keyed File

Creation of Multi-Keyed files is like the creation of other file types, except with the addition of the required EMT= clause:

```
MULTI FILENAME$,NUMRECS,RECSIZE,FMT=FORMAT$,ERR=0320
```

RECSIZE and ERR= are optional. RECSIZE will be determined by the variables contained in the format.

In this example, FILENAME$ has been previously set to the name of the file to be created.

NUMRECS has been previously set to the maximum number of records to be allowed in the file. The "number of records" has the intuitively obvious meaning: If one writes a record which doesn't replace a record that was there previously, this increases the number of records in the file by one. If one writes a record which replaces a previously-existing record, this has no effect on the record count. If one removes a record, this decreases the number of records in the file by one.

If one attempts to add a record so that the number of records exceeds NUMRECS, then an error 2 occurs.

RECSIZE (optional) has been previously set to the maximum number of bytes allowed in each record. If this value is omitted, BASIC will calculate the value as the smallest number of bytes which allows all the fields in the format string to be present. If a record which is to be written to the file contains more bytes than specified by the explicit or implicit RECSIZE, error 17 will occur.

ERR= (optional) shows the number of the statement to be executed next if the system is unable to create this Multi-Keyed file. This overrides any SETERR statement which may have been executed but does not override SETESC if the error was caused by the <ESCAPE> being pressed.

Format String

FMT=FORMAT$ shows how each record in the file is divided into fields, via (in this example) FORMAT$. FORMAT$ has been previously assigned something which we will call the "format string".

Generally, all records in the same file will be divided into fields in the same way; just how that division is performed as described in the format string. The format string describeas each field in the record layout. We show here a sample format string. For purposes of clarity we show each field on a separate line; in practice, they're typically all part of the same string, and are separated from each other by one or more spaces:

```
EMPLNUM#  = N5 PRIMARY
SOCIALSN# = N9 ALTKEY
FRSTNAME# = S16
LASTNAME# = S20
FULLNAME# = LASTNAME# + FRSTNAME# DUPKEY
DEPTNUM#  = N4 DUPKEY
COMMENT#  = S*200
```

The first part of each field description is a field name ending with #, followed by =. The part of the field name before the # is subject to the same restrictions as BASIC numeric variable names (begins with a letter, etc .). Note that the # symbol is not available on all keyboards. On those keyboards which do not have the # symbol, the sym- bol (English pound sign) may be used instead because it generates the same character code. Table B-l shows the ISO-646 standard characters which generate the necessary character code.

Table B-l. ISO-646 Standard Characters

Language	Symbol	Language	Symbol
ISO646-002 (U.S.)	#	ISO646-017 (SPN.)	
ISO646-021 (GER.)	#	ISO646-015 (ITAL.)	
ISO646-060 (NOR.)	#	ISO646-004 (U.K.)	
ISO646-011 (SWD.)	#	ISO646-016 (PORT.)	#
ISO646-025 (FRN.)		ISO646-019 (GREEK)	#
NEN-646 (DUTCH)	#		

After the equal sign (=) comes the field information, fol- lowed by optional keyset information.

Multi-Keyed files, as the name implies, can have more than one key per record. The keys are derived from the values in the fields of each record. The above example would have a keyset based on each of these fields: EMPLNUM#, SOCIALSN#, FULLNAME#, and DEPTNUM#. Each record would have a key in each of these keysets. The keys for a given record would be equal to the value of each of these four fields in the record.

The keyset information shows whether a given field is represented by a keyset, and what kind of keyset that is. The keyset information may be changed after the file has been in use. The field names and field information are not allowed to change for the life of the file; nor can new fields be added once the file has been created.

Keyset information, if present, is either PRIMARY, ALTKEY, DUPKEY, or NOKEY. If no keyset information is present, then NOKEY is implied.

- PRIMARY means that this field forms the primary keyset of the file. Every Multi-Keyed file must have one and only one PRIMARY keyset. The keyset may not be deleted, and does not allow duplicate values. Other special properties of the keyset are mentioned in this document at the appropriate places.

  Suppose a record is written to a Multi-Keyed file, and the PRIMARY key for that record matches the PRIMARY key for another record that is already in the file. That earlier record will be automatically removed from the file at the same time that the new record is written, unless DOM= is specified in the WRITE statement; in that case, the earlier record will stay, the new record won't be written at all, and the COM branch will be taken.

- ALTKEY means that this field forms a keyset, and no two records in the file may have the same value for this field; this insures that all keys in the keyset are different. The keyset may be deleted, and there may be more than one of them, or none at all.

  Suppose a record is written to a Multi-Keyed file, and an ALTKEY-type key for that record matches the corresponding key for another record that is already in the file. That earlier record will stay in the file; an error 11 has occurred.

- DUPKEY means that this field forms a keyset, and it is permitted for several records in the file to have the same value for this field; this implies that several keys in the keyset may have the same value also . The keyset may be deleted, and there may be more than one of them , or none at all.

  Suppose a record is written to a Multi-Keyed file, and a DUPKEY-type key for that record matches the corresponding key for another record that is already in the file. That earlier record will stay in the file, and the new record will also get written; the records will coexist with duplicate keys in the same keyset.

- NOKEY means that this field is not represented by a keyset at all. If this field is either fixed-length or composite, a keyset (either ALTKEY or DUPKEY) may be added for this field at a later time. Even if it is not represented by a keyset, it may be part of a composite field which is represented by a keyset. In the above example, LASTNAME is not presented by a keyset, but FULLNAME is, and LASTNAME is part of FULLNAME.

  A PRIMARY, ALTKEY, or DUPKEY field may not be longer than 80 bytes. This limit does not apply to a NOKEY field.

  A total of 40 keysets are allowed in a Multi-Keyed file not including NOKEY.

Field Information

Field information in a format string shows (or implies) where the field starts, how large the field is, and what kind of information is in the field.

Where the field starts in the record usually follows the simple rule: "Each field begins just after the end of the field defined just before it in the format string." Exceptions to this rule are discussed later.

When choosing how large the field is, and what kind of in**Format**ion is in the field, one can choose from five general categories:

- a fixed-length string field;
- a variable-length string field;
- a fixed-length numeric field;
- a variable-length numeric field; or
- a composite field.

The following examples illustrate most of the variations available.

N5	a numeric field with an implied "00000" mask
UNSIGNED N5	the same
N5.2	a numeric field with an implied "00000.00 mask
+N5.2	a numeric field with an implied "+00000.00 mask
-N5.2	the same
SIGNED N5.2	the same

N*10	a variable-length numeric field taking an average of 10 bytes, with no implied mask; a terminating character (a line feed) will be added when the field is written and removed when the field is read back in
S10	a 10-byte string, padded with trailing null bytes which will be removed when the field is read back in
LEFT S10	same
RIGHT S10	a 10-byte string, padded with leading null bytes which will be removed when the field is read back in
RIGHT X10	the same, except that the null bytes will be retained when the field is read back in
RIGHT C20	a 20-byte string, padded with leading spaces which will be removed when the field is read back in
S*20	a variable-length string, with no padding; a terminating line feed will be added when the field is written and removed when the field is read back in
LAST#+FIRST#	a composite field composed of two other fields. This does not carve out a new area in the record, but simply gives a new "unifying" name to two other fields which have already been carved out. These two fields need not be adjacent to each other. This is an exception to the "each field follows the previous one" rule.

Note that the specification for S, X, and C type fields may all start with the LEFT or RIGHT modifier, and that LEFT is implied if neither appears.

Variable-length Fields

The length specified for a variable-length field is not enforced when writing that field; but it is used for calculating the maximum record length when creating a file. This means that if the format string describes each record as containig two variable-length fields, and one of the variable-length fields in a particular record is longer than specified in the format string, then the other variable-length field must be shorter, so that the overall record length does not exceed the maximum specified for the file.

In general, variable-length fields must follow all fixed-length fields in the record; in other words, all fields which follow a variable-length field must be variable-length fields as well. Exceptions are described later.

Variable-length fields may not be keysets; that is, they may not be declared to be PRIMARY, ALTKEY, or DUPKEY. The alert reader, however, may think there is a loophole, that it is possible for the first variable-length field to contribute to a composite field. That will not work; the system checks for, and prevents, this situation. Indeed, if any field overlaps a variable-length field, even by one byte, then that field may not be PRIMARY, ALTKEY, or DUPKEY either.

A variable-length field may only make up part of a composite field under the following conditions:

- only a specified number of bytes in the variable-length field is used, thus forcing the number of bytes in this component of the composite field to be fixed also.
- it is the first variable-length field in the record, thus forcing the position within the record as a whole to be fixed.

There are also restrictions on the use of a variable-length field within that strange animal known as a "starting-position clause", described below .

Composite Fields

One might assume that a composite field simply consists of two or more other fields. This assumption is logical, but not quite complete. A composite field may consist also of parts of other fields, or one part of one field. A maximum of eight subfields are allowed in one composite field.

A composite field may not be used to place data into a record (via, for example, WRITE, PRINT, or PACK); it is primarily intended for getting data from a record (via, for example, READ, INPUT, or UNPACK), and can be used as a key.

A composite field is specified by specifying one or more components separated by "+". Suppose a format string specifies a fixed-length field called FIXEDFLDt. Suppose also that the first variable-length field in the format string is called FIRSTVAR#. In that situation, here are examples of variations allowed in a component of a composite field. The first variant is usually used only to combine two or more fields into one. Variants 2, 3, and 4 are normally used to specify part of a field, although it is permissible to "spill over" into the next field.

Variant 5 (not highly recommended) is used to specify part of a record without referring to a particular field at all. There are other variants allowed by the general syntax, but they do not offer any functionality beyond what is provided by these five. We'll discuss each of these five variants in more detail later.

```
1)  FIXEDFLD#
2)  FIXEDFID# (7)
3)  FIXEDFLD# (7) : 5
4)  FIRSTVAR# (3) : 5
5)  39 : 6
```

It is best to view these variants not as complete specifications of composite fields, but as specifications of components of composite fields. To illustrate :

```
0030 LET FORMAT$="FIELD1# ? N5  PRIMARY"
0030:          +" FIELD2# = S15"
0030:          +" FIELD3# = S10"
0030:          +" FIELD4# = FIELD1# + FIELD2# (7)"
0030:                              +" + FIEID3# (2) : 5"
0030:          +" FIELD5# = S12"
```

In this example, FIELD5# will physically follow FIELD3# in each record. FIELD4# does not define a new area in the record; being a composite field, it "comprises" its data from other fields or from unnamed regions of the record.

Note that in a composite field declaration, a component may not name any variable-length field after the first one. It's easy for the system to compute where the first variable-length field begins, and that location is the same for every record in the file. For other variable-length fields, the location is dependent on the lengths of previous variable-length fields, and so can vary from record to record.

Note that there is no indication in these specifications whether the component of the composite field is of type "N", "S", "X", or "C"; all that appears after the colon or the conma is an integer. In practice, all components of a composite field are of type "LEFT S".

Using composite fields as components of other composite fields is not permitted.

How is an excessively long component (one that overflows into the next field) handled? In most of these examples, there is no verification that the component actually fits in the named field, or even that it fits in the whole record . If the component spills over into the next field, then it is useful to keep in mind how the component will be built. To do so, remember that:

- o if the component spills over the end of a fixed-length field, the next byte in the component will normally be the first byte of the next field;

- o if the component spills over the end of a variable-length field, the next byte in the component will normally be a line feed, which will be followed by the first byte of the next field;

- o if the component spills over the end of the record (which will end with a line feed if the last field was variable-length), then the remaining bytes of the component will be null (all bits off).

Keeping those things in mind, here's a discussion of the above examples:

1) FIXEDFLD#

   In this example, the component is simply some other field. This is the usual way to combine two fields as the components for a composite field.

2) FIXEDFLD# (7)

   In this example the component doesn't consist of all of FIXEDFLD#, but only of that part of it which starts at byte 7. This is similar to the BASIC substring notation in which one can refer to all the bytes after byte 6 of a string by saying STRING$ (7). An error occurs in this example if FIXEDFLD# is less than 7 bytes long.

3) FIXEDFLDt (7) : 5

   In this example the component doesn't consist of all of FIXEDFLD#, but only of the five bytes beginning at byte 7. There is no requirement that FIXEDFLD# be 11 bytes long, or even 7 bytes long. If it isn't 11 bytes long, then this subfield will extend beyond the end of FIXEDFLD#; if it isn't even 7 bytes long, then this

   subfield will actually begin beyond FIXEDFLD#.

4) FIRSTVAR# (3) : 5

   This example works the same way as FIXEDFLD# (7) : 5. There is no requirement that FIRSTVAR# be 7 bytes long, or even 3 bytes long.

   Each component of a composite field must be of fixed length; therefore, FIRSTVAR# (3) won't work, because the length depends on how long FIRSTVAR# itself is.

5) 39 : 6

Suppose the whole record as it is stored on disk were accessible as a BASIC variable called RECORD$. Then this example would define the corrponent as RECORD$ (39,6). There is no requirement that the record be 45 bytes long, or even 39 bytes long.

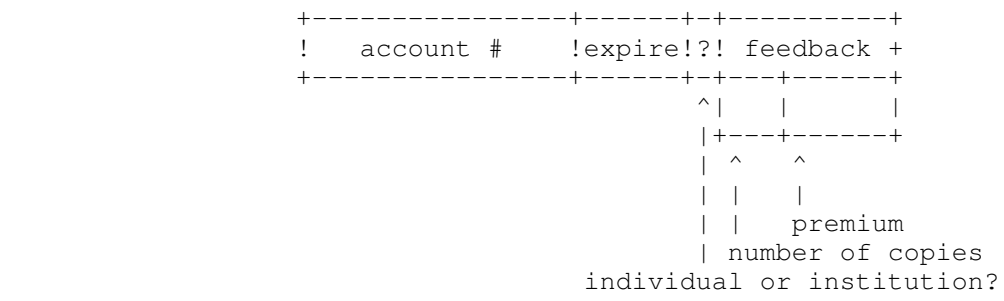
Fields That Don't Follow Each Other

There are two situations in which the actual order of fields in the data record is different from the order in which the fields were specified in the format string. The first situation is the use of a composite field. If a Format string contains (perhaps among other items) the following adjacent items:

- a non-composite field
- a composite field
- a non-composite field

Then the second non-composite field shown above will normally follow the first non-composite field.

The second situation is a sort of "multiple viewpoint" of the record. In some contexts it is referred to as "multiple logical record types", and it works as follows: Suppose sometimes the record could be divided into certain fields, but in other situations it would be more appropriate to divide it into different fields, with different names and at different locations in the record. Or suppose that part of the record should always have the same layout but the rest of the record could have one of several layouts depending on what's found in the first part of the record. We might have a record, for example, that looks like this:



In this partial example of a record layout for a file of subscribers to a magazine, there is a field which identifies whether the subscriber is an individual or an institution. Based on that field, either the next field shows details of whether the individual subscriber has sent letters to the editor of the magazine, or the next two fields show how many copies are to be delivered to the institution and what premium to the normal subscription price must be paid by that institution.

There is a way to use the format string to specify this sharing of space in the record. One does this by using an explicit "starting position" on one of the fields, in this case OOPIES#. The above (simplified) example might be expressed as follows:

```
0100 FORMAT$ = "ACCOUNT#  = N8 PRIMARY"
0100:        +" EXPIRE#   = N6 DUPKEY"
0100:        +" ACCTTYPE# = C1 DUPKEY"
0100:        +" FEEDBACK# = S8"
0100:        +" COPIES#   = FEEDBACK# : N3"
0100:        +" PREMIUM#  = +N1.2"
```

What happens in the above format string is that COPIES* does not come after FEEDBACK# in the normal manner; rather, it starts where FEEDBACK* starts. PREMIUM* continues right after COPIES*, following the normal each-field-follows-the-previous-one rule.

There are two variations on the previous example:

1. COPIES# = FEEDBACK# (3) : N3

   This makes the first byte of COPIES# coincide with the third byte of FEEDBACK#.

2. COPIES# = 17 : N3

   This makes the first byte of COPIES* coincide with the 17th byte of the whole record. This sort of dead reckoning without using field names is harder to do correctly, and is not recommended .

Actually, it is recommended that the practice of overlaying field definitions (as shown in this appendix) be avoided entirely for two reasons. The first is that files organized this way could be difficult incorporate into a database management system which may be implemented in the future. The second is that it is easier to

reorganize a file to change (typically lengthen) fields which have not been overlaid than it is to change fields which share space with other fields.

For applications where starting positions are needed, though, there are three additional points to keep in mind.

The first point is a minor restriction. If two fields are defined to occupy exactly the same space in each record (whether they are composite or not), one or the other may be a keyed field (PRIMARY, ALTKEY, or DUPKEY) but not both. This restriction encourages organization of the file so that writes occur faster, because there are fewer keysets.

The second point is to differentiate between two kinds of field specification. Suppose we have the above example of a format string, but with one minor change:

```
0100 FORMATS = "ACCOUNT#     N8 PRIMARY"
0100:        +" EXPIRE#      N6 DUPKEY"
0100:        +" ACCTTYPE#    CI DUPKEY"
0100:        +" FEEDBACK#    S8"
0100:        +" COPIES#      FEEDBACK# : 3"
0100:        +" PREMIUM#     +N1.2"
```

The difference is that COPIES# is now defined as FEEDBACK# : 3, not FEEDBACK# : N3. If the part of the specification after the ":" is an integer, as it is here, then the ":" is an N-, S-, X-, or C-type specification, then the field is not a composite field. In the above example, since COPIES* is a composite field, PREMIUM* is located immediately after FEEDBACK*, and would not be a correct implementation of the diagrammed record layout; but if COPIES* is not a composite field, as in the previous example, then PREMIUM* is located immediately after COPIES*.

The third point to keep in mind when using starting positions is the effect that this has on rules concerning variable-length fields. When using a starting position, one usually starts at a fixed point in the record, by using either an integer, a fixed field, or the first variable-length field; if this is so, then even if prior fields in the format string are variable-length, one may now revert to using fixed fields again. When the starting position is fixed like this, then the first variable-length field declared at or after the starting position has a fixed beginning point within the record; so this variable-length field enjoys the same distinctions accorded to the "first variable-length field" as described elsewhere in this document.

Suppose, on the other hand, that the starting position is not at a fixed point in the record; that is, it is described in terms of a "non-first" variable-length record. In that case, fields declared at or after the starting position must be variable-length fields, and none of them qualifies as the "first variable-length field". This restriction applies only until a subsequent starting position is specified which refers to a fixed point in the record.

Gaps In The Record

It is possible to describe areas of the record which are to contain no information whatsoever. It is difficult to describe a situation in which one might want to do such a thing, but for completeness we offer the following summary of the syntax of such an unused field:

```
FILLER = 39 : 6
FILLER = FIXEDFLD# : 5
FILLER = FIRSTVAR# : 5
FILLER = FIXEDFLD# (7) : 5
FILLER = FIRSTVAR# (3) : 5
```

Note that "FILLER" is not followed by #. Also note the similarity between these examples and some of the examples above.

READING RECORDS FROM A MULTI-KEYED FILE

First we give a few simple examples. Then we go into significant new capabilities such as expanded KEY=, FIELD ALIAS, RETAIN, and UNPACK.

Examples

Suppose a file has the following format string:

```
F1#=S10 PRIMARY
F2#=S5
F3#=S8
F4#=S15
F5#=+N5.2
F6#=S3
F7#=S3
F8#=N3
F9#=S3
```

and suppose that the following READ operation is performed on the file:

```
READ (5) A$, B$, F4#=C$, D, E$, #F8, Q$, #=F3$
```

Then the following variables receive values from the corresponding fields in the record:

```
AS  gets its value from Fl#
B$  gets its value from F2#
C$  gets its value from F4#
D   gets its value from F5#
E$  gets its value from F6#
F8  gets its value from F8#
Q$  gets its value from F9#
F3$ gets its value from F3#
```

Note that the fields do not need to be read in the order in which they appear in the record; indeed, not all of the fields need be read. If the 10 list contains a variable but does not associate a field name with that variable, this implies that the field to be used is the one that follows (in the format string) the field used for the previous variable in the I/O list. If a field name is explicitly shown in an item in the I/O list, it causes a new "current field" to be defined, and may alter the field-to-variable correspondences of later items that may be in the long form (e.g., F4#=C$) or in the short form (e.g., #F8). In the example above, variable D gets its value from field F5# because the previous item in the I/O list explicitly named field F4#. Similarly, Q$ gets its value from F9# because the previous item referenced field F8#.

Also note that numeric variables must correspond to N-type fields, and string variables must correspond to S-, C-, and X-type fields; an error 17 will result if this correspondence is not maintained.

Three equivalent forms of the above READ statement are:

```
READ(5)Fl#=AS,F2#=B$,F4#=C$,F5#=D,F6$=E$,F8#=F8,F9#=Q$,F3#=F3$

READ(5)    AS,    BS,F4#=C$,    D,    E$,  #=F8,    Q$,  #=F3$

READ(5)    AS,    BS,F4#=C$,    D,    E$,   #F8,    Q$,   #F3$
```

One may place everything after the "READ (5)" in the above READ statement into an IOLIST statement, and instead say:

```
    READ (5) IOL=1230
```

Expanded KEY= Capabilities

When reading using the KEY= clause, one may specify the searching of any field which is either PRIMARY, ALTKEY, or DUPKEY (that is, anything but NOKEY). For example, one may say "read the record whose key in keyset F1# is 'Jones'" by using clause:

```
    KEY=Fl#="Jones"
```

Of course, it is also permissible to have the key value in a string variable and use this clause:

```
    KEY=Fl#=STRING$
```

If the variable name matches the field name (except for the # at the end of the field name), certain shortcuts may be taken. The following examples are equivalent:

```
KEY=F1#=F1$
KEY=#=F1$
KEY=#F1$
```

These five examples will only work if field Fl# is of type S, C, or X, but not N (numeric). If the field is of type N, then the following examples will work, with the last three being identical in effect:

```
KEY=Fl#=-987.33
KEY=Fl#=PAYMENT
KEY=F1#=F1
KEY=#=F1
KEY=#F1
```

It is also acceptable not to specify the keyset, in which case the PRIMARY keyset is used. Here are two examples:

```
KEY="Jones"

KEY=-987.33
```

If one does not use the KEY= clause, then the "next" record is read. Since different keysets place different ordering on the records, the keyset which is used to find the "next" record is the last keyset which was previously used in a KEY= clause, whether that previous KEY= clause was in a REMOVE statement or in a READ statement (or variant such as EXTRACT or READ RECORD). If no KEY= clause has been used for this logical unit since it was opened, then the PRIMARY keyset is used for ordering purposes on sequential reads.

Reading Using FIELD ALIAS

Note that field names are hard coded into a BASIC program; that is, there is no way (without using an EXECUTE statement) to input from the user (or compute from scratch) a field name, place it into a string variable, and use that string variable in place of the usual field name, There is, however, a way around this: the FIELD ALIAS statement:

```
0530 FIELD ALIAS (1,ERR=650) X#=F$, Y#=G$, Z#=H$ ...
```

If this statement is in the program, one can then have READ statements containing field names X#, Y#, Z#, etc . The actual fields read from the record, however, are not X#, Y#, and Z#; the actual fields have the names which were in F$, G$, and H$ at the time statement 530 was executed. Assuming that the contents of F$, G$, and H$ have not changed since the execution of the FIELD ALIAS

statement, the following two statements perform the same action:

```
8720 READ (1,KEY=X#=A$) Y#=B$, Z#=C$

8720 EXECUTE "READ (1,KEY="+F$+"=A$) "
8720:                      +G$+"=B$, "
8720:                      +H$+"=C$"
```

Note that this example only specifies aliases for unit 1; a READ statement for unit 2 can also refer to field X#, but this will actually pull field X# from the record that's read from unit 2 (if there is such a field), unless a FIELD ALIAS statement naming #X has also been executed for unit 2.

A statement such as statement 0530 above can be executed more than once; each time there can be different values in F$, G$, and H$, and each time a subsequent read referring to fields X#, G#, and H# will actually read different fields.

An error 17 occurs if any of the following happens:

- reading a numeric field into a string
- reading a field string into a numeric
- performing a FIELD ALIAS where the field name specified in the string doesn't exist in the file to which the specified unit number is open
- performing a FIELD ALIAS where the field name specified in the string isn't even a valid field name (including the "#")

For every logical unit which is open to a Multi-Keyed file there is a RETAIN buffer. When reading data from fields into BASIC variables, it is possible at the same time (i.e., in the same READ statement) to copy the whole record from which those fields come into the RETAIN buffer.

Suppose one reads a record into the RETAIN buffer, and then reads one or more records from the same unit without specifying RETAIN for these later read operations. If this is done, the RETAIN buffer will continue to hold the data from the prior read which specifies RETAIN; the later

reads don't change the buffer at all.

The RETAIN option is used like this:

```
READ (1,KEY=NEWKEY$,RETAIN) F1#=A$,F2#=X
```

It isn't necessary to use the KEY= clause just because one wants to use the RETAIN clause; they are independent features. Just use each one when it's helpful.

After placing the data into the RETAIN buffer, one can pull additional data out of it with the UNPACK statement, which works just like READ except that it pulls the data out of the RETAIN buffer, not out of the file. (Since the system already knows which record is desired, the KEY= clause is not used here.) For example:

```
UNPACK (1,ERR=0970) F3#=B$,F4#=Y
```

Other Variations On the READ Statement

READ RECORD works as READ does, except that no fields are specified (except optionally in the KEY= clause), and that the destination is simply a string variable. This reads the record as it is stored on disk, and is of limited usefulness.

EXTRACT works just like READ, with two differences: (a) The record will be locked until the next operation on the same logical unit, (b) If the next operation on the same logical unit is a REMOVE, no key is required.

If a DOM= clause is in a READ or READ RECORD statement, it works the same as for DIRECT files; if the specified key is not found, the error branch is taken. If there is no DOM= clause, then an error 11 is generated .

INPUT and INPUT RECORD work exactly the same as READ and READ RECORD do, respectively.

WRITING RECORDS TO A MULTI-KEYED FILE

Most of the discussion here is about differences (syntactically and otherwise) between reading and writing.

In general, the syntax for a WRITE statement is the same as that for a READ. One difference is the same for Multi-Keyed files as it is for other file types: one may write string expressions and numeric expressions, but one may not read into them. The first two items after the ")" in the following example may appear in a READ statement, but not the last four; but all six may appear in a WRITE statement:

```
WRITE (5) F1#=A$,F2#=A,F3#=B$+"123",F4#=3*X,F5#="Now"
,F6#=98.6
```

(In this example, the odd-named fields are of type S, C, or X; the even-named fields are of type N.)

An IOL= clause, with a corresponding IOLIST statement, may be used here, just as with the READ statement.

Do not use a KEY= clause in a WRITE statement to a Multi-keyed file; the key(s) is (are) generated directly from the record, according to the format string for the file.

If a DOM= clause appears in the WRITE statement, then that branch is taken if either the PRIMARY key for the new record matches an already existing PRIMARY key, or an ALTKEY-type key for the new record matches an already existing key in the corresponding keyset. Note that this prevents a record from overwriting a previously existing one with the same PRIMARY key; without the DOM= statement, the overwriting would take place.

In order to rewrite a record, it is necessary to execute a WRITE statement without the DOM= clause. The PRIMARY key value of the record being written must match the PRIMARY key value of the record in the file which is to be overwritten. It is not necessary to execute an EXTRACT statement (on the record which is to be overwritten) prior to the WRITE, but the EXTRACT ... WRITE sequence is highly recommended in any case where the file is shared.

If, during an EXTRACT ... WRITE sequence, the PRIMARY key value changes before the WRITE, then effectively a brand new record is being written, and the ordinary rules for writing a brand new record will apply. Note that in this situation the record lock will be removed from the old record when the WRITE takes place.

When a record is written or rewritten, it may cause duplicate key values to be created in those fields which are defined to be DUPKEY. These duplicate key values are stored in the corrosponding keyset along with the other keys with identical values. Further, all of these key entries with the same value ate stored chronologically, based on the order of which the KEY was added to the file. For example, if a record is rewritten so that a field defined to be DUPKEY is changed, the old key value of that field will be removed from the corresponding keyset and the new value will be inserted at the end of all other keys with that same value. Thereafter, when reading sequentially through that keyset, within a given set of duplicate key values, records will be returned in the order in which the keys were added to the keyset.

Note that certain operations do not preserve the chronological ordering of duplicate key values. These operations would include file utilities that alter the number of records in the file (UPDATE on BOSS/VS and fchange on BOSS/IX), reconstruction utilities (DISKANALYZER or RECONSTRUCT on BOSS/VS and frepair on BOSS/IX), and utilities that copy files one record at a time.

If the chronological ordering of duplicate key values is to be preserved despite the use of the above-mentioned utilities, or if other orderings are preferred, it is necessary to append a sequencing field to the end of the main field being defined. When this is done, the field need not be defined DUPKEY and should properly be defined ALTKEY, because the addition of the sequencing field will cause all key values of the composite to be unique.

A FIELD ALIAS statement works for subsequent WRITE statements, just as it does for READ statements.

PACK may be used to modify the RETAIN buffer for a logical unit, just as UNPACK may be used to copy fields in the retain buffer into variables. BEWARE, though: a PACK without the RETAIN clause will reinitialize the RETAIN buffer. The following erroneous example will leave just E$ in the buffer:

```
0530 PACK (1) Fl#=A$
0540 PACK (1) F2#=B$; REM WRONG!
0550 PACK (1) F3#=C$; REM WRONG!
0560 PACK (1) F4#=D$; REM WRONG!
0570 PACK (1) F5#=E$; REM WRONG!
```

Each of the following examples will place all five values into the RETAIN buffer. The one on the left will leave ONLY those values in the buffer; the one on the right will also leave untouched any other fields in the buffer which had values before this example ran:

```
0530 PACK(1       )F1#=A$      0530 PACK(1,RETAIN)Fl#=A$
0540 PACK(1,RETAIN)F2#=B$      0540 PACK(1,RETAIN)F2#=B$
0550 PACK(1,RETAIN)F3#=C$      0550 PACK(1,RETAIN)F3#=C$
0560 PACK(1,RETAIN)F4#=D$      0560 PACK(1,RETAIN)F4#=D$
0570 PACK(1,RETAIN)F5#=E$      0570 PACK(1,RETAIN)F5#=E$
```

The example on the left above may be replaced by the first of these two statements, and the example on the right above may be replaced by the second:

```
0530 PACK (1       ) Fl#=A$,F2#=B$,F3#=C$,F4#=D$,F5#=E$
0530 PACK (1,RETAIN) F1#=A$,F2#=B$,F3#=C$,F4#=D$,F5#=E$
```

When the retain buffer contains the record just the way one wants it, one may WRITE using the RETAIN clause:

```
WRITE (1,RETAIN)
```

If the retain buffer contains the record almost the way one wants it, one may modify any desired fields to produce the desired record at the same time one writes the record:

```
WRITE (1,RETAIN) F7#=98.6, #F8$
```

WRITE RECORD works as WRITE does, except that no fields are specified, and that the source data is simply a string variable. The contents of the string variable are used as the exact contents of the record; the string is considered to be divided into fields as specified in the format string. This feature is of limited usefulness.

The RETAIN clause is not allowed on WRITE RECORD.

PRINT and PRINT RECORD work exactly the same as WRITE and WRITE RECORD do, respectively.

REMOVING RECORDS FROM A MULTI-KEYED FILE

One can remove the record which has a given PRIMARY key like this:

```
REMOVE (1,KEY=A$)
```

Since one cannot remove a record by knowing just the value of a non-PRIMARY key for the record, one should not specify a field name in the REMOVE statement:

```
REMOVE (1,KEY=F1#=A$); REM WRONG!
```

One can remove a record which has been EXTRACT 'ed without specifying the key at all:

```
REMOVE (1)
```

If the previous operation to this logical unit was not EXTRACT, an error will result and no record will be removed from the file.

NEW LANGUAGE FEATURES

The following paragraphs address the new elements included in BB86 for Multi-Keyed files. Refer to the appropriate sections of the main text of this manual for the general syntax.

The KEY Function

The KEY function returns the next key in the current keyset (that is, the keyset which was last specified in a KEY= clause for this logical unit). The field will be treated as type "S", with trailing nulls removed. This means, for example, that if the current keyset is of type N5.2, and the next value in that keyset is 3.5, then KEY (that unit) is "00003.5".

The FMTINFO Function

Suppose a BASIC program is to function as a sort of utility to work with files whose format string is unknown. It uses the FMTTNFO function to become acquainted with such a Format string. The FMTINFO function takes 1, 2, or 3 parameters. The first parameter is the number of the logical unit which has been opened to the Multi-Keyed file; this parameter is required. The second parameter is optional and is used to specify individual fields, or that information for all fields is desired. The third parameter, also optional, shows the form in which the program wants the information returned.

Suppose we have a file whose format string was originally formed as follows. (Normally it would not be split into many lines like this, but this arrangement makes it easier to read each field definition.)

```
0530 FORMATS = "F1#=S5 PRIMARY "
0530:        + "F2#=N5 ALTKEY "
0530:        + "F3#=X5 DUPKEY "
0530:        + "F4#=C5 NOKEY "
0530:        + "F5#=S5 "
0530:        + "G5#=F1#+F2# "
0530:        + "F6#=N*10 "
0530:        + "F7#=S*20"
```

Suppose further that this file is open on unit 7 and we perform one of the following operations (they're all equivalent):

```
AS = FMTINFO  (7)
A$ = FMTINFO  (7,0)
AS ? FMTINFO  (7,0,0)
```

Then A$ will contain the format string in much the same Format as it was originally defined (and as it may have been altered by the SETFIELD)operation). A$ is a single string, but for ease of reading we break it up into several pieces, as follows:

```
"F1#=S5 PRIMARY  "
"F2#=N5 ALTKEY  "
"F3#=X5 DUPKEY  "
"F4#=C5  "
"F5#=S5  "
"G5#=F1#+F2#  "
"F6#=N*10  "
"F7#=S*20"
```

Note that two spaces separate each adjacent field description, and that there are no other occurrences of two adjacent spaces in the format string. Also note that the

specification of "NOKEY" for F4# has disappeared; the meaning, however, is the same.

Suppose that we change the call to FMTINFO as follows:

```
A$ = FMTINFO (7,0,1)
```

Then AS will contain the format string in a form which is much easier for a BASIC program to read. In hex, the contents of A$ are as follows:

```
2010 1020 2230 2100 2000 F000 5000 6000
```

Each field is described by two bytes. In the following discussion of the contents of these two bytes, "x" means four bits which may contain anything, and are not guaranteed to contain zero.

The first byte shows what the field type is. The allowable values are:

```
$1x$     "N"  (fixed length)
$20$     "S"  (fixed length)
$21$     "C"  (fixed length)
$22$     "X"  (fixed length)
$5x$     "N*" (variable length)
$6x$     "S*" (variable length)
$Fx$     a composite field
```

The second byte shows what the key type is. The allowable values are:

```
$0x$    NOKEY (not a key)
$1x$    PRIMARY
$2x$    ALTKEY
$3x$    DUPKEY
```

The reader is encouraged to verify that the hex string as shown above in A$ corresponds to the format string originally used to create the file. The most convenient formula for the number of fields defined in the file is the following:

```
3280 A$ = EMTINFO ( 7, 0, 1 )
3290 NUMFLDS = LEN(A$)/2
```

The remaining variations simply allow one to access individual field definitions. The following code:

```
5500 FOR I = 3 TO 4
5510 A$ = FMTINFO ( 7, I, 0 )
5520 B$ = FMTINFO ( 7, I, 1 )
5530 PRINT ,"'",A$,","'"
5540 PRINT HTA(B$)
5550 NEXT I
```

will produce the following output (keeping in mind that in each case the last four bits of B$ are not guaranteed):

```
'F3#=X5 DUPKEY'
2230
'F4#=C5'
2100
```

If the third argument to EM TINPO is 0, it may be omitted. The following variations are identical:

```
A$ = EMTINPO ( 7, 3, 0 )
A$ = FMTINFO ( 7, 3 )
```

The following forms are also allowed, and identical:

```
A$ = FMTINFO ( 7, SOCIALSN#, 0 )
A$ = FMTINFO ( 7, SOCIALSN# )
```

If a BASIC program is reading field names from a terminal, EMTINPO can be used in combination with FIELD ALIAS to obtain information (for example, is this field numeric? does it have a keyset?) about the field whose name has just been typed:

```
0040 INPUT "Enter the field name: ", F$
0050 IF F$ = "" GOTO 40 ELSE IF F$(LEN(F$)) <> "#"
0050: THEN F$=F$+"#"
0060 FIELD ALIAS (5,ERR=80) X#=F$
0070 GOTO 100
0080 PRINT "No such field"
0090 GOTO 40
0100 INFO$ = EMTINPO (5,X#,1);
```

(now look at INFO$ for the information)

INITFILE

The INITFILE directive, when used on a Multi-Keyed file, removes all the records from the file, but keeps the **Format** string unchanged; even though SETFIELD operations may have been performed on the file since it was first created, these operations are not undone by the INITFILE statement.

```
0730 INITFILE "FILE",ERR=0900
```

The ERR= clause is optional.

SETFIELD

It is possible to change a fixed-length or composite field from one to another of these key types: NOKEY, ALTKEY, DUPKEY. The statement looks like this:

```
SETFIELD "FILE",FMT="FIELD3#=DUPKEY",MSG="Progress: "
,ERR=1230
```

The MSG= and ERR= clauses are optional.

The file must not currently be open by anyone.

The PRIMARY field may not be changed to any other keyset type; no other field may be declared as PRIMARY.

An error will occur if changing a field from NOKEY or DUP-KEY to ALTKEY if two or more records have the same data in the specified field; this would generate duplicate keys.

The MSG= clause, if present, specifies that a message

should be displayed on the terminal, followed by a running percentage-complete tally. At the end of the operation, this tally will show 100%. If the clause is omitted, the percentage-complete tally will not show.

The message will be displayed at the screen's current cursor position, unless a new one is specified as in this example:

```
MSG=@(65,20)+"Progress: "
```

It is permissible, but not very useful, to change a field from ALTKEY to ALTKEY, from DUPKEY to DUPKEY, and from NOKEY to NOKEY.

FIELD ALIAS

Suppose that a BASIC programmer is working with a multi-keyed file and doesn't want to hard code the field names into his program. This might happen if the end user of the program will be typing field names at the terminal, or if the program computes the field names (F001#, F002#, etc.).

The ingenious programmer can find a way to achieve this effect, but the result is generally inefficient. The FIELD ALIAS directive is provided to do this efficiently:

```
0210 FIELD ALIAS ( 1) X#=F$
       .
       .
0550 READ ( 1, KEY=X#=G$ ) AS, B$, C$
```

Statement 210 says the following:

From this point on, whenever performing operations on unit 1, which is opened to a Multi-Keyed file, if field X# is specified, the field actually desired is the field in F$ as of the time this statement (statement 210) is executed. This is to be effective until (a) some other FIELD ALIAS on the same unit number (1 in this case) and the same alias name (X# in this example, not the contents of F$) is performed, or (b) the unit (unit 1 in this example) is CLOSE'd.

It is important to remember that the FIELD ALIAS is performed on whatever expression is on the right-hand side of the equals sign as of the time of execution of the FIELD ALIAS statement. For example:

```
7700 AS = "BROWN#"
7710 FIELD ALIAS ( 1) J# = AS
7720 AS = "GREEN#"
7730 READ ( 1, KEY=J#=B$) X$, Y$, Z$
```

will read using field BR0WN#, not field GREEN#.

FIELD ALIAS can be used for more than READ statements; in fact, it can be used effectively in any statement containing a field name, except (of course) another FIELD ALIAS statement.

Miscellany

Modifications to the syntax of Business BASIC elements are described in the main text of this manual, but we note the following.

The syntax for the OPEN and CLOSE statements is the same for Multi-Keyed files as it is for other file types.

Neither the IND function nor the IND= clause is allowed in operations on Multi-Keyed files.

FILE CREATION Examples

The following paragraphs give some examples of creating a Multi-Keyed file.

Sample Creation #1:

This example creates a fairly simple file, DEPTFILE, that has five fields, two of which are keys. An actual department file would have many more fields than this, but this illustrates basic file creation.

DEPTFILE's Record:

DEPTNAME# (S12) (ALTKEY)	DEPTNUM* (N5) (PRIMARY)	DRCTRNUM# (N6)	BUDGET# (N8.4)	EXPENSE# (N8.4)

```
0110 DEPTFMT$ =
0110      "DEPTNAME# = S12 ALTKEY
0110       DEPTNUM#  = N5 PRIMARY
0110       DRCTRNUM# = N6
0110       BUDGET#   = N8.4
0110       EXPENSE*  = N8.4"
0120 MULTI "DEPTFILE",100, FMT = DEPTFMT$
```

DEPTFMT$ = is the beginning of a normal BASIC string assignment. The string on the right-hand side specifies all of the field names and field attributes that will be found in a file which we'll soon create.

DEPTNAME# is the name of a field whose attributes appear after the equals sign.

S12 indicates a string field that's 12 bytes long.

ALTKEY shows that this field is an alternate keyset which doesn't allow duplicate keys.

N5 is a 5 byte numeric field.

PRIMARY specifies the field that's the primary key (this is required).

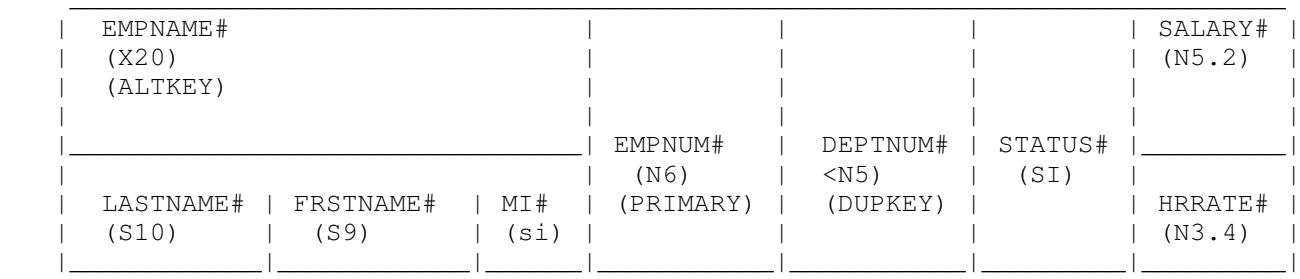
N8.4 is a 13-byte numeric field of the form "00000000.0000".

MULTI creates a Multi-Keyed file named "DEPTFILE" using the format string found in DEPTFMT$ and allocates 100 records in that file. The record size is implied by the **Format** string (49 bytes per record in this example).

Sample Creation #2

This next example creates a more complex record. This record has two fields, EMPNAME# and OMSKEY#, that are composites of other fields, and two overlaid fields, SALARY* and HRRATE# (this is really two separate record types that reside in the same file... a record type for hourly employees if STATUS#="H", and another type for salaried employees if STATUS#="S").

EMPFILE's Record:



OMSKEY# is an S6 field that is composed of the first 5 bytes of LASTNAME# and the first byte of FRSTNAME#.

```
0100 EMPFMT$ = "LASTNAME# = S10
0100   FRSTNAME#   S9
0100   MI#         SI
0100   EMPNUM#     N6 PRIMARY
0100   DEPTNUM#    N5 DUPKEY
0100   STATUS#     SI
0100   SALARY#     N5.2
0100   EMPNAME#    LASTNAME# + FRSTNAME# + MI# DUPKEY
0100   HRRATE#     SALARY#: N3.4
0100   OMSKEY#     LASTNAME#(1,5) + FRSTNAME#(1,1)
0100               DUPKEY"
0110 MULTI "EMPFILE",5000,FMT=EMPFMT$
```

DUPKEY specifies an alternate keyset which allows duplicates.

EMPNAME# is a "composite" field which is made up of three subfields, LASTNAME#, FRSTNAME# and MI#. Note that composite fields may be read or used as a keyset (as EMPNAME# is), but that they may never be written.
SALARY# specifies that the field being defined, HRRATE#, is to begin at the same byte that the SALARY# field started at. This, in essence, specifies two alternate record types... one which uses the field SALARY# and the second which uses the field HRRATE#.

OMSKEY# is a composite field that is made up of parts of two other fields: bytes 2 for 5 of LASTNAME# and byte 1 for 1 of FRSTNAME# (the (1,5) acts just like the substring specifier of a regular string variable). This field is used as a key (a DUPKEY to be precise) and, like all composite fields, is a "read-only" field which may not be written. Note that it would have been nice, but not practical, to have specified here that both OMSKEY# and EMPNAME# were ALTKEYs rather than DUPKEYs.

SAMPLE PROGRAMS

The following programs illustrate the sort of use to which Multi-Keyed files can be put. The examples use a fair number of the facilities, and give a good flavor for the flexibility of Multi-Keyed files. The first two programs use the files just created above.

Program 1: This example references the two previously created files. First, two new records will be added to "DEPTFILE". Finally, a record is read from "EMPFILE", the amount of pay calculated, and the result totaled in the appropriate "DEPTFILE" record.

```
1030 OPEN  (2) "DEPTFILE"
1500 WRITE (2) DEPTNAME# "LANGUAGES", DEPTNUM# 1099
1510 WRITE (2) DEPTNAME# "LARGE SYSTEMS SW", DEPTNUM#
1510:1092
2020 OPEN INPUT (1) "EMPFILE"
2030 READ (1,KEY=EMPNUM# 12345, RETAIN) STATUS# S$,
2030:                                          # DEPTNUM
2040 IF S$="S" THEN UNPACK (1) SALARY# S ; PAY=S
2040:          ELSE UNPACK (1) HRRATE# P ; PAY= R *
2040:HOURS
2050 EXTRACT(2, KEY= #DEPTNUM, RETAIN) #BUDGET,
2050:#EXPENSE
2060 EXPENSE = EXPENSE + PAY
2070 WRITE (2, RETAIN) # EXPENSE
2080 IF EXPENSE > BUDGET THEN PRINT "Department",
2080:DEPTNUM, " blew its budget
```

OPEN... notice that there's nothing special about OPENing a Multi-Keyed file.

WRITE... again notice nothing special.

DEPTNAME# "LANGUAGES" specifies that the field DEPTNAME# is to be given the value "LANGUAGES". Since this is an S12 field, there will be 3 null bytes tacked onto the end Of "LANGUAGES".

DEPTNUM# 1099 puts the value 1099 into the field DEPTNUM#. This is an N6 field so "001099" is actually stuck into the record. Note that the other three fields that weren't specified will be initialized to nulls and the DEPTNUM# and DEPTNAME# fields will be automatically used as keys... a KEY= clause is not allowed on a WRITE since the keys are already part of the data.

KEY=EMPTNUM# 12345 specifies that the READ is to use the EMPNUM# keyset (which was defined as a PRIMARY in the Format... although other keysets defined with ALTKEY or DUPKEY may also be used in this manner) and that the EMPNUM# value is to be 12345 (actually "012345" since this is an N6 type field). Subsequent READs advance through the EMPNUM# keyset until the next KEY= clause is given.

RETAIN causes the read operation to save the "raw" I/O record for later use by UNPACK, PACK RETAIN and WRITE RETAIN statements. Note that each logical unit can have its own "retain" buffer.

STATUS# S$ sets S$ to the contents of the STATUS# field. It would have been an error if 'STATUS# "abc"' had been used here instead, since STATUS#'s value can't very well be read into "abc".

DEPTNUM is exactly equivalent to saying DEPTNUM# DEPTNUM. This takes the value of the DEPTNUM# field and puts it into the program's numeric variable, DEPTNUM. Note that DEPTNUM and DEPTNUM# are two distinct variables in much the same way that A, A$, A(10) and A# are all distinct.

Once S$ has been examined to see if it's "H" or "S", then we'll know whether to use UNPACK to get SALARY# or HRRATE#. UNPACK can be thought of as a kind of reread operation which allows different record types to be handled properly. Note that the proper I/O buffer is available.

EXTRACT works with Multi-Keyed files. Its RETAIN specifies a different retain buffer than what's being used by "EMPFILE" I/O. DEPTFILE's retain buffer will be used later on a WRITE.

KEY=#DEPTNUM is equivalent to KEY=DEPTNUM# DEPTNUM. Since DEPTNUM# is also the PRIMARY keyset, it is also permissible to use KEY=DEPTNUM.

WRITE with RETAIN allows the user to update only one field and retain the values of all others. This is similar to a rewrite, but only implies that values are to be retained, not that the same record is going to be updated .

Program 2: This example is almost identical to the previous one, but introduces the FIELD ALIAS and IOLIST directives.

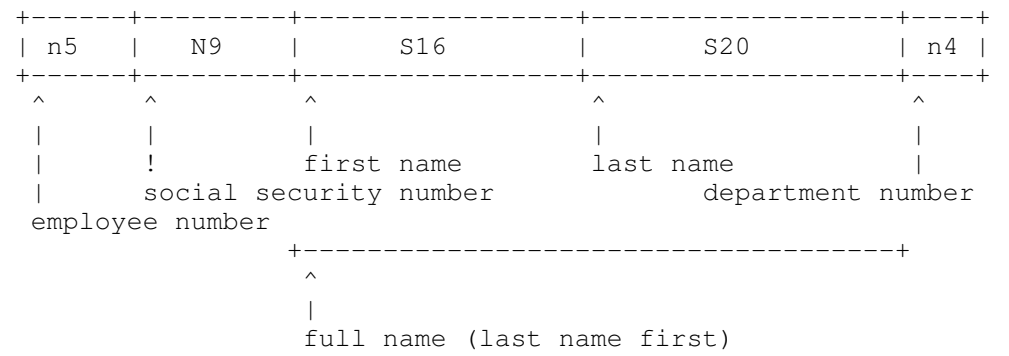
```
1010 OPEN INPUT (1) "EMPFILE"
1020 FIELD ALIAS (1) DEPTNO# = "DEPTNUM#", EMPNO# =
1020:"EMPNUMt"
1030 OPEN (2) "DEPTFILE"
1040 FIELD ALIAS (2) DEPTNOt = "DEPTNUM#"
1500 WRITE (2) DEPTNAMEt "LANGUAGES", DEPTNOt 1099
1510 WRITE (2) DEPTNAMEt "LARGE SYSTEMS SW", DEPTNO#
1510:1092
2000 IOLIST EMPNAMEt ENAME$, #DEPTNO
2030 READ (1,KEY=EMPNO# 12345, RETAIN) STATUS# S#,
2030:IOL=2000
2040 IF S$="E" THEN UNPACK (1) SALARY# S ; PAY=S
2040:          ELSE UNPACK (1) HRRATEt R ; PAY= R *
2040:HOURS
2050 EXTRACT (2,KFY=#DEPTNO, RETAIN ) #BUDGET , #EXPENSE
2060 EXPENSE = EXPENSE + PAY
2070 WRITE (2, RETAIN) #EXPENSE
2080 IF EXPENSE > BUDGET THEN PRINT "Department",
2080:EDEPT, "blew its budget!"
```

FIELD ALIAS (1) sets up future references of the field-variable DEPTNO# to actually use the field DEPTNUM# and EMPNO# to actually use EMPNUM#. Note that this alias only applies to logical unit l's field-variables DEPTNO# and EMPNO#... not to other field-variables of the same name on other logical units. For more information, refer to the topic "FIELD ALIAS" in this appendix.

FIELD ALIAS (2) specifies that DEPTNOt, on logical unit 2 only, actually be DEPTNUM# within the format string.

IOLIST has been extended to allow fields to be associated with BASIC expressions. The IOL=2000 clause then becomes a short-hand way of saying "EMPNAME# ENAME$, DEPTNO# DEPTNO". Since this is almost a textual type of substitution, an IOLIST may be used by any file and on any logical unit (unlike FIELD ALIAS).

The remaining programs all work with the same file. The record layout is as follows:



Within each box in this diagram, "N" means "numeric", "S" means "string", and the number shows how many bytes are reserved for the field.

The layout of the record is described by the format string shown in this statement:

```
 0030 LET FORMAT$="EMPLNUM#  = n5  PRIMARY"
 0030           +" S0CIALSN# = N9  ALTKEY"
 0030              FRSTNAME# = S16"
 0030              LASTNAME# = S20"
 0030              FULLNAME# = LASTNAME#+FRSTNAME# DUPKEY"
 0030              DEPTNUM#  = N4 DUPKEY"
```

EMPLNUM# is the employee number. It is a numeric field, and has been designated the PRIMARY key because the employee number is the principal means of identifying each record in the file (that is, identifying each employee in the company). When a record is deleted, for example, it is the employee number which will be used to identify the

record not the social security number or the last name of the employee. Note thet the PRIMARY keyset of a file may never have duplicate values; in this case, no two employees will have the same employee number.

SOCIALSNt is the social security number. It is a numeric field, and has been declared to be ALTKEY; a keyset will be maintained using this field, so that employee records may be accessed directly by social security number. Should the need arise, the keyset may also be used to print the records so that the social security numbers are in order. Since this field is ALTKEY, not DUPKEY, duplicate values are not allowed in this field; no two employees are allowed to have the same social security number.

FRSTNAME# is the employee's first name. It is a string (non-numeric) field. It has not been declared to be PRIMARY, ALTKEY, or DUPKEY; therefore this field is of type NOKEY, and as it exists can never be used to access records directly by employee's first name ("give me Fred's record") or list the records in order of first name.

LASTNAME# is the employee's last name; like FRSTNAME#, it is a string field, and is of type NOKEY.

FULLNAME# is a combination of FRSTNAME# and LASTNAME#, and is therefore called a "composite" field. It has been declared to be DUPKEY; a keyset will be maintained using this field, so that employee records may be accessed directly using first and last name. ("Get me the records for all Fred Smiths in this company.") Should the need

arise, the keyset may also be used to print the records in alphabetical order, last name first. "Last name first" means that Fred Smith comes after Jim Jones, but if the record is printed as it appears in the file, the first name will appear before the last name on the listing. Since this field is DUPKEY, not ALTKEY, duplicate values are allowed in this field; there may be two Fred Smiths working for the company.

DEPTNUM# is the number of the department to which the employee belongs. It is a numeric field, and has been declared to be DUPKEY; a keyset will be maintained using this field, so that employee records may be accessed directly using the department number. ("Get me the records for all employees in department 1099.") Should the need arise, the keyset may also be used to print the records of all employees, ordered by department number. This field is DUPKEY, not ALTKEY; otherwise, it would not be possible to have any department contain more than one employee, which causes more than one record to contain the same department number.

The three programs which follow are designed more for easy reading than for user friendliness. The first one prints a report showing all the records in the file, ordering them by any field, even if that field is currently declared NOKEY. The second one allows a user to update records which are already in the file. The third one shows how to create the file in the first place, converting from a DIRECT or SERIAL file.

Printing a Multi-Keyed File

This program lists the whole file, in order by any field in the file. If there is no keyset for the field yet, it produces one, lists the file, and then deletes the keyset. (Note that in real life no new keyset could be added if anyone had the file open.)

Lines 0120, 0130, 0140, 0170, and 0180 show how FMTINFO can be used to find out information about an otherwise unfamiliar field.

```
0010 REM "PRINT THE FILE IN A SPECIFIED ORDER"
0020 BEGIN
0030 OPEN (1)"EMPLOYEES"
0040 INPUT "On which field should we sort the file? ",
0040:F$
0050 IF F$ = "" GOTO 40 ELSE IF F$(LEN(F$)) <> "#"
0050: THEN F$=F$+"#"
0060 FIELD ALIAS (1,ERR=80) X#=F$
0070 GOTO 100
0080 PRINT "No such field"
0090 GOTO 40
0100 REM "Position at the beginning of
0100: the requested keyset"
0110 REM "First we must ask: is there a keyset already?
0120 INFO$ = FMTINFO  (1,X#,1)
0130 FLDINFO = INT(ASC(INFO$(1,1))/16)
0140 KEYINFO = INT(ASC(INFO$(2,1))/16)
0150 IF KEYINFO <> 0 GOTO 270
0160 FLDNAME$ = FMTINFO  (1,X#,0)
0170 FLDNAME$ = FLDNAME$ (1,POS("#"=FLDNAME$))
0180 CLOSE (1)
0190 SETERR 220
0195 PRINT "Please wait for keyset generation.",
0200 SETFIELD "EMPLOYEES",FMT=FLDNAME$+"=DUPKEY",MSG=
0200:" "
0205 PRINT 'LF','Generation complete."
0210 GOTO 240
0220 PRINT 'LF','Couldn't create a keyset for this
0220:field"
0230 SETERR 0; GOTO 30; REM "We must reopen the file"
0240 SETERR 0; REM "We didn't catch an error from line
0240: 0190 onward."
0250 OPEN (1)"EMPLOYEES";
0260 FIELD ALIAS (1,ERR=80) X#=F$
0270 REM "This field has a keyset; is this field
0270: numeric or string?"
0280 IF FLDINFO <> 1 GOTO 320
0290 REM "Position at beginning of numeric keyset"
0300 EXTRACT (1,KEY=X#=0)
0310 GOTO 330
0320 REM "Position at beginning of string keyset"
0325 EXTRACT <1,KEY=X#="")
0330 OPEN (2)"P*"
0340 PRINT (2)"EMPLNUM#",
0340: (11),"SOCIALSN#",
0340: (21),"FRSTNAME#",
0340: (41),"LASTNAME#",
0340: (62),"DEPTNUM#"
0350 PRINT (2)""
```

```
0360 READ (1,END=390) #EMPLNUM, #SOCIALSN, #FRSTNAME$,
0360: #LASTNAME$, #DEPTNUM

0370 PRINT (2) EMPLNUM:"00000",
0370: (11), SOCIALSN:"000000000",
0370: (21), ERSTNAME$,
0370: (41), LASTNAME$,
0370: (62), DEPTNUM:"0000"
0380 GOTO 360
0390 CLOSE (1)
0400 CLOSE (2)
0410 IF KEYINFO = 0 THEN SETFIELD "EMPLOYEES",
0410: FMT=FLDNAME$+"=NOKEY"
0420 END
```

Updating a Multi-Keyed File

This program updates a Multi-Keyed file, field by field. As with the other sample programs, most of the human engineering aspects have been grossly neglected, so that the essentials of handling Multi-Keyed files could be seen as plainly as possible.

```
0010 REM "UPDATE A SINGLE FIELD"
0020 BEGIN
0030 OPEN (1)"EMPLOYEES"
0040 INPUT "Employee number: ",EMPLOYEE
0050 IF EMPLOYEE = 0 THEN END
0060 EXTRACT (1,KEY=EMPLOYEE,RETAIN,DQM=80)
0070 GOTO 100
0080 PRINT "No such employee number"
0090 GOTO 40
0100 INPUT "Field name:      ",FLDNAME$
0105 REM "<return> means no more field changes;
0105: rewrite record"
0110 IF FLDNAME$="" THEN WRITE (1,RETAIN); GOTO 40
0120 IF FLDNAME$(LEN(FLDNAME$))<>"#" THEN
0120: FLDNAME$=FLDNAME$+"#"
0125 IF FLDNAME$ = "EMPLNUM#" THEN
0125: PRINT "Can't change employee number"; GOTO 100
0130 FIELD ALIAS (1,ERR=150) X#=FLDNAME$
0140 GOTO 170
0150 PRINT "No such field"
0160 GOTO 40
0170 INFO$=FMTINFO(1,X#,1)
0180 FLDINFO = INT(ASC(INFO$(1,1))/16)
0190 IF FLDINFO = 1 GOTO 270
0200 UNPACK (1) X#=VALUE$; REM "from the RETAIN buffer"
0210 PRINT "Current value:   ",VALUE$
0220 IF FLDINFO = 15 GOTO 100;
0220: REM "Can't rewrite composite fields"
0230 INPUT "New value:       ",VALUES
0240 IF VALUE$="" GOTO 100;
0240: REM "Doesn't want to change the field"
0250 PACK (1,RETAIN) X#=VALUE$
0260 GOTO 100
```

```
0270 REM  "Numeric value"
0280 UNPACK (1) X#=VALUE; REM "from the RETAIN buffer"
0290 PRINT "Current value :   ",STR(VALUE)
0300 INPUT "New value:       ",VALUE$

0310 IF VALUE$="" GOTO 100;
0310: REM "Doesn't want to change the field"
0320 PACK (1,RETAIN) X#=NUM(VALUE$,ERR=340)
0330 GOTO 100
0340 PRINT "Invalid numeric value"
0350 GOTO 290
```

Loading Data into a Multi-Keyed File

This program is fairly straightforward; it initializes the file and does a record-by-record transfer into it.

```
0010 REM "CONVERT TO MULTIKEYED FILE"
0020 BEGIN
0030 LET FORMAT$="EMPLNUM#  = N5  PRIMARY"
0030:          +" SOCIALSN# = N9  ALTKEY"
0030:          +" FRSTNAME# = S16"
0030:          +" LASTNAME# = S20"
0030:        +" FULLNAME# = LASTNAME#+FRSTNAME# DUPKEY"
0030:          +" DEPTNUMt  = N4 DUPKEY"
0040 MULTI "EMPLOYEES",200,FMT=FORMAT$
0050 OPEN (1)"OLDDATA"
0060 OPEN (2)"EMPLOYEES"

0070 READ  (1,END=0100) EMPLNUM,SOCIALSN, FRSTNAME$,
0070: LASTNAME$, DEPTNUM
0080 WRITE (2) #EMPLNUM ,#SOCIALSN,#FRSTNAME$,
0080: #LASTNAME$,#DEPTNUM
0090 GOTO 70
0100 CLOSE (1)
0110 CLOSE (2)
0120 END
```

CONVERTING EXISTING APPLICATIONS

The following points should be considered when planning a conversion of an existing application program from using Direct and Sort files to the use of Multi-Keyed files.

Select an Appropriate Program

Some applications will benefit greatly from conversion to Multi-Keyed file use. The most obvious candidates are those that currently emulate Multi-Keyed file functionality. These are discussed under the topic "APPLICATIONS FOR MULTI-KEYED FILES" in this appendix.

Conversion Approaches

When considering the modification of an existing application program to the use of Multi-Keyed files, the first decision that must be made is how much modification will be required. For some programs, a straightforward substitution approach may be taken. For others, modification may be impractical and redesigning and rewriting the application or portions of it may be required . To make this determination, the application programmer would have to analyze the complexity of the existing application and the difficulty in isolating the references to the individual files that make up the user data base.

Another decision that must be made is how to distribute the development costs. Rewriting an application represents the largest investment in development costs but could result in lower maintenence costs. Another approach would be to locate and replace all references to the main Direct file and its associated Sort files with references to the new Multi-Keyed file. This would require less initial investment, but could be slightly more difficult to maintain than a rewritten version. A variant of this approach would be to replace the Direct file with an equivalent one-keyset Multi-Keyed file, and to replace Sort files with additional keysets on the Multi-Keyed file as time permits. This approach would have the lowest initial cost, distributing the conversion effort over a longer period, but might also present more maintenance difficulties.

Selection of Keysets

When designing the characteristics of a Multi-Keyed file, one of the most important decisions to be made is the selection of which fields will become keysets. A good starting point is to define the primary key of a Multi-Keyed file to be the same field as was used to reference the Direct file and define alternate keysets (either ALTKEY or DUPKEY) for each of the Sort files. During the conversion design process, it may be determined that there are other fields that should also be keyed to allow access to the data in ways that were not possible before. At the same time, it must be remembered that each keyset that is maintained as a permanent part of the file will increase the overhead required for modifications to the file. Therefore, a subset of the keysets defined as useful may be designated as temporary keysets that will be added to the file only when necessary.

Selecting NOKEY Fields

Not all fields which would be useful to have as keysets should be made into "permanent" keysets. If a field is used as a means of locating records very infrequently, it is usually inappropriate to include it as one of the standard keysets. For example, suppose a monthly report is required that selects records from an Employee file based on a field (EXEMPT#) which indicates whether or not the employee is salaried. If the EXEMPT# field were keyed, then it could be used to facilitate the selection of records. However, the fact that the report is required only monthly, presuming the field is not used as a key for other purposes, would argue against defining this field as a keyed field for the entire month .

Finding Records By NOKEY Fields

Several options are available for handling cases where an access path is required on a field, but only infrequently. The easiest solution is to make the field into a keyed field when necessary with the use of the SETFIELD directive. Once the field is a keyed field, it may be used normally to select records. Sometimes, however, it is impractical to perform the SETFIELD operation because that operation requires that the file be used at the time. In this case, it may be necessary to process all of the records in the file, selecting only those records that satisfy the selection criteria. If the output is to be in order by a non-keyed field, an intermediate file should be created with all of the selected records and then that file, on BOSS/VS, could be sorted using the Sort utility.

On BOSS/VS, the Sort utility could be used to sort the entire Multi-Keyed file on the contents of a non-keyed field. This technique would be appropriate if the character positions being sorted on had not been defined as a field within the Multi-Keyed file format. The Sort utility has the same restriction as the SETFIELD operation, in that the file must not be used by others at the time of the sort.

Suggestions for Conversion

A few techniques are useful during the conversion process. These are outlined in the following paragraphs.

Data Layout Diagrams

A pictorial representation of the fields within the. data record will facilitate the design of the data records within a Multi-Keyed file. It is useful to have a picture of the layout of the data records within the Direct file as well as the layout of the new record(s). This is especially helpful if the record design contains more than one logical record (record type).

It is important to define as fields any combination of character positions that might at some point be used as a keyset. Fields that are not initially defined to be keyed may be converted into keyed fields (either ALTKEY or DUPKEY) by using the SETFIELD directive. However, new fields may not be added to a format once the file has been created, making it important to anticipate future needs. When planning the field layout for the Multi-Keyed file, the field for the primary key must be defined. This is important because the field may not have been part of the data record of the Direct file. Similarly, Sort file key values may not have been part of the Direct data record, but must be included in the Multi-Keyed file key.

Having a pictorial representation will make it easier to deal with some of the special considerations described below.

Field Separator Characters

It is important to remember the field separator characters. These will be present between fields of the Direct file record. Because the logical structure of the records within a Multi-Keyed file is described in the Format string, field separator characters are not required for fixed-length fields and are not included in the physical data records. In order to eliminate field separator characters when converting a Direct file to a Multi-Keyed file, a BASIC program should be written that READs records from the Direct file, using the standard IOLIST; and WRITES each record to the Multi-Keyed file, also using an IOLIST. An example of such a program is given above. The IOLIST used for output might be slightly different from the IOLIST used for input (see the next section for one example). The field separator characters in the input data record will be used to assign data to BASIC variables, but will not be included in the output data record.

Subfields

Subfields, for the purpose of this discussion, are sequences of characters that are defined as substrings of other fields. For example, if field 'PHONUM ' (phone number) is a 10-character string, it might consist of two subfields: PHONUM (1,3) (area code) and PHONUM (4,7) (local number). In order to use the subfields, the characters must be extracted by substring reference from the larger field. Subfields are allowed on either Direct or Multi-Keyed files. If subfields are present in the Direct file and they are to be eliminated in the Multi-Keyed file (so that each subfield becomes a field that can be referenced explicitly without a substring reference to another field), there are two possible approaches.

The first (and preferred) solution is to READ the input record into an IOLIST which specifies a variable for the larger field; then extract the subfields into other variables; finally to WRITE the subfield variables to the Multi-Keyed file as part of the IOLIST. To clarify this using the example above, first READ the record specifying a variable, PHONUMS to hold the entire 10-digit number. Then separate the parts into individual variables. Finally, write the record to the Multi-Keyed file specifying both AREA$ and PHONE$ in the IOLIST:

```
READ  (1) A, B,      PHONUMS, ...
AREAS  = PHONUMS (1,3)
PHONES = PHONUMS (4,7)
WRITE (2) A, B,      AREAS, PHONES,
```

The second approach would be to define a new field in the Multi-Keyed file format string which overlaid all of the subfields (which had been described as separate fields in the new record). For example:

```
MKFORM$ =
       AREA#   = S3
       PHONE#  = S7
       PHONUM# = AREA#: S10"
```

Here, the field PHONUM# overlays the previous two fields. To convert a record using this technique, the data would be READ the same way as in the first approach, but the subfields would not be extracted. Instead, the larger field would be written to the Multi-Keyed file specifying the redefined field name:

```
READ  (1) A, B,      PHONUMS,
WRITE (2) A, B,      PHONUM#=PH0NUM$, ...
```

After writing the record this way, the parts of the phone number may be referenced by name (AREA# and PHONE#) or the while field may be referenced my the name of the redifined filed (PHONUM#). The abiity to dreference a combination of fields by a single name can be desirable, but the addition of overlaid fields implies a more complicated record structure than really exists. Note that the use of a composite field instead of an overlay field wouldnot work because of the restriction that composite fields may not be included in the IOLIST for a WRITE. The use of overlay fields is not recommended.

The WriteThru File Attribute on BOSS/VS

To speed up the conversion of a Direct file into a Multi-Keyed file, it is important to set the WriteThru attribute for the Multi-Keyed file to False. This will have no effect if the system-wide WriteThru parameter is set to True, so it may be necessary to change the system-wide parameter also. Once the conversion is complete, the WriteThru attribute on the Multi-Keyed file should be reset to True to improve recoverability of the file when it is in use.

Definition of Keysets for Conversion

Another technique for speeding up the conversion of a Direct file is to set up the format string for the Multi-Keyed file to define all of the fields but only one keyset (the primary keyset is required). This will accelerate the WRITES to the new file. After all records have been converted (written to the new file) the other keysets should be added using the SETFIELD directive. This operation uses a more efficient technique to build the other keyset structures.

RECOVERY OF MULTI-KEYED FILES ON BOSS/VS

This section deals with issues that arise when a system failure causes a Multi-Keyed file to "lack integrity". Because the internal structure of Multi-Keyed files are more complicated than those of other file types, some of the integrity characteristics and recovery techniques are different and require special attention.

Concurrency and Integrity

In handling the critical resources of file operations, there is a tradeoff between concurrency (the ability to have multiple users access a resource simultaneously) and integrity. As concurrency is increased, "throughput" and general system performance are improved. At the same time there is an increase in the likelihood that, at any instant in time, the file is in an inconsistent state. This is especially true of Multi-Keyed files. When several users of a file are allowed to simultaneously modify the file (assuming they are operating on different records), their modifications to file structures will overlap. Each file contains an indicator, called the lack-of-integrity indicator, which represents the current state of the integrity of the file. When modifications to a Multi-Keyed file cause the file to be temporarily in an internally inconsistent state, the lack-of-integrity indicator is set to True. This indicator is used to prevent access to a file if a system failure or system load occurs while a file is in this state. Files in this state must be reconstructed before they can be accessed. Because of the interrelated nature of Multi-Keyed file structures, Multi-

Keyed files will have the lack-of-integrity indicator set to True a greater percentage of the time than is true for Direct files. Therefore, they will require reconstruction following a system failure a much greater percentage of the time than is true for Direct files.

The lack-of-integrity indicator for Multi-Keyed files is unaffected by whether or not WriteThru is enabled for the file. As mentioned previously, it is recommended that WriteThru be enabled for Multi-Keyed files to maximize the amount of data that will be recovered, but this will not reduce the need for reconstruction.

Tools Available

After a system crash or other problem that leads to a reloading of the system, the system administrator is concerned with providing access to the system data base as quickly as possible, while guaranteeing the integrity of that data base. The sequence of steps covered in the following section describes the quickest way to reestablish the integrity of the user data base. The major steps involve repairing or reconstructing files. There are two tools available to reconstruct files: Diskanalyzer and the Reconstruct utility (not to be confused with the Validation/Restruction option within Diskanalyzer). Both will reconstruct a keyed file that lacks integrity, but they each use different techniques and have different limitations and applications. Specifically, the Reconstruct utility operates much more quickly than does Diskanalyzer, but its limitations are that it may require more disk space than Diskanalyzer and it may not be able to recover all files that Diskanalyzer can recover.

File Recovery Sequence

The recommended sequence of operations for recovery of user files is:

1. On the system load immediately following a system failure, the system operator should select type 2 load. This provides a single-user environment in which to verify and correct certain problems. The Diskanalyzer option 5, "DISK SPACE USAGE", should then be selected and run on each family to verify that the system files are intact. If this operation reports any errors, the user must immediately take steps to correct the errors. The Diskanalyzer functional specification describes these steps, but to summarize the important points:

   a) Reconstruct the Directory of the family(ies) that received errors using the Diskanalyzer option "Reconstruct Directory File" (option 10).

b) Reconstruct the available space file for each member of the family(ies) affected using the option "Reconstruct Available Space Files" (option 9).

c) Select and run option 5 again to verify that the problems detected have been corrected.

2. Run Diskanalyzer option 1, "Find All Files That Lack Integrity". The list of files that this option generates will be used in subsequent steps. This procedure can be run while still under type 2 load.

3. At this point it will be necessary to reload the system under type 1 load in order to use the Reconstruct utility. If normal access to filrd must be avoided because there are files that lack integrity, steps must be taken to inform or prevent users from restarting their work. With the new 8.6 release, this can be accomplished with the job manager GROUP enable initial command. Suggestions on how to control access to the system through the startup procedures are contained in the System Startup Overview document.

4. Run the Reconstruct utility to reconstruct all keyed files that lack integrity - using the output of step 2 to select the files.

5. Use the Diskanalyzer Validate/Reconstruct option to reconstruct those files which either are not keyed files (e.g., Serial or Indexed files) or are damaged in such a way that the Reconstruct utility fails to recover them.

6. Finally, if after steps 1 through 5 have been performed and there are problems with starting up the application that seem bo be attributable to file corruption problems, the customer may be advised to take other, more time-consuming steps such as running the Diskanalyzer option "Validate All Files", validation of specific files or, depending on the circumstances, restoration of files from the most current backup.

7. When recovery of all files is complete, including any renaming that must be done following the use of the Diskanalyzer reconstruction option, access to the system by others can be allowed. If access to the system was prevented by not enabling the terminal groups (using the Job Manager) it can be restored by enabling them at this time.

As can be seen from the sequence above, the reconstruction of a Multi-Keyed file using the Diskanalyzer option is not recommended until after the Reconstruction utility has been tried and has failed. The validation of all files or the validation of a specific file is recommended only in the unusual circumstance that the other recovery mechanisms fail.

RECOVERING MULTI-KEYED FIILS ON BOSS/IX

This section describes the mechanism for analysis and reconstruction of Multi-Keyed files under the BOSS/IX operating system. The utility that performs this function is the File Analysis and Repair Utility (frepair). The EREPAIR utility can be used to determine if there is any corruption in a given file. If the file is corrupted the utility may also be used to repair the file. The goal of the repair is to retrieve as much data as possible from the corrupted file and to copy this data into a new file. The new file has the same name as the original file and

replaces the original file. The operation of this utility requires that the file system is in good working order. No attempt is made within FREPAIR to detect or repair problems in the file system. Detection and repair of file system problems is done automatically at system load time. If a file system problem is detected or suspected after system load, the FSCHK command may be performed to verify the integrity and to repair the file system if necessary.

Template File

The File Analysis and Repair Utility requires certain inFormation about the damaged file, such as file size, data record size, key descriptors and format string in order to create a new file with the same attributes as the file being repaired. This information is normally found in the file descriptor for the damaged file. (Note: For simplicity, the term file descriptor also refers to the file definition information which is stored in the associated "index" file. the "index" does not appear in the directory.) However, sometimes the file descriptor itself has been damaged. In this case, the vital information is conveyed to the utility in the form of a template file which must have the same characteristics as the damaged file, but does not need to contain the same data . The most convenient way to generate the template file is to restore a recent backup copy of the damaged file.

Disk Space Requirements

Repairing a file requires disk space at least equal to the size of the file being repaired. Repairing a Multi-Keyed file could require twice the disk space of the file being repaired if it is necessary to use a separate template file for the file attributes.

User Interface

All user input to the utility is completed before the analysis or analysis and repair operations begin. To repair a Multi-Keyed file, the user is prompted to enter a template file name. The default tei rplate file name is the name of the file being repaired, but when the file descriptor of the file being repaired is damaged, the user should specify a separate template file, which can be the name of a backup copy of the damaged file. Normally, the default file name should be used. However, if this results in errors which are related to the file descriptor, a copy of the damaged file should be used as a template. The types of errors which would require a separate template file include:

1. The file cannot be opened and locked

2. The record size is out of range (1 < Record size < 32756)

3. The key descriptor contains values that are out of range

4. Invalid information in the Format string.

If the user specifies a template file which was created with different attributes than the file being repaired (such as with different keys, or data record lengths or different format string, etc .), or if corruption in the file descriptor area of the template file cannot be detected, the repaired file will be built on the basis of a false description of the file. This will result in the repaired version of the file being corrupted itself.

The analysis option will only analyze the given file(s)

and send the report to the specified output device. The analysis and repair option will first analyze the file and then try to repair the file. The report for both the analysis and repair will be sent to the specified output device. If an error occurs, a message describing the nature of the error will be displayed on the screen.

Single User Mode

When repairing a file, it is highly recommended that the operation be performed in single user mode. This will prevent the possibility of a bad block being reallocated (to some other file for another user) by the file system during a repair. Bad blocks found during a repair operation will be added to the bad blocks list when the repair is complete.

Operating in single user mode should also eliminate the possibility of running out of memory for temporary working space during the operation.

REPAIRING A MULTI-KEYED FILE

The objective of repairing a file is to recover as much of the data in the file as possible, and to allow the user to access the file without getting any errors. There is a possibility of data loss, so the user should attempt to back up the file first.

If the user selects the repair mode, the file will be repaired even if it is not corrupted.

When a Multi-Keyed file is being repaired, the utility first performs an analysis of the file structure. The first step of the analysis is to read the entire file one block at a time to detect any bad blocks. Any bad blocks encountered are reported to the user and the block number is saved so that the block can be added to the bad blocks list at the end of the repair. Following the block read validation, the file is checked for errors such as discrepancies in the B-Trees that implement the various keysets, or errors in reading each data record.

Following the analysis phase, the data is extracted from the file and a new file is created with the records read. To do this, the damaged file is read again, one record at a time. Contained within each data record are the keys associated with that record, which will be used to build the keysets within the new file. This process continues until all records from the damaged file have been processed. When the repair is complete, the corrupted file will be deleted. All bad blocks discovered earlier will be added to the bad blocks list.

Following the completion of the repair operation, the newly repaired file may be referenced under its previous name. The only data loss which might occur would be the result of the inability to read certain data records out of the corrupted file.

NOTES

APPENDIX C - VARIABLE TABLES FOR BOSS/IX

There are eight variable tables used in the BOSS/IX implementation of Business BASIC. The following description of the tables is intended to help in the use of the CPL and LST functions.

The tables must start on an even boundary, so if the hash field ends on an odd boundary, a 1-byte hold exists between the end of the hash field and the first variable table. Each variable table begins with a 4-byte field which gives the length of that table. The length includes the 4 bytes of the length field.

The variable tables give information about the variables in the current BASIC environment. The tables are:

1. Numeric id table: contains the names of numeric variables as entered by the user. There are no delimiters or length fields (except the length field of the entire table). The variable names are stored in ASCII form, one right after another.

2. Numeric sort table: lexicographically sorts the numeric variables. This information is useful because when a BASIC program is LOADed or RUN, the variables in the incoming program must be merged with the variables in the current environment. Sorting greatly speeds up LOADs and RUNs.

3. Numeric offset table: contains two-byte fields which are used as offsets into the numeric id table to determine where one variable id ends and the next begins.

4. String id table: similar to the numeric id table, used for strings and numeric arrays.

5. String sort table: similar to the numeric sort table,

   used for strings and numeric arrays.

6. String offset table: similar to the numeric offset table, used for strings and numeric arrays.

7. Numeric location table: gives location of the data of a numeric BASIC variable.

8. String location table: similar to numeric location table, used for strings and numeric arrays.

NOTES

CHARACTER CODES

Character representations differ slightly on BOSS/IX and BOSS/VS systems. On BOSS/IX systems, characters are represented by low-order, 7-bit ASCII codes, with the eighth bit turned off (b8=0) On BOSS/VS systems, characters are represented by high-order, 8-bit ASCII codes, with the eighth bit turned on (b8=1).

Tables D-1 and D-2 show the character codes for BOSS/IX and BOSS/VS systems, respectively.

BINARY LSB →		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
MSB ↓	HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	NUL <0>	SOH <1>	STX <2>	ETX <3>	EOT <4>	ENQ <5>	ACK <6>	BEL <7>	BS <8>	HT <9>	LF <10>	VT <11>	FF <12>	CR <13>	SO <14>	SI <15>
0001	1	DLE <16>	DC1 <17>	DC2 <18>	DC3 <19>	DC4 <20>	NAK <21>	SYN <22>	ETB <23>	CAN <24>	EM <25>	SUB <26>	ESC <27>	FS <28>	GS <29>	RS <30>	US <31>
0010	2	SP <32>	! <33>	" <34>	# <35>	$ <36>	% <37>	& <38>	' <39>	(<40>	) <41>	* <42>	+ <43>	, <44>	- <45>	. <46>	/ <47>
0011	3	0 <48>	1 <49>	2 <50>	3 <51>	4 <52>	5 <53>	6 <54>	7 <55>	8 <56>	9 <57>	: <58>	; <59>	< <60>	= <61>	> <62>	? <63>
0100	4	@ <64>	A <65>	B <66>	C <67>	D <68>	E <69>	F <70>	G <71>	H <72>	I <73>	J <74>	K <75>	L <76>	M <77>	N <78>	O <79>
0101	5	P <80>	Q <81>	R <82>	S <83>	T <84>	U <85>	V <86>	W <87>	X <88>	Y <89>	Z <90>	[<91>	\ <92>	] <93>	^ <94>	_ <95>
0110	6	` <96>	a <97>	b <98>	c <99>	d <100>	e <101>	f <102>	g <103>	h <104>	i <105>	j <106>	k <107>	l <108>	m <109>	n <110>	o <111>
0111	7	p <112>	q <113>	r <114>	s <115>	t <116>	u <117>	v <118>	w <119>	x <120>	y <121>	z <122>	{ <123>	| <124>	} <125>	~ <126>	DEL <127>

Table D-1. BOSS/IX Low-order ASCII Character Codes

ASCII Code Set																	
B_4-B_1 → / B_8-B_5 ↓		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
		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]

Table D-2. BOSS/VS High-order ASCII Character Codes

EXPLANATION OF CODES The early codes are unprintable characters. The representations shown in the charts have the following standard meanings.

NUL	NULL
SOH	Start of Heading
STX	Start of Text
ETX	End of Text
EOT	End of Transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	BELL
BS	Back Space
HT	Horizontal Tab
LF	Line Feed
VT	Vertical Tab
FF	Form Feed
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape

DC3	Device Control 1
DC2	Device Control 2
DC1	Device Control 3
DC4	Device Control 4
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

NOTES

The following list includes all the Business BASIC 86 KEYWORDS.

ABS
ADD
ADDE
ADDR
ALIAS
ALL
AND
ASC
ASCII
ATH
ATN
ATTR
BEGIN
BIN
BLK
BNK
CALL
CHAR
CHR
CLASS
CLEAR
CLOSE
CONSOLE
CONSOLELOCK
COPIES
COS
CPL
CRC
CREATE
CSW
CTL
DAY
DEC
DEF
DELETE
DEVINFO
DIM
DIRECT
DOM
DROP
DSZ
EDIT
ELSE
ENCRYPT
END
ENDIF
ENDTRACE
ENDTRANS
ENTER
EPT
ERASE
ERR
ERROR
ESCAPE
EXCEPT
EXECUTE
EXIT
EXITTO
EXP
EXTEND
EXTRACT
EXTRACTRECORD
FI
FID
FIELD
FIND
FINDRECORD
FLD
FLOATING
FLOATINGPOINT
FMT
FMTINFO
FN
FOR
FPT
GAP
GO
GOSUB
GOTO
HELP
HSA
HSH
HTA
IF
IND
INDEXED
INIT
INITFILE
INPUT
INPUTRECORD
INT
IOL
IOLIST
IOR
ISZ
KEY
LEN
LET
LIB
LIST
LISTPROGRAM
LOAD
LOCK
LOG
LRC
LST
LVL
MAKE
MAKEPROGRAM
MAX
MERGE
MIN
MOD
MSG
MULTI
NEXT
NO
NOEXTEND
NOT
NUM
ON
OPEN
OPTS
PACK
PFX
PGM
PNM
POINT
POS
PRC
PREFIX
PRECISION
PRINT
PRINTRECORD
PRIORITY
PROGRAM
PSAVE
PSZ
PUB
QUIT
RANDOMIZE
READ
READRECORD
RECORD
RELEASE
REM
REMOVE
RENAME
RESET
RETAIN
RETRY
RETURN
RND
RUN
SAVE
SEO
SEQUENCE
SERIAL
SET
SETCTL
SETDAY
SETERR
SETESC
SETFIELD
SETTIME
SETTRACE
SETTRANS
SGN
SIN
SIZ
SORT
SPX
SOR
SSN
SSZ
START
STEP
STOP
STR
STRING
SYNTAX
SYS
SYSTEM
TABLE
TBL
TCB
THEN
TIM
TIME
TRACE
TRANS
TRX
TSK
UNLOCK
UNPACK
UNT
VMERGE
WAIT
WHO
WRITE
WRITERECORD
XOR

NOTES

OVERVIEW

This appendix provides a summary of the Business BASIC features. It includes the availability of the feature in each level of Business BASIC, the statement type, and a brief description of the feature. The key used is:

-	=	not present
*	=	differences exist
+	=	has more features
?	=	system specific
ig	=	ignored/minimal support
IX	=	available in pre-BB86 versions of BOSS/IX
VS	=	available in pre-BB86 versions of BOSS/VS
BB86	=	available in BB86

BUSINESS BASIC FEATURE SUMMARY

Feature	Availability			Type	Description
!	IX	VS	BB86 ?	statement	Perform system command
'	-	VS	-	statement	Abbreviation for EDIT
/	-	VS	-	statement	Abbreviation for LIST
?	-	VS	-	statement	Abbreviation for PRINT
ABS	IX	VS	BB86	function	Return absolute value
ADD	IX	VS ig	-	directive	Cache a program
ADDE	IX	-	-	directive	Set up error handling program
ADDR	IX	VS ig	-	directive	Cache a program & make it resident
ALL	IX	VS	BB86	special	used to specify an entire array
AND	IX	VS	BB86	function	Combine the bits of two strings
AND	IX	VS	BB86	operator	Condition within IF statements
ASC	IX	VS	BB86	function	Convert character to number
ASCII	-	-	BB86	function	Maps character to ASCII code
ATH	IX	VS	BB86	function	Convert "hex" to hex
ATN	-	VS	-	function	Arctangent function
ATTR	-	-	BB86	function	Return file information
ATTR=			BB86	clause	Used within CREATE for file info
ATTR=	-	VS new	-	clause	Used within OPEN for spool info
BEGIN	IX	VS	BB86	directive	Reset system
BIN	IX	VS	BB86	function	Return binary string value
BLK=	-	VS ig		clause	Specify block size on OPEN

Feature	Availability			Type	Description
BNK=	-	VS ig	-	clause	Set memory bank
CALL	IX	VS	BB86	directive	Transfer control to another pgm
CHAR	-	-	BB86	function	Convert number to BOSS/IX,/VS character
CHR	IX	VS	BB86	function	Number to character
CLASS=	IX	VS		clause	Set spooler class on OPEN
CLEAR	IX	VS	BB86	directive	Clear a program's variables
CLOSE CONSOLE	IX	VS	BB86	directive	Release file or device
LOCK	-	-	BB86	directive	Inhibit Console mode
COPIES=	-	VS	-	clause	Set copies to spool on OPEN
COS				function	Cosine function
CPL	IX	-	-	function	Return compiled form of source
CRC	IX	VS	BB86	function	Return cyclic redundancy check
CREATE	-	-	BB86	directive	Create a new file
CSW	-	VS	BB86	variable	Return called program flag
CTL	IX	VS	BB86	variable	Return last field te rminator
DAY	IX	VS	BB86	variable	Return system date
DEC	IX	VS	BB86	function	Convert binary string to number
DEF FNx	IX	VS	BB86	directive	Define numeric/string function
DELETE	IX	VS	BB86	directive	Remove statement(s) from program
DEVINFO	-	-	BB86	variable	Return device information
DIM	IX	VS	BB86	directive	Dimension an array or
DIRECT	IX +	VS	BB86	directive	Create a single-keyed file
DOM=	IX	VS	BB86	I/O option	Transfer if dup or missing key
DROP	IX	VS ig	-	directive	Remove program from cache
DSZ	IX	VS ig	-	variable	Return size of data area left
EDIT	IX +	VS +	BB86	directive	Change a program state - ment
ENCRYPT	IX	-	BB86	directive	Encrypt a BASIC program
END	IX	VS	BB86	directive	Terminate a program
END=	IX	VS	BB86	I/O option	Transfer if end of file

Feature	Availability			Type	Description
END=	-	-	BB86	clause	Allowed within an IND function
ENDIF	-	-	BB86	directive	Replaces FI (end of IF)
END TRACE	IX	VS	BB86	directive	Terminate SETTRACE listing
END TRANS	-	-	BB86	directive	Terminate translation
ENTER	IX	VS	BB86	directive	Receive arguments from CALL
EPT	IX	VS	BB86	function	Return power of ten of argument
ERASE	IX	VS	BB86	directive	Delete disk file
ERR	IX	VS	BB86	variable	Return number of last error
ERR	IX	VS	BB86	function	Return position of error in list
ERR=	IX	VS	BB86	I/O option	Transfer on error
ERROR	IX *	VS *		directive	List last error
ESCAPE	IX	VS	BB86		Interrupt program
EXECUTE	IX	VS +	BB86	directive	Generate/modify state-ments
EXIT	IX	VS	BB86	directive	Terminate a CALLed program
EXITTO	IX	VS	BB86	directive	Break out of POR/GOSUB
EXP		VS		function	Returns the exponential value requested
EXTEND		VS		directive	Begin extended mode
EXTRACT	IX	VS	BB86 +	directive	Read and lock
FI	IX	VS		directive	Show end of most recent IF statement
FID	IX *	VS *		function	Return file information
FIELD ALIAS			BB86	M/K directive	Reassign meaning of field variable
FILE	IX	VS		directive	Uses string to define file type
FIND	IX	VS		directive	
			BB86 +		Read, don't advance if missing
FLOATING POINT	IX	VS	BB86	directive	Initate floating point mode
FMT=			BB86	M/K option	Specify format string for file
FMTINFO			BB86	M/K function	Return open file's format string
FNx	IX	VS	BB86	function	User defined numeric function
FNx$	IX	VS	BB86	function	User defined string function
FOR/NEXT	IX	VS	BB86	directive	Loop control

Feature	Availability			Type	Description
FPT	IX	VS	BB86	function	Return fractional part of expression
GAP	IX	VS	BB86	function	Generate odd-parity string
GOSUB	IX	VS	BB86	directive	Transfer into internal routine
GOTO	IX	VS	BB86	directive	Transfer to another statement
HELP	-	VS	-	directive	Give on-line help
HSA	IX	VS ig	-	function	Return highest sector available
HSH	IX	VS	BB86	function	Return hash of argument
HTA	IX	VS	BB86	function	Convert hex to "hex"
IF/THEN/ ELSE IF/...	IX	VS	BB86	directive	Conditional control
/ENDIF	IX	VS	BB86 +	directive	Conditional control
IND	IX	VS	BB86 +	function	Return position in file
IND=	IX	VS *	BB86	clause	Specify position in file
INDEXED	IX +	VS	BB86	directive	Create a relative file
INITFILE	-	VS	BB86	directive	Initialize existing file
INPUT	IX	VS	BB86 +	directive	Read
INT	IX	VS	BB86	function	Return whole part of argument
IOL=	IX	VS	BB86	I/O option	Use an IOLIST
IOLIST	IX	VS	BB86 +	directive	Define list of variable/values
IOR	IX	VS	BB86	function	Combine bits of two strings
ISZ=	IX	-	-	clause	Set I/O size
KEY	IX	VS	BB86 +	function	Return key of next . record
KEY=	IX	VS	BB86 +	clause	Specify key on I/O operation
LEN	IX	VS	BB86	function	Return length of string
LEN=	IX	VS	BB86	clause	Specify length range on input
LET	IX	VS	BB86	directive	Assign value to a variable
LIB	IX	-	-	directive	Include 'C library functions
LIST	IX *	VS	BB86	directive	Print statement(s)
LIST PROGRAM	-	-	BB86	directive	Convert program to a serial file
LOAD	IX	VS	BB86	directive	Bring a program into memory
LOCK	IX	VS	BB86	directive	Protect a file from others

Feature	Availability			Type	Description
LOG		VS	-	function	Returns the logarithm of the number specified
LRC	IX	VS	BB86	function	Return longitudinal redundancy
LST	IX	-	-	function	Convert compiled statement into list format
LVL	IX	-	-	variable	Return system level
MAKE PROGRAM	-	-	BB86	directive	Convert serial file to program file
MAX	-	VS	-	function	Maximum argument value
MERGE	IX	VS	BB86	directive	Add serial file to
MIN	-	VS	-	function	Minimum argument value
MOD	IX	VS	BB86	function	Return remainder of division
MSG=	-	-	BB86	clause	Specify messages
MULTI	-	-	BB86	M/K directive	Create a multi-keyed file
NEXT	IX	VS	BB86	directive	Used with FOR for looping
NO EXTEND	-	VS	-	directive	Begin no extend mode
NOT	IX	VS	BB86	function	Return inverse of a string
NUM	IX	VS	BB86	function	Convert ASCII to number
ON..GOSUB	IX	VS	BB86	directive	Conditional transfer to subroutine
ON..GOTO	IX	VS	BB86	directive	Conditional transfer to statement
OPEN	IX	VS	BB86	directive	Access a file or device
OPTS=	IX	-	-	clause	Set spooler options on OPEN
PACK	-	-	BB86	M/K directive	Pack variables into buffer
PFX	IX	VS		variable function	Return user's prefix list
PGM	IX	-	-		Return compiled code
PGM(O)	IX	-	-	variable	Return name of pgm in memory
PNM	IX	VS	BB86 ?	variable	return name of pgm in memory
POS	IX	VS	BB86	function	Returns string position
PRC	IX	VS	BB86	variable	Returns current precision
PRECISION	IX	VS	BB86	directive	Set number of digits rounded
PREFIX	IX		_	directive	Set user's prefix list
PRINT	IX	VS	BB86 +	directive	Print to file or device
PRIORITY^	-	VS		clause	Set priority of spooler on OPEN

Feature	Availability			Type	Description
PROGRAM	IX	VS	-	directive	Create a BASIC file
PSAVE	-	VS	BB86	directive	Protected SAVE
PSZ	IX	VS	BB86	variable	Return program size
PUB	IX	-	-	function	Return public programs
QUIT	-	VS	BB86	directive	Close files and exit BASIC
RANDOMIZE	-	VS	-	directive	Starting value for a pseudo-random number
READ	IX	VS	BB86 +	directive	Read data
RELEASE	IX *	VS	BB86	directive	Close files and log off
REM	IX	VS	BB86	directive	Remark
REMOVE	IX	VS	BB86 +	directive	Delete a record from keyed file
RENAME	IX	-	BB86	directive	Rename a file
RESET	IX	VS	BB86	directive	Reset some task variables
RETAIN	-	-	BB86	M/K I/O Option	Use "raw" I/O buffer
RETRY	IX	VS	BB86	directive	Transfer back to last error statement
RETURN	IX	VS	BB86	directive	Return after GOSUB
RND	-	VS	-	function	Returns a pseudo-random number
RUN	IX	VS	BB86	directive	Execute/continue a program
SAVE	IX	VS	BB86	directive	Save pgm in memory to a file
SEQ=	IX	VS	BB86	clause	Positions tape on an OPEN
SEQUENCE	-	VS	-	directive	Begin auto statement numbering
SERIAL	IX +	VS	BB86	directive	Create a sequential file
SETCTL	IX	VS	BB86	directive	Branch when ctrl-Y
SETDAY	IX	VS	BB86	directive	Set system date
SETERR	IX	VS	BB86	directive	Branch when error encountered
SETESC	IX	VS	BB86	directive	Branch when escape encountered
SETFIELD	-	-	BB86	M/K directive	Add/Remove keysets on MULTI file
SETTIME	IX	VS	BB86	directive	Set system time
SETTRACE	IX *	VS	BB86	directive	List statements as they execute
SETTRANS	-	-	BB86	directive	Begin translation
SGN	IX	VS	BB86	function	Return sign of value
SIN		vs		function	Sine function
SIZ=	IX	VS	BB86	I/O option	Set maximum size of input
SORT	IX +	vs	BB86	directive	Creates a "sort" keyed file

Feature	Availability			Type	Description
SPX	-	VS	-	variable	Return system prefix list
SQR	-	VS	-	function	Returns the square root of the number specified
SSN	IX	VS	BB86 ?	variable	Return system serial number
ssz	-	VS ig	-	variable	Return sector size
START	IX	VS +	BB86	directive	Reset system/start tasks
STOP	IX	VS	BB86	directive	Terminate program (like END)
STR	IX	VS	BB86	function	Convert number to ASCII string
STRING	IX	-	-	directive	Create a UNIX "stream" type file
SYNTAX	-	-	BB86	directive	Validate syntax of expression
SYS	IX *	VS *	BB86		Return O/S level
SYSTEM	IX	VS	BB86 ?	directive	Execute system command
TABLE	IX	VS	BB86		Define table translation
TBL	IX	VS	BB86	function	Return table translation
TBL=	IX	VS	BB86	I/O option	Perform table translation on I/O
TCB	IX	VS	BB86 ?	variable	Return task information
TIM	IX	VS	BB86	variable	Return system time
TIM=	IX	VS	BB86	I/O option	Set timeout value for I/O
TRANS	-	-	BB86	function	Return translated string
TRX	-	-	BB8?	variable	Return current translation file name
TSK	IX	-		function	Return device information
UNLOCK	IX	VS	BB86	directive	Free a locked file
UNPACK	-		BB86	M/K directive	Unpack from buffer to variables
UNT	IX	VS	BB86	variable	Return lowest available lun
VMERGE	IX	-	-	directive	Merge in a "STRING" type file
WAIT	IX	VS	BB86	directive	Wait for specified number of seconds
WHO	-	VS	BB86 ?	variable	Return logon account's name
WRITE	IX	VS	BB86 +	directive	Output to a file or device
XOR	IX	VS	BB86	function	Exclusive OR of two string
""""	-	VS	BB86		Specifies a string with one "
functions		vs			Optional $ at end of string function

NOTES

SYSTEM VARIABLES

Format	OPERATION
CSW	Return CALLed program flag
CTL	Field terminator last used
DAY	Current system date
DEVINFO	Return device information
ERR(code-1,...,code-n)	Error that occurred last
PNM	Program currently in memory
PRC	Current precision
PSZ	Return program size
SSN	System Serial Number
SYS	BASIC release and version levels
TCB(numeric expr 0-14)	Task information
TIM	Current system time
TRX	Current translation file name
UNT	Lowest unused logical unit number
WHO	Return logon account's name

Additional Variables For BOSS/IX Systems:

DSZ	Return size of data area left
ISZ=recsz	Set redefined record size
PFX	Return user's prefix list

Additional Variables For BOSS/VS Systems:

DSZ	Always returns 32,767
PFX	Return user's prefix list
SPX	Return system prefix list
SSZ (disk number)	Returns 1024 bytes per sector

INPOT/OUTPUT OPTIONS

Format	OPERATION
DOM=stno	Branch if duplicate or missing key
END=stno	Branch at end of file
ERR=stno	Branch on error
IND=numeric expr	Specify index of record to be accessed
IOL=stno	Specify stno of the IOLIST to be used
KEY=field.var# expr	Specify key of record to be accessed
LEN=minimum, maximum	Specify length range of variable
RETAIN	Use raw I/O buffer
SEQ=numeric expr	Positions tape on an OPEN
SIZ=numeric expr	Set maximum characters to be input
TBL=stno	Specify TABLE statement number to be used
TIM=numeric expr	Specify number of seconds allowed for input

Additional I/O Options For BOSS/IX Systems:

CLASS= "str-expr"	Set spooler class on OPEN
OPTS= "str-expr"	Set spooler options on OPEN

Additional I/O Options For BOSS/VS Systems:

ATTR="str-expr"	Use with OPEN to specify Spooler (print job) attributes
CLASS="str-expr"	Set spooler class on OPEN
OOPIES=int-exr	Use with OPEN to specify number of copies to be printed
PRIORITY=int-expr	Set priority of Spooler on OPEN

OPERATORS

-	minus
+	plus
*	multiplied by
/	divided by
^	raised to the power of (exponentiation)
AND	conjunction
OR	disjunction
=	is equal to
>	is greater than
<	is less than
<>	is not equal to
>= or =>	is greater than or equal to
<= or =<	is less than or equal to

BASIC COMMAND LINE ARGUMENTS

For BOSS/IX Systems:

```
username> basic {-h} {-nr} {pgm=} {s=} {-e} {-q } {-x} {lib=} {trans=}
           {'command string'}
```

For BOSS/VS Systems:

EXTEND Mode	NO EXTEND Mode
!BASIC {pgm} {RUN=pgm} {OOMMAND='string'}	!BASIC
!BASIC	>N0 EXTEND
>	]

Format	**OPERATION**
ABS (numeric expression)	Absolute value
AND ("str-expr", "str-expr")	Combine Strings
ASC ("str-expr" {,ERR=stno})	String to ASCII decimal value
ASCII("str-expr" i,ERR=stno}	Returns ASCII numeric code
ATH ("str-expr" {,ERR=stno})	"hex" to hex
ATTR (fileno ,"{ALL} { NAME} { OWNER} { USAGE RIGHTS} { ORGANIZATION} { RECORD_SIZE} { RECORDS_ALD0WED} { RECORDS USED} { KEY SIZE} { INITIAL} { GROWTH} { LONG} { SHORT} { WRITE THRU}" {,ERR=stno})	Return file information
BIN (num-expr, int-expr)	Binary string value
CHAR (num-expr {,ERR=stno})	Convert ASCII numeric code to character
CHR (num-expr {,ERR=stno}	Convert num-expr to ASCII
CRC ("str-expr" {,2-byte string}	Return cyclic redundancy check
DEC ("str-expr" {,ERR=stno})	Binary to signed decimal
EPT (num-expr)	Return exponent of num-expr
FMTINFO (fileno {,field-selector {,info-selector}})	Return multi-keyed file format information
FNx {$} (arg-list)	User defined functions
FPT (num-expr)	Fractional part of num-expr
GAP ("str-expr")	Generate odd-parity string
HSH ("str-expr" {,2-byte string})	HASH; data integrity check
HTA ("str-expr")	Convert hex to "hex"
IND (fileno {,END=stno} {,ERR=stno})	Index of current record position
INT (num-expr)	Integer part of num-expr
ICR ("str-expr", "str-expr")	Combine bits of two strings
KEY (fileno {,ERR=stno} {,END=stno} {,IND=recno})	Key of current record position
LEN ("str-expr")	Length of "str-expr"
LRC ("str-expr")	Longitudinal redundancy check information
MOD (num-expr-a, num-expr-b)	Return remainder of division
NOT ("str-expr")	Inverse of string, bit-by-bit
NUM ("str-expr" {,ERR=stno})	Convert "str-expr" to numeric value
POS (scan-str relational-op target-str {,step })	Return string position
SGN (num-expr)	Sign of num-expr
STR (num-expr {:mask})	Convert num-expr to string
TBL ("str-expr",stno)	Translates string expression
TRANS ("str-expr")	Return translated string
XOR ("str-expr", "str-expr")	Exclusive OR of two strings

FUNCTIONS (cont'd)

Additional Functions For BOSS/IX Systems:

CPL ("str-expr" {,"str-expr"} {,str-var} {,ERR=stno})	Return compiled form of source
FID (fileno)	Return file information
LST ("str-expr" {,'str-expr"} {,ERR=stno})	Convert compiled statement into list format
LVL (num-expr)	Return system release level
PGM (stno)	Return compiled format of stno; pgm name if stno=0.
PUB (int-expr)	return public programs
TSK (int-expr)	Return device information

Additional Functions For BOSS/VS Systems:

ATN(num-expr)	Arctangent function
COS(num-expr)	Cosine function
EXP(num-expr)	Return exponential value
FID (fileno)	Return file information
LOG(num-expr)	Return logarithm of num-expr
MAX (num-exprl, num-expr2 {,...,num-expm })	Maximum argument value
MIN (num-exprl, num-expr2 {,...,num-expm })	Minimum argument value
RND {(num-expr)}	Return a pseudo-random number
SIN(num-expr)	Sine function
SQR(num-expr)	Return square root of num-expr

ERRORS

ERROR NUMBER	MESSAGE
00	FILE/RECORD/DEVICE BUSY OR INACCESSIBLE
01	END OF RECORD
02	END OF FILE
03	DISK READ ERROR
04	DISK NOT READY
05	PERIPHERAL DATA TRANSFER ERROR
06	INVALID DISK DIRECTORY
07	CORRUPTED FILE
09	POWER FAILURE
10	ILLEGAL FILE NAME SIZE OR USAGE/ILLEGAL OVERLAID CALL
11	MISSING OR DUPLICATE KEY
12	MISSING OR DUPLICATE FILE NAME/NON-CONFIGURED DEVICE
13	IMPROPER FILE OR DEVICE ACCESS
14	IMPROPER FILE OR DEVICE USAGE
15	DISK SPACE ALLOCATION ERRORS
16	DISK OR PUBLIC PROGRAMMING DIRECTORY IS FULL
17	INVALID PARAMETER/NON-CONFIGURED DISK
18	ILLEGAL CONTROL OPERATION

ERROR NUMBER	MESSAGE
19	INVALID PROGRAM SIZE
20	STATEMENT SYNTAX
21	INVALID STATEMENT NUMBER
23	MISSING VARIABLE/NON-DIMENSIONED STRING
24	DUPLICATE FUNCTION NAME
25	UNDEFINED FUNCTION
26	INCORRECT VARIABLE USAGE
27	RETURN WITHOUT GOSUB/DELETE WITH ACTIVE GOSUB OR FOR-NEXT
28	NEXT WITHOUT FOR
29	INVALID MNEMONIC
30	USER PROGRAM INCORRECT CHECKSUM
31	INSUFFICIENT MEMORY WITHIN TASK
32	STACK OVERFLOW
33	INSUFFICIENT MEMORY CAPACITY
34	VDT BUFFER OVERFLOW
35	COMPILER OUT OF MEMORY
36	CALL/ENTER VARIABLE MISMATCH
38	ILLEGAL COMMAND IN A PUBLIC PROGRAM
39	ESCAPE IN A PUBLIC PROGRAM
40	NUMERIC VALUE OVERFLOW
41	INVALID INTEGER RANGE
42	NON-EXISTENT NUMERIC SUBSCRIPT
43	INVALID FORMAT MASK SIZE
44	STEP SIZE OF ZERO
45	INVALID STATEMENT USAGE
46	INVALID STRING SIZE
47	SUBSTRING REFERENCE OUT OF RANGE
48	INVALID INPUT
49	NON-TRANSLATABLE STATEMENT
50	GENERAL MEMORY ERROR
54	OPEN OF A SERIAL FILE WITH INVALID HEADER/PROGRAM OR FILE HAS INVALID FORMAT
59	NON-BASIC ERROR OCCURRED
60	FEATURE NOT YET IMPLEMENTED
61	RESTRICTED OPERATION ON A RESTRICTED-ACCESS PROGRAM
62	INTERNAL SYSTEM LIMIT EXCEEDED
63	SOURCE BUFFER/SYMBOL TABLE OVERFLOW
64	FILE LACKS INTEGRITY
66	FILE SYSTEM HAS NO MORE CACHE RECORDS
68	BAD SECOND ARGUMENT TO CPL OR LST
69	MISSING VARIABLE ID
70	THIRD ARGUMENT TO CPL FUNCTION IS NOT AN ACTIVE VARIABLE
71	THIRD ARGUMENT TO CPL FUNCTION IS NOT LONG ENOUGH
90	INVALID JUMP INTO PROGRAM CODE
91	CONSOLE-MODE FOR WITHOUT NEXT
95	LAN ERROR
98	SPOOLER ERROR
99	COMM ERROR
100	BASIC COMPILER INTERNAL ABNORMALITY
103	CATASTROPHIC READ FAILURE/FILE POINTERS DAMAGED

ERRORS (cont'd)

ERROR NUMBER	MESSAGE
104	CATASTROPHIC DISK FAILURE/FILE POINTERS DAMAGED
123	CATASTROPHIC PARITY ERROR/FILE POINTERS DAMAGED
124	PARITY ERROR
126	CTL+Y KEY USED
127	ESCAPE
254	PROGRAM SAVE ERROR
255	UNKNOWN ERROR

MNEMONICS

MNEMONIC	FUNCTION
@(x)	Horizontal position
@(x,y)	Horizontal & vertical position
'10'	10 pitch
'16'	16 pitch
'6L'	6 lines per inch
'8L'	8 lines per inch
'B1'	Bin 1 (BOSS/IX only)
'B2'	Bin 2 (BOSS/IX only)
'BB'	Begin blinking
'BE'	Begin echo
'bg'	Begin to generate error 29
'BI'	Beqin input transparency
'BO'	Begin output transparency
'BR'	Set reverse video
'BS'	Backspace
'BT'	Begin Input Buffering
'BU'	Begin underline
'CE'	Clear screen to end of page
'CF'	Clear foreground
'CH'	Move cursor home (0,0)
'CI'	Clear input buffer
'CL'	Clear line
'CR'	Carriage return
'CS'	Clear screen
'DC'	Delete character
'dn'	Cursor down (BOSS/IX only)
'DACS'	Disable alternate character set
'DBLW '	Double width print
'DBLH'	Double height print
'DPM'	Reset to default character printing mode
'EB'	End blinking
'EE'	End echo
'EG'	End generation of error 29
'ET'	End input transparency
'EL'	End Load (VFU)
'ED'	End output transparency
'EP'	Expanded print
'EPM'	Even dot plot mode

MNEMONIC	FUNCTION
'ER'	End reverse video
'ES'	Escape
'ET'	End input buffering
'EU'	End underline
'FF'	Form feed
'IC	Insert character
'KL'	Keyboard lock
'KU'	Keyboard unlock
'LD '	Line delete
'LF'	Line feed
'LI'	Line insert
'LT'	Cursor left (BOSS/IX only)
'NL'	New line
'OP'	Overprint
'OUT(n)'	Output (n) characters without
'PE'	End protect mode
'PG'	Print screen
'PM'	
'PS'	Begin plot mode Start protect mode
'RB'	Ring bell
'RC	Read cursor position
'RT'	Cursor right
'S2'	Slew to channel 2
'S3'	Slew to channel 3
'S4'	Slew to channel 4
'S5'	Slew to channel 5
'S6'	Slew to channel 6
'S7'	Slew to channel 7
'S8'	Slew to channel 8
'SACS '	Start alternate character set
'SB'	Start background mode
'SET6'	6 LPI
'SET8'	8 LPI
'SF'	Start foreground mode
'SL'	Start laod (VFU)
'SN'	Screen narrow
'SP'	
'SPM1'	. Superscript
'SPM2'	Set print mode 1 Set print mode 2
'SPM3'	Set print mode 3
'SPM4'	Set print mode 4
'SPM5'	Set print mode 5
'SS'	Subscript
'sw'	Screen wide
'TL'	Transmit line
'TP'	Transmit line protected
'TR'	Transmit terminal screen
'TS'	Transmit screen protected
'UP'	Cursor up (BOSS/IX only)
'VT'	Vertical tab
'WPM'	Letter quality emulation mode

```
BEGIN  { {EXCEPT} var-list}

CALL "prog ID" {,ERR=stno} {,arg-list}
CLEAR { {EXCEPT} var-list}
CLOSE (fileno {,ERR=stno} {,IND=num-expr})
CONSOLE LOCK {"str-expr" {,MSG="str-expr"}}
CREATE ATTR= "str-expr" {,EM T="str-expr"} {,ERR=stno}

DEF FNx (var-list)=arithmetic-expr
DEF FNx$ (var-list)="str-expr"
DELETE {first stno} {,} {last stno}
DIM array-name (rangel {,range2 {,range3}})
DIM string-name (int-expr {,"str-expr"})
DIRECT "file-id", keysz, recno, recsz {,ERR=stno}

ENCRYPT "source prog-id", "destination prog-id" {,ERR=stno}
END
ENDTRACE
ENDTRANS
ENTER arg-list
ERASE "file-ID" {,ERR=stno}
ESCAPE
EXECUTE "str-expr"
EXIT {int-expr}
EXITTO stno
EXTRACT (fileno {,RETAIN} {,ERR=stno} {,END=stno} {,DOM=stno} {,IND=int-expr}
         {,key={{field-var} } expr} {,TBL=stno} {,SIZ=int-expr}
         {,TIM=num-expr}) {arg-1ist} {,IOL=stno}
EXTRACT RECORD (fileno {,ERR=stno} {,END=stno} {,DOM=stno}
                {,IND=int-expr} {,key={{field-var} } expr} {,TBL=stno}
                {,SIZ=int-expr} {,TIM=int-expr}) string-variable

FIELD ALIAS (logical unit [,ERR=stmt_num ] ) field_var# [ = ] name_string$
             [, field_var# [ = ] name_string$ ...]
FIND (fileno  {,RETAIN} {,ERR=stno} {,END=stno} {,DOM=stno}
              {,key={{field-var} } expr} {,TBL=stno} {SIZ=int-expr})
              {arg-1ist} {,IOL=stno}

FIND RECORD (fileno {,ERR=stno} {,END=stno} {,DOM=stno} {,IND=int-expr}
             {,key={{field-var} } expr} {,TBL=stno} {,SIZ=size})
             string-variable
FLOATING POINT
FOR numeric-variable = num-expr TO num-expr {STEP num-expr}

GOSUB stno
GOTO stno

IF log-expr {THEN} stno-a {ELSE stno-b} {ENDIF}
INDEXED "file-ID", recno, recsz {,ERR=stno{
INITFILE "file-ID" {,ERR=stno}
INPUT  {(fileno  {,RETAIN} {,ERR=stno} {,END=stno} {,DOM=stno} {,IND=int-expr}
       {,key={{field-var} } expr} {,TBL=stno} {,TIM=time-expr}
       {,SIZ=int-expr})} {,mnemonic} {,string-const} {,variable} {,IOL=stno}
```

```
INPUT RECORD (fileno {,ERR=stno} {,END=stno} {,DCM=stno} {,IND=int-expr}
              {,key={{field-var} } expr} {,SIZ=int-expr} {,TBL=stno})
              string-variable
IOLIST arg-list {fIOL=stno}

{LET} assignment-list
LIST {(fileno  {,ERR=stno} {,IND=recnoj {,TBL=stno})} {stno-a} {,} {stno-b}
LIST PROGRAM "prog-id", "file-ID" {,ERR=stno}
LOAD "prog-id"
LOCK (fileno {,ERR=stno})

MAKE PROGRAM "file-ID", "prog-id" {,ERR=stno}
MERGE (fileno {,ERR=stno} {,IND=int-expr} {,TBL=stno})
MULTI "file-ID", recno {,recsz}, FMT="str-expr" {,ERR=stno}


NEXT num-variable

ON int-expr GOSUB stno-list
ON int-expr GOTO stno-list
OPEN  {INPUT} (fileno {,ERR=stno} {,SEQ=int-expr}) "file/device ID"

 PACK {(fileno {,RETAIN} {,ERR=stno})} {tvar-list}
 PRECISION int-expr
 PRINT {(fileno {,RETAIN} {,ERR=stno} {,END=stno} {,IND=int-expr}
       {,key={{field-var} } expr} {,DOM=stno} {,TBL=stno} {,TIM=time})}
       {,mnemonic} {,var-list} {,IOL=stno} {,}
 PRINT RECORD (fileno {,END=stno} {,ERR=stno} {,TIM=time} {,SIZ=int-expr}
               {,DOM=stno} {,IND=int-expr} {,key={{field-var} } expr}
              S,TBL=stnof) string-variable
 PSAVE "prod-id" {,ERR=stno}

 QUIT

 READ {fileno {,RETAIN } {,ERR=stno} {,END=stno} {,IND=int-expr}
      {,key={{field-var} } expr} {,TBL=stno} {,DOM=stno} {,SIZ=int-expr}
      {,TIM-time-expr})} {,mnemonic} {var-list} {,IOL=stno}

 READ RECORD (fileno {,DOM=stno} {,ERR=stno} {,END=stno} {,IND=int-expr}
              {,key={{field-var} } expr} {,TBL=stno} {,DOM=stno} {,TIM=time}
             {,SIz~int-expr}) string-variable
 RELEASE {"task-id"}
 REM {{"}str-expr{"}}
 REMOVE  (fileno, {,KEY=expr} {,DOM=stno} {,ERR=stno} {,END=stno})
 RENAME "old-file-ID", "new-file-ID" {,ERR=stno}
 RESET
 RETRY
 RETURN
 RUN {"prog-id"}

 SAVE {"prog-id"} {,int-expr}
 SERIAL "file-ID", av-recno, av-recsz {,ERR=stno}
 SETCTL stno
 SETDAY "str-expr"
```

```
SETERR stno
SETESC stno
SETFIELD file-ID, EM T="str-expr" {,MSG="str-expr"} {,ERR=stno}
SETTIME num-expr
SETTRACE {(fileno)}
SETTRANS "file-ID" {,ERR=stno}
SORT "file-ID", keysz, recno {,ERR=stnoj
START {pages} {,ERR=stno} {,"prog-id"} {,"task-id"}
STOP
SYNTAX "str-expr" {,ERR=stno}
SYSTEM "str-expr"

TABLE hexadecimal-string

UNLOCK (fileno {,ERR=stno})
UNPACK (fileno {,ERR=stno}) var-list

WAIT seconds
WRITE {(fileno {,RETAIN} {,ERR=stno} {,END=stno} {,DOM=stno}
       {,IND=int-expr} {,key=H field-var} { expr} {,SIZ=int-expr}
       {,TBL=stno} {,TIM=time})} {,mnemonic} {,variable-list} {,IOL=stno}
WRITE RECORD (fileno {,ERR=stno} {,END=stno} {,DCM=stno} {,IND=int-expr}
              {,TIM=time} {,key={{field-var} } expr} {,SIZ=int-expr}
              {,TBL=stno}) {string-variable}

Additional Directives For BOSS/IX Systems:

 ! {unquoted BOSS/IX command line}
 ADDE "prog-ID" {,ERR=stno}
 ADDR "prog-ID" {,ERR=stno}
 DROP "prog-ID" {,ERR=stno}
 EDIT stno {C[copy through value]} {D[delete through value]}
           {[Rtreplace value]} {[insert value]}
 ERROR
 FILE "str-expr"
 IF log-expr {THEN} statement-a {ELSE statement-b} {FI}

 PREFIX "directory/path...directory/pathn"
 PROGRAM "file ID", prog-size {,diskno} {,sectno} {,init_alloc}
                                {,add alloc} {,ERR=stno}
 STRING "file-ID" {,diskno} {,ERR=stno}
 VMERGE "file-ID"

 Additional Directives For BOSS/VS Systems;

 ! {unquoted BOSS/VS command line}
 ' (Abbreviation for EDIT)
 / (Abbreviation for LIST)
 ? (Abbreviation for PRINT)
 EDIT stno {C[copy through value]} {D[delete through value]}
           {[R[replace value]} {[insert value]}
```

```
ERROR
EXTEND
FILE "str-expr" {,ERR=stno}
HELP {identifier or error number}
IF log-expr {THEN} statement-a {ELSE statement-b} {FI}
NOEXTEND
PROGRAM "file-ID" {,prog-size}
RANDOMIZE {num-expr}
SEQfUENCE} {stno}{,integer}
```

NOTES

INDEX

NOTES