

Business BASIC Level 7 (7.2) Reference Manual

M6252E

MA/BasicFour
Computers That Mean Business

006252-050

This manual includes original works of authorship and proprietary information protected by copyright and trade secret laws. The Copyright Act of 1976 provides for civil liability and severe criminal penalties for unauthorized copying or disclosure of protected works. Possession and use of this manual is restricted and subject to the terms of the MAI Basic Four, Inc. license agreement for this manual ("Agreement"). You shall not publish, reproduce or transmit, in whole or in part, this manual in any form or by any means, whether electronic, mechanical, photographic, or otherwise. You shall not disclose, or fail to make your best efforts to prevent the inadvertent disclosure of, this manual to any third party except to your employees on a need-to-know basis, which employees you shall bind in writing to the terms of the Agreement and be responsible for their compliance therewith. If you fail to act in accordance with this notice and the Agreement, your license to this manual will be immediately and automatically terminated, and you may expose yourself to civil and criminal liability.

Copyright 1983, Rev. 1986 MAI Basic Four, Inc. All rights reserved. MAI and BASIC FOUR are registered trademarks of MAI Basic Four, Inc.

MAI Basic Four, Inc., 14101 Myford Road, Tustin, CA 92680 (714) 730-5100

Business BASIC Level 7 (7.2) Reference Manual

NOVEMBER, 1986
006252-050

M6252E



PAGE STATUS

<u>Chapter</u>	<u>Page No.</u>	<u>Effective Date</u>
Cover/Title Page		May - 86
Page Status	iii through v/vi	May - 86
Table of Contents	vii through xiv	May - 86
Preface	xiii through xiv	Nov - 86
Chapter 1	1-1	Nov - 86
	1-2	May - 86
	1-3	Jul - 84
	1-4 through 1-5	May - 86
	1-6	Jul - 84
Chapter 2	2-1 through 2-8	May - 86
	2-9	Jul - 84
	2-10	Sep - 84
	2-11	Jul - 84
	2-12 through 2-13/14	May - 86
Chapter 3	3-1 through 3-3/4	May - 86
Chapter 4	4-1	Jul - 84
	4-1A through 4-8	May - 86
	4-9	Nov - 86
	4-10 through 4-11	May - 86
	4-12	Aug - 84
	4-13 through 4-18	May - 86
	4-19 through 4-21	Aug - 84
	4-22	Sep - 84
	4-23	Nov - 86
	4-24 through 4-25	May - 86
	4-26	Feb - 85
	4-27 through 4-32	May - 86
	4-33 through 4-40	Jul - 84
	4-41 through 4-42	May - 86
	4-43	Aug - 84
	4-44 through 4-46	Jul - 84
	4-47 through 4-54	May - 86
	4-55 through 4-56	Jul - 84
	4-57 through 4-58	May - 86
	4-59 through 4-60	Jul - 84
	4-60A through 4-60D	May - 86
	4-61 through 4-62	Jul - 84
	4-63 through 4-64	May - 86
	4-65 through 4-68	Jul - 84
	4-68A through 4-68B	May - 86
	4-69 through 4-70	Jul - 84
	4-71 through 4-79	May - 86

PAGE STATUS (continued)

<u>Chapter</u>	<u>Page No.</u>	<u>Effective Date</u>
Chapter 4 (Continued)	4-80 through 4-84	Jul - 84
	4-85 through 4-86	May - 86
	4-87	Feb - 85
	4-88 through 4-94	Jul - 84
	4-95 through 4-100	May - 86
	4-101	Nov - 86
	4-102 through 4-111B	May - 86
	4-112 through 4-116	Jul - 84
	4-117 through 4-118	May - 86
	4-119	Nov - 86
	4-120 through 4-121	Jul - 84
Chapter 5	5-1 through 5-5	Jul - 84
	5-6	Aug - 84
	5-7 through 5-8A	May - 86
	5-9 through 5-12	Jul - 84
	5-13 through 5-14	May - 86
	5-15 through 5-16	Jul - 84
	5-17 through 5-18	May - 86
	5-19 through 5-24	Jul - 84
	5-25 through 5-26	May - 86
	5-27 through 5-28	Feb - 85
	5-29 through 5-30	May - 86
	5-31 through 5-35	Jul - 84
	5-35A through 5-35B	May - 86
	5-36	Nov - 86
	5-37 through 5-38	Jul - 84
Chapter 6	6-1 through 6-2	Jul - 84
	6-3	Nov - 86
	6-4	Jul - 84
	6-5	Nov - 86
	6-6 through 6-7	Jul - 84
	6-7A through 6-9	May - 86
	6-10	Jul - 84
	6-11 through 6-14	May - 86
	6-15	Jul - 84
Chapter 7	7-1	Jul - 84
	7-2	May - 86
	7-3 through 7-4	Jul - 84
	7-5 through 7-6	May - 86
	7-7 through 7-10	Jul - 84
	7-11 through 7-14	May - 86

PAGE STATUS (continued)

<u>Chapter</u>	<u>Page No.</u>	<u>Effective Date</u>
Chapter 8	8-1	Nov - 86
	8-2	Jul - 84
	8-3 through 8-4	Jul - 84
	8-5	Feb - 85
	8-6 through 8-8	Jul - 84
	8-9 through 8-11	May - 86
	8-12	Jul - 84
	8-13 through 8-15/16	May - 86
Chapter 9	9-1 through 9-2	Jul - 84
	9-3 through 9-5	May - 86
	9-6	Jul - 84
	9-7	May - 86
	9-8	Jul - 84
	9-9	May - 86
	9-10	Aug - 84
	9-11 through 9-17	May - 86
	9-18	Jul - 84
	9-19	Feb - 85
	9-20 through 9-22	Jul - 84
	9-23 through 9-24	May - 86
	9-25	Jul - 84
	9-26 through 9-29	May - 86
	9-30	Aug - 84
Chapter 10	10-1 through 10-3	Aug - 86
Appendix A	A-1	Nov - 86
	A-2	Jul - 84
	A-3	Aug - 84
	A-4	Jul - 84
	A-5 through A-7	May - 86
	A-8	Jul - 84
	A-9 through A-10	May - 86
	A-11 through A-16	Jul - 84
Appendix B	B-1 through B-3	Jul - 84
Appendix C	C-1 through C-7/8	May - 86
Appendix D	D-1 through D-4	May - 86
Appendix E	E-1 through E-2	May - 86
Index	I-1 through I-17/18	May - 86

TABLE OF CONTENTS

<u>Chapter</u>	<u>Page</u>
CHAPTER 1 - INTRODUCTION	
Overview.	.BASIC REF 1-1
Scope	1-1
Compatibility	1-1
Contents.	1-2
Conventions	1-3
CHAPTER 2 - FEATURES OF BUSINESS BASIC LEVEL 7	
Console Mode vs. Program Mode	.BASIC REF 2-1
Variables, Constants and Expressions.	2-2
Numbers.	2-2
Variable Names	2-2
Simple Numeric Variables	2-3
Subscripted Numeric Variables (DIM).	2-3
Arithmetic Expressions	2-3
String Constants	2-4
String Variables	2-4
Subscripted String Variables (DIM)	2-5
String Expressions	2-5
String Comparison.	2-6
Logical Operations.	2-6
Output Data Formatting.	2-7
Positioning.	2-7
Numeric Editing.	2-8
Non-formatted Printing of Numeric Values	2-11
I/O Directives and Allowable Devices.	2-12
Command Mode.	2-13
Entering BASIC.	2-13
CHAPTER 3 - STATEMENT FORMATS	
Overview.	.BASIC REF 3-1
Format.	3-1
Statement Numbers	3-1
Directives.	3-2
Parameters and I/O Options.	3-2
Compound Statements	3-2

TABLE OF CONTENTS

<u>Chapter</u>	<u>Page</u>
CHAPTER 4 - DIRECTIVES	.BASIC REF 4-1
CHAPTER 5 - FUNCTIONS	.BASIC REF 5-1
CHAPTER 6 - SYSTEM VARIABLES	.BASIC REF 6-1
CHAPTER 7 - INPUT/OUTPUT OPTIONS	.BASIC REF 7-1
CHAPTER 8 - MNEMONICS.	.BASIC REF 8-1
CHAPTER 9 - ERROR PROCESSING	
Overview.	.BASIC REF 9-1
Non-catastrophic Errors	9-1
Catastrophic Errors	9-2
Error Codes	9-4
CHAPTER 10 - COMMAND INTERPRETER LINE OPTIONS	
FOR BASIC	.BASIC REF 10-1
APPENDIX A - FEATURES OF THE BUSINESS BASIC PROGRAMMING	
ENVIRONMENT	
Ghost Tasks	.BASIC REF A-1
Public Programming.	A-4
BASIC Directory Access and Mounting Procedures.	A-6
Specifying Print Job Attributes	A-8
Input Buffering	A-11
Branching	A-14
Field Protection.	A-14
Null Output Characters.	A-16
Special Key Controls.	A-16

TABLE OF CONTENTS

<u>Chapter</u>	<u>Page</u>
APPENDIX B - INTERNAL CHARACTER CODE.	.BASIC REF B-1
APPENDIC C - PROGRAM FILE FORMAT.	.BASIC REF C-1
APPENDIX D - ALPHABETICAL SUMMARY OF DIRECTIVES, FUNCTIONS, VARIABLES, AND I/O OPTIONS	.BASIC REF D-1
APPENDIX E - BASIC PROGRAM CONVERSION FOR 7.2	.BASIC REF E-1
INDEX	.BASIC REF I-1

LIST OF TABLES

<u>Table</u>	<u>Title</u>	<u>Page</u>
2-1	BASIC DIRECTIVES AND APPLICABLE ACCESS MODES. .	2-1
2-2	INPUT/OUTPUT DIRECTIVES AND APPLICABLE DEVICES.	2-12
4-1	CALL/ENTER	4-6
4-2	TABLE STATEMENT	4-113
5-1	FID FORMAT.	5-13
5-2	PUB(0) FORMAT	5-33
5-3	TSK(0) FORMAT	5-37
6-1	TERMINATOR KEY CONTROL VALUES	6-2
6-2	TCB VARIABLE FORMAT	6-13
8-1	ALPHABETICAL LISTING OF MNEMONICS	8-13
B-1	ASCII CHARACTER CONVERSION (U.S. ENGLISH) . . .	B-1
B-2	NON-PRINTABLE ASCII CODES	B-2
B-3	ASCII CHARACTER CONVERSION (INTERNATIONAL). . .	B-3

PREFACE

This manual provides reference information on the Level 7 Business BASIC programming language. It is assumed that the user already knows how to write programs in BASIC and is familiar with Business BASIC.

The major topics covered in this manual are:

- Chapter 1 - An overview of Business BASIC and the manual
- Chapter 2 - Features of Level 7 Business BASIC
- Chapter 3 - Business BASIC Statement formats
- Chapter 4 - Descriptions of the directives
- Chapter 5 - Descriptions of the functions
- Chapter 6 - Descriptions of the system variables
- Chapter 7 - Descriptions of the input/output options
- Chapter 8 - Descriptions of the mnemonics
- Chapter 9 - Error processing
- Chapter 10 - Command interpreter line options for BASIC
- Appendix A - Features of the Business BASIC programming environment
- Appendix B - ASCII character codes
- Appendix C - BASIC Program file format
- Appendix D - Alphabetic summary of directives, functions, system variables, and i/o options
- Appendix E - Program file conversion for 7.2

Several enhancements have been made to the Level 7, BOSS/IX version of Business BASIC for the 7.2 release. A summary of the changes is given below.

The motivation behind most of the changes to BOSS/IX Business BASIC has been a move towards compatibility with Level 8, BOSS/VS Business BASIC and away from the older Level 4 Business BASIC. This move has resulted in the addition or enhancement of several directives, functions and system variables, and the removal of a few directives and functions that are obsolete.

As with BOSS/VS BASIC, several directive formats accept parameters required by Level 4, but ignore them. This is done to maintain compatibility with Level 4 where most important.

Along with certain changes, the format for BASIC program files has been changed between level 7.1 and 7.2. This necessitates converting any programs written under 7.1 before they can be run under 7.2 Business BASIC. A command is provided to perform the conversion. It is described in Appendix E, included in the change package.

The major enhancements are:

- Line numbers are allowed from 0 to 16000.
- Start sizes are allowed from 10 to 65535, and the size is optional.
- User defined variable, array, and function names may be up to 8 characters long.
- Support for "logical disks" has been removed. The directives, functions and system variables dealing with logical disks have been deleted. Directives still accept the logical disk and sector parameters, but ignore them.
- The ON/GOSUB directive has been added.
- The IF directive has been enhanced to provide full nesting of conditions.
- The BEGIN and CLEAR directives have been enhanced to provide for initializing selected variables only.
- Command language commands can be executed from BASIC programs using the "!" prefix and the SYSTEM directive.
- Mnemonics may now occur as string expression.
- The SAVE directive has been enhanced, requiring only a program name to create a new program file.

OVERVIEW

This manual describes Business BASIC Level 7 for the BOSS/IX 7.2 operating system.

Programmers trained in the original Dartmouth University/ General Electric Business BASIC language will discover major differences in MAI Basic Four's system which provide extended capabilities.

A few aspects of the operating system that affect BASIC programming are described in Appendix A. Note, however, that this is a language manual; a more extensive description of the system is located in the user guide. The filesystem and file types are described in Volume I, Section 5 of the User Guide. Included here are only the BASIC directives for using them.

SCOPE

This reference manual is written as a tool for programmers. The explanations in this manual are presented in a simplified 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 manual is specifically directed toward users of Basic Four systems who develop, program and support business applications.

COMPATIBILITY

Business BASIC Level 7.1 was the same as Level 3/4 BASIC with only a few exceptions. For level 7.2, enhancements have been made to accomplish Level 8, BOSS/VS Business BASIC compatibility.

The Business BASIC file types are all supported on Level 7, and the same format is used for directives that operate on them.

There is no need for bank switching (a memory management procedure on previous systems), since memory management is handled entirely by the operating system.

The user does not have primary access to physical devices as on Level 3/4 systems. Access to devices is handled entirely by the operating system, so that the user may not have direct access to individual sectors on disks or records on magnetic tape. Hence, the GET and PUT directives have been deleted from the language.

Logical disks, which were used in 7.1 for Level 3/4 compatibility, have been deleted from 7.2 Business BASIC.

CONTENTS

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

- Chapter 1: "Introduction" - provides an overview of the purpose of the manual. Defines the intended audience, briefly describes the contents, and defines style conventions.
- Chapter 2: "Features of Business BASIC" - describes variables, constants, expressions, logical operations and output data formatting.
- Chapter 3: "Statement Formats" - explains each component of a statement and defines parameters, common parameter abbreviations, and input/output (I/O) options. Included also here are symbols, compound statements and input terminators.
- Chapter 4: "Directives" - lists and describes each directive in alphabetical order.
- Chapter 5: "Functions" - lists and describes each function in alphabetical order.
- Chapter 6: "System Variables" - lists and describes each system variable in alphabetical order.
- Chapter 7: "Input/Output Options" - lists and describes each input/output option in alphabetical order.
- Chapter 8: "Mnemonics" - lists and describes each mnemonic and its devices by category.
- Chapter 9: "Error Processing" - Lists each error, describes what the error number and message mean, and outlines the procedures to follow for correction.
- Chapter 10: "Command Interpreter Line Options for BASIC" - describes the options that may be specified with the command to enter BASIC.
- Appendix A describes features of the operating system that may be used in the BASIC language environment.
- Appendix B describes the BASIC program header format.
- Appendix C provides a table of internal character codes for use in converting characters from ASCII to hexadecimal and vice-versa.
- Appendix D is an alphabetical summary of the directives, functions, variables, I/O options and system options available in Business BASIC.

CONVENTIONS

This manual uses standard style conventions established for all MAI Basic Four Inc documentation. Symbols used are defined as follows:

Symbols

Symbols used in the examples in this manual 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 used only in the EDIT statement.
- `" "` = Parameters enclosed in quotation marks are required. Quotation marks are to be entered with the parameters they surround.

NOTE

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

```
    {"file ID"}  
    {(logical unit number)}
```

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.
- argument 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.
- devno - the logical unit number that identifies a device, typically terminals and printers. Assigned numbers may be from 0 to 63 only.
- directory - a string expression that specifies a directory name.
- diskno - can be either a number from 0 to 7 or the number 255. This parameter is ignored by 7.2 BASIC.

file ID	-	a character string containing from 1-20 characters (or a string variable of similar length) that identifies a file. A file name may be a full or partial path name. A full path name also contains the directory names, separated by slashes, and may be up to 128 characters long.
fileno	-	the logical unit number of a file, limited to numbers from 0-63 (also called "luno").
init_alloc	-	the number of records initially allocated to a file when the file is defined.
keysz	-	the size of a key in a keyed file; minimum=1, maximum=56.
logical expr	-	a comparison between variables and/or values, using a relational operator.
numeric expr	-	a numeric variable or constant, or an expression containing any combination of both. May also contain arithmetic operators.
prefix list	-	a list of directories to be searched first 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. A prefix list may contain up to 9 directories and can be up to 255 characters long.
prog ID	-	the name of a program.
recno	-	the number of records in a file.
recsz	-	the size of each record of a file, in bytes.
secno	-	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 level 7 BASIC.
stno	-	statement number.
string expr	-	a string variable or literal, or an expression containing a combination of both. May also contain a "+" for concatenation.

Input
Terminators

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

CONSOLE MODE
COMPARED TO
PROGRAM MODE

The BASIC environment has two programming modes: console mode and program mode. First, commands can be entered directly into the system without statement numbers, which causes an immediate execution of the command upon striking of the RETURN or ENTER key. This type of operation provides immediate response to input. While these commands are being entered the system is in console mode.

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

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

NOTE

To start BASIC with other than 32 pages of memory, key in the following at the command interpreter prompt:

```
basic s=###
```

where "###" equals an integer from 10 to 65535, inclusive.

Most directives can be used in either program or console mode; those that cannot are listed below:

TABLE 2-1. BASIC DIRECTIVES AND APPLICABLE ACCESS MODES

DIRECTIVE	PROGRAM MODE ONLY	CONSOLE MODE ONLY
DEF FN	X	
EDIT		X
ENCRYPT		X
ERROR		X
EXECUTE	X	
EXIT	X	
EXITTO	X	
GOSUB	X	
IOLIST	X	
LOAD		X
ON/GOTO	X	
ON/GOSUB	X	
RETURN	X	
TABLE	X	
FOR/NEXT	X	
RETRY	X	
RUN w/o arguments		X

FEATURES OF BUSINESS BASIC LEVEL 7

VARIABLES, CONSTANTS, AND EXPRESSIONS

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

NUMBERS

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

The caret (^) indicates that the following number will represent how many times the preceding number will be multiplied by itself, as in 3^2 for three squared.

Example:

3	3.000
003	.3E1
3.	3^1

These are all valid ways to represent the same number.

Numbers can range in magnitude from $-10^{63}+1$ to $10^{63}-1$. Numbers outside this range result in an ERROR 40. The system retains up to 14 significant digits. Integers and decimal places in excess of 14 digits return an ERROR 26. Exponents out of the range of + or - 63 result in an ERROR 20.

If 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

Level 7.2 Business BASIC allows long variable names, up to eight characters long (plus "\$" for string variables).

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

For example, a user defined variable cannot be TIM, which is a system variable, but it can be TIME. 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.

FEATURES OF BUSINESS BASIC LEVEL 7

SIMPLE NUMERIC VARIABLES

A simple numeric variable is denoted by between 1 and 8 letters or numbers; the first character must be a letter. ACCOUNT2 and EZ4ME are examples of names for simple numeric variables.

A simple numeric variable requires between 21 and 28 bytes of data area when it is assigned any value: 12 for the data itself, 4 for the location table entry, 2 for the sort table entry, 2 for the offset table entry and between 1 and 8 for the id table entry. Once assigned a value these bytes cannot be released without clearing the entire data area. The variable will remain in the variable stack until the stack is cleared, even if the value is zero. 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, Chapter 4).

ARITHMETIC EXPRESSIONS

Business BASIC uses common mathematical symbols, numeric variables and numeric constants to form arithmetic expressions. An arithmetic expression can be used wherever a numeric variable is valid, except to the left of an equal (=) sign. A string variable cannot be used in an arithmetic expression unless converted to numeric format (see NUM and ASC functions in Chapter 5).

Arithmetic expressions are evaluated according to a predefined priority:

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

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

FEATURES OF BUSINESS BASIC LEVEL 7

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

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

STRING CONSTANTS

A string constant can be any length (subject to memory limits of the task) and can be represented in two ways. Characters that can be entered from the keyboard are enclosed in quotation marks ("").

Characters that cannot be generated from the keyboard can be represented by their hexadecimal value. Hexadecimal string constants are enclosed in dollar signs. Two hexadecimal characters are required to represent each single character, i.e., \$01\$ (see Appendix B for assigned values). If only one character is used, then it is right justified (i.e. \$1\$ is treated as \$10\$). If three characters are used, it is left justified (i.e., \$010\$ is treated as \$0010\$).

STRING VARIABLES

A string variable is identified by 1 to 8 letters or numbers 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 user memory, to the number of characters that can be stored in a string variable.

Example:

A\$ = "LOTSOFCHARACTERS"

SUBSCRIPTED
STRING
VARIABLES
(DIM)

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

Examples:

```
00300 DIM B$(5)
```

-B\$ is 5 characters in length.

```
00300 DIM B$(5,"*")
```

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

STRING
EXPRESSIONS

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

Example:

```
00010 LET A$="HEAD"  
00020 LET B$="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.

In this case, the most overhead used during the operation is the length of A\$+B\$ (the size of the temporary storage area) plus the length of B\$ (which must be added to the area occupied by A\$). If the statement "C\$=A\$+B\$" is used, and C\$ has not been defined previously, the overhead required is twice the size of A\$ and B\$ combined (once for the temporary storage area and once for the variable C\$).

FEATURES OF BUSINESS BASIC LEVEL 7

String expressions can include mnemonics. For example:

```
00010 SCREEN$='CS' + @(10,30) + Operator Id_____ " +  
      @(12,32) + "Password_____"
```

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

STRING COMPARISON

When compared, strings of unequal length do not compare as equal. 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.

Example:

```
00100 LET A$="SOME"  
00110 LET B$="SOMEMORE"  
  
-B$ is greater in value
```

LOGICAL OPERATIONS

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

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

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

00100 IF A=1 AND B=2 OR C=3 THEN GOTO 0200

See the IF directive in Chapter 3.

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

OUTPUT DATA
FORMATTING

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

POSITIONING

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

```
00020 PRINT (0,ERR=00100) @ (horizontal position expr
                             {,vertical position expr}) {,output expr}
```

where:

horizontal
position = a numeric constant, variable or
 arithmetic expression the value of which
 defines the horizontal position where the
 first character of the printed or
 displayed value is to be placed.

vertical
position = a numeric constant, variable or
 arithmetic expression the value of which
 defines the line on which input is to be
 placed (applicable only to video display
 devices).

output
expr = an expression that defines the value to
 be printed or displayed (see individual
 statement descriptions for allowable
 forms).

FEATURES OF BUSINESS BASIC LEVEL 7

The following are valid positioning expressions:

```
00010 LET A$="STAGGER"
00020 PRINT @(5,10),A$,@(20),A$
00030 LET R=2,B=5
00040 PRINT @(R*5,B+10),A$
```

>RUN

STAGGER STAGGER

STAGGER

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

The vertical position value is also an integer value, but in this case, reference is made to the top line (line 0) of the VDT screen. The VDT has 24 lines (0-23). If the vertical position is greater than or equal to 24, the display appears on line 0 (the top line) of the VDT screen.

NUMERIC EDITING

Numeric values are normally printed or displayed in a formatted manner. In this system the editing of numeric values to be printed or displayed is provided by a form expression which includes a form operator (:) and a format mask (or the name of a format mask). The form expression follows a numeric expression and precedes a parameter as follows:

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

or

```
PRINT numeric expr: A$
```

FEATURES OF BUSINESS BASIC LEVEL 7

Following are numeric editing options:

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

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

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

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

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

FEATURES OF BUSINESS BASIC LEVEL 7

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.
- = outputs a blank if the value is positive and - if the value is negative.
- DR = outputs DR if the value is positive and CR if the value is negative.
- CR = outputs two blanks if the value is positive and CR if the value is negative.
- 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 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.

NOTE

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

Examples:

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

```
>PRINT A:A$  
+0.05
```

```
>PRINT 1000:A$  
> +1,000.00
```

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

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

FEATURES OF BUSINESS BASIC LEVEL 7

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

Examples:

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

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

I/O DIRECTIVES
AND ALLOWABLE
DEVICES

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

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

File Type/Devices							
Disk Files							
	Serial/ Indexed	Direct/ Sort	Program	Printers (LP,Pn)	Terminal (Tn)	String	Tape (R0)
OPEN	YES	YES	YES	YES	YES	YES	YES
CLOSE	YES	YES	YES	YES	YES	YES	YES
LOCK	YES	YES	YES	NO	NO	YES	NO
UNLOCK	YES	YES	YES	NO	NO	YES	NO
EXTRACT	YES	YES	NO	NO	NO	NO	NO
FIND	NO	YES	NO	NO	NO	NO	NO
INPUT	YES	YES	NO	NO	YES	YES	NO
PRINT	YES	YES	NO	YES	YES	YES	NO
READ	YES	YES	YES*	NO	YES	YES	**
REMOVE	NO	YES	NO	NO	NO	NO	NO
WRITE	YES	YES	YES*	YES	YES	YES	**
LIST	YES	NO	NO	YES	YES	YES	NO
MERGE	YES	NO	NO	NO	YES	NO	NO
VMERGE	NO	NO	NO	NO	NO	YES	NO
FILE	YES	YES	YES	NO	NO	YES	NO
* Program files may be accessed by READ/WRITE if opened with the ISZ= option. Care should be exercised when using this facility. ** Tape supports these directives in RECORD mode only (e.g., READ RECORD but not READ); ERROR 13 is generated if not in RECORD mode.							

FEATURES OF BUSINESS BASIC LEVEL 7

COMMAND MODE

The user may execute command interpreter operations while in BASIC console mode by entering the exclamation point (!) first (i.e., at the BASIC prompt), followed by the command mode directive. For example, the command "pwd" may be executed by entering:

```
>!pwd
```

The result is the same as entering the command from the command interpreter console mode. Following execution of the command, control returns to the BASIC environment.

For Level 7.2, the exclamation point can also be used in BASIC programs:

```
00500 !pwd
```

This is compatible with Level 8 Business BASIC.

A BASIC program may also execute these commands by using the EXECUTE and SYSTEM directives. These are described in Chapter 4.

ENTERING BASIC

There are two methods for entering the BASIC programming environment. If you enter the utility menu system when you log on to the system, select the BASIC mode from the Initial Menu. If you are in the command language environment, type "basic" at the user name prompt:

```
yourname>basic
```

Refer to the Chapter 10, "Command Line Options For BASIC" for other options that you may exercise as you execute the "basic" command.

OVERVIEW

A BASIC program statement is a string of text, following a numeric marker, that contains one or more BASIC directives and concludes with a carriage return. A simple statement contains only one directive. A compound statement contains several directives, each separated by a semicolon (;). The statement numbers specify the order in which the statements will be performed.

FORMAT

Every BASIC program statement contains three elements: a statement number, a directive, and one or more parameters. Console mode statements do not have a statement number. BASIC statements are in the following format:

Example: 500 PRINT "EXPRESSION"

<u>500</u>	<u>PRINT</u>	<u>"EXPRESSION"</u>
Statement Number:	Directive:	Parameter(s):
A number that uniquely identifies the statement and places it within the program in its proper sequence.	The type of operation being requested.	Required and/or optional values used in association with the directive to further define the action to be taken. (Some directives need no parameters)

Each element must be separated from the preceding and following elements by a space in Level 7.2 BASIC.

SPACE
SENSITIVITY

Level 7.2 Business BASIC is space sensitive, making spaces significant even outside of double quotations marks. This is necessary to accommodate long variable names.

For example, PRINT2 can be a variable:

00010 PRINT2 = 5

In order to print "2", this will generate an error:

00020 PRINT2

What is needed is:

00020 PRINT 2

The general rule to follow for blank spaces is: when you type a letter or a number after a keyword, it is interpreted as a long variable. To avoid this interpretation, a space must be entered immediately following the keyword.

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

When entering statement numbers, or any other numeric entry, leading zeros need not be entered.

DIRECTIVES

The directive is the key element of the BASIC statement in that it instructs the system to perform specific operations such as PRINT or READ. Directives can be executed in both Console and program modes, unless otherwise noted in the description of the directive.

LET and THEN are optional parts of the directives and need not be entered (see the LET and IF/THEN directives in Section 3).

PARAMETERS AND I/O OPTIONS

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

Abbreviations for parameters are defined in the CONVENTIONS portion of the introduction.

For a detailed explanation of input and output parameters, see INPUT/OUTPUT OPTIONS in Chapter 6.

COMPOUND STATEMENTS

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

Example:

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

The following rules apply to compound statements:

1. Both console and program mode statements may be compound.
2. DEF, TABLE and IOLIST cannot be part of a compound statement.
3. A 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 followed by any command other than REM:

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

5. An ESCAPE check occurs at each semicolon during execution.
6. RETURN causes a return to the next statement in the compound sequence following a GOSUB routine.
7. RETRY re-executes the appropriate statement within a compound sequence after an error has been corrected.

DIRECTIVES

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

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

The Directives are presented in alphabetical order.

DIRECTIVES

FORMAT

! {unquoted BOSS/IX command}

DESCRIPTION

! is used in console or program mode to execute a BOSS/IX command while remaining in BASIC.

EXAMPLES

```
00010 PRINT "YOUR DIRECTORY IS",; !pwd
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 will translate 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
>
```

This page left blank intentionally.

DIRECTIVES

ADDR
ADDE

ADDR
ADDE

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 memory 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. Thus it is available for use by your tasks without time spent in directory searches or loading the program when you execute a CALL, RUN or LOAD.

Another terminal user who wants to use the same program that you have added to memory should also ADDR the program. Otherwise a directory search will be executed and the program then found to be already in memory.

If you try to ADDR a program that you have already added into memory, and ERROR 14 results.

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 dropped by means of the DROP directive.

ADDE - Adds into memory an error-handling program which is automatically called whenever BASIC falls into console mode to display an !ERROR=nn message. The ADDE program may use the error number variable to display a more detailed message about an error occurrence. For example, the system automatically displays the message for ERROR 41 as "INVALID INTEGER RANGE". You can use an ADDE'd program to tell the operator what was done wrong and how to go about the procedure properly.

DIRECTIVES

ADDR (continued)
ADDE

ADDR (continued)
ADDE

NOTE

An error handling program that has been ADDE'd 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 from another terminal.

EXAMPLES

00100 ADDE "YOUGOOFED"

00110 ADDR "OFTENUSED"

DIRECTIVES

BEGIN

BEGIN

FORMAT

BEGIN {{EXCEPT} var1, ... ,varN}

where:

var1, ..., varN = a list of up to 63 variables

DESCRIPTION

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

- Resets the ERR and CTL system variables to zero.
- Resets incompleted GOSUB and FOR/NEXT loops (clears the stack).
- Resets precision to 2.
- Clears the user data area, eliminating all variable definitions.
- Closes all OPEN files and devices.
- Resets the contents of TCB(11)

The BEGIN directive with a variable list resets only the variables in the list. The BEGIN EXCEPT directive followed by a variable list resets all and only variables not in the list.

EXAMPLES

00020 BEGIN

>BEGIN

00150 BEGIN A,B\$,TIME\$

00250 BEGIN EXCEPT C,DTE\$

DIRECTIVES

CALL

CALL

FORMAT

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

where:

argument list = 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 variable in the argument list is referenced in the called program by the name specified in the corresponding ENTER statement.

A program which is called is loaded into the shared memory area of main memory. The memory management unit first checks the shared memory area to make sure that only one copy of the program is resident in shared memory at one time. The shared memory segment identification is then passed to the BASIC interpreter. Then control is passed to the called program. When that program terminates at the EXIT, its entry is dropped unless it is in use by another task.

If a program that is called is already resident in the shared memory area (meaning it has been added by means of the ADDR or ADDE directives), the program simply has control passed to it. Adding the program by means of ADDR or ADDE is recommended to reduce disk search time if a program is to be used frequently.

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

Programs using CALL should have provisions for handling execution of an ESCAPE statement, pressing of the ESCAPE key, and the occurrence of an error. If one of these conditions occurs, and the program is not designed to handle it, the called program is exited and the system returns to the main program. If the main program is not designed to handle these conditions, the system returns to console mode.

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

An attempt to pass the same variable twice in the same CALL results in an ERROR 38.

DIRECTIVES

CALL (continued)

CALL (continued)

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

EXAMPLES

01000 CALL "MEACAB"

01010 CALL "MEABUS",ERR=2000,A\$,B

DIRECTIVES

CALL (continued)

CALL (continued)

Table 4-1. CALL/ENTER Directives

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

DIRECTIVES

CLEAR

CLEAR

FORMAT

CLEAR [{EXCEPT} var1, ..., varN]

where:

var1, ..., varN = a list of up to 63 variables

DESCRIPTION

The CLEAR directive used without a variable list resets the system by performing the same functions as the RESET directive, and clearing the user data area.

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

CLEAR performs the following functions:

- Resets the ERR and CTL system variables to zero.
- Resets incompletd 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 all and only the variable in the list. The CLEAR EXCEPT directive with a variable list resets all and only variables not in the variable list.

EXAMPLE

00020 CLEAR

>CLEAR

00250 CLEAR A, C\$, TIME\$

DIRECTIVES

CLOSE

CLOSE

FORMAT

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

where:

index expr = used only for 1/2 inch magnetic tape to indicate the position of the tape after the CLOSE. The following options are available:

IND = 0 - rewinds tape to load point
or
IND = 2

IND = 1 - rewinds tape to load point and
takes tape off-line

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.

NOTE

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

DESCRIPTION

The CLOSE directive releases use of a file or device. The total number of open files and devices cannot exceed 64 in level 7 BASIC. File number 0 is always open to the terminal.

The ERR= parameter is used on Business BASIC Level 4. It remains here to maintain compatibility between the levels, but the Level 7 BASIC interpreter ignores it.

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

The CLOSE directive unlocks files that were locked by the LOCK directive.

When closing a file on 1/2 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 READ RECORDs, the file is closed and no filemarks are written to tape. If one or more WRITE RECORDs were used, then at least one filemark is written, at the point writing was stopped.

An ERROR 13 is reported if and IND= is specified other than 0, 1, 2, or 9.

EXAMPLES

01200 CLOSE (1)

01200 CLOSE (1,ERR=0150) The ERR= is ignored.

This page left blank intentionally.

DIRECTIVES

DEF FNx
DEF FNx\$

DEF FNx
DEF FNx\$

FORMAT

DEF FNx (argument list) = arithmetic expr

DEF FNx\$ (argument list) = string expr

where:

x = a function name that uniquely defines the DEF statement, where x is letter from A - Z followed by 0 to 7 letters or digits.

argument list = a list of variables where the position of each variable is correlated to a corresponding item positioned in the same relative location within the argument list of the statement using the DEF FNx directive.

DESCRIPTION

The DEF statement is used to define up to 63 functions in a program. These functions are in addition to the predefined functions which are part of the Business BASIC language (see "FUNCTIONS" in Chapter 5).

The DEF FNx directive defines an arithmetic operation; 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. 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.

DIRECTIVES

DEF FNx
DEF FNx\$ (continued)

DEF FNx
DEF FNx\$ (continued)

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

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

```
>RUN  
-1 100 10
```

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

EXAMPLES

DEF FNx - 00010 DEF FNA(A,B)=(A+B)/A

00020 LET C=FNA(2,6)

- Statement 00020 assigns A=2, B=6 and C=(2+6)/2=4

DEF FNx\$ - 00010 DEF FNA\$(A\$,B\$)=B\$+"-"+A\$
01000 LET X\$="SIDO",Y\$="DOE"
01010 PRINT FNA\$(X\$,Y\$)

```
>RUN  
DOE-SIDO
```

DIRECTIVES

DELETE

DELETE

FORMAT

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

where:

stno a = the number of a statement to be removed, or
 the first in a series of statements to be
 removed.

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

DESCRIPTION

The DELETE directive is used to remove one or more statements from a program. It cannot be used in a called or public program. The DELETE directive can specify nothing (to delete all statements); it can specify a single statement number to delete that line only; a statement number followed by a comma deletes that statement and all following statements; a comma preceding the statement number deletes that and all preceding statements; and a statement number followed by a comma and a second number deletes the first statement specified and all statements up to and including the second statement specified. The examples below illustrate the use of this directive.

Attempting to delete a non-existent line number with this directive has no effect and does not generate an ERROR 21. However, attempting to delete a nonexistent statement number by typing the number followed by a carriage return will generate an ERROR 21 (STATEMENT NUMBER ERROR).

EXAMPLES

DELETE	- removes all statements from the program.
DELETE 10	- removes only statement 10 from the program (simply typing 10 followed by a carriage return performs the same task).
DELETE 10,	- removes statement 10 and all following statements.
DELETE 10,100	- removes all statements between 10 and 100, inclusive.
DELETE ,100	- removes all statements up to and including statement 100.

DIRECTIVES

DIM array

DIM array

FORMAT

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

where:

array name = name of the numeric array (a letter followed by 0 to 7 letters or digits).

range of dimensions = specifies the number of elements in each of the three possible dimensions. The number of elements in the subscript may be specified by a number or a variable, but the value must be an integer.

DESCRIPTION

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

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

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

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

DIM is allowed on ENTERed variables.

DIRECTIVES

DIM array (continued)

DIM array (continued)

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. For example:

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

For Level 7.2, like Level 8 BASIC, the name of an array may be up to eight characters and digits long, but must begin with a letter.

EXAMPLES

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

```
00010 DIM A(1)
```

```
00010 DIM A(2,2,2)
```

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

```
00200 DIM A(5)
00210 FOR I=0 TO 5
00215 LET A=37
00220 LET A(I)=I*10; NEXT I
00225 PRINT A(5),A(4),A(3),A(2),A(1),A(0),A
```

```
>RUN
```

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

DIRECTIVES

DIM string

DIM string

FORMAT

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

where:

string variable name	= name of the string (a letter followed by 0 to 7 letters or digits and \$).
length	= length of the string (up to the limit of available user memory). Data area for the string variable is released if the length is 0.
string expr	= the character used to fill the string. If no character is specified, the string is filled with blanks.

DESCRIPTION

The DIM directive creates the string variable of the specified length. The string variable is initialized with the fill character specified in the string expression. If no fill character is specified, the string is filled with blanks. If the fill character expression is more than one character long, only the first is used.

Dimensioning a string of underlines, for example, is useful in programming input routines in which you want to prompt the operator by specifying the allowable length of the input. For each field you need only call the required number of underlines by printing the subscripted string.

If B\$ is dimensioned to twenty underlines, you can call five of them in this manner: PRINT @(X,Y), B\$(1,5). The directive takes from the first to the fifth underlines and prints them at the specified location. An alternate method of subscripting specifies B\$(16), which takes the sixteenth fill character to the end of the string (i.e., five underlines).

EXAMPLES

01200 DIM A\$(5)	--assigns 5 blanks to A\$
01200 DIM B\$(5,"A")	--assigns "AAAAA" to B\$
01200 DIM C\$(5,"BC")	--assigns "BBBBB" to C\$

DIRECTIVES

DIRECT

DIRECT

FORMAT

DIRECT "file ID", keysz, recno, recsz {,diskno, sectno
, init_alloc, add_alloc} {,ERR=stno}

where:

keysz	= the size of a key in a keyed file: minimum=1, maximum=56.
recno	= the maximum number of records for the file (cannot exceed 8,388,608).
recsz	= the size, in bytes, of each record in the file (cannot exceed 32,256).
diskno	= used only for compatibility with other Basic Four systems. Otherwise, it is ignored.
sectno	= used only for compatibility with other Basic Four systems. Otherwise, it is ignored.
init_alloc	= the number of records initially allocated to a file when the file is defined; this cannot exceed the recno parameter.
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 until it reaches its defined number of records.

NOTE

If init_alloc and add_alloc are not specified,
then the system will define the entire file at
creation time.

DIRECTIVES

DIRECT (continued)

DIRECT (continued)

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

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

The Direct file structure is described in the User Guide.

NOTE

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

EXAMPLES

DIRECT "HIT",10,100,50

-defines a DIRECT file named "HIT" with a key size of 10 bytes, 100 records of 50 bytes each, in the user's current working directory. The filename may have specified an absolute filename indicating a specific directory.

DIRECT "ORY",8,10,80,12,2,1

-defines a DIRECT file named "ORY" with a key size of 8 bytes, 10 records of 80 bytes each in the user's working directory. The sector number parameter, 12 in this case, is ignored. Two of the ten records are allocated at file creation time, and as the file grows and needs additional space that additional space is allocated in increments of one record.

DIRECTIVES

DISABLE

DISABLE

The DISABLE directive has been deleted from 7.2 BASIC.

DIRECTIVES

DROP

DROP

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.

Programs that are in use cannot be dropped. Therefore, if the program cannot be dropped, the error should be handled within the program to prevent falling to console mode.

EXAMPLE

1200 DROP "ALINE" -removes "ALINE" from memory.

DIRECTIVES

EDIT (Text Editor)

EDIT (Text Editor)

FORMAT

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

where:

copy through value	= indicates the point in the text of the original statement after which a change is to occur.
delete through value	= indicates the final point in the text of the original statement through which everything is to be deleted.
replace value	= specifies the text that is to replace the existing characters in the original statement, 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

This EDIT directive is compatible with Level 4 BASIC and is used to alter existing program statements. The parameters allow you to add, delete or replace any character or string of characters in any program statement.

There is also a screen editor for changing program statements. It is described just after the Text Editor.

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

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

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

DIRECTIVES

EDIT (continued)

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

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

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

(The series of "C[E]" characters tells the cursor to pass over four E's in the text before any editing takes place. "R..." causes the remaining text to be replaced by the spaces and characters within the brackets. The final set of brackets calls for those characters to be inserted.)

NOTE

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

```
0020 REM "23"  
>EDIT 20 C[2]R[4]  
0024 REM "23"
```

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

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

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

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

DIRECTIVES

EDIT (continued)

EDIT (continued)

EXAMPLES

ORIGINAL STATEMENT: 1200 PRINT(1)"CHANGER"

1. "Delete" EDIT:

EDIT 1200 C["] D[H]

result:

1200 PRINT(1)"ANGER"

2. "Replace" EDIT:

EDIT 1200 C[(] R[2]

result:

1200 PRINT(2)"ANGER"

3. "Insert" EDIT:

EDIT 1200 C[)] [CH]

result:

1200 PRINT(2)"CHANGER"

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

ORIGINAL STATEMENT: 0150 PRINT (1) "ALPHABET SOUP"

EDIT command:

>EDIT 150 C[]R[WRITE]C[(]R[2]C["]
R[CHICKEN][NOODLE]

result:

0150 WRITE (2) "CHICKEN NOODLE SOUP"

DIRECTIVES

EDIT (screen editor)

EDIT (screen editor)

In addition to the previously described editing procedure, compatible with other BASIC FOUR systems, Level 7 BASIC enhanced "screen editing" feature (similar to Level 8 BASIC).

FORMAT

EDIT stno

EDIT -help

DESCRIPTION

The statement line will be displayed, and the cursor will appear at the beginning of the statement. By pressing the control key (CTRL) simultaneously with any of the letter keys in the following list, the statement line will be changed according to the directions for that key. The "-help" option displays the following list of editing procedures:

PRESS THESE KEYS:

EFFECT ON THE STATEMENT LINE:

CTRL + A	-moves the cursor to the beginning of the current line.
CTRL + C	-exits the editor, abandons edits.
CTRL + D	-deletes the character at the cursor.
CTRL + E	-moves the cursor to the end of the current line.
CTRL + F	-moves the cursor one space to the right.
CTRL + H	-moves the cursor one space to the left.
CTRL + I	-activates the insert mode.
CTRL + N	-moves the cursor down one line.
CTRL + O	-deactivates the insert mode, activates overstrike mode.
CTRL + P	-moves the cursor up one line.
CTRL + T	-tabs forward (moves the cursor eight spaces to the right).
CTRL + V	-exit edit mode (deactivates the edit process without saving any of the changes).

DIRECTIVES

EDIT (screen editor, continued)

EDIT (screen editor, continued)

CTRL + W	-tabs backward (moves the cursor eight spaces to the left).
DELETE	-deletes character left.
ESCAPE	-exits editor, abandons edits.
RETURN	-exit edit mode (deactivates the edit process and saves all the changes).

When the Insert Mode is on, each subsequently entered character is inserted into the statement line at the cursor, and the remainder of the line moves to the right one space for each inserted character. Turn off the Insert Mode when you are finished inserting text.

Typing characters when Insert Mode is off simply overwrites the existing text. With the control key pressed, typing a character other than those in the list will have no effect.

Typing characters without pressing the control key will have an effect depending on whether or not the Insert Mode is on. If the Insert Mode is on, the character will be inserted into the text. If the Insert Mode is off, the existing text will be overwritten by the typed characters.

Edits made by either the line method or the screen method must be saved to the program file or they will be lost when the program is deleted from memory.

DIRECTIVES

ENABLE

ENABLE

The ENABLE directive has been deleted from 7.2 BASIC.

DIRECTIVES

ENCRYPT

ENCRYPT

FORMAT

ENCRYPT "Oldname", "Newname"

where:

"Oldname" = the name of the program file that contains the program to be encrypted.

"Newname" = the name of the program file that will contain the encrypted program.

DESCRIPTION

The ENCRYPT directive encodes the program statements of the source BASIC program, and stores the result in the second (destination) file. Both files must have been already defined before the ENCRYPT directive, and both files must be PROGRAM files.

The encrypted program will RUN in the same manner as the source program. However, the function PGM and the directives LIST, EDIT, and SAVE will not work, but will generate an ERROR 18, when referring to the encrypted program.

Statements may not be added to or deleted from an encrypted program while it is loaded. However, the encrypted program itself can insert or delete statements by means of the EXECUTE directive.

When you use this format for the ENCRYPT directive, the original "Oldname" PROGRAM file remains intact (unencrypted). The destination file, "Newname" contains the new encrypted version of the original PROGRAM file.

An optional format for the ENCRYPT directive uses "Oldname" as both the source file and the destination file:

ENCRYPT "Oldname", "Oldname"

In this case, the original, unencrypted version of the program is deleted and replaced by the encrypted version. The encrypted version retains the original file name.

Note that ENCRYPT is only available in console mode.

EXAMPLE

ENCRYPT "OPEN", "HIDDEN"

The ENCRYPT directive will take the BASIC program "OPEN", encrypt it, and store the encrypted program in the file "HIDDEN".

DIRECTIVES

END

END

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
- CLOSEs all open files and devices, except when done in a CALLED program
- Performs a RESET operation
- Returns the terminal to console mode
- Executed from a CALLED program, END performs EXIT

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

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

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

EXAMPLE

09999 END

DIRECTIVES

ENDTRACE

ENDTRACE

FORMAT

ENDTRACE

DESCRIPTION

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

EXAMPLE

>ENDTRACE

00200 ENDTRACE

DIRECTIVES

ENTER

ENTER

FORMAT

ENTER argument list

where:

argument

list

= 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).

NOTE

Only one ENTER directive can be used for each called program.

DESCRIPTION

The ENTER directive defines a set of variables in a called program that corresponds to a set of variable names in the argument list of the calling program. ENTER is used for passing arguments (values) from the calling program to the called program, and back again.

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

DIRECTIVES

ENTER (continued)

ENTER (continued)

EXAMPLES

01000 ENTER A\$,B,C	-passes parameters A\$,B and C to the called program.
01000 ENTER A(ALL)	-passes the entire array of parameters to the called program.

Calling Subroutine

00010 CALL "SUB",1,2,3	-note that only three variables are used to call the subroutine.
------------------------	--

Routine Called

00010 A=1, B=2, C=-3, D=-4	however, the resulting values of A,B,C and D are:
00020 ENTER A, B, C, D	
	A=1
	B=2
	C=3
	D=-4

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

DIRECTIVES

ERASE

ERASE

FORMAT

ERASE "file ID" {,ERR=stno}

DESCRIPTION

The ERASE directive deletes a file entry from the disk directory. It also de-allocates 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 the matching file found in the user's current working directory will be erased.

EXAMPLES

01000 ERASE "AGOOFF" - deletes the file "AGOOFF" from the disk directory and de-allocates the disk space only if it is found in the user's current working directory.

04000 ERASE "/usr/seq/LOGS" - deletes the file "LOGS" only from the "seq" directory. The erase has no effect on a file of the same name in another directory.

DIRECTIVES

ERROR

ERROR

FORMAT

ERROR

DISCUSSION

When you execute this directive a message is displayed that describes which BASIC error occurred last and which file system error corresponding to it, if any.

The error code field is cleared when you enter BASIC or execute a BEGIN or CLEAR. The message is displayed but no code numbers.

The BASIC error code corresponds to the contents of the ERR variable, and the file system error code corresponds to the TCB(12) described in Chapter 5.

EXAMPLE

1.

```
100 INPUT (0, ERR= 800 )"TYPE A NUMBER: ", A
800 PRINT "HERE'S THE ERROR CODE: "
810 ERROR
820 GOTO 100
```

2.

```
>OPEN (9)"NO FILE"
!ERROR=12 : MISSING OR DUPLICATE NAME/NON-CONFIGURED DEVICE
>ERROR
Basic error code: 12
File system error code: -5
```

DIRECTIVES

ESCAPE

ESCAPE

FORMAT

ESCAPE

DESCRIPTION

When executed in program mode (either as a single statement or within a compound statement), 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 executed in console mode, ESCAPE causes the system to list the next statement (if any) in the line to be executed in the currently running program.

EXAMPLE

2000 ESCAPE

DIRECTIVES

EXECUTE

EXECUTE

FORMAT

EXECUTE string argument

where:

string = a string expression that duplicates
argument either a console mode command or a line

DESCRIPTION

The EXECUTE directive can only be used in program mode. It cannot be used in console mode, nor can it be invoked in a public program.

EXECUTE provides the ability to generate or modify program statements within a program.

EXECUTE can be used to build statements. When used within a program it permits the use of directives and commands which are normally available only in console mode and in command interpreter mode, as in 100 EXECUTE "!pwd", which will print the working directory.

If a user presses ESCAPE during the running of a program that is executing a "!" command, it is ignored. Use CTRL + "C" to terminate a "!" command execution.

EXAMPLES

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

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

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

```
0100 LET X=Q+3
0200 IF X=4 THEN GOTO 400
.
.
0400 EXECUTE "EDIT 0100 C[+]R[5]"
```

A third use of the EXECUTE directive is to create statements to be executed later in a program:

```
0100 LET X=Q+3
0200 IF A=X THEN GOTO 400
.
.
0400 EXECUTE "0150 LET A=B"
```

DIRECTIVES

EXIT

EXIT

FORMAT

EXIT {expr}

where:

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

DESCRIPTION

The EXIT directive is used to exit a called program and return control to the calling program.

The first statement executed after an EXIT directive is the statement following the CALL statement in the calling program. If the CALL was made from console mode, EXIT returns control to console mode.

If the BASIC variable ERR is used as EXIT's numeric expression, the error status of the called program becomes the error status of the calling program.

Even if there is no error in the called program, EXIT X will return an error status of X to the calling program.

EXAMPLES

9999 EXIT

9999 EXIT ERR

1000 EXIT A + B

DIRECTIVES

EXITTO

EXITTO

FORMAT

EXITTO stno

DESCRIPTION

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

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

EXAMPLE

```
0010 FOR I=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 0060, and the top entry is cleared from the FOR/NEXT stack.

DIRECTIVES

EXTRACT

EXTRACT

FORMAT

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

where:

argument list = string or numeric variables into
 which extracted data is to be
 inserted.

NOTE

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

DESCRIPTION

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

EXTRACT differs from READ in two ways:

1. it prevents other users from accessing the record until
another operation is performed on the file;
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
released for access by other users.

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

EXAMPLE

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

-reads and locks a record, setting record pointer to the
extracted record.

DIRECTIVES

EXTRACT RECORD

EXTRACT RECORD

FORMAT

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

where:

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

DESCRIPTION

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

EXTRACT RECORD differs from READ RECORD in two ways:

1. it prevents other users from accessing the record until another operation is performed on the file;
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 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 directive is executed to modify the file, the pointer is already in position to return data to the correct field.

DIRECTIVES

FILE

FILE

FORMAT

FILE string

where:

string = 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).

NOTE

The FILE directive cannot be used to recover a file as it could on previously released BASIC FOUR products.

EXAMPLES

```
0010 OPEN (1)"ADOOR"  
0020 LET F$=FID(1)  
0030 CLOSE (1)  
0040 ERASE "ADOOR"  
0050 FILE F$
```

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.

DIRECTIVES

FIND

FIND

FORMAT

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

where:

argument

list

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

NOTE

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

DESCRIPTION

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

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

EXAMPLE

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

DIRECTIVES

FIND RECORD

FIND RECORD

FORMAT

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

where:

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

DESCRIPTION

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

EXAMPLE

00200 FIND RECORD(1,KEY=K\$,ERR=00500)A\$

DIRECTIVES

FLOATINGPOINT

FLOATINGPOINT

FORMAT

FLOATINGPOINT

DESCRIPTION

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

Business Basic Level 7 allows a range of numbers from E-63 to E+63.

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

EXAMPLE

```
00010 FLOATINGPOINT
00020 FOR I=0 TO 5
00030 PRINT 2^I;NEXT I
```

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

DIRECTIVES

FOR/NEXT

FOR/NEXT

FORMAT

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

where:

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

DESCRIPTION

The FOR/NEXT loop is used to repeat a series of statements in a program.

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

DIRECTIVES

FOR/NEXT (continued)

FOR/NEXT (continued)

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

The reverse is true when you use a negative step value. For example, "FOR I = 100 to 1 STEP -1", control passes to the next statement as long as the start expression is greater than or equal to the end value.

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

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

is invalid.

During the execution of a FOR statement, you cannot delete the statement. Escaping into console mode to delete the statement and deleting the statement through the EXECUTE directive both generate an ERROR 27.

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

DIRECTIVES

FOR/NEXT (continued)

FOR/NEXT (continued)

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

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

Nested FOR/NEXT loop:

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

```
>RUN
11 12 13

21 22 23
```

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

FOR/NEXT loop

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

DIRECTIVES

GOSUB

GOSUB

FORMAT

GOSUB stno

DESCRIPTION

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

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

During the execution of a GOSUB statement, you cannot delete the statement. Escaping into console mode to delete the statement and deleting the statement through the EXECUTE directive both generate 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$="####0.00"
0060 GOSUB 1000
0070 READ (1,END=0500,ERR=0600)A,B,C,D
0080 LET L=L+1
0090 IF L>50 THEN GOSUB 1000
0100 PRINT (7)A:P$,@(10),B:P$,@(20),C:P$,@(30),D:P$
0110 GOTO 0070
0500 PRINT "END OF RUN"
0510 STOP
0600 PRINT "ERROR: ",ERR:"000"," OCCURRED ON READ"
0610 STOP
1000 REM "SUBROUTINE TO PRINT HEADINGS"
1010 LET P=P+1,L=0
1020 PRINT(7)'FF',"ITEM",@(10),"QUANTITY",@(20),
1020: "COST",@(30),"PRICE",@(70),"PAGE",P,'LF'
1030 RETURN
```

DIRECTIVES

GOTO

GOTO

FORMAT

GOTO stno

DESCRIPTION

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

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

EXAMPLE

```
00100 OPEN (7)"LP"  
00110 LET X=L+1  
00120 GOTO 00500  
00130 PRINT (7)"THIS"
```

```
>OPEN(7) "LP"  
>GOTO 0130  
>RUN  
THIS
```

DIRECTIVES

IF

IF

FORMAT

IF logical expr {THEN} statement a {ELSE statement b} {FI}

where:

logical expr = a comparison between variables and/or
 values using a relational operator
 sign or a logical compound of
 relations

statement a = the statement, such as GOTO 0250 or
 WRITE A\$, to execute if the
 comparison in the logical expression
 is "true"

statement b = the statement, such as GOTO 0275 or
 WRITE B\$, to execute if the
 comparison in the logical expression
 is "false"

FI = a delimiter, used to terminate the
 range of the immediately preceding
 unterminated IF statement

DESCRIPTION

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

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

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

DIRECTIVES

IF (continued)

IF (continued)

Some examples of logical expressions are:

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

C>=B A/B=E

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

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

Example:

```
00010 LET A=1,B=2,C=3
00200 IF A=1 OR B=2 AND C=0 THEN PRINT "20 IS
00020:TRUE"
00030 IF A=1 OR (B=2 AND C=0) THEN PRINT "30 IS
00030:TRUE"

>RUN

30 IS TRUE
```

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

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

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

DIRECTIVES

IF (continued)

IF (continued)

IF/ELSE commands can be fully nested into a single statement in 7.2 BASIC. The subsequent IF can occur in either the THEN or the ELSE clause of the preceding IF.

```
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. So, in the above scheme, the first ELSE terminates the second IF, and the second ELSE terminates the first IF.

The FI terminator is used in compound IF statements to terminate a nested IF clause not terminated with an ELSE.

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

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

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

```
00300 IF A=20 THEN GOSUB 09000; C=3
00400 IF A=20 THEN GOSUB 09000 FI; C=3
```

Line 300 set C=3 only if A=20, but 400 sets C=3 in either case.

DIRECTIVES

IF (continued)

IF (continued)

EXAMPLES

0010 IF A=B THEN GOSUB 6000 ELSE GOTO 9999

It is not necessary to type the word "THEN" as part of a THEN clause, if another directive is involved (e.g., GOTO, GOSUB, etc.) nor is it necessary to add preceding zeros. The system adds them automatically.

10 IF A=B GOSUB 60 -as entered (no THEN)

>LIST

00010 IF A=B THEN GOSUB 00060 -as displayed by the system

DIRECTIVES

INDEXED

INDEXED

FORMAT

INDEXED "file ID", recno, recsz {, diskno, sectno
, init_alloc, add_alloc } {,ERR=stno}

where:

recno	= the maximum number of records for the file (cannot exceed 8,388,608).
recsz	= the size, in bytes, of each record in the file (cannot exceed 32,256).
diskno	= used only for compatibility with other MAI Basic Four systems. Otherwise, it is ignored.
sectno	= used only for compatibility with other MAI Basic Four systems. Otherwise, it is ignored.
init_alloc	= the number of records initially allocated to a file when the file is defined; this cannot exceed the recno parameter.
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 grows in increments until it reaches its defined number of records.

NOTE

If init_alloc and add_alloc are not specified, then the system will define the entire file at creation time.

DIRECTIVES

INDEXED (continued)

INDEXED (continued)

DESCRIPTION

The INDEXED directive defines a file comprised of records located in contiguously numbered sectors. These records 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.

EXAMPLES

00130 INDEXED "FINGER", 100, 50

- creates the Indexed file "FINGER" in the user's working directory with 100 records of 50 bytes each.

00130 INDEXED "BIGTOE", 100, 50, 3, 7, 25, 5

- creates the Indexed file "BIGTOE" in the user's working directory with 100 records of 50 bytes each. Note that the logical disk number, 25, and the sector number, 7, are ignored. When the file is created, 25 of the 50-byte records are allocated. As the file grows, it will extend in increments of 5 (50-byte) records at a time.

DIRECTIVES

INPUT

INPUT

FORMAT

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

where:

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

NOTE

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

The INPUT directive is essentially identical to the READ directive.

The INPUT directive provides for two-way communication between the operator and the program. An INPUT may contain messages to the operator to be displayed on the terminal device. The operator's response goes into the variables included as parameters in the INPUT statement.

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

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

DIRECTIVES

INPUT (continued)

INPUT (continued)

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

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

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

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

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

```
00010 LET F$=FID(0)
00020 OPEN (2)F$
00030 INPUT (2,ERR=00030)@(0,10),"ENTER QUANTITY
00030:SOLD-",B
00040 PRINT (0,ERR=00030)@(0,11),B:"00000"
>RUN

ENTER QUANTITY SOLD-
000123
```

DIRECTIVES

INPUT (continued)

INPUT (continued)

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

Example:

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

Input Verification

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

Numeric Verification

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

where:

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

DIRECTIVES

INPUT (continued)

INPUT (continued)

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

Examples:

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

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

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

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

String
Verification

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

where:

branchlist = branchlist is one or more items whose syntax is: string literal =stmt no. (e.g., "END"=100). Branchlist items are separated by commas. If a true condition is found (i.e., N\$ = string literal), statement execution is transferred to the specified statement number.

Min Max = Min and Max specify the inclusive range of legal lengths for N\$. Min must be less than or equal to Max, or an ERROR 20 results.

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

DIRECTIVES

INPUT (continued)

INPUT (continued)

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

Examples:

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

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

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

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

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

```
00100 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.

```
00050 INPUT "NEXT KEY OR CR",A$:( ""=1000, LEN=8,10)
```

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

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

DIRECTIVES

INPUT RECORD

INPUT RECORD

FORMAT

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

where:

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

DESCRIPTION

The INPUT RECORD and READ RECORD directives are identical in all respects, except that INPUT RECORD is not allowed for reads from 1/2 inch tape; only READ RECORD can be used. INPUT RECORD is generally used to read from the terminal (using a SIZ=clause) and READ RECORD is generally used to read a record from a file.

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

00010 INPUT RECORD(2,ERR=00100,SIZ=5)A\$

DIRECTIVES

IOLIST

IOLIST

FORMAT

IOLIST argument list {,IOL=stno}

where:

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

DESCRIPTION

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

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

The IOLIST statement cannot be part of a compound statement.

EXAMPLES

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

DIRECTIVES

LET

LET

FORMAT

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

DESCRIPTION

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

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

EXAMPLES

0010 LET A=2

0010 B=5,Q=2

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

DIRECTIVES

LIB

LIB

FORMAT

```
LIB "String Expression" {,ERR=stno} {argument list}
                        {,RET=libret}
```

where:

String Expression - the name of a C program being called.

argument list - one or more lib arguments
 separated by commas. (Lib
 arguments are described below.)

```
libret      - a simple numeric variable or a
              numeric array element.
```

DESCRIPTION

The LIB directive is used to transfer program control from a BASIC program to a program written in C language. This directive is similar to the CALL directive which transfers control from one BASIC program to another. The error branch is provided to allow recovery in case the C program is not found. Errors occurring within the called C program do not activate this error branch.

Just as arguments such as variables and arrays may be passed from the calling program to the called program, so may they be passed from a BASIC to a C program. However, certain identifying elements must be specified to make the arguments compatible with the C program.

The argument list consists of one or more lib arguments, separated by commas. A lib argument consists of a lib descriptor, followed by a colon, followed by a numeric or string expression. The lib descriptor is necessary because the data must be transformed from the way it is stored in BASIC into the equivalent C data type.

The descriptors also determine whether or not the passed arguments that are being modified in the called subprogram may be returned in the modified form. For instance, the passed argument A may be modified and returned, but the passed argument A+B may not. Only data types identified with an asterisk (*) can be returned to the BASIC program in a modified form.

The lib descriptors used to identify the type of data are integer, double, string (char), null-terminated string and pointers to data addresses of these types. Refer to the following list:

DIRECTIVES

LIB (continued)

LIB (continued)

<u>LIB DESCRIPTOR</u>	<u>C ARGUMENT</u>
I	- integer
D	- double
*I	- pointer to integer
*D	- pointer to double
*S	- pointer to string
*Z	- pointer to null-terminated string which BASIC null-terminates again when passed

The following are valid lib arguments: I:2, *I:A, D:NUM(A\$), *D:A+ABS(B-C), *S:STR(A:"0000"), *Z"A\$. The first four examples allow only numeric expressions; the last two allow string expressions.

LIB can pass an argument between programs either as the argument itself or as an address pointer to the argument. In LIB syntax (as in C syntax) an asterisk ("*") denotes a pointer. For example, the lib argument *I:7 indicates that when the argument 7 is passed, its memory address is sent to the called program; the value is not passed. The lib argument I:7, on the other hand, passes the value 7 to the C program, not a pointer.

The following list shows the kinds of C program arguments that correspond to each lib descriptor:

<u>LIB DESCRIPTOR</u>	<u>C ARGUMENT</u>
I	char, short, int
D	float, double
I	short, int*
D	float, double*
S	char
Z	char

A "double" is a double-precision, floating-point number, usually with fourteen digits of precision, and stored as a 64-bit number. A "float" is a floating-point number with six digits of precision, stored as a 32-bit number. A "short" is essentially the same as "int". The "int" is simply an integer number. A string character is identified by "char".

DIRECTIVES

LIB (continued)

LIB (continued)

For a parameter or argument to be modifiable, it must satisfy the requirements of both BASIC and C. To satisfy the requirements of BASIC, refer to Table 4-1 for the CALL directive. To satisfy the requirements of C syntax, a parameter or argument that is to be modified by the subprogram must be passed by its address, i.e., as a pointer.

The only difference between parameters that CALL can modify and parameters LIB can modify is that LIB can modify a numeric array element (such as H(4)) whereas CALL cannot modify a numeric array element. Unlike CALL, LIB cannot pass an entire array (such as A(ALL)).

LIB converts its numeric arguments to the format a C program expects to receive. Arguments with a lib descriptor I or *I are converted to C integers, those with D or *D are converted to C doubles. Strings are always passed as pointers, and no conversion takes place.

Because strings are always passed by pointers to their addresses, the C program could overwrite the data at that address. If the string is a simple BASIC variable, the overwriting done by the C subroutine becomes part of that variable. For example, if the lib argument *S:A\$ is used to call a C program, the contents of A\$ may be altered when returned to the calling program.

The *Z descriptor is used to coordinate the passing of strings to a C program. It may be necessary because of the C convention that terminates strings with a null. BASIC will null terminate a string, such as A\$, that is passed by the lib descriptor *Z. Then when the C program returns, the length of A\$ will be adjusted to the number of characters currently preceding the first null (not including the null).

The optional RET= parameter is provided so that the integer or double value returned by many C programs can be passed to BASIC in the numeric variable or numeric array element when control is returned to the calling program.

DIRECTIVES

LIB (continued)

LIB (continued)

EXAMPLE

LIB "circle", I:A+B, *D:C, *S:A\$, *Z:B\$, RET=X

The C program "circle" is called with these parameters:

A+B (passed as an integer)
C (passed on the stack as a pointer to a double)
A\$ (passed by address on the stack)
B\$ (passed by address on the stack--when returned the length will be adjusted to the number of characters preceding the first null)
X (when returned to the calling BASIC program the value will be placed in the BASIC numeric variable X)

The arguments passed may appear in the C program "circle" in the following manner:

circle(a,b,c,d)

<u>LIB</u>		<u>C</u>
I:A+B	will become	int a
*D:C	will become	double *b
*S:A\$	will become	char *c
*Z:B\$	will become	char *d

DIRECTIVES

LIST

LIST

FORMAT

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

where:

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

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

DESCRIPTION

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

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

When listing to a disk file, the file must be an INDEXED, SERIAL or STRING file with at least as many records as there are lines in the program that are to be listed. The number of records is not required for the STRING file.

The rules for determining which line or lines are to be listed are the same as for DELETE in this section. See the following examples:

EXAMPLES

>LIST - lists all statements.

>LIST 10 - lists statement 10.

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

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

>LIST ,100 - list all statements through 100.

DIRECTIVES

LIST (continued)

LIST (continued)

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

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

>LOAD "THUMB"

READY

>INDEXED "TOE",100,80,0,0

>OPEN (1)"TOE"

>LIST (1)

- this routine sets up the Indexed file "TOE", opens it, and copies the statements in "THUMB" to "TOE."

DIRECTIVES

LOAD

LOAD

FORMAT

LOAD "prog id"

DESCRIPTION

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

When a LOAD command is issued, the current program in the user area is deleted, all FOR/NEXT/GOSUB/SETERR/SETESC return addresses are cleared, precision is set to 2, the ERR variable is reset to 0, and the program is READ into the user area. The program can then be executed or modified. The execution of a LOAD command has no effect on the user data area.

If insufficient program area is available, an ERROR 19 (PROGRAM SIZE) displays. The program area is not cleared until it has been determined that the specified program can be loaded.

If a corrupted or miscoded program is loaded, and the BASIC interpreter detects the error, the user program and variable stack area is cleared (as with START), and an ERROR 30 is returned.

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

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

129

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

EXAMPLE

```
>LOAD "INZONE"
```

Loads the file name INZONE in the user's current working directory.

```
>LOAD"/absolute/path_name/INZONE"
```

loads the file from the specified directory, regardless of what the user's current working directory is.

DIRECTIVES

LOCK

LOCK

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 locked file is released by an UNLOCK or CLOSE statement.

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

EXAMPLE 00100 LOCK (1,ERR=00200)

DIRECTIVES

MERGE

MERGE

FORMAT

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

where:

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

DESCRIPTION

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

The statements of the two programs are merged together. If both programs have a statement with the same statement number, the one in the 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, an ERROR 2 (END OF FILE) is displayed upon reaching a record in the file that contains no statement number. If you try to then run the merged program an ERROR 21 (STATEMENT NUMBER MISSING) is displayed.

MERGE cannot be used in a public program.

EXAMPLE

Follow these steps to perform a MERGE:

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

```
>LOAD "PGM1"

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

DIRECTIVES

MERGE (continued)

MERGE (continued)

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

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

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

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

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

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

DIRECTIVES

MERGE (continued)

MERGE (continued)

5. LIST the combined programs:

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

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

NOTE

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

DIRECTIVES

NEXT

NEXT

FORMAT

NEXT control variable

where:

control
variable = the variable to be incremented or
decremented, if the step value is
negative.

DESCRIPTION

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

See FOR/NEXT in this section.

EXAMPLE

See FOR/NEXT in this section for examples.

DIRECTIVES

ON/GOSUB

ON/GOSUB

FORMAT

ON expr GOSUB stno a {,stno b} {,stno c}...{,stno n}

where:

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

DESCRIPTION

The ON/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 numeric value of the expression, and the relative position of the statement numbers after the GOSUB determines which statement number is to be executed next. During execution, the value of the expression must be an integer. Execution of the subroutine continues until a RETURN is executed.

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

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

DIRECTIVES

ON/GOSUB (continued)

ON/GOSUB (continued)

EXAMPLE

```
00100 ON X GOSUB 0200,0300,0400,0500
```

-if X=0 or less (negative), the next statement executed is 00200.

-if X=1, the next statement executed is 00300.

-if X=2, the next statement executed is 00400.

-if X=3 or more, the next statement executed is 00500.

The ON/GOSUB principle may be used in conjunction with the SETERR directive and the ERR variable described in Chapter 6. The SETERR directive may be initiated early in the program so that errors that are not trapped within a statement may be controlled.

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

EXAMPLE

```
00020 SETERR 9000
```

```
09000 ON ERR(12, 14, 17) GOSUB 09100, 09200, 09300, 09400
```

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

<u>ERR</u>	<u>NEXT STATEMENT EXECUTED</u>
12	9200
14	9300
17	9400
OTHER	9100

DIRECTIVES

ON/GOTO

ON/GOTO

FORMAT

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

where:

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

DESCRIPTION

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

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

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

DIRECTIVES

ON/GOTO (continued)

ON/GOTO (continued)

EXAMPLE

0100 ON X GOTO 0200,0300,0400,0500

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

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

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

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

The ON/GOTO principle may be used in conjunction with the SETERR directive and the ERR variable described in section 6. The SETERR directive may be initiated early in the program so that errors that are not trapped within a statement may be controlled.

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

EXAMPLE

0020 SETERR 9000

9000 ON ERR(12, 14, 17) GOTO 9100, 9200, 9300, 9400

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

<u>ERR</u>	<u>NEXT STATEMENT EXECUTED</u>
12	9200
14	9300
17	9400
OTHER	9100

DIRECTIVES

OPEN

OPEN

FORMAT

```
OPEN {INPUT} (fileno/devno {,ERR=stno} {,ISZ=recsz}  
             {,CLASS="string expr"} {,OPTS="string expr"}  
             {SEQ=sequence number}) "file/device ID"
```

where:

ISZ
(record
size) = an arithmetic expression representing a temporarily redefined record size for a file. The file is accessed as if it were an Indexed file with a record size equal to the arithmetic expression. The ISZ= option is used with READ RECORD and WRITE RECORD to handle multiple records or partial record. 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 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.

CLASS= = the name of an entry in the "/etc/class"
(string file that specifies the attributes of a
expression) print job. See CLASS= in section 7 of
this manual

OPTS= = a string expression representing all the
(string attributes desired for a given print job.
expression) The string must be in the same format as
an entry in the "/etc/class" file. See
OPTS= in section 7 of this manual.

SEQ= = used for magnetic tape, SEQ indicates the
(sequence file to be accessed by specifying the
number) sequential number of that file on the tape.

DIRECTIVES

OPEN (continued)

OPEN (continued)

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, or
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 1 - 63) at any given time. File number 0 is always opened to the terminal, so cannot be opened or closed through BASIC.

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

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

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

EXAMPLE

```
00010 OPEN (1)"ADOOR"
```

```
00020 OPEN INPUT (2,ERR=00050) "DONTCHANGE"
```

```
00030 OPEN (7, OPTS=A$)"LP"
```

DIRECTIVES

PRECISION

PRECISION

FORMAT

PRECISION expr

where:

expr = an integer value between 0 and 14.

DESCRIPTION

The PRECISION directive is used to change the number of places of rounding. PRECISION is always reset to 2 when a RESET, END, STOP, RUN or LOAD statement is executed and when BEGIN or CLEAR are executed without a variable list.

EXAMPLE

```
00010 PRECISION 2
00020 LET A=.55555
00030 FOR I=0 TO 5
00040 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:

```
00020 LET A=0+.55555
```

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

DIRECTIVES

PRECISION (continued)

>RUN

1

.6

.56

.56

.56

.56

00100 REM "CODE 3-6"

00200 PRECISION 2

00220 LET A=.5,B=.01,C=4

00230 LET D=A*B*C,E=B*C*A

00240 PRINT D,E

>RUN

.04 .02

PRECISION (continued)

DIRECTIVES

PREFIX

PREFIX

FORMAT

PREFIX "prefix_list"

where:

prefix_list = a list of directories to be searched when accessing files for which a full path name has not been given. The length of the list is limited to 255 characters and cannot include more than 9 directories.

DESCRIPTION

The PREFIX directive defines a set of directories called a prefix list. When searching for various files, the system restricts itself to the specific set of directories. The first directory in this list becomes the user's current working directory. If a file is referenced without a full path name, it is first searched for in the current working directory.

If the PREFIX list is the null string, the previous PREFIX command is undone, and the current working directory changes accordingly. Executing PREFIX with a null string argument is required to make logical disks available to the user. For more information on the alternate directory access methods, refer to Appendix A of this manual.

When the user first enters BASIC, an implicit PREFIX is executed: the user's prefix list is automatically set to the current working directory.

Alternate directory names (not on the same nodal path) in the prefix list are separated by spaces, as follows:

```
"/usr/dir2 /sys/tmp/ /usr/fred"
      |           |
      space      space
```

Each directory, in the sequence specified in the prefix list, is searched for the absolute filename. A file not found error is given only when a search for the file in every one of the directories fails.

File creation and deletion operations are attempted only in the first (rightmost) directory in the prefix list.

EXAMPLES

1. PREFIX "/usr/fin/acct/rec" -- instructs a file search to follow the specified nodal path.
2. PREFIX "/usr/dir/ /sys/tmp/" -- note that each directory may optionally be followed by a slash.

DIRECTIVES

PRINT

PRINT

FORMAT

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

where:

list = one or more numeric or string constants or variables or arithmetic or string expressions defining the data items to be printed or displayed. Each such data item can employ a positioning expression and/or a form expression as required. Mnemonic constants can be inserted at points where I/O device control is required.

= a comma can be used at the end to suppress the otherwise automatic line feed (LF).

NOTE

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

DESCRIPTION

The PRINT directive is used to PRINT to a file or device. PRINT does not add a character to either the beginning or end of string fields. However, it will precede a numeric field with a blank.

Use of PRINT suppresses automatic generation of a field mark (line feed (LF) character). A comma (,) at the end of all items suppresses the terminating LF character.

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

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

EXAMPLE

```
00130 PRINT (3,ERR=00340)@(5),A$,@(35),B:X$
```

DIRECTIVES

PRINT RECORD

PRINT RECORD

FORMAT

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

DESCRIPTION

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

EXAMPLE

```
00130 PRINT RECORD(3,ERR=00340)A$
```

DIRECTIVES

PROGRAM

PROGRAM

FORMAT

PROGRAM "file ID" prog size {, diskno, sectno,
init alloc,add alloc} {,ERR=stno}

where:

prog size	= the maximum size of the program in bytes (can not exceed 32,256 bytes).
diskno	= used only for compatibility with other MAI Basic Four systems; otherwise it is ignored.
sectno	= used only for compatibility with other MAI Basic Four systems; otherwise, it is ignored.
init_alloc	= number of bytes initially allocated to a file when the file is defined.
add_alloc	= additional number of bytes 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 number of bytes.

NOTE

If init_alloc and add_alloc are not specified, then the program size space is not allocated on the disk. Space is allocated only as needed.

DIRECTIVES

PROGRAM (continued)

PROGRAM (continued)

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. For more information on program files, refer to the User Guide.

EXAMPLES

>20 PROGRAM "NOVA", 2000,255,0,ERR=0100

-defines program "NOVA", with a maximum size of 2000 bytes in the user's working directory. The logical disk specification, 255, and the sector specification, 0, are ignored.

>PROGRAM "DALLAS",3000

-defines a program named DALLAS, with a maximum size of 3000 bytes, in the user's working directory.

DIRECTIVES

READ

READ

FORMAT

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

where:

@ expr,	= horizontal and vertical positioning
espr	expression

NOTE

A comma is inserted before IOL= only when both IOL= and a variable list, or multiple IOL entries, or @ pos construct is used.

DESCRIPTION

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

If a field contains non-numeric information, and the corresponding variable is numeric, an ERROR 26, INCORRECT VARIABLE USAGE, results. A numeric field can be read into a string variable, and a field that has been written as a string, but that contains only valid numeric data, can be read into a numeric variable.

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

For non-terminal devices, string constants, positioning expressions, and mnemonic constants are not allowed.

For more information on file types, see the User Guide.

DIRECTIVES

READ (continued)

READ (continued)

Direct File READ

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

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

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

Examples

Reading and writing a Direct file:

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

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


DIRECTIVES

READ (continued)

READ (continued)

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

DIRECTIVES

READ (continued)

READ (continued)

Sort File READ

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

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

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

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

DIRECTIVES

READ (continued)

READ (continued)

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

DIRECTIVES

READ (continued)

READ (continued)

Reading From Indexed or Serial Files and Peripheral Devices

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

Example:

```
00010 REM "PROGRAM TO PRINT LABELS
00020 BEGIN
00030 OPEN (1)"ADDRES"
00040 OPEN (7)"LP"
00050 READ (1,END=00100)A$,B$,C$,D$
00060 PRINT (7)'FF',A$
00070 PRINT (7)B$
00080 IF LEN(C$)>0 THEN PRINT (7)C$
00090 PRINT (7)D$
00100 GOTO 00050
00110 CLOSE (1)
00120 CLOSE (7)
00130 END
```

String File READ

A string file is a continuous stream of characters. A string file record is considered to be one line; thus records are separated by linefeeds (\$OA\$). A READ RECORD of the string file returns the next line. A READ of a string file returns the first field within that line.

READ with the ISZ= option allows you to READ more than one line at a time from a string file.

DIRECTIVES

READ RECORD

READ RECORD

FORMAT

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

where:

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

DESCRIPTION

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

When READ RECORD is done from the 1/2 inch tape device R0, the string variable must be previously allocated to be at least as big as the record to be read from tape. The reason for this is that the READ RECORD is done directly from the tape into the string variable. This is done to improve the performance of tape reads and to make it possible for the tape to stream.

EXAMPLE

```
00100 READ RECORD(1,END=00900)A$
```

DIRECTIVES

RELEASE

RELEASE

FORMAT

RELEASE {"ghost task ID"}

where:

ghost task ID = the task identifier corresponding to the task identifier used with a prior START command. The Task ID is not used when RELEASE is executed by a task to RELEASE itself.

DESCRIPTION

The RELEASE directive closes files and releases a task's memory. The RELEASE directive can release any ghost task or the process executing the RELEASE, ie., themselves. Releasing oneself will return the user to the originating process.

EXAMPLES

0010 RELEASE "G1"

This directive, used within a program, releases the first ghost task.

>RELEASE

This directive releases the Business BASIC process. If you accessed BASIC through the Initial Menu, you are returned to the Initial Menu, which was the originating process. However, if you chose the Command Interpreter option from the Initial Menu and then accessed BASIC, the RELEASE directive would return you to the Command Interpreter, which was the originating process in this case.

DIRECTIVES

REM

REM

FORMAT

REM {"}remark{"}

where:

remark = a comment

DESCRIPTION

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

EXAMPLE

0100 REM "PROGRAM TO DISEMBARK ARK"

DIRECTIVES

REMOVE

REMOVE

FORMAT

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

DESCRIPTION

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

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

The REMOVE statement must specify a key.

EXAMPLE

0100 REMOVE (1,KEY=K\$)

DIRECTIVES

RENAME

RENAME

FORMAT

RENAME "old_id","new_id"

DESCRIPTION

The RENAME directive permanently changes a file name to a second file name. Any file may be renamed.

If the second file name exists prior to the RENAME directive, an ERROR 12 is generated.

After RENAME is executed, the first file name no longer exists.

EXAMPLE

>LOAD "NEWNAME"

Verifies that "NEWNAME" already exists.

>DELETE

>LOAD "OLDNAME"

>RENAME "OLDNAME","NEWNAME"

!ERROR 12

>ERASE "NEWNAME"

>RENAME "OLDNAME", "NEWNAME"

>LOAD "OLDNAME"

!ERROR 12

Verifies that "OLDNAME" no longer exists.

>LOAD "NEWNAME"

>

DIRECTIVES

RESET

RESET

FORMAT

RESET

DESCRIPTION

The RESET directive performs a low-level system reset that affects only the task that issued the statement. RESET resets the ERR and CTL system variables to zero, and any GOSUB or FOR/NEXT loops that have not been fully executed are reset. Additionally, the RESET statement reestablishes the arithmetic mode at PRECISION 2 and any statement numbers active for SETESC, SETCTL, or SETERR are cleared.

Execution of the RESET statement does not clear the user data area, nor close any open files or devices, nor reset the program execution pointer which identifies the statement to be next executed.

EXAMPLE

0900 RESET

DIRECTIVES

RETRY

RETRY

FORMAT

RETRY

DESCRIPTION

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

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

EXAMPLE

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

DIRECTIVES

RETURN

RETURN

FORMAT

RETURN

DESCRIPTION

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

EXAMPLE

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

DIRECTIVES

RUN

RUN

FORMAT

RUN {"prog ID"}

DESCRIPTION

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

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

>RUN "AMOK"

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

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

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

0400 RUN "PRGM"

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

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

EXAMPLE

>RUN

0400 RUN "AMOK"

DIRECTIVES

SAVE

SAVE

FORMAT

SAVE {"prog id"} {,prog size, diskno, sectno,
init alloc, add alloc}

where:

prog id	= the name of the program.
prog size	= the program file size in bytes (cannot exceed 32,256 bytes). Ignored by the system.
diskno	= retained for compatibility only
sectno	= retained for compatibility only
init_alloc	= retained for compatibility only
add_alloc	= retained for compatibility only

DESCRIPTION

The SAVE directive is used to copy a program from user memory to a Program file on disk. If the program file does not exist already, it is created automatically; the program size, disk number, and so on do not need to be specified for 7.2 BASIC. (The file size and location parameters are retained for level 4 BASIC compatibility only.)

SAVE cannot be used in a public program.

EXAMPLES

>SAVE "/all/my/FAVORITE_STAMPS"

This command copies the contents of user memory to the program file. If the program file does not exist, it is created, and if it is too small, it is enlarged.

>SAVE "GAS",8000,4,0,2000,1000

This command creates a Program file named "GAS" in the user's working directory. Note that all parameters except the file name are ignored by 7.2 BASIC.

DIRECTIVES

SELECT

SELECT

The SELECT directive has been deleted from 7.2 BASIC.

DIRECTIVES

SERIAL

SERIAL

FORMAT

SERIAL "file ID", av_recno, av_recsz {, diskno, sectno,
init_alloc, add_alloc} {,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.
diskno	= used only for compatibility with other MAI Basic Four systems; otherwise it is ignored.
sectno	= used only for compatibility with other MAI Basic Four systems; otherwise it is ignored.
init_alloc	= the number of records initially allocated to the file when the file is defined. This cannot exceed the average number of records.
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 number of records.

NOTE

If init_alloc and add_alloc are not specified, then a file with av_recno is created.

DIRECTIVES

SERIAL (continued)

SERIAL (continued)

DESCRIPTION

The average record size and average number of records parameters are used to define the total space required for the file. However, they do not limit either quantity to that amount. For example, a Serial file defined with 100 records with an average size of 60 bytes can actually contain 200 records of 30 bytes, or 50 records of 120 bytes, or any other combination which totals 6000 bytes. Note that there is a 2 byte system overhead per record.

Rules for using Serial files:

1. The maximum record size for a serial file is 32,767 bytes.
2. The file must be locked in order to WRITE to it; otherwise, an ERROR 13, ILLEGAL FILE USE/ACCESS, results.
3. Indices can be used to access records in a Serial file as they are in an INDEXED file. Record-to-record movement of the index can be forward or backward. There is no need to close and reopen the file to reset the pointer.
4. Each record has a system overhead of two bytes that contain the length of the that plus the preceding record.

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.

>SERIAL "SPOONS", 50, 20, 255, 0

defines a serial file named "SPOONS" with an average number of records equal to 50 and an average record size equal to 20, in the user's working directory. Note that the logical disk number, 255, and the sector number, 0, are ignored.

DIRECTIVES

SETCTL

SETCTL

FORMAT

SETCTL stno

DESCRIPTION

The SETCTL directive is used to cause program control to branch or transfer to another part of the program when the operator enters |CTRL| + 'Y'. When the operator enters this combination, and SETCTL is in effect, branching occurs to the statement number specified in SETCTL, and the ERR variable is set to 126. If no SETCTL is in effect, and the operator enters |CTRL| + 'Y', input is ignored and no action is taken.

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

For 7.2, error stacking has been implemented to preserve the current BASIC error status. For example, if an error sets the system variable ERR to 40, and a |CTRL| + 'Y' is then executed, ERR is set to 126 (if SETCTL has been used). Another error may be generated during the SETCTL routine, setting ERR to 12. Upon returning from the SETCTL routine, ERR is returned to the value 40 it had immediately prior to entering the SETCTL routine. In this way, SETCTL can be used without affecting the way the program runs.

EXAMPLE

00700 SETCTL 14200

DIRECTIVES

SETDAY

SETDAY

FORMAT

SETDAY "string expr"

where:

string expr = 8 characters in the string length.

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, and must be in the format specified for the system. (See the "date" command in the BOSS/IX User Reference Manual for setting the format.) The usual default format is: mm/dd/yy. Incorrect lengths and formats result in an ERROR message. The incorrect length of the string results in an ERROR 46. If the format is wrong, an ERROR 17 results.

Only an operator with system administrator status may use the SETDAY directive. Other users will be issued an ERROR 18.

EXAMPLE

SETDAY "12/31/84"

DIRECTIVES

SETERR

SETERR

FORMAT

SETERR stno

DESCRIPTION

The SETERR directive is used to branch to a general error routine. RETRY can then be used to return to the statement at which the error occurred for 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 error statement.
- SETERR is cleared by a RUN, LOAD, RESET, BEGIN, CLEAR, END or SETERR 0.
- When the system takes the SETERR branch, it automatically performs a SETERR 0 and saves the statement number to RETRY (unless the error occurred on an ERR= branch and returns to the same statement where the error occurred). This allows limited error branching within an error routine without losing the original RETRY address. When the RETRY statement is executed, the SETERR is restored to its original value. This design prevents an error within an error routine causing an infinite loop.
- If an ERR= option (that does not branch to itself) is executed within an error routine, the RETRY address is set to that statement (losing the original RETRY address) and the SETERR is not reset.
- If a SETERR is used for handling errors in a routine, a SETERR 0 should be executed after completion of the routine. This protects future errors from falling under control of the first SETERR.

EXAMPLE

00010 SETERR 00100

DIRECTIVES

SETESC

SETESC

FORMAT

SETESC stno

DESCRIPTION

The SETESC directive is used to prevent an operator from escaping out of a program. SETESC causes program control to transfer to the specified statement number when ESCAPE is pressed. The operating system executes a GOSUB to the SETESC line number and begins processing and sets the system variable ERR to 127. Following a RETURN, the operating system resumes processing at the point the SETESC branch was taken.

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

For 7.2, error stacking has been implemented to preserve the current BASIC error status. For example, if an error sets the system variable ERR to 40, and an |ESC| is then executed, ERR is set to 127 (if SETESC has been used). Another error may be generated during the SETESC routine, setting ERR to 12. Upon returning from the SETESC routine, ERR is returned to the value 40 it had immediately prior to entering the SETESC routine. In this way, SETESC can be used without affecting the way the program runs.

EXAMPLE

```
00050 SETESC 09000
00059 REM "ESCAPE KEY WILL BE PRESSED DURING EXECUTION OF 60
00060 LET A=A+1,B=B+1,C=C+1
00070 GOTO 00060
09000 REM
09001 REM "ESCAPE ROUTINE"
09002 REM
09003 PRINT "YOU CANNOT ESCAPE"
09004 RETURN
```

DIRECTIVES

SETTIME

SETTIME

FORMAT

SETTIME numeric expr

where:

numeric expr = an expression representing a value between 0 and 24 entered in decimal form (i.e., 13.50 = 1:30 p.m.). The following formula should be used to determine the proper format:

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

where:

H = Hours
M = Minutes
S = Seconds

DESCRIPTION

SETTIME is used to change the value of the TIM system variable. The TIM variable is set to 0 whenever the system is loaded. Only a user with administrator status may use the SETTIME; others are returned an ERROR 18.

EXAMPLE

```
00010 REM "PROGRAM TO SET TIME AND DAY"
00020 BEGIN
00030 INPUT (0,ERR=00030)"HOUR = ",H:(23)
00040 INPUT (0,ERR=00040)"MINUTES = ",M:(59)
00050 INPUT (0,ERR=00050)"SECONDS = ",S:(59)
00060 PRECISION 4
00070 SETTIME H+M/60+S/3600
00080 INPUT (0,ERR=00080)"MONTH= ",M:(12)
00090 IF M<1 THEN GOTO 00080
00100 INPUT (0,ERR=00100)"DAY = ",D:(31)
00110 IF D<1 THEN GOTO 00100
00120 IF POS(STR(M:"00")="04060911",2)<>0 AND D>30
      THEN GOTO 00100
00130 IF M=2 AND D>29 THEN GOTO 00100
00140 INPUT (0,ERR=00140)"YEAR = ",Y:(99)
00150 IF Y<1 THEN GOTO 00140
00160 IF FPT(Y/4)<>0 AND M=2 AND D>28 THEN GOTO 00100
00170 SETDAY STR(M:"00")+ "/" +STR(D:"00")+
      "/" +STR(Y:"00")
00180 REM "PRINT THE DATE AND TIME"
00190 PRECISION 4
00200 LET T=TIM, H=INT(T), S=INT(FPT(T)*3600),
      M=INT(S/60), S=S-M*60
00210 PRINT "DATE IS",DAY
00220 PRINT "TIME IS",H:"00",":",M:"00",":",S:"00"
00230 STOP
```

DIRECTIVES

SETTRACE

SETTRACE

FORMAT

SETTRACE {(fileno/devno)}

DESCRIPTION

The SETTRACE directive is used to initiate the listing of statements as they are executed. It is especially useful when debugging a program that appears to be branching in an unforeseen or undesirable manner. The resulting listing delineates the exact sequence in which program statements are being executed. The SETTRACE command can be used as a statement within the program at selected points until the program is debugged. The output of the SETTRACE is in the same format as the LIST command.

SETTRACE can be entered in console mode to begin the listing of executed statements. In any case, the listing continues until terminated by execution of an ENDTRACE, END or STOP.

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

Note that there is no difference between the display of traced statements which come from a main program and statements which come from a called program.

EXAMPLE

```
00010 FOR I=1 TO 3
00020 LET A=I+1; NEXT I
>SETTRACE
```

```
>RUN
```

```
00010 FOR I=1 TO 3
00020 LET A=I+1; NEXT I
00020 LET A=I+1; NEXT I
00020 LET A=I+1; NEXT I
END
```

```
>READY
```


DIRECTIVES

SORT

SORT

FORMAT

SORT "file ID", keysz, recno {, diskno, sectno,
init_alloc, add_alloc} {, ERR=stno}

where:

keysiz	= the size of a key in a keyed file; minimum=1, maximum=56.
recno	= the maximum number of records for the file (cannot exceed 8,388,608).
diskno	= used only for compatibility with other MAI Basic Four systems; otherwise it is ignored.
sectno	= used only for compatibility with other Basic Four systems. Otherwise , it is ignored.
init_alloc	= the number of records initially allocated to the file when the file is defined. This cannot exceed the average number of records.
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 number of records.

NOTE

If init_alloc and add_alloc are not
specified, then space for recno is
allocated at creation time.

DIRECTIVES

SORT (continued)

SORT (continued)

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.

EXAMPLES

0100 SORT "ACUTE",15,100

```
00010 REM "PROGRAM TO INPUT CUSTOMERS"
00020 BEGIN
00030 GOTO 00070
00040 REM "THE TWO FILES ARE"
00050 DIRECT "CUST",5,1000,64
00060 SORT "ZIP",10,1000
00070 OPEN (1)"CUST"
00080 OPEN (2)"ZIP"
00090 INPUT (0,ERR=00090)"CUSTOMER NUMBER",A:(99999)
00100 LET A$=STR(A:"00000")
00110 INPUT (0,ERR=00110)"CUSTOMER NAME",A1$(LEN=0,20)
00120 INPUT (0,ERR=00120)"CUSTOMER ZIP",Z:(99999)
00130 WRITE (1,KEY=STR(A:"00000"))A,A1$,Z
00140 REM "FOLLOWING EXTRACTS RECORD TO ENABLE
      OBTAINING OF INDEX VALUE"
00150 EXTRACT (1,KEY=A$)*
00160 LET I=IND(1)
00170 WRITE (2,KEY=STR(Z:"00000")+STR(I:"00000"))
00180 GOTO 00090
```

```
00010 REM "PROGRAM TO READ ZIP FILE AND PRINT CUSTOMER
      NAME AND NUMBER"
00020 BEGIN
00030 OPEN (1)"CUST"
00040 OPEN (2)"ZIP"
00050 LET A$=KEY(2,END=00100)
00060 READ (1,IND=NUM(A$(6)))A,A1$
00070 PRINT "CUST #",A,@(15),"CUSTOMER NAME",
      A1$,@(33),"ZIP",A$(1,5)
00080 READ (2)
00090 GOTO 00050
00100 END
```

DIRECTIVES

START

START

FORMAT

START {pages} {,ERR=stno} {,"prog ID"} {,"task ID"}
{"driver ID"}

where:

- pages - the number of pages assigned to the task
 (min=10; max=65535).
- prog ID - the full path name of the program.
- task ID - the task identifier refers only to those
 ghost tasks that can be STARTed by the
 user. Ghost tasks are numbered G0, G1,
 etc.
- driver ID - the name of the driver for the new task.

DESCRIPTION

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

The START directive will only affect the currently executing task (ie. the one executing the START directive) or a ghost task depending upon the statement. It can be executed either in console mode or program mode.

If no number of pages is assigned, the amount of memory assigned to the task is not changed.

Note that the string used to identify the program name must be a full path name.

The following paragraphs describe how START works with zero, one, two, or three optional string parameters:

No Option

No optional parameters: If only START and the number of pages are specified, the BASIC user area is cleared and resized to the number of pages. Each page contains 256 bytes (1/4 K).

Example: START 40 allocates 40 pages (48 pages on 7.2), or 10 K bytes (12 KB on 7.2), to the BASIC user area.

One Option

One optional parameter: It can be either the name of a task to be started or the name of a BASIC program to be run. The system first checks to see if it is a task.

Example: START 40, "G1" will start the task G1 with 40 pages.

DIRECTIVES

START (continued)

START (continued)

Two Options

Two optional parameters: The system tries to use the first parameter as a task and the second as a communication link (the driver). If there is an error when attempting to use the first parameter as a task or the second as a driver, the system assumes the second to be a task and the first a BASIC program to be run.

Example: START 20, "G3", "X25" will START a task named G3 with 20 pages, using the X25 driver.

Three Options

Three optional parameters: The first parameter is the BASIC program to be run, the second is the task, and the third is the driver.

Example: START 60, "/full/path_name/PROG", "G2, "X25" will START the task G2, using the X25 driver, running the BASIC program "PROG".

NOTE

In 7.2 BASIC the amount of memory allocated by the START directive is increased by 20%. This corresponds to the average increase in program size between 7.1 and 7.2. For example, START 40 allocates 48 pages of memory on 7.2 ($1.20 * 40 = 48$).

DIRECTIVES

STOP

STOP

FORMAT

STOP

DESCRIPTION

The STOP directive is used to terminate a program at any point other than the end of that program. The termination process resets the program execution pointer to the first statement of the program, closes all open files and devices, performs a RESET operation, and returns the terminal to console mode.

The logical termination point established by the STOP statement does not discontinue MERGE operations, and for this reason the STOP statement should be used for all program terminations except the one that occurs at the end of the program.

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

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

EXAMPLE

06510 STOP

DIRECTIVES

STRING

STRING

FORMAT

STRING "file ID" {,diskno}{,ERR=stno}

where:

diskno - used only for compatibility with other MAI
Basic Four systems; otherwise it is
ignored.

DESCRIPTION

The STRING directive defines a UNOS string file. This file differs from other Business BASIC files in that there is no implied file record structure. That is why there is no need to specify any size parameters for the file; the memory management unit allocates space as needed.

EXAMPLE

>20 STRING "ME_ALONG", ERR=00100

-defines string file "ME_ALONG".

DIRECTIVES

SYSTEM

SYSTEM

FORMAT

SYSTEM "command string"

DESCRIPTION

The SYSTEM directive allows execution of a command mode command from a BASIC program or BASIC console mode. It is essentially the same as the "!" directive, except that it is followed by a proper string expression rather than an unquoted command string.

Unlike the EXECUTE directive, SYSTEM can be executed from console mode as well as from a BASIC program, and can be executed from a public program.

Since the argument is a proper string expression, a command can be dynamically constructed and then executed.

EXAMPLES

>SYSTEM "who"

```
01100 A$="mcssave cs "  
01120 INPUT B$, "Type a file name to save, <CR> to end: "  
01130 IF B$<>" " THEN A$ = A$ + B$ + " ";GOTO 01200  
.  
.  
.  
  
01200 SYSTEM A$
```

This page left blank intentionally.

DIRECTIVES

TABLE

TABLE

FORMAT

TABLE hexadecimal string

DESCRIPTION

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

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

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

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

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

The 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.

DIRECTIVES

TABLE (continued)

TABLE (continued)

NOTE

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

EXAMPLE

The following paragraphs provide 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 value. There are usually numerous holes in the table (marked with an *). These holes must be filled with some hexadecimal values, such as blanks, or another hexadecimal value that is not in the output character set, such that they can be later removed. Once this table is complete, it can be written in BASIC by appending the mask byte to the front of the hexadecimal values.

DIRECTIVES

TABLE (continued)

TABLE (continued)

Table 4-2. Table Statement Table

COLUMN 1	COLUMN 2	COLUMN 3	COLUMN 4	COLUMN 5
EBCDIC CHARACTER	EBCDIC BINARY VALUE	MASKED DECIMAL VALUE	ASCII CHARACTER EQUIVALENT	OUTPUT HEX VALUE
A	1100 0001	1	A	41
B	1100 0010	2	B	42
C	1100 0011	3	C	43
D	1100 0100	4	D	44
E	1100 0101	5	E	45
F	1100 0110	6	F	46
G	1100 0111	7	G	47
H	1100 1000	8	H	48
I	1100 1001	9	I	49
J	1101 0001	17	J	4A
K	1101 0010	18	K	4B
L	1101 0011	19	L	4C
M	1101 0100	20	M	4D
N	1101 0101	21	N	4E
O	1101 0110	22	O	4F
P	1101 0111	23	P	50
Q	1101 1000	24	Q	51
R	1101 1001	25	R	52
S	1110 0010	34	S	53
T	1110 0011	35	T	54
U	1110 0100	36	U	55
V	1110 0101	37	V	56
W	1110 0110	38	W	57
X	1110 0111	39	X	58
Y	1110 1000	40	Y	59
Z	1110 1001	41	Z	5A
Masked Decimal Value				
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				
through 63				
Output Hex Value 20*20*20*20*53 54 55 56 57 58 59 5A 20				
are all 20's				
0100 TABLE 3F20414243444546474849202020202020				
204A4B4D4E4F5051522020202020202020				
535455565758595A2020202020202020				
202020202020202020202020202020				

DIRECTIVES

TABLE (continued)

TABLE (continued)

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.

DIRECTIVES

UNLOCK

UNLOCK

FORMAT

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

DESCRIPTION

The UNLOCK directive allows other tasks to access files previously locked. A locked file automatically becomes unlocked when the file is closed using the CLOSE directive.

EXAMPLE

0200 UNLOCK (1,ERR=0200)

DIRECTIVES

VMERGE

VMERGE

FORMAT

VMERGE "file id"

where:

file id = the name of the String file to be merged.

DESCRIPTION

The VMERGE command is used to retrieve a program in LIST format from any other input device and to add that program to the program currently existing in the user memory area.

The statements of the two programs are merged together. If both programs have a statement with the same statement number, the one in the String file replaces the one in the user memory area.

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 into the program in numerical order according to its statement number.

VMERGE cannot be used in a public program.

Follow these steps to perform a VMERGE:

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

```
>LOAD "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 "ROPE", 1  
>OPEN (1) "ROPE"  
>LIST (1)  
>END
```

DIRECTIVES

VMERGE (continued)

VMERGE (continued)

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

```
>LOAD "PGM2"  
>LIST  
00010 REM "PGM2, LINE 10"  
00030 REM "PGM2, LINE 30"  
00040 REM "PGM2, LINE 40"
```

4. Enter the VMERGE command:

```
>VMERGE "ROPE"
```

5. LIST the result:

```
>LIST  
00010 REM "PGM1, LINE 10"  
00020 REM "PGM1, LINE 20"  
00030 REM "PGM1, LINE 30"  
00040 REM "PGM2, LINE 40"
```

Statements 10 and 30 appear in both PGM1 and PGM2; the merged program (PGM1) supersedes the program in user memory (PGM2), so statements 10 through 30 are from PGM1.

Statement 40 was from PGM2 only, so it remains in user memory.

DIRECTIVES

WAIT

WAIT

FORMAT

WAIT number of seconds.

where:

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

DESCRIPTION

The WAIT directive is used to suspend task execution for a specified number of seconds. The pause can range from 0 to 255 seconds. The actual elapsed time waited by the task is dependent on system load factors.

NOTE

Level 4 BASIC adds approximately .9 seconds to every WAIT. Level 7 BASIC does not. If you execute a WAIT 0 on this system, there is no pause.

EXAMPLE

0200 WAIT 2

DIRECTIVES

WRITE

WRITE

FORMAT

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

NOTE

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

DESCRIPTION

The WRITE directive is similar to the PRINT directive except that the system automatically appends a one byte line feed (\$OA\$) as a record terminator after every record field it outputs. Also, unlike the PRINT directive, WRITE will not precede a numeric field with a blank.

NOTE

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

Direct File WRITE

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

If the key already exists, the new record is written over the old record (only if DOM= clause was not specified). The operation is then complete. If the key does not exist, the system must find space for the key and data.

For more information on file types, refer to the BOSS/IX User Reference Manual.

DIRECTIVES

WRITE RECORD

WRITE RECORD

FORMAT

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

where:

string variable = the string variable to be written

DESCRIPTION

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

EXAMPLE

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


OVERVIEW

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 forementioned. 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 fifty-two other functions using the FNX directive.

FUNCTIONS

ABS
(ABSOLUTE VALUE)

ABS
(ABSOLUTE VALUE)

FORMAT

ABS (numeric 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) - assigns the value 6 to X

FUNCTIONS

AND
(COMBINE STRINGS)

AND
(COMBINE STRINGS)

FORMAT AND (string expr, string expr)

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

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

EXAMPLE LET X\$=AND(\$OF\$, \$DC\$)

then: \$OF\$ = 0000 1111
 \$DC\$ = 1101 1100

PRINT HTA(X\$) = \$OC\$ = 0000 1100

FUNCTIONS

ASC
(STRING TO DECIMAL)

ASC
(STRING TO DECIMAL)

FORMAT ASC (string expr {ERR=stno})

DESCRIPTION

The ASC function converts a single string character to a decimal number. If the string expression is longer than one character, the value returned is the ASC of the first character in the string.

The function also converts a single (two character) hexadecimal number to a decimal number. As with the character string, the function uses only the leftmost hexadecimal number when the input includes more than two character (unless it is substringed).

For this function, the high order bit is inconsequential. The system assumes preceding hexadecimal zeroes (\$00\$).

It is possible to substring the results of the ASC function to get the ASC of another position:

```
0100 LET A$="EAGLE"  
0110 PRINT ASC(A$(2))  
65
```

EXAMPLES

```
0500 LET X=ASC("A")        - returns a value of 65 to X  
0500 LET X=ASC("1")        - returns a value of 49 to X  
0500 LET X=ASC("$79$")     - returns a value of 121 to X  
0500 LET X=ASC("$79C1$") - returns a value of 121 to X
```

FUNCTIONS

ATH
(ASCII TO HEXADECIMAL)

ATH
(ASCII TO HEXADECIMAL)

FORMAT

ATH (string expr {,ERR=stno})

DESCRIPTION

The ATH function converts pairs of characters (representing a hexadecimal number) in a string to a hexadecimal value. The string can contain only the numbers 0-9 and letters A-F.

If the string contains an odd number of characters, a zero is added to the left of the first character before the conversion takes place.

```

100 LET A$="41"      stores: 00000100      00000001 (2 bytes)
110 LET B$=ATH(A$)    stores:      0100 +      0001 (1 byte)
120 PRINT B$          returns 01000001 and prints A
                        one
                        ASCII
                        character
                        ---->      |-----+64
                                   |-----+65 = A

```

The system cannot print a hexadecimal value. So, when asked to print the result of the ATH function, the system returns the single ASCII character associated with each ASCII character string input as a hexadecimal number. The ATH function thus resembles the CHR function, but requires more processing time.

The result of the ATH function, when placed in a variable, may be substringed, as in the following example:

EXAMPLE

```
100 LET A$=ATH("4541474C45")
110 PRINT A$           returns      EAGLE
120 PRINT A$(2,1)      returns      A
```


FUNCTIONS

BIN
(BINARY)

BIN
(BINARY)

FORMAT

BIN (numeric expr, length)

where:

length = 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, the string is right justified and truncated.

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".

"Two's complement" means that the bit structure of the positive integer is inverted bit by bit and then "1" is added. For example, "-1" takes the binary bit structure "0001", inverts each bit to form "1110", and adds "1". The result is "1111", which indicates "-1".

The Binary to Hexadecimal Conversion Table is as follows:

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

EXAMPLES

```
LET X$=BIN(50,2)      - X$ is $0032$
LET X$=BIN(1024,2)    - X$ is $0400$
LET X$=BIN(-50,2)     - X$ is $FFCE$
LET X$=BIN(193,1)     - X$ is $C1$
```

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

FUNCTIONS

CHR
(NUMERIC TO ASCII)

CHR
(NUMERIC TO ASCII)

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

DESCRIPTION The CHR function converts the numeric expression to an ASCII character. The number must be between 0 and 128. The allowable range of numbers is 0 - 255, but numbers beyond 128 will yield unpredictable results depending on what characters were assigned to each number, if any.

EXAMPLES 00100 LET X\$=CHR(65)
 - stores "A" in X\$
00100 LET X\$=CHR(49)
 - stores "1" in X\$

FUNCTIONS

CPL
(COMPILE)

CPL
(COMPILE)

FORMAT

```
CPL (string expr {, string expr} {, sting variabel}
    {, 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, if the statement number is included.

In Level 7 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.

If it is desired that the CPL reflect the environment of the second, or target program, CPL may take a second string parameter. This string contains information about the target program. The information is in the format of the eight variable tables described in Appendix C: 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 DEF statement number table, and the DEF id table. Once these descriptions are understood, the following examples will help.

EXAMPLES

1. This example shows the result of a compiled statement that does not require a variable table:

A\$=CPL("1 END") - A\$ = \$0005000143\$

```
| | |  
| | |  
| | |-compiled "END"  
| |-statement number in  
|   binary  
|-length of compiled  
  statement in binary
```

To print the value of A\$ as listed, enter:

```
PRINT HTA(A$)
```

FUNCTIONS

CPL (Continued)

CPL (Continued)

2. The following example applies to both the CPL and LST functions:

```
00005 DIM MISSING$(12)
00010 LET ARG$=$0000000F46495253545345434F4E440100000000800
010002000C000A00040009000F00000004000C000C40000000060004000C4
000A0008060153455448$
00020 LET B$=CPL("10 FIRST=SECOND", ARG$, MISSING$)
00030 LET C$=CPL("10 PRINT FNSETH(7)", ARG$, MISSING$)
00060 ARG$(5,1)="G"
00070 PRINT LST(B$, ARG$)
00080 ARG$(LEN(ARG$),1)="X"
00090 PRINT LST(C$,ARG$)
```

The above program prints:

```
00010 LET GIRST=SECOND
00010 PRINT FNSETX(7)
```

In this example, ARG\$ contains the eight tables listed above and described in Appendix C. These tables describe the variables and user-defined functions available to the CPL function which uses ARG\$.

If a variable or user-defined function passed to the CPL function is not described in ARG\$, the third argument to the CPL function is used to return the id of the missing variable or user-defined function. MISSING\$ will contain the id, which is null-terminated. MISSING\$ needs to be at least 12 characters in length, because the largest possible id is 11 characters (a user-defined string function could have "FN" plus an 8-character id plus "\$"), and after the 11 characters comes the null terminator. If the id is that of a numeric array, the first byte of MISSING\$ has its high bit turned on, to distinguish the numeric array id from the id of a simple number.

FUNCTIONS

CRC
(CYCLIC REDUNDANCY CODE)

CRC
(CYCLIC REDUNDANCY CODE)

FORMAT

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

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

DESCRIPTION

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

EXAMPLES

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

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

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

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

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

NOTE

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

FUNCTIONS

DEC
(BINARY TO DECIMAL)

DEC
(BINARY TO DECIMAL)

FORMAT DEC (string expr [, ERR=stno])

DESCRIPTION

The DEC function converts a binary string expression into a signed decimal number (either positive or negative). The DEC function can take either an ASCII character or a hexadecimal number.

The leftmost bit is considered the "sign" bit. If "on" (1), the number is negative. Thus, for numbers with one in the high order position (decimal numbers from 128-255), the DEC statement must add hexadecimal zeroes to the high order position.

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

When dealing with a multi-character string, substringing is necessary if you want to extract the DEC of only part of the string. In this case the second coordinate is required:

```
0100 LET B$="HATTRICK"  
0110 LET A=DEC(B$(2,1))  
0120 PRINT A  
65
```

EXAMPLES

LET X=DEC(\$0032\$)	- X is 50
LET X=DEC(\$FFCE\$)	- X is -50
LET X=DEC(\$0400\$)	- X is 1024
LET X=DEC("A")	- X is 65
LET X=DEC(\$00FF\$)	- X is 255
LET X=DEC(\$FF\$)	- X is -1

FUNCTIONS

EPT
(EXPONENT)

EPT
(EXPONENT)

FORMAT

EPT (numeric expr)

DESCRIPTION

The EPT function returns the exponent of the numeric expression.

EXAMPLES

LET X=EPT(55)	=	.55*10 ²	then: X=2
LET X=EPT(5.23)	=	.523*10 ¹	then: X=1
LET X=EPT(-500)	=	-.5*10 ³	then: X=3
LET X=EPT(0)	=	0*10 ⁰	then: X=0
LET X=EPT(.00001)	=	.1*10 ⁻⁴	then: X=-4

FUNCTIONS

FID
(FILE INFORMATION)

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 (e.g. "G0"), the two-byte device name is returned. If the number refers to a disk file, information about the file is returned.

The format for the FID function when identifying files is displayed in Table 5-1 on the following page.

Bytes 25 through 28 of the FID give the highest number of records written in an INDEXED file or the actual number of records in use in a keyed file.

If the file type is a program, the defined maximum number of records is the number of bytes used by the program, the record size reflects the maximum number of bytes, and the number of records used is always zero.

EXAMPLE

```
1.
>LET A$=FID(2)
>PRINT DEC(A$(12,3))    -displays the defined maximum
                        number of records for the file
                        opened on logical unit 2.

2.
>A$=FID(0)
>PRINT A$
T3                      -displays the terminal number.
```

FUNCTIONS

FID (continued)

FID (continued)

Table 5-1. FID FORMAT

BYTES	NUMBER OF BYTES	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 \$04\$ = Program file \$07\$ = String file \$14\$ = Basic Encrypted Program File \$17\$ = Directory
11	1	Defined key size
12-14	3	Defined maximum number of records
15-16	2	Record size (=1 for STRING)
17-19	3	\$000000\$ (for compatibility with other levels only)
20	1	Disk number
21-24	4	Unused
25-28	4	Number records written/used
29-31	3	Number of records in the initial extent
32-34	3	Number of records in the growth extent
35-162	6+	fully qualified file name

FUNCTIONS

FNX
(DEFINE FUNCTION)

FNX
(DEFINE FUNCTION)

FORMAT

FNx {\$} (argument list)

where:

x = function name, up to 8 characters

\$ = specified for string functions

argument list = values provided for use by the DEF
statement

DESCRIPTION

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

EXAMPLE

00230 LET A=FNA(B,D)

FUNCTIONS

FPT
(FRACTIONAL PART)

FPT
(FRACTIONAL PART)

FORMAT FPT (numeric 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

FUNCTIONS

GAP
(ODD PARITY)

GAP
(ODD PARITY)

FORMAT GAP (string variable or literal)

DESCRIPTION This function generates odd parity for the specified string variable or constant on a byte-for-byte basis. The resultant string is the same length as the specified string.

EXAMPLE 0200 LET A\$=GAP(\$0FDC\$) - A\$ is equal to \$8FDC\$

FUNCTIONS

HSH
(HASH)

HSH
(HASH)

FORMAT

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

where:

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

DESCRIPTION

The HSH function is used by the system to compute a hash function. It may be used by the programmer to do a data integrity check.

NOTE

Level 7 BASIC programs do not have a hash field at the end of the program.

EXAMPLES

00600 LET A\$=HSH(B\$)

00600 LET A\$=HSH(B\$,C\$)

00600 LET A\$=HSH(B\$(10,LEN(B\$)-11))

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

FUNCTIONS

HTA
(HEXADECIMAL TO ASCII)

HTA
(HEXADECIMAL TO ASCII)

FORMAT HTA (string expr)

DESCRIPTION The HTA function converts the hexadecimal value of a string expression to pairs of characters representing a hexadecimal number.

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, since a hexadecimal value cannot be displayed.

The function requires ASCII characters in the statement. For each character in the string, the associated hexadecimal value is determined and two characters are returned. The result represents a hexadecimal number.

Thus, each character in the string is converted to two characters in the output string. For example:

```
500 LET X$="A"          stores: 01000001 (1 byte)
510 LET Y$=HTA(X$)      stores: 00110100 00110001 (2 bytes)
520 PRINT Y$            returns      |      |
    41                  2 ASCII characters: 4    &    1
```

The result of the HTA function, when placed in a variable, may be substringed:

EXAMPLES

```
00400 LET A$="EAGLE"
00410 LET B$=HTA(A$)
00420 PRINT B$          returns 4541474C45
00430 PRINT B$(3,2)     returns 41
```

FUNCTIONS

IND
(INDEX)

IND
(INDEX)

FORMAT

IND (fileno {,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.

EXAMPLE

LET A=IND(1,ERR=0500)

FUNCTIONS

INT
(INTEGER)

INT
(INTEGER)

FORMAT

INT (numeric expr)

DESCRIPTION

The INT function returns the integer part of the numeric expression. Any fractional digits are removed, and rounding does not occur.

EXAMPLES

0100 LET X=INT(5.84)	- X is 5
0200 LET Y=INT(.333)	- Y is 0
0300 LET Z=INT(-6.22)	- Z is -6

FUNCTIONS

IOR
(LOGICAL OR)

IOR
(LOGICAL OR)

FORMAT

IOR (string expr, string expr)

DESCRIPTION

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

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

EXAMPLE

LET X\$=IOR(\$OF\$, \$DC\$)

then: \$OF\$ = 0000 1111
 \$DC\$ = 1101 1100

X\$ = \$DF\$ = 1101 1111

FUNCTIONS

KEY

KEY

FORMAT

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

DESCRIPTION

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

EXAMPLE

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

FUNCTIONS

LEN
(LENGTH)

LEN
(LENGTH)

FORMAT

LEN (string expr)

DESCRIPTION

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

EXAMPLES

0010 LET A\$="ABC"

0020 LET B\$="DEFG"

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

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

FUNCTIONS

LRC
(LONGITUDINAL
REDUNDANCY
CHECK)

LRC
(LONGITUDINAL
REDUNDANCY
CHECK)

FORMAT

LRC (string variable or constant)

DESCRIPTION

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

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

EXAMPLE

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

FUNCTIONS

LST
(LIST)

LST
(LIST)

FORMAT

LST (string expr [, string expr] [, ERR=stno])

DESCRIPTION

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

On Level 7 BASIC, 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.

If it is desired that the LST reflect the environment of the second, or target program, 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 C.

Once the variable id tables described in Appendix C are understood, the following example and example 2 for the CPL function will help.

EXAMPLE

```
>10 PRINT A$LST(B$)
>PRINT LST(PGM(10))
00010 PRINT A$
```

(See the CPL function for an example using LST with two arguments.)

FUNCTIONS

LVL
(LEVEL)

LVL
(LEVEL)

FORMAT

LVL (numeric expression)

where numeric expression = the integers 0 or 1

DESCRIPTION

The LVL function returns one of two strings depending on which number is specified in the parameter. LVL(0) returns a string representing the current level of the BASIC software. LVL(1) returns a string representing the current level of the file system.

EXAMPLES

Example 1:

```
>PRINT LVL(0)  
EBS7XXX
```

Example 2:

```
>PRINT LVL(1)  
EOS7XXX
```

FUNCTIONS

MOD
(MODULO)

MOD
(MODULO)

FORMAT

MOD (numeric expr a, numeric expr b)

where:

numeric expr a = the number on which to
 perform the modulo
 calculation; the dividend

numeric expr b = the number representing
 the base or divisor

DESCRIPTION

Whenever "numeric expr a" is a positive number, the MOD function returns the result of the following formula:

$$X = \text{expr a} - (\text{INT}(\text{expr a} / \text{expr b}) * \text{expr b})$$

where: INT is the BASIC function that returns the integer portion of a numeric expression.

In this case, the value returned by the function is equivalent to calculating the remainder of the division of the two numeric expressions. Given the example, MOD (7,3), you could simply divide 7 by 3. The quotient is 2 with a remainder of 1. You can see that we obtain the same result, 1, when we apply the formula to the example, MOD (7,3).

$$X = \text{expr a} - (\text{INT}(\text{expr a} / \text{expr b}) * \text{expr b})$$

$$X = 7 - (\text{INT}(7 / 3) * 3)$$

$$X = 7 - (\text{INT}(2.333) * 3)$$

$$X = 7 - (2 * 3)$$

$$X = 7 - 6$$

$$X = 1$$

The MOD function becomes slightly more complicated when "numeric expr a" is a negative number. The result, "X", of the formula, will always be negative in this case. Since the MOD function must return a positive value, the following formula is used to complete the calculation:

$$X = X + |\text{expr b}|$$

The system adds the absolute value of "expr b" to the result, "X", of the first formula to obtain a positive value.

Let's apply both formulas to the following example, MOD (-7,3), which contains a negative value for "numeric expr a":

$$X = \text{expr a} - (\text{INT}(\text{expr a} / \text{expr b}) * \text{expr b})$$
$$X = -7 - (\text{INT}(-7 / 3) * 3)$$
$$X = -7 - (\text{INT}(-2.333) * 3)$$
$$X = -7 - (-2 * 3)$$
$$X = -7 - (-6)$$
$$X = -1$$

Since "X" is negative, we need to continue with the second formula to obtain a positive result:

$$X = X + |\text{expr b}|$$
$$X = -1 + |3|$$
$$X = 2$$

The final result of the function, MOD (-7,3), is 2.

EXAMPLES

0010 LET X=MOD(7,3) - function returns a value of 1

0020 LET X=MOD(7,-3) - function returns a value of 1

0030 LET X=MOD(-7,3) - function returns a value of 2

0040 LET X=MOD(-7,-3) - function returns a value of 2

0100 LET X=MOD(26,7) - function returns a value of 5

0100 LET X=MOD(22,11) - function returns a value of 0

FUNCTIONS

NOT
(INVERSE STRING)

NOT
(INVERSE STRING)

FORMAT NOT (string expr)

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

NOT 0 = 1
NOT 1 = 0

EXAMPLE 00100 LET X\$=NOT(\$DC\$)

if: \$DC\$ = 1101 1100
then: PRINT HTA(X\$) = \$23\$ = 0010 0011

FUNCTIONS

NUM
(NUMERIC VALUE)

NUM
(NUMERIC VALUE)

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

DESCRIPTION The NUM function returns the numeric value of the characters in the string expression. All characters in the string must be numeric, or related to numbers; e.g., "+", "-", ".", "E" are legal. The exception is that commas are not allowed.

EXAMPLE 00100 LET A\$="224"

 00200 LET B=NUM(A\$, ERR=8000)

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

FUNCTIONS

PGM
(PROGRAM)

PGM
(PROGRAM)

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 statement number is zero, PGM returns the program name. Thus, PGM (0) is equivalent to the Function PNM.

EXAMPLES

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

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

FUNCTIONS

POS
(POSITION)

POS
(POSITION)

FORMAT

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

where:

scan string = the string (in constant or
 variable form) being searched

relational operator= one of the valid symbols of comparison:

=	<> or ><
<	<= or =<
>	>= or =>

target string = the string (in constant or
 variable form) to be searched

```
step value      = the increment defining the
                  intervals at which the target
                  string is examined for each
                  subsequent comparison (default
                  value is 1)
```

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.

EXAMPLE

```
LET A$="ABCDEFGHJKLM"
      (target string)
then: LET X=POS("D"=A$)           - X is 4

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

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

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

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

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

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

FUNCTIONS

PUB
(PUBLIC PROGRAMS)

PUB
(PUBLIC PROGRAMS)

FORMAT

PUB (numeric expression from 0-15)

DESCRIPTION

The PUB function returns a string representing all of the programs in the shared memory space. For each public program, a string of 136 bytes in binary format is returned.

For compatibility with other Basic Four products, the numbers 0-15 are accepted. Any number other than zero (0) returns a null string.

PUB (0) returns information about all programs in shared memory. The format, with conversion code, is as follows:

Table 5-2. PUB(0) FORMAT

<u>BYTE</u>	<u>CONTENTS</u>	<u>CONVERSION</u>
1	ownership indicator:	0 = someone else 1 = current user
2	type of public program:	0 = a called program that is not added by means of ADDR or ADDE 1 = a program added to public memory by means of ADDR or ADDE
3-6	length of the program in bytes	
7-136	absolute path name of the public program	

EXAMPLE

```
LET X$=PUB (0)
PRINT X$(7)
```

This returns the full program name.

FUNCTIONS

SGN
(SIGN)

SGN
(SIGN)

FORMAT

SGN (numeric 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.

EXAMPLES

```
LET X=SGN(-77)    -  X=-1
LET X=SGN(6)      -  X=1
LET X=SGN(0)      -  X=0
```

FUNCTIONS

STR
(STRING)

STR
(STRING)

FORMAT

STR (numeric expr {:mask})

where:

mask = see "NUMERIC EDITING" in the
 introduction.

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"
```


FUNCTIONS

TBL
(Translation Table)

TBL
(Translation Table)

FORMAT TBL(string expr, statement number)

where:

string expr = a string to be translated by the
translation table

statement = the statement number containing the
number translation table

DESCRIPTION The TBL function translates the specified string expression based on the contents of the TABLE statement referenced. This function performs the same operation as the TBL= input/output option with the exception that this function may be used independently of any input or output directive. Refer to the TABLE directive and the TBL= input/output option for more information.

EXAMPLE 00200 A\$=TBL(B\$ + C\$, 100)

This page left blank intentionally.

FUNCTIONS

TSK
(Display Configured Devices)

TSK
(Display Configured Devices)

FORMAT TSK (numeric expression)

where:

numeric expression = an integer from 0 - 15

DESCRIPTION The TSK function is used to access information about each device that is configured on the system, such as terminals and printers.

Level 7 BASIC uses only TSK(0). the other numbers are accepted only for compatibility with other BASIC FOUR products. When used, the numbers 1 - 15 return the null string.

The string that is returned by TSK(0) consists of six-byte entries, one after the other. There is one entry for each configured device. Each six-byte field of information can be accessed by placing the string in a variable and substringing it.

The format for the TSK(0) function is displayed in Table 5-3. TSK(0) FORMAT on the following page.

EXAMPLE 00100 LET A\$=TSK(0)
 00110 FOR I=1 TO LEN (A\$) STEP 6
 00120 LET B\$=A\$(I,6)
 00130 PRINT "DEVICE NAME IS: ",B\$(1,2)
 00140 IF B\$(3,1)="0" THEN PRINT "AVAILABLE"
 00150 IF B\$(3,1)="2" THEN PRINT "IN USE"
 00160 NEXT I

FUNCTIONS

TSK (continued)

TSK (continued)

Table 5-3. TSK(0) FORMAT

<u>BYTES</u>	<u>NUMBER OF BYTES</u>	<u>DESCRIPTION</u>
1,2	2	Device name in ASCII (e.g., T0,G1)
3	1	Device status in ASCII: 0 = available 2 = in use
4-5	2	Bytes 4 and 5 are not used
6	1	Byte 6 is always binary zero

FUNCTIONS

XOR
(EXCLUSIVE OR)

XOR
(EXCLUSIVE OR)

FORMAT XOR (string expr, string 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\$, \$DC\$)

then: \$OF\$ = 0000 1111
 \$DC\$ = 1101 1100

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

OVERVIEW

A system variable is a function whose use is established by the operating system. Certain commonly used pieces of information are placed in a variable with a mnemonic-type identifier for use in programming. System variables are used to determine the value of specific system operations, such as the time (TIM) and the date (DAY).

System variables are also used to determine the number of unused bytes in the user area of memory, the value of the last occurring error, and more.

SYSTEM VARIABLES

CTL
(Control Variable)

CTL
(Control Variable)

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):

Table 6-1. TERMINATOR KEY CONTROL VALUES

<u>KEY</u>	<u>VALUE</u>	<u>ASCII CHARACTER</u>	<u>CTL VALUE</u>
(None)	\$00\$	NULL	0
LINEFEED	\$0A\$	LF (linefeed)	0
RETURN	\$0D\$	CR (carriage return)	0
CTL I (or SHIFT + CTRL +'L')	\$1C\$	FS (field separator)	1
CTL II (or SHIFT + CTRL +'M')	\$1D\$	GS (group separator)	2
CTL III (or CTRL +'N')	\$1E\$	RS (record separator)	3
CTL IV (or CTRL +'O')	\$1F\$	US (unit separator)	4
none (SIZ=satisfied)(none)			5

EXAMPLES

1. 0100 PRINT CTL
2. 0100 IF CTL=4 THEN GOTO 9000

SYSTEM VARIABLES

DAY
(Date)

DAY
(Date)

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 configured for the system (see the "date" command in the BOSS/IX User Reference Manual. The default format is:

mm/dd/yy where - mm = month

dd = day

yy = year

EXAMPLES

>PRINT DAY

00100 LET X\$=DAY

00200 PRINT X\$(1,2) - substrings allow you to print
just the portion of the date (in this
case, the month).

SYSTEM VARIABLES

DSZ
(Available User 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

SYSTEM VARIABLES

ERR
(Error)

ERR
(Error)

FORMAT

ERR {(code 1, code 2,...,code n)}

where:

code = an error code.

DESCRIPTION

The ERR variable contains the value of the last error that occurred. This can be a number from 0 to 127.

ERR can be used by itself, as demonstrated in Example 1 below, to display the previous error number.

ERR can also be used to branch to a specified statement number, based upon the error code of the previous error, as demonstrated in Example 2 below.

The ERR variable contains information relating only to the BASIC programming environment. BOSS/IX implements other error messages, such as the Business File System errors.

EXAMPLES

1. 0100 PRINT "ERROR CODE = ", ERR
0999 EXIT ERR

2. 0050 ON ERR(11,12,47) GOTO 100,200,300,400

-branch to 100 if error is other than 11, 12
or 47

-branch to 200 if error=11

-branch to 300 if error=12

-branch to 400 if error=47

The same operation can be written using a LET statement:

0050 LET E=ERR (11,12,47)
0060 ON E GOTO 100,200,300,400

SYSTEM VARIABLES

PFX
(Prefix List)

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 chapter 5. It is a list of directories to be searched first in the event that a file is requested without a full path name.

EXAMPLE

```
>PRINT PFX  
/usr/username
```

SYSTEM VARIABLES

PNM
(Program Name)

PNM
(Program Name)

FORMAT PNM

DESCRIPTION PNM returns the name of the program currently in main memory. PNM is equivalent to the function PGM(0) described in chapter 5. It is included in Level 7 BASIC in order to be compatible with Level 8.

EXAMPLE >LOAD "BOXCARS"
 >PRINT PNM
 BOXCARS
 >

SYSTEM VARIABLES

PRC
(Precision)

PRC
(Precision)

FORMAT

PRC

DESCRIPTION

The PRC system variable returns the value of the current arithmetic precision of BASIC. The value returned is either an integer between 0 and 14 or $-.1E+01$ indicating floating point.

EXAMPLE

>PRINT PRC
2

This page left blank intentionally.

SYSTEM VARIABLES

PSZ
(Program Size)

PSZ
(Program Size)

FORMAT

PSZ

DESCRIPTION

The PSZ variable contains the number of bytes used by the resident program, not including data. If PSZ is referenced in a called program, the value is the size of the calling program.

PSZ variable contains the user program area overhead. Therefore, PSZ always equals at least 62. PSZ includes space for console-mode-only variables, and will therefore always be at least as large as the currently loaded program.

Despite these variations, the PSZ value is always large enough to use as a parameter to the SAVE directive when saving a program.

EXAMPLE

Example 1:

```
>PRINT PSZ
3724
```

SYSTEM VARIABLES

SEL

SEL

The SEL system variable has been deleted from 7.2 BASIC.

SYSTEM VARIABLES

SSN
(System Serial Number)

SSN
(System Serial Number)

FORMAT SSN

DESCRIPTION The SSN variable contains the system serial number.

EXAMPLE >PRINT SSN

SYSTEM VARIABLES

SYS
(Operating System Level)

SYS
(Operating System Level)

FORMAT SYS

DESCRIPTION The SYS function contains the level of the operating system.
It provides a string expression showing the operating system
level.

EXAMPLE >PRINT SYS
EBS7XXX

SYSTEM VARIABLES

TCB
(Task Control Block)

TCB
(Task Control Block)

FORMAT

TCB (n)

where:

n = a numeric value ranging from 0-12

DESCRIPTION

The TCB variable contains information that pertains to a particular task. Some TCB's must be converted into decimal or hexadecimal format to be useful. The table on the following page shows the contents of each TCB and the appropriate equation for conversion, if required.

EXAMPLE

```
01000 INPUT (0,ERR=08000)@(5,10)'CL',A
.
.
.
08000 PRINT @(0,21),'CL',"YOU GOOFED. ERR = ",
      ERR,"AT LINE    ",TCB(5);INPUT*;RETRY
```

- displays the line number where
the error occurred.

SYSTEM VARIABLES

TCB (Continued)

TCB (Continued)

Table 6-2. TCB VARIABLE FORMAT

TCB(n)	DESCRIPTION	BYTE	
		LENGTH	CONVERSION EQUATION
0	disk number	1	\$00\$ (retained for BB 4 compatibility only)
1	sector number	2-	- \$00\$ (retained for BB 4 compatibility only)
2	sector number	1-	
3	system status	2	S\$=HTA(BIN(TCB(3),2))
4	current statement number	2	
5	statement number of last error	2	
6	statement number SETESC references	2	
7	statement number SETERR references	2	
8	Unused	11	
9	Selected logical disk (always 0 on 7.2, unused)	2	
10	Unused	12	
11	*Last channel used (other than 0)	1	
12	Most recent file system error code	1	

*This byte field refers to the logical unit (channel) number (0-63) that was accessed last. Following a successful START, BEGIN or END, the number is always zero. Following an error, the number refers to the logical unit number for the current process. References to logical unit number zero (0) are not updated and always refer to the user's console.

SYSTEM VARIABLES

TIM
(Time of Day)

TIM
(Time of Day)

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:

```
00100 LET T=TIM
00200 LET H=MCD(INT(T),12)
00300 LET S1=INT(FPT(T)*3600)
00400 LET M=INT(S1/60)
00500 LET S=S1-M*60
```

- where H=hours, M=minutes, S=seconds

EXAMPLES

1.

```
>PRINT TIM
10.94028

>PRINT H+" ":"M+" ":"S (using the translation program above)
10:53:57
```

2.

```
00500 OPEN (1)"LP"
00510 OPEN (2)"FILENAME"
00520 READRECORD (2)A$
00530 PRINT (1),@(58),TIM
00540 PRINT (1),@(20),A$
00550 END
```

SYSTEM VARIABLES

UNT
(Lowest Available Unit)

UNT
(Lowest Available Unit)

FORMAT

UNT

DISCUSSION

The UNT variable returns the lowest logical unit number that is available. For example, if logical unit numbers from 1 to 5 are already opened by a running program, the UNT variable contains the number 6, representing the next available logical unit number. If logical unit 3 were to be closed, the UNT variable would be updated to contain 3, since that would now be the lowest available unit.

The UNT variable is particularly useful in modularized programs because a called program need not know which units have already been opened by the calling program. The next logical unit number to be opened may be referred to by the variable UNT, and the next available channel will automatically be opened.

EXAMPLES

1.

```
>END
>PRINT UNT
1
>OPEN (1)"P1"
>PRINT UNT
2
>OPEN (3)"P3"
>PRINT UNT
2
>OPEN (2)"P3"
>PRINT UNT
4
>CLOSE (1)
>PRINT UNT
1
```

2.

```
0100 A=UNT
0110 OPEN (A)"LP"
0120 B=UNT
0130 OPEN (B)"FILENAME"
.
.
.
0900 EXIT
```


OVERVIEW

Input/Output options are used to improve the execution of an I/O directive. Specified within the parentheses, immediately following the file number, these optional parameters can cause branching within the program. They can also set up controls to override system defaults, specify a record to be accessed, specify a length for the range of a variable, and more.

Multiple I/O options in a statement are separated by commas.

INPUT/OUTPUT OPTIONS

CLASS=
(Specify Printer Attributes)

CLASS=
(Specify Printer Attributes)

FORMAT

CLASS= "string"

where:

"string" = the name of a print job
attribute list

DESCRIPTION

It is possible to specify attributes for a print job, which affect the spooling operation, as a parameter of the OPEN directive. Spooling is an operation that the operator need not be concerned with, since it operates transparently.

Spooling is the feature that controls which print jobs are printed in what priority and in what manner. Files to be printed are held in a separate disk file while waiting to be printed. Therefore, a program may continue without waiting for or being interrupted by a busy printer. For more information on the spooling feature, refer to the User Guide.

A system file, "/etc/class", is provided to hold regularly used spooling formats. Print job priority, the number of copies, whether or not to notify the operator upon completion are examples of selectable options. A name is assigned to each set of options in the "class" file, and the desired set may be invoked by specifying CLASS= in the OPEN statement. The string argument refers to the "class" file entry name.

The OPTS= option allows you to specify all desired print queue attributes within the OPEN statement without calling a defined set from the "class" file. See OPTS= in this section.

Appendix A of this manual describes procedures for using this option and making an entry in the file, "/etc/class". See "Specifying Print Job Attributes".

EXAMPLE

00110 OPEN (1, CLASS="QUICK")"LP"

- This statement causes a channel to the printer "LP" to be opened and invokes the options list entry in the "/etc/class" file named "QUICK".

INPUT/OUTPUT OPTIONS

DOM=
(Duplicate or Missing Key)

DOM=
(Duplicate or Missing Key)

FORMAT

DOM= stno

DESCRIPTION

The DOM= option transfers control to the specified statement if the key specified in an INPUT or REMOVE operation is not found in the file, or if the key specified in a WRITE operation is already in the file. If a DOM= option is not used, an ERROR 11, MISSING OR DUPLICATE KEY, is generated when the specified key is not found.

EXAMPLES

0100 READ (2,KEY=A\$)R\$ - If the KEY is not in the file, an ERROR 11 occurs.

0100 READ (2,KEY=A\$,DOM=500)R\$ - If the KEY is not in file, the DOM=branch is taken, and ERR=11 is set.

0100 WRITE (2,KEY=A\$)R\$ - If the KEY is in the file, old data is overwritten.

0100 WRITE (2,KEY=A\$,DOM=500)R\$ - If the KEY is in the file, the DOM=branch is taken, and ERR=11 is set. Old data is not overwritten.

0100 WRITE (2,KEY=A\$,DOM=500,ERR=400)R\$

- If the KEY is not in the file, but another error occurs, branch to statement 400.

INPUT/OUTPUT OPTIONS

END=
(Branch at End of File)

END=
(Branch at End of File)

FORMAT END= stno

DESCRIPTION The END= option transfers program control to the specified statement number when the end of the file is reached. If an END= option is not used, an ERROR 2, END OF FILE, is generated.

EXAMPLES 0200 READ (1,END=0500)A\$
 0200 LET K\$=KEY(1,END=9000)

INPUT/OUTPUT OPTIONS

ERR=
(Error Exit)

ERR=
(Error Exit)

FORMAT

ERR= stno

DESCRIPTION

The ERR= option transfers program control to the specified statement number if an error occurs while executing the statement. For the statement containing it, the ERR= option overrides a SETERR statement. Specific error control clauses, such as END= and DOM=, override an ERR= option. Errors greater than 99 are not trapped by an ERR= option; rather, they cause an immediate exit to console mode, due to the nature of these errors (exceptions: ERROR 126: CTRL+Y KEY USED, and ERROR 127: ESCAPE, are trapped by SETERR and SETESC).

NOTE

Use of DOM= is recommended in statements performing READ or WRITE directives.

EXAMPLE

00200 READ (1,ERR=00500)A\$

INPUT/OUTPUT OPTIONS

IND=
(Record Index)

IND=
(Record Index)

FORMAT

IND= expr

where:

expr = a numeric expression that specifies the
 position of the record in a file,
 relative to zero.

DESCRIPTION

The IND= option specifies the index (record number) of the record to be accessed by the input/output statement. The first record in a file has an index of 0.

IND= can be used with Indexed, Direct, Sort, Serial or String files. Use of IND= when reading Direct or Sort files speeds record access by using the relative (to 0) record number. However, files are not sorted when this method is used.

If the IND= clause is used in a CLOSE directive, it can have the values 0, 1, 2, or 9. These are used when CLOSEing a unit which had been OPENed to the half-inch tape device. These numbers take 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 the CLOSE is preceded by a WRITE RECORD, 2 file marks are written on tape.
- 9 - writes 2 file marks on tape, then rewinds the tape

EXAMPLE

00200 READ(1,IND=10)

INPUT/OUTPUT OPTIONS

IOL=
(IOLIST Statement)

IOL=
(IOLIST Statement)

FORMAT

IOL= stno

DESCRIPTION

The IOL= option specifies the statement number of the IOLIST to be used. The IOLIST contains a list of variables and/or constants.

EXAMPLES

```
0100 IOLIST A$,B,C,IOL=0200
0200 IOLIST D,E
0300 READ (1,KEY=A$)IOL=0100
0400 PRINT (7)IOL=0100
```

INPUT/OUTPUT OPTIONS

ISZ=
(Access File as if INDEXED)

ISZ=
(Access File as if INDEXED)

FORMAT

ISZ=recsz

where:

recsz = the redefined record size for a file

DESCRIPTION

The ISZ option allows any file to be accessed as if it were an Indexed file with the record size specified.

ISZ= is used in conjunction with READ RECORD and WRITE RECORD to handle multiple records or partial records (e.g., 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)"BOOK"  
>READ RECORD(1)A$  
  
>PRINT HTA(A$)
```

INPUT/OUTPUT OPTIONS

KEY=

KEY=

FORMAT

KEY= string expr

DESCRIPTION

This option specifies the key of the record to be accessed by the input/output statement containing the KEY= option.

EXAMPLES

0500 READ(1,KEY=A\$)X\$

0500 WRITE(1,KEY=STR(A:"00000"))A,B\$

INPUT/OUTPUT OPTIONS

LEN=
(Length of Variable)

LEN=
(Length of Variable)

FORMAT

LEN= min,max

where:

min = the minimum length allowable for the variable.

max = the maximum length allowable for 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.

EXAMPLE

0100 INPUT (0,ERR=0300)A\$:(LEN=2,3)

0300 IF ERR=48 THEN GOTO 8000 ELSE GOTO 7000

INPUT/OUTPUT OPTIONS

OPTS=
(Specify Printer Attributes)

OPTS=
(Specify Printer Attributes)

FORMAT

OPTS= "string expr"

where:

string expr = list of print job options

DESCRIPTION

As with the CLASS= option described earlier in this section, the OPTS= option allows you to specify within the OPEN statement the manner and priority of the print job. The string format uses the options for the command "lpr". Refer to appendix A under "Specifying Print Job Attributes" for details on the use of "lpr" options with the OPTS= parameter.

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. For more information regarding spooling, refer to the User Guide.

EXAMPLE

```
00100 LET A$="alias=report copies=2 -notify"  
00110 OPEN (7, OPTS=A$)"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.

INPUT/OUTPUT OPTIONS

SEQ=
(SEQUENTIAL FILE NUMBER)

SEQ=
(SEQUENTIAL FILE NUMBER)

FORMAT

The following description applies only when SEQ= is used in an OPEN directive which is OPENing the half-inch tape device, R0.

The numeric expression taken by SEQ= denotes where to position the tape. The positioning is with reference to filemarks as follows:

- 0 - positions the tape at the Beginning of Tape mark
- 1 - positions the tape at the first filemark
- 2 - positions the tape at the second filemark
- .
- .
- .

EXAMPLE

00500 OPEN(1,SEQ=3) "R0"

Positions the tape at the third filemark.

This page left blank intentionally.

INPUT/OUTPUT OPTIONS

SIZ=
(Input Size)

SIZ=
(Input Size)

FORMAT

SIZ= numeric expression

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 is pressed. The CTL variable is set to five (5) if input is terminated due to a SIZ= option.

EXAMPLE

00700 INPUT (0,SIZ=1)A\$

INPUT/OUTPUT OPTIONS

TBL=
(Translation Table)

TBL=
(Translation Table)

FORMAT TBL= stno

DESCRIPTION This option specifies the number of the TABLE statement to be used to translate data. The statement number specified must contain a TABLE statement. See the TABLE directive and the TBL function for further information.

EXAMPLES 00100 READ(1,TBL=02000)A\$

 00100 WRITE(2,TBL=05000)A\$,B

INPUT/OUTPUT OPTIONS

TIM=
(Set Time Out)

TIM=
(Set Time Out)

FORMAT TIM= numeric expr

DESCRIPTION This option specifies the number of seconds allowed for completion of input. After that interval has passed, an ERROR 0 is generated. There is no default time out for keyboard input. The maximum TIM= value is 255 seconds. "TIM=0" returns almost immediately.

EXAMPLE 00100 INPUT (0,ERR=00500,TIM=60)"NAME",A\$

- allows 60 seconds for input;
 otherwise, control passes to
 statement 500

OVERVIEW

Mnemonics are standardized operations for use in program statements and in console mode. There are two main types of mnemonic operations: one type controls certain operations on devices, and the other type is a series of control switches for functions in the operating system.

The use of alphabetic characters serves two purposes: it helps you to remember which mnemonic goes with which operation (hence the term "mnemonic"), and it makes it possible for the operating system to interface with differently configured terminals. For terminal operations, one need not necessarily program operations by the use of hexadecimal equivalents. Automatically BOSS/IX interprets the mnemonics for each configured terminal.

Mnemonic constants are subject to conversion by the TBL function and TBL= input/output option, and are passed as data to the software driver for the device.

Each mnemonic consists of two alphabetic characters enclosed by primes (single quotation marks) and is inserted in a statement at the point where the stated operation is desired. The format and use of the mnemonics is illustrated by the following example:

```
00100 PRINT @(35,5), A$, 'LF', B$
```

In this example the 'LF' mnemonic is used to perform a line feed on the user's terminal after printing the value of A\$ at character position 35 on line 5. If the mnemonic is inserted in the statement immediately following the PRINT directive, the line feed occurs prior to printing the value of A\$.

Device mnemonics are used to direct certain operations on terminals and printers, such as Foreground and Background routines, cursor control, expanded print, and "screen dumps" to a printer. Operating system mnemonics control such operations as echo, escape, and input buffering.

MNEMONICS

The mnemonic constants available for each type of I/O device appear on the following pages. Unless the mnemonic is listed as applicable for a device, an ERROR 29 UNDEFINED MNEMONIC, is generated upon statement execution (ERROR 29's can be turned off for VDT's.)

Mnemonics can be used as string expressions. When so used, they are evaluated to the 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.

EXAMPLES

```
200 PRINT 'CS', 'SB'
210 PRINT @(12,5),"1. CUSTOMER NUMBER OR END: "
220 PRINT 'CF', @(37,5), 'CL', "_____"
230 INPUT @(37,5),A
```

Statement 200 clears the terminal screen and sets the Background Mode. Statement 210 provides horizontal and vertical positions for the print statement. Statement 220 clears any characters printed in foreground, ends Background Mode, clears the line from vertical position 37 to the end of the line, and positions a series of underlines there. Statement 230 positions a numeric input prompt.

Using mnemonics as string expressions, the above sequence can be placed in a variable:

```
00200 SCREEN$= 'CS' + 'SB' + @(12,5) + "1. CUSTOMER NUMBER OR
END: " + 'CF' + @(37,5) + 'CL' + "_____"
00210 PRINT SCREEN$
00220 INPUT @(37,5), A
```

Following the discussion is an alphabetical listing of all mnemonics, divided into categories, and the name of each.

MNEMONICS

CURSOR CONTROL: This category of mnemonics is a function of the VDT. The various commands direct the position of the cursor, generally for input procedures.

Cursor control mnemonics are useful in input routines using formatted screens. A program can be written using considerably less memory overhead since there is no need to reconstruct the formatted screen.

A brief discussion of each follows:

- | | |
|---|--|
| @(X) - Horizontal Position | - The next data is displayed at the horizontal column position defined by X. |
| @(X,Y) - Horizontal and Vertical Position | - The next data is displayed at the horizontal column position X and the vertical line position of Y. In a single program statement where there are multiple displays on the same line, the vertical position need only be specified in the first set of parameters. |
| 'BS' - Backspace | - A destructive backspace is performed. The cursor moves back one space, prints a space, then backs up again to the deleted position. This backspace is not equivalent to the ASCII backspace, which is non-destructive. |
| 'CH' - Cursor Home | - Two functions are performed: the cursor is positioned at "home" (0,0), and the Foreground Mode is set. |
| 'CR' - Carriage Return | - The cursor moves down one line and back to position zero. Note that this operation code is not the same as the ASCII character. (This is also a printer operation. The exact function varies from printer to printer; it may or may not perform a concurrent linefeed. |
| 'DN' - Cursor Down | - The cursor moves down one line, retaining the same horizontal position. It is non-destructive. |
| 'LT' - Left Cursor | - The cursor moves one space to the left. This is a non-destructive backspace that leaves the existing character intact. |

MNEMONICS

CURSOR CONTROL (continued):

'RC' - Read Cursor

- The current cursor position coordinates are provided for use with input statements. By converting to "X,Y" variables the program can reuse the coordinates for printing or further input. If a screen format change occurs during input routines (as when calling a help text) the cursor must return to the last input position when the original screen format is recalled. Echo is suppressed during the remainder of the input routine, and restored after. The actual coordinates are the returned coordinates less 32. And the coordinates are returned in the reverse order of the "@(x,y)" mnemonic, i.e., line first and column second.

Example:

```
100 OPEN (1)FID(0)
110 INPUT (1)'RC',A$
120 LET X=ASC(A$(2))-32
130 LET Y=ASC(A$(1))-32
```

Now the variables x and y contain the coordinates where the cursor was positioned when the 'RC' was invoked. And the coordinates are in the same order used in the "@(x,y)" mnemonic which uses column first and line second.

Note that it may be necessary to lock the keyboard to insure that operator entry does not cause the wrong coordinates to be sent.

'RT' - Right Cursor

- The cursor moves one space to the right. This is a non-destructive control, leaving existing characters intact.

'UP' - Up Cursor

- The cursor moves up one space, retaining the same horizontal position. It is non-destructive.

MNEMONICS

PRINTER CONTROL:

These mnemonics instruct the associated 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. |
| '8L' - Eight Lines Per Inch | - The printer's default lpi setting is overridden and set to 8 lines per inch. |
| '10' - 10 Pitch | - Documents are printed with 10 lines per horizontal inch. |
| '12' - 12 Pitch | - Documents are printed with 12 lines per horizontal inch. |
| '16' - 16 Pitch | - Documents are printed with approximately 16 characters per horizontal inch. |
| 'B1' - Bin 1 | - Paper from bin 1 (front 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 feed into the out-tray. |
| 'B2' - Bin 2 | - Paper from 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 feed into the out-tray. |
| '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. |
| '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 <u>height</u> to two spaces. For 120 line printers 'EP' expands horizontally to two spaces per character. |
| 'FF' - Form Feed | - The printer vertically spaces to the top of the next page. |

MNEMONICS

PRINTER CONTROL (Continued):

- | | |
|--------------------------------------|--|
| 'LF' - Line Feed | - The printer performs a line feed and a carriage return. |
| 'NL' - New Line | - Start a new line. |
| '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. |
| 'PM' - Plot Mode
(with line feed) | - This command allows direct dot-line addressing on matrix printers. A line feed then causes the paper to advance only a single dot row, instead of a normal character line space. This mnemonic allows graphics printing to be initiated. It is used for each line of plot data and is terminated with a line feed. Plot data consists of hexadecimal characters 40 through 7F. |
| 'S2' - Slew 2 | - Slew to channel 2 |
| 'S3' - Slew 3 | - Slew to channel 3 |
| 'S4' - Slew 4 | - Slew to channel 4 |
| 'S5' - Slew 5 | - Slew to channel 5 |
| 'S6' - Slew 6 | - Slew to channel 6 |
| 'S7' - Slew 7 | - Slew to channel 7 |
| 'S8' - Slew 8 | - Slew to channel 8 |
| 'SL' - Start Load | - The electronic VFU (vertical print unit) is loaded. This is only available on some printers. |
| 'SP' - Superscript Print | - This allows the printer to print superscripted text. See the examples on the next page. |
| 'SS' - Subscript | - The following data is printed subscripted to the previously printed data. (This mnemonic allows a continuous underline to be printed on serial printers.) See the examples on the next page. |
| 'VT' - Vertical Tab | - Slew to channel 6 |

MNEMONICS

NOTE

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.

SUBSCRIPT AND SUPERSCRIP T EXAMPLES

The Tritel and Whisper printers require that the subscripted or superscripted data be sent separately. The printer backs up and prints the sub- or superscripted data in the spaces left for it.

For example,

Example 1:

```
>PRINT (7)"THIS LINE SHOULD HAVE WORDS IN  
SUBSCRIPT",'SS',@ (9),"SUB",@ (30),"SUB",'LF'
```

Resulting line:

THIS LINE SUB SHOULD HAVE WORDS SUB IN SUBSCRIPT

Example 2:

```
>PRINT (7)"NEXT LINE WILL HAVE  
SUPERSCRIPTS",'LF','SP',  
>PRINT @ (4),"1",@ (19),"2",@ (39),"3",'LF',"BOOK",  
@ (13),"NUMBER",@ (33),"OFFICE"
```

Resulting line:

BOOK¹ NUMBER² OFFICE³

The Diabolo printer shifts into subscript or superscript mode each time it receives the mnemonic in a print line. There is no need to back up and reprint the line.

Example:

```
>PRINT (7)"THE NEXT WORD IS",'SS'," SUBSCRIPTED",  
" AND THE NEXT IS",'SP'," SUPERSCRIPTE
```

Resulting line:

THE NEXT WORD IS SUBSCRIPTED AND THE NEXT IS
SUPERSCRIPTE

MNEMONICS

VDT CONTROLS:

The mnemonics in this category perform a variety of functions relating directly to the terminal display.

- | | |
|---------------------------------------|---|
| '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. |
| 'BU' - Begin Underline | - All of the following characters are underlined. |
| 'CE' - Clear Screen to
End of Line | - The screen is cleared from the cursor position to the bottom of the screen. |
| 'CF' - Clear Foreground | - All the characters printed in foreground mode are replaced with spaces. Background mode is cancelled, if it is in effect. |
| 'CL' - Clear line | - All the characters from the cursor to the righthand end of the line are replaced with spaces. |
| 'CS' - Clear Screen | - All characters are cleared from the screen, and the cursor is positioned at home (0,0). |
| '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. Background Mode is cancelled, if it is in effect. |
| '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 and dark print (begun by 'BR'). |
| 'EU' - End Underline | - Text following is no longer printed with underlining (begun by 'BU'). |

MNEMONICS

VDT CONTROL (continued)

- | | |
|-------------------------|--|
| '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. Foreground Mode is reset. |
| 'KL' - Keyboard Lock | - Transmission of data from the keyboard is halted. |
| 'KU' - Keyboard Unlock | - Resumes data transmission from the keyboard following 'KL'. |
| 'LD' - Line Delete | - The line where the cursor is positioned is removed, all lines below it scroll up one line, a line of spaces is printed at the bottom, and Foreground is reset. |
| '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, the bottom line is deleted, and Foreground Mode is reset. |
| '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, and screen scrolling through the protected field is prevented. |
| 'RB' - Ring Bell | - The terminal buzzer sounds. |
| '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 <u>not</u> begin. |

MNEMONICS

VDT CONTROL (Continued)

- | | |
|----------------------------------|---|
| 'SF' - Start Foreground | - Foreground Mode is initiated. This is the normal (default) I/O mode. |
| 'SN' - Screen Narrow | - Sets the screen to 80 characters. |
| 'SW' - Screen Wide | - Sets the screen to 132 characters. |
| 'TL' - Transmit Line | - Transmits all unprotected data on the screen from the beginning of the line through the cursor position. |
| 'TP' - Transmit Line Protected | - Transmits all data on the screen, 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:

0120 INPUT 'TR', A\$

(A\$ contains everything that was displayed on the screen.) |
| 'TS' - Transmit Screen Protected | - Transmits all data on the screen, including protected fields, from the home position through the cursor position. |

MNEMONICS

O.S. CONTROL

The mnemonics in this section direct the operating system to perform certain functions which control how or whether devices receive data.

- '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'.) 'BI' is a data transfer mnemonic that causes the data being input from a device or communication process to pass through the driver without interference. This means that certain terminal commands, such as ESC, CAN, CTL+X,Y,S, or Q, will not be intercepted and executed by the driver.
- 'BG' - Begin Generation
 - 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 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.

MNEMONICS

O.S. CONTROL (continued)

- | | |
|--------------------------------------|---|
| 'EG' - End Generation of
Error 29 | - The generation of ERROR 29 (Undefined 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 VDT. |
| '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 resetting will clear it up.

MNEMONICS

Table 8-1. ALPHABETICAL LISTING OF MNEMONICS

<u>MNEMONIC</u>	<u>MNEMONIC NAME</u>	<u>MNEMONIC</u>	<u>MNEMONIC NAME</u>
@(x)	Horizontal Position	'CH'	Cursor Home
@(x,y)	Horizontal and Vertical Position	'CI'	Clear Input Buffer
'8L'	Eight LPI	'CL'	Clear Line
'10'	10 Pitch	'CR'	Carriage Return
'12'	12 Pitch	'CS'	Clear Screen
'16'	16 Pitch	'DC'	Delete Character
'B1'	Bin 1	'DN'	Down Cursor
'B2'	Bin 2	'EB'	End Blink
'BB'	Begin Blink	'EE'	End Echo
'BE'	Begin Echo	'EG'	End Generation of ERROR 29
'BG'	Begin Generating ERROR 29	'EI'	End Input Transparency
'BI'	Begin Input Transparency	'EL'	End Load
'BO'	Begin Output Transparency	'EO'	End Output Transparency
'BR'	Begin Reverse Video	'EP'	Expanded Print
'BS'	Backspace	'ER'	End Reverse Video
'BT'	Begin Input Buffering (Type-Ahead)	'ES'	Escape
'BU'	Begin Underline	'ET'	End Input Buffering
'CE'	Clear Screen to End of Page	'EU'	End Underline
'CF'	Clear Fore-ground	'FF'	Form Feed
		'IC'	Insert Character
		'LD'	Line Delete
		'KL'	Keyboard Lock

MNEMONICS

Table 8-1. ALPHABETICAL LISTING OF MENMONICS (continued)

<u>MNEMONIC</u>	<u>MNEMONIC NAME</u>	<u>MNEMONIC</u>	<u>MNEMONIC NAME</u>
'KU'	Keyboard Unlock	'S6'	Slew 6
'LD'	Line Delete	'S7'	Slew 7
'LF'	Line Feed	'S8'	Slew 8
'LI'	Line Insert	'SB'	Start Back-ground
'LT'	Cursor Left	'SF'	Start Fore-ground
'NL'	New Line	'SL'	Start Load
'OP'	Overprint	'SN'	Screen narrow
'PE'	End Protect	'SP'	Superscript
'PG'	Print Screen	'SS'	Subscript
'PM'	Plot Mode	'SW'	Screen Wide
'PS'	Start Protect Mode	'TL'	Transmit Line
'RL'	Ring Bell	'TP'	Transmit Line Protected
'RC'	Read Cursor	'TR'	Transmit Screen
'RT'	Cursor Right	'TS'	Transmit Screen Protected
'S2'	Slew 2	'UP'	Cursor Up
'S3'	Slew 3	'VT'	Vertical Tab
'S4'	Slew 4		
'S5'	Slew 5		

MNEMONICS

Most of the mnemonics just described work on each of the terminals that may be used with the MAIR^R 2000 Series system. Those that apply to one or more, but not to all, are listed in the following table. The allowable terminals are 7270, The Model 4309 Ergonomic Video Display Terminal (EVDT), and the Model 4310 Ergonomic Video display Terminal (EDT).

Table 9-2. MNEMONICS AND SUPPORTED TERMINALS

MNEMONIC	7270	EVDT	EDT
'BB'	NO	YES	YES
'BO'	YES	YES	NO
'BR'	NO	YES	YES
'BU'	NO	YES	YES
'EB'	NO	YES	YES
'EP'	NO	YES	YES
'ER'	NO	YES	YES
'EU'	NO	YES	YES
'TL'	NO	YES	YES
'TR'	NO	YES	YES
'TS'	NO	YES	YES

OVERVIEW

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, an error message is displayed on the user terminal in the following form:

```
!ERROR=nn : ERROR MESSAGE
```

where:

nn = a number identifying the type of error that has occurred.

ERROR
MESSAGE = a short message describing the error.

The statement causing the error is printed directly below the error number and/or message, and the system enters console mode.

The proper procedure is to correct the error as necessary, then type "RUN" to continue.

ERROR PROCESSING

If it is necessary to continue the program at a different statement, enter the following:

GOTO n

where:

n = the number of the statement to be executed

Then enter RUN.

The ERR (Code 1, Code 2, Code 3,...,Code n) function assists in determining which error occurred. The ERR function generates an integer which can be used in an ON/GOTO statement to construct a multiple branch. See ON/GOTO in Section 4.

CATASTROPHIC ERRORS

Catastrophic errors (ERROR 100 series; e.g., ERROR 103) occur during execution of certain statements which require more than one disk WRITE to complete. If an unrecoverable disk error occurs before the successful completion of all required WRITE directives (except for data records), an appropriate disk error code (103 or 104) is issued. Issuance of an ERROR 100 type diagnostic returns the task to console mode. ERR options are not taken.

If an ERROR 100 diagnostic appears, the user must assume that either the disk directory or Direct file Key Area has been written incorrectly, and take the appropriate action. The type of error must be determined and corrected before proceeding (see ERROR 3 and 4, which correspond to 103 and 104).

Some errors, though not catastrophic in nature, are in the 100+ range. The most notable are ESCAPE (ERROR 127) and CTL+Y (ERROR 126).

In most cases, correction of a Series 100 error requires contact with a Marketing Service Representative.

Following is a list of BASIC statements for which ERROR 100 diagnostics are issued:

ERROR PROCESSING

SAVE (when defining a file)

INDEXED

SERIAL

DIRECT

PROGRAM

SORT

ERASE

WRITE OR WRITE RECORD (Direct file, and only if
a new key is created)

REMOVE

The following subsection defines the causes of error codes generated by the system. The error codes are listed in numerical order. The paragraph title for each code illustrates the format in which the error code along with the message appears on the terminal.

When an error message displays, turn to the following subsection and locate the error. Then, review the list following that error until the cause of the problem is found. In some cases, correcting action is suggested, while in others, the procedure is obvious; e.g., an ERROR 21, INVALID STATEMENT NUMBER, results from the statement:

>LIST 99991

Correcting action in this case is the re-entering of the statement with the proper statement number, which cannot be greater than 16000.

ERROR PROCESSING

ERROR CODES This subsection describes error codes and what they mean.

!ERROR=0 FILE/RECORD/DEVICE BUSY OR INACCESSIBLE

This error occurs (usually after a few seconds delay) when an attempt is made:

1. To access a peripheral device (printer, floppy, 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 ERASE an open file. Do an END on all active terminals.
3. 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.
4. 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.
5. To LOCK a file already opened by another user. To correct, the file must be closed by the user that opened the file.
6. 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.
7. To START a task which had already been started.

ERROR PROCESSING

!ERROR=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 to 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.

ERROR PROCESSING

!ERROR=2 END OF FILE

This condition occurs when an attempt is made:

1. To READ/WRITE to a record using an IND value greater than the total number of records defined. To correct, redefine the IND of the READ/WRITE statement or enlarge the file.
2. To WRITE a greater number of records than are defined. To correct, define a new file using a new name and with a number of records greater than the current value. Then transfer the data from the old file to the new one.
3. To READ sequentially past the highest indexed record or the highest key. To trap this error, that is to keep the system from returning to console mode, enter an END= option in the READ statement.
4. 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.
5. On SERIAL files, to READ or WRITE a record too large to fit in the remaining file space.
6. To READ or WRITE a file opened with an ISZ= beyond the last record of a file. No error is given when attempting to READ or WRITE the last record of the file, even if it is smaller than the ISZ= value. To correct, adjust the ISZ= option.
7. By a non-ghost task to READ from a ghost task which is not in output mode.
8. To print to a spool file which is filled.
9. To READ a Serial file when the last access was a WRITE.

ERROR PROCESSING

!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.

!ERROR=4 DISK NOT READY

This error occurs when an attempt is made to:

1. Use an inoperative disk drive unit. To avoid/correct an ERROR 4 occurrence, do not use the inoperative disk drive unit; have it repaired.
2. WRITE to a disk when the media or drive is in the READ ONLY state (i.e., a floppy disk with the write protect tab on).
3. Perform operations on a drive when the media are "not ready" (e.g., a floppy disk is improperly installed or missing).

ERROR PROCESSING

!ERROR=5 PERIPHERAL DATA TRANSFER ERROR

This error occurs when:

1. A parity error occurs upon transmission to or from a terminal. A persistent error is indicative of a device malfunction.
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.

!ERROR=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 SECTOR POINTER OUT OF RANGE

This error occurs when an attempt is made to access a keyed file which has an out-of-range data pointer. A utility is provided to correct such a case and to recover the data in the file. See "File Utilities" in the User Guide.

ERROR PROCESSING

!ERROR=9 POWER FAILURE

Programs currently running on the MAI 2000 Series system are not managed during a power fail and cannot be restarted from a power-fail condition. The system is not 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 are specified as a file identification field of an INDEXED, SERIAL, DIRECT, 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 file name, combined with the prefix or selected directory name strings, exceeds the total allowable path name length, which is 128 characters.
3. 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".

ERROR PROCESSING

!ERROR=11 MISSING OR DUPLICATE KEY

This error occurs when an attempt is made to access a record of a Direct file using a KEY whose value is not equal to the key defined for any record of the file.

After taking the DOM= option on a WRITE statement, the ERR variable is set to 11.

!ERROR=12 MISSING OR DUPLICATE FILE NAME/ NON-CONFIGURED DEVICE

This error occurs when an attempt is made to:

1. OPEN 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, PROGRAM, SORT, SERIAL, FILE or SAVE statement.
2. ERASE a file that does not exist on the first directory in the current "select" 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, SERIAL, PROGRAM, SORT or SAVE statement when a file of the same name already exists on the first selected directory.
5. Define a disk data file or program by means of a DIRECT, INDEXED, SERIAL, PROGRAM, SORT or SAVE statement where the file name is the same as the name reserved for a system device (i.e., LP, P1, P2...P7, T0, T1...TF, MO, M1, G0...G3, SY).
6. ADDR/ADDE or DROP a program that is not found.
7. Use a path name containing a non-existent directory in any of the following directives: SAVE, PROGRAM, INDEXED, SERIAL, DIRECT, FILE, STRING.

ERROR PROCESSING

!ERROR=13 IMPROPER FILE OR DEVICE ACCESS

This error occurs when an attempt is made to:

1. READ or INPUT on an output-only device such as a printer.
2. WRITE or PRINT on an input-only device.
3. WRITE or PRINT to a Direct file when the statement does not include an IND or KEY option and the subject record has not been extracted.
4. READ or INPUT a disk data file using a statement that contains a constant or mnemonic.
5. WRITE to Serial file, if the file is not locked.
6. WRITE from a non-ghost task 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. See Appendix A for more information on ghost tasks.
7. ADDR/ADDE a non-program file to the public memory.
8. DROP a peripheral device or a nonresident program.
9. 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 as opposed to WRITE RECORD).

ERROR PROCESSING

!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, 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.
9. ADDR a program that you have already made resident in the public programming memory area. Other terminals may ADDR the same program without generating an error.
10. START a device (rather than a terminal or ghost task).
11. This error is also issued for a communications error that corresponds to command interpreter errors 803 and 805.

ERROR PROCESSING

!ERROR=15 DISK SPACE ALLOCATION ERRORS

No additional disk space is available for file growth or creation.

!ERROR=16 DISK OR PUBLIC PROGRAMMING MEMORY IS FULL

This error occurs when:

1. An attempt is made to define a disk data file or program using a DIRECT, INDEXED, 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. ADDE, ADDR, LOAD, RUN or SAVE a file that is not of type PROGRAM.
2. LIST or MERGE from anything other than an Index, String, or Serial file or a device.
3. VMERGE from anything other than a string file.
4. Use AND, IOR or XOR functions with different length arguments.
5. Execute a FILE statement with bad file parameters.
6. SAVE, without parameters, a null program area.
7. Use a KEY= option on an Indexed file.
8. Perform an input or output operation to a program file unless opened with the ISZ= option.
9. Specify 10 or more directories for the PREFIX directive.

ERROR PROCESSING

!ERROR=18 ILLEGAL CONTROL OPERATION

This error occurs when an attempt is made to:

1. DROP a program which is busy or which is not ADDR'ed.
2. SAVE a program that has been ADDR'ed.
3. SAVE a program to an OPEN program file.
4. Access a "protected" program (LIST, SAVE, PGM function, etc.)
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. LOAD or RUN a program with insufficient data space.
2. ENCRYPT a program whose actual size (PSZ) is close to the defined maximum size of the new program file. Encryption requires a few extra bytes.

ERROR PROCESSING

!ERROR=20 STATEMENT SYNTAX

ERROR 20 is a general catch-all error for the compiler. Illegal punctuation, non-existent or misspelled directives and incorrect syntax are just some of the causes of an ERROR 20.

In addition to compiler errors, several other situations can cause an ERROR 20. This error can occur when an attempt is made to:

1. 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 A\$(-1) or DIM A\$(33000).

NOTE

A distinction is made between entering a number that is out of range and executing a number that is out of range. The following merely enters a number that is out of range:

```
>00010 DIM A$(33000)
```

The next example, however, attempts to execute a number that is out of range:

```
>00010 LET A=33000  
>00020 DIM A$(A)
```

The first example results in an ERROR 20, whereas the second results in an ERROR 41. An ERROR 31 may occur due to limited data area for strings approaching the limit.

ERROR PROCESSING

7. Enter an exponent that is equal to or greater than 63 or less than or equal to -63. For example,

```
>00100 LET A= .0004E63
```

results in an ERROR 20. Note that the number can be reentered as .004E62 without generating an error.

8. Attempt to use a system variable as a variable name. For example:

```
>ATH = 8
```

!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, ON/GOSUB 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. Execute a CPL function on a text string which has no statement number.

ERROR PROCESSING

!ERROR=23 MISSING VARIABLE/
NON-DIMENSIONED STRING

This error occurs when an attempt is made to enter or execute a statement whose structure implies the absence of a variable, for example:

- o 00010 *ERR 23
00010 FOR5=1T010
- o 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, when the function name within the statement has been previously defined.

!ERROR=25 UNDEFINED FUNCTION

This error occurs when an attempt is made to execute a statement containing a user-defined function 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 PROCESSING

!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 an out-of-range number will sometimes generate an ERROR 26. Example:
INPUT A\$:(LEN=1,33000).
6. Convert non-hexadecimal characters via the ATH function or to input non-hexadecimal characters to a hexadecimal string (A\$=\$ZZ\$).

!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.

ERROR PROCESSING

4. Execute an EXIT from the main (not Public) program.
5. 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.

!ERROR=29 UNDEFINED MNEMONIC

This error occurs when an attempt is made to:

1. Enter or execute a statement containing a mnemonic constant or a positioning expression on an inappropriate device, for example "@ (10,10) on a printer.
2. 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.
3. EXIT back to a calling program which has been modified.
4. Run a program in which the interpreter detects invalid or corrupt code.

NOTE

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 BASIC interpreter is corrupted, and so it was made a separate error.

ERROR PROCESSING

!ERROR=31 INSUFFICIENT MEMORY WITHIN TASK

This error occurs when an attempt is made to:

1. ENTER or MERGE a statement which, if added to the program, would make the program too large to fit in the available user area. To correct, see number 2 below.
2. EDIT an existing statement to increase its length to the extent that the additional program area required would make the program too large to fit in the available user area. To correct:
 - a. SAVE the program, enlarge the user area, LOAD the program and continue; or
 - b. Split the program and add statements to initiate overlay; or
 - c. Reduce the size of the existing program to provide space for the coding to be included.
3. Execute a program whose operation has filled the user area. The specific action that caused the error is usually the addition of a new variable or the lengthening of an existing variable. In either case, it is likely that a loss of data has occurred and the program must be rerun following corrective action. It is possible that the error was due solely to failure to CLEAR the user data area prior to the execution of the program.
4. Execute string manipulations within a program which temporarily require more data area than is available. After the error occurs, the data area is returned to the size remaining prior to the string manipulation.
5. Enter a statement via a terminal keyboard when the user area is almost full (this is a less common cause for the error). In this instance, the error results from the fact that all console mode keyboard entries are stored in a buffer within the user area prior to processing of the carriage return terminator. To correct:
 - a. Enlarge the size of the user area using the START command.
 - b. CLEAR the data area (if possible) prior to execution (or revision) of the program. If clearing is not appropriate, select one or more unnecessary string variables of sufficient length and set their values to null; or

ERROR PROCESSING

- c. Modify the program so that less data is required (e.g. remove REM statements).
6. LOAD or ADDR a Program file that has non-valid data.
7. Execute a CALL statement with insufficient data space available to store the CALL stack information.

!ERROR=32 STACK OVERFLOW

This error occurs when internal storage for BASIC program management has been exhausted. This is caused rarely, if ever, and is due to a number of variables, such as the use of recursion, such as an IOLIST statement that loops on itself.

For example:

```
0010 PRINT IOL=0020
0020 IOLIST IOL=0030
0030 IOLIST IOL=0020
>RUN
```

ERROR=32

!ERROR=33 INSUFFICIENT MEMORY CAPACITY

This error occurs when an attempt is made to execute a START or CALL statement requesting allocation of task memory space when the available (unused) memory is less than that required to satisfy the request. To correct:

1. Reduce the size of the user area requested by the START statement.
2. RELEASE any terminals not in use; or
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 may be advisable to purchase more memory.

ERROR PROCESSING

!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; or
3. Design an application program such that INPUT statements are executed more frequently.

!ERROR=35 COMPILER OUT OF MEMORY

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.

!ERROR=36 CALL/ENTER VARIABLE MISMATCH

This error occurs when:

1. The number of variables or the mode of the variables are not consistent between CALL and ENTER statements.
2. ENTER is executed more than once in a called program.
3. An attempt is made to execute ENTER in a main (not public) program.

ERROR PROCESSING

!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:

EXECUTE	LIST	RUN	ESCAPE
DELETE	MERGE	SAVE	VMERGE

2. Attempt to pass the same argument more than once in a CALL.
3. Restart from within a public program.

!ERROR=39 ESCAPE IN PUBLIC PROGRAM

This error occurs when ESCAPE is pressed in a public program and there is no active SETESC.

!ERROR=40 NUMERIC VALUE OVERFLOW

This error occurs when an attempt is made to execute a statement involving arithmetic operations that result in an absolute numeric value less than $10^{-63}+1$ 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 if a zero value divisor was generated.

ERROR PROCESSING

!ERROR=41 INVALID INTEGER RANGE

This error occurs when an attempt is made to:

1. 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 $2^{23}-1$ records).
 - c. The record size (maximum 32256 bytes).
 - d. An IND or ISZ= value (maximum $2^{23}-1$).
 - e. At position @ (maximum 255).
 - f. A subscript (range = 1 to 32767).
 - g. A program size (maximum 32256 bytes).
 - h. A PRECISION (maximum 14).
 - i. An ON/GOTO statement whose expression field results in a value greater than 32K.
 - j. A power (^)(maximum 255).
 - k. A key size in a DIRECT or SORT statement (maximum 56, minimum 1).
 - l. An increment length in a POS statement (maximum 32K).
 - m. A START size where size is less than 10 or greater than 65535.
 - n. A BIN function length (max=32767)

Entering a statement with out-of-range numbers sometimes results in an ERROR 20. An attempt to execute a statement that uses an out-of-range value results in an ERROR 41.

2. Execute the CHR code conversion function of a value that is less than zero or greater than 255.
3. Close file 0.

ERROR PROCESSING

!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. Return a POS function with a length field of zero.

!ERROR=43 INVALID FORMAT MASK SIZE

This error occurs when an attempt is made to execute a 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 have 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.

ERROR PROCESSING

!ERROR=45 INVALID STATEMENT USAGE

This error occurs when an attempt is made to:

1. Enter a statement which is restricted to console mode only, including a statement number (indicating program mode).
2. Enter a DELETE or LIST command that references descending statement numbers.
3. Execute a statement with a TBL= option or TBL function that references a statement number which is not a TABLE statement.
4. Enter a statement (EXECUTE, FOR, NEXT, GOSUB, RETURN or RETRY) in console mode which is available in program mode only.
5. Enter a statement with an IOL= option that references a statement which is not a valid IOLIST statement.

!ERROR=46 INVALID STRING SIZE

This error occurs when an attempt is made to:

1. Execute a statement whose KEY= field defines a key to a Direct or Sort data file whose length exceeds the key size inferred by the keysz field of the associated DIRECT or Sort statement.
2. Execute the ASC function with a null argument (string length = 0).
3. Enter other than eight characters with the SETDAY statement. This error is issued only if the length of the string is incorrect.
4. Enter a SETDAY string expression in a format other than mm/dd/yy.

ERROR PROCESSING

!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

This error occurs when a program is being translated from one level of BASIC to another and the translator encounters a statement that cannot be translated.

!ERROR=50 GENERAL MEMORY ERROR

An ERROR 50 indicates that a problem exists in the operating system. If this error occurs, please call a Product Support Representative.

ERROR PROCESSING

!ERROR=54 OPEN OF A SERIAL FILE WITH INVALID HEADER

This error occurs when an attempt is made to open a Serial file with an invalid header.

!ERROR=68 BAD SECOND ARGUMENT TO CPL OR LST

This error occurs when an attempt is made to use the CPL or the LST functions with a second string parameter that is not in the correct format. For the correct format, see the description of the CPL and LST functions in Chapter 5.

!ERROR=69 MISSING VARIABLE ID

This error occurs when the CPL function has the second argument in the correct format, but one or more of the variables in the first argument do not exist in either of the two variable id tables that comprise the second argument.

If an ERROR 69 occurs, the third argument to the CPL function, which is a string variable, is given the id of the first variable or user-defined function which was in the first argument but not described by the second argument. This id is null-terminated. If the id is that of a numeric array, the first byte has hex 80 added to it.

!ERROR=70: THIRD ARGUMENT TO CPL FUNCTION IS NOT AN ACTIVE VARIABLE

This error indicates that the string variable given as the third argument of the CPL function has length 0.

!ERROR=71: THIRD ARGUMENT TO CPL FUNCTION IS NOT LONG ENOUGH

This error indicates that the string variable passed as the third argument for the CPL function is not long enough to accommodate the particular id. For example, if the id is 3 characters long, the third argument to the CPL function must have length of at least 4: 3 for the id and 1 for the null terminator. The maximum length that could be needed for the third argument is 12 bytes.

ERROR PROCESSING

!ERROR=95 LAN ERROR

This error indicates that an error has occurred in the LAN (Local Area Network) process. It corresponds to the command interpreter error codes that are generated during LAN operations.

!ERROR=98 SPOOLER ERROR

This error indicates that an error has occurred in the spooling process, which is the process that coordinates the multi-user, multi-task access to the configured printers.

The ERROR 98 corresponds to the command interpreter error codes generated during spooling operations.

!ERROR=99 COMMUNICATIONS ERROR

This error indicates that an error has occurred in the communications process. It corresponds to the command interpreter error codes generated during the communications process.

!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 PROCESSING

!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 CTL+Y KEY USED

Use of the |CTRL|+Y operation 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

The ERROR 254 occurs only during a SAVE operation on a BASIC program that has become corrupted, as does ERROR 30. However, ERROR 254 may indicate that the BASIC interpreter itself has become corrupted. Thus, it 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.

OVERVIEW

To enter BASIC console mode from the command interpreter mode, the user types "basic" following the prompt. The word "basic" may be followed by one or more of the following options:

-help	-e
-nr	-q
pgm=	-trans
s=	-x
lib=	

{any string expression} (this option is referred to as the "command field")

A discussion of each option follows:

COMMAND FIELD

The command field is made up of a single string expression. The string, which may be in quotes, contains any BASIC directive that may be entered in console mode. Immediately upon entering the BASIC console mode, that directive will be executed. For example:

```
username>basic 'A$="HELLO";PRINT A$'
```

The first thing that occurs on entering BASIC, even before it prints "READY", is that "HELLO" is assigned to A\$ and then is printed. Semicolons are considered part of the string.

The main restriction on command line options is that any of the switches described below must precede the command field string, if there is one. A list of the possible switches and a brief discussion of each follows:

-help This field displays the command line options for the BASIC command.

-nr This field may be used with pgm= and the command field. If it is not used, the system returns to the command interpreter mode after running the specified BASIC program or executing the specified directives.

If the -nr option is used, the user remains in the BASIC environment, and the effects of the command that was executed remain (such as variable assignment).

COMMAND LINE OPTIONS

pgm= This option takes a string argument containing the name of a BASIC program. The program is loaded and then run. When used with command string, the command is executed before the program is run.

s= This option requires a numeric argument (for {size} example, `-size 40`). The option is equivalent to the `START` directive, and determines the number of pages allocated to the user upon entering BASIC. The option `"s= 60"` initializes the user area to 60 pages (or 15K bytes). The maximum is 65535 pages.

If the `"s="` field is not specified, BASIC will be initialized with a default user area of 32 pages (7.5K bytes).

lib= This option takes a string argument which is the name of a BASIC library containing one or more C programs which may be called from BASIC.

If the string argument is a BASIC library, BASIC loads the library into memory before the user is presented with the BASIC environment. The programs in the library are then available to be called by the `LIB` command.

-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 initiates "quiet mode". The text of error messages will not appear, nor will the initial BASIC release level upon entering BASIC.

If a syntax error occurs, for example, the system will display only:

`!ERROR 20`

(without explanatory text.)

-trans translation file name

This parameter specifies a translation file used for translating file names. When a program references a file, the translation file is checked. If the file is specified as a source file, the program obtains the translated file name from the translation file. If the file name does not occur, no translation is done.

COMMAND LINE OPTIONS

The translation file is a string file with the following format:

- The first line must contain the seven characters:

EBS001:

- Each of the remaining lines has the following format:

<delimiter><source file name><delimiter>
<translated file name>

The delimiter can be any character. For example:

:TEST:/sta/xx/source/TEST

- x This parameter will prevent escapes and CTRL + C from terminating a program. The only way to terminate a runaway program, when the -x option is in effect, is to kill it from another terminal. Therefore, it should be used with caution.

EXAMPLES

1. username>basic -help

This option displays the menu of options.

2. username>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. username>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. username>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.

APPENDIX A - FEATURES OF THE BUSINESS BASIC PROGRAMMING ENVIRONMENT

OVERVIEW

The appendix describes features of the operating system that may be used within the BASIC environment. For a discussion of the operating system, command interpreter, filesystem and file structures, device handling, etc., please refer to the User Guide.

Described here are the following: public programming, ghost tasking, directory access methods, specifying print job attributes, and input buffering. Also described are a few elements of BASIC programming: field protection, null output effect, and special key controls.

MULTI-TASKING

The BOSS/IX operating system permits multi-tasking, that is, the concurrent execution of more than one operation. The BASIC programming environment allows you to take advantage of this feature.

Some programs (such as report printing) do not require any action by an operator, and therefore do not require a video display terminal. Such programs can be activated as "ghost tasks" as described below.

GHOST TASKS

A ghost task is one which is not dependent on a terminal for operation. Examples are print programs and file updating programs. Ghost tasks are started from any other task, or by the operator using a terminal. The START command is used to start a ghost task. The task name must be a full path name.

Example:

```
0010 START 20, "/fin/reports/PRINT", "G0"
```

where "START 20" indicates 20 pages of user memory, "/fin/reports/PRINT" is the name of the program and G0 is the name of the ghost task. Up to eight ghost tasks can be configured on this system (G0 through G7).

When a ghost task is finished, it should execute a RELEASE statement. This releases the ghost task's memory for reassignment to another task.

The following code allows a task to 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:

- o The program cannot attempt to communicate with a controlling terminal because none is assigned; and
- o A SETERR should be executed at the beginning of the program to prevent an error which might cause a return to console mode (which requires a terminal for output of the error message, the ">", etc.).

COMMUNICATION WITH A GHOST TASK FROM A TERMINAL

It is possible to communicate with a ghost task from a "standard task" (one with a controlling 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 task is trying to read from a keyboard, the user may only WRITE to the ghost. An attempt to READ results in an ERROR.
- b. If the task is trying to write to a VDT screen, the user 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. These 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 user's procedures 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 A$
```

3. The terminal then displays:

ENTER DATA HERE:

which satisfies the output directive.

4. The ghost task then halts on the following statement,
INPUT A\$

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 (1)
```

and the task continues until it releases itself or is reopened by another user.

PUBLIC PROGRAMMING

The main objective of public programming is to reduce the time required to access frequently called programs. This is done by putting one copy of frequently used programs, utilities, and subroutines into the shared memory area so that subsequent calls need not perform a directory search, go out to disk and load the program into memory. When you ADDR a program, it resides in the shared memory area, and the BASIC driver has pointers to its location. It does not affect your data area, and a CALL or RUN transfers control to the program in shared memory very quickly.

When another terminal task performs and ADDR on the same program, it, too, has pointers to the program in shared memory. If the second terminal task does not ADDR the program, a directory search is performed, the program found to be already resident, and control passed to it. Only one copy is resident at a time. If the first user executes a DROP, the program remains in public memory only until the second user exits.

The operating system manages an area of main memory for maintaining the public (shared) programs and all open files or devices. There are control block entries available for this purpose, 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 attempt to open a file or device or to CALL or ADDR a program results in an ERROR 16.

The ADDR command can, thus, be used 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 public programs.

For programs not entered, 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 PROGRAMMING

The following statements cannot be executed from a public program. If an attempt is made to do so, an ERROR 38 results.

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.

CALLING C PROGRAMS

The BASIC LIB directive causes the execution of a C program. As with the CALL directive, arguments may be passed from one program to another. However, the arguments must be specially formatted. The C program may also return values to the BASIC program through those parameters and by the optional RET= clause.

The advantages to calling C programs include:

1. Improved speed and performance. C programs can be written to run faster than BASIC programs that do the same thing.
2. C programs may be able to do things like graphics that are unavailable in Business BASIC.
3. Existing C software applications may be run.

A BASIC library of C programs is required to call these routines because they must be linked in a special way. More than one C program may be included in the library. To convert C programs to a BASIC library, a command interpreter utility called "makelib" is provided.

If BASIC is to make use of a BASIC C library, the "lib=" option is used on the command line which invokes BASIC from the command interpreter mode. The argument to "lib=" is the library name.

INFORMATION DELETED

INFORMATION DELETED

SPECIFYING PRINT JOB ATTRIBUTES

SPOOLING

Spooling is a feature that is built into the operating system to make it easier to manage all the print jobs that a system with multiple users or many different jobs would require.

Essentially all the user needs to do is specify the number of copies, the priority, the format, and so on. Here is a list of the kinds of attributes that you can select:

- key - a one to six character name that may be one of the following:
1. User name (for a personal defaults file)
 2. The name of another class file for options to be selected before reading the rest of this file
 3. A printer name for its defaults file
- These file entries must be created by the user.
- priority - a number ranging from a low of 0 to a high of 9 to be assigned to the print job.
- form name - a one to six character name entered in the / etc/forms file defining the form attributes. These file entries must be created by the user.
- copies - a number from 1 to 999 for the number of copies to be printed.
- delete - either "y" or "n" (yes or no) to tell the system whether or not to delete the file upon completion of the print job.
- requeue - either "y" or "n" (yes or no) to tell the system whether or not to place this print job back in the queue again after completion.
- notify - either "y" or "n" (yes or no) to tell the system to notify the user when the print job is completed.
- spooling - either "y" or "n" (yes or no) to tell the system to process the job through the spooling queue or send it directly to the device.
- despool wait- - either "y" or "n" (yes or no) to wait on printing this job or not to wait for reception of the entire file.

For detailed information on the spooling system, refer to the appropriate chapter in the User Guide. The next paragraphs describe the format for entries in the /etc/class file to be used as parameters for the OPEN statement.

CLASS FILE

The I/O Option used with the OPEN directive called CLASS="string" specifies the manner in which a print job is to be done. The names used in the string must be entries in the class file.

The system file "/etc/class" is designed to hold a selection of the print job options described on the last page. You may specify a name for the file entry and one or more of the other options. You may leave any field empty except for the "key" or name field.

The system takes any options found in the class file, or key, first. If some options are not selected in the class file, it will next take the options specified in the options list, if any, (see below). It will then search for any other required options in the printer default file or, last, the system default file. These last two are entries in the /etc/defaults file.

The Printer Parameters Utility is supplied with 7.2 to maintain the class file and other printer parameter files.

OPTIONS

The I/O Option used with the OPEN directive called OPTS="string" allows you to use the same selections for a print job. However, the format for making selections is that used in the "lpr" command. These are the options:

- alias= (name of the job used when notifying the user)
- class= (name of an entry in the /etc/class file)
- copies= (number of copies to be printed)
- delete (this switch causes the file to be deleted)
- form= (name of an entry in the /etc/forms file)
- list= (printer name)
- off (this switch turns spooling off)
- notify (this switch causes your terminal to be notified when the job is finished)
- priority= (a number from 1 to 9)
- raw (no mnemonics processing or character translation)
- requeue (this switch causes the job to be requeued upon completion)
- spool (this switch turns spooling on)
- time= (earliest time/date to begin despooling)
- vfu (this switch causes the activation of the vertical form unit)
- wait (this switch tells the despooler to wait on printing the job until the whole file is received)

Procedure

The procedure for using the OPTS="string" option on the OPEN statement is as follows: first, define a string variable with any of the above options, and then use the variable in the OPTS= string position. The order is not important. Simply omit unwanted options. Here is the statement format:

Format

```
OPTS="alias=(name) class=(name) -delete form=(name)
list=(name) -off -notify priority=(number) -raw -requeue
-spool time= -wait -vfu"
```

Example Format

```
>LET A$="alias=expense_report class=system -notify"
>OPEN (7,OPTS=A$)"PO"
```

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.

CLEARING THE INPUT BUFFER

'CI', the "clear input" mnemonic, provides a means to insure that no unprocessed input is used at critical prompt points in a program. The execution of 'CI' in a statement clears all data in the input buffer. A statement such as:

```
INPUT 'CI', "PLEASE REENTER DATE:  ", A$
```

clears any data in the input buffer, prints the character string, and waits for the operator to enter the field. Subsequent inputs are then buffered as they were before the execution of this mnemonic. 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 Section 8).

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.

PROGRAMMING

When operator verification of system output is required, the 'CI' mnemonic should be used on the input statement. This forces the operator to wait for the system prompt before keyboard input is accepted.

Example:

```
0090 PRINT (0,ERR=1010)"BALANCE=", A
0100 INPUT (0,ERR=100) "CORRECT? (YES/NO)", 'CI',
0100: D$:( "YES"=650,"NO"=725)
```

Input buffering can be disabled by use of the 'ET' mnemonic.

EXAMPLE OF INPUT
BUFFERING

```
0010 BEGIN
0020 SETERR 0500
0030 FOR X = 1 TO 20000
0040 REM "THIS LOOP IS TO SIMULATE PROCESSING TIME
0050 NEXT X
0060 INPUT "ENTER A:",A
0070 INPUT "ENTER B:",B
0080 INPUT "ENTER C:",C
0090 PRINT 'CI',
0100 INPUT "ENTER D:",D
0110 INPUT "ENTER E:",E
0120 PRINT "HERE ARE THE RESULTS:",A,B,C,D,E,
0200 STOP
0490 PRINT 'CI'
0500 ON ERR (26, 34) GOTO 0510, 0530, 0550
0510 PRINT "PROGRAM TERMINATED BECAUSE OF ERROR",
0510:ERR; STOP
0530 PRINT "ENTER ONLY NUMERIC DATA,"; WAIT 2; RETRY
0550:PRINT "YOU HAVE EXCEEDED THE INPUT BUFFER AREA.
0550:PLEASE REKEY DATA"; WAIT 2; RETRY
1000 END
```

The preceding program can be used as a sample method of handling input buffer overflows and other errors that affect the state of the input buffer. The loop beginning at statement 30 is used as a timing loop to allow the filling of the input buffer. To overflow the buffer, key in more characters within the time of the loop.

When statement 60 is executed (the first I/O statement encountered after the buffer overflow), an error branch occurs at statement number 0550 and the overflow error message is printed. The input buffer is cleared automatically, and all input accumulated in the buffer is cleared.

An example of the 'CI' mnemonic appears in statement 90. This means that the buffer area is cleared at this point and the next input line, "ENTER D:", always waits for a response.

In the example, an ERROR 26 occurs if an alpha character is entered. An error branch takes the program to statement 530 and the error message is printed. Since error processing does not clear the input buffer, input statements after an error condition takes their data from the input buffer. Consequently, the 'CI' mnemonic should be used in the statements processing the error (see Examples 2 and 3).

The following data tests the example:

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:

BRANCHING

Some directives cause program control to transfer to another statement number when certain conditions exist, as a method of program control. These directives are:

GOTO	ON/GOTO
GOSUB	SETCTL
EXITTO	SETESC
	SETERR

Some I/O options also transfer program control. These include:

DOM=	ERR=
END=	

FIELD PROTECTION

The terminal driver supports mnemonics which protect display fields from being overwritten. Protected fields are written in Background Mode, and once 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; then Protect Mode must be initiated('PS').

The following mnemonics are associated with Field Protection, and are fully described in Section 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 terminal from Background ('SB') to Foreground mode:

'CE'	'DC'
'CF'	'IC'
'CL'	'LD'
'CS'	'SF'

Start of screen scroll

3. Use of the following mnemonics when 'PS' (protect mode on) is in effect are ignored by the VDT:

'LD'
'LI'
'CL'

4. Use of the following mnemonics reset protect mode:

'CS'
'CF'
'PE'

5. Following execution of 'PS', the cursor is at home position (0,0).

NULL OUTPUT
CHARACTERS

Level 7 systems do not count NULL characters (\$00\$ or \$80\$) as printable characters on the display screen.

\$00\$ does not move the cursor.

SPECIAL KEY
CONTROLS

Level 7 provides additional keyboard controls accessed by use of the CONTROL key in conjunction with X, Y, S or Q. Use of one of these combinations results in the following action:

- |CTRL| + X - used to generate an ERROR 5.
- |CTRL| + Y - used with SETCTL to shift program control to a specified statement.
- |CTRL| + S - causes task to stop processing. The task can be restarted from where it left off by use of |CTRL| + Q. It is often used with the LIST directive.
- |CTRL| + Q - used to begin processing at the point where processing was stopped by use of |CTRL| + S.

APPENDIX B - INTERNAL CHARACTER CODE

OVERVIEW

The following is a conversion table based on industry standard 8-bit ASCII character codes. The far left column indicates the binary representation of the most significant bits (MSB), otherwise referred to as the high order position or "nibble". In the column immediately to the right of each binary number is its hexadecimal representation.

Across the top of the chart is the binary representation of the least significant bits (LSB), otherwise referred to as the low order position. Immediately below each binary number is its hexadecimal representation.

Reading across and down to form an 8-bit word, you arrive at a box containing in the upper portion the corresponding character or code and in the lower portion the corresponding decimal representation.

HOW TO USE

To use the chart simply select a binary or hexadecimal number from the left column and a corresponding one from the top row and read across and down to the character or decimal representation of the combined four-bit words. To reverse the procedure, select a number or character and read to the left column and top row to find the corresponding binary and/or hexadecimal representation.

Table B-1. ASCII Character Conversion Table (U.S. English)

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	MUL <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>

EXAMPLE

Find the capital letter "A" in the chart. Reading to the left we find a "4" in the hexadecimal column; reading up we find a "1" in the hexadecimal row. The hexadecimal number that corresponds to "A" is "41".

The binary representation is "01000001". This number is derived by combining the left column binary digits with the top row binary digits.

The number "65" immediately below the "A" is the decimal number that corresponds to "A".

SORT SEQUENCE

The ranking of each character in a list of sorted alphabetic characters depends upon the character's ASCII value (e.g., the lowest ranked printable character is a space, ASCII value 00000000; and the highest ranked printable character is a lower-case underlined o, ASCII value 11111110). Key values are compared character by character, left to right, and the key with the first higher ranking character is ranked higher; but if the ASCII values of two keys are equal up to the length of the shortest key, then the longer key is ranked higher. When accessing keys in sorted order, lowest ranked keys are accessed first.

EXPLANATION OF CODES

Table B-2. Non-Printable ASCII Codes

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

The the non-printable ASCII codes are repeated on the conversion table. They are assigned also to the hexadecimal numbers from \$80\$ through \$A0\$ and \$FF\$, \$FF\$ being the delete code.

INTERNATIONAL CHARACTERS

The hexadecimal numbers from \$A1\$ through \$FE\$ have been assigned international and special characters, as shown in the following chart. Refer to the terminal operator guide for the 4309 and 4310 models of the Ergonomic Video Display Terminal for procedures to invoke these characters.

Other MAI Basic Four computers have the special characters assigned to the low order ASCII values, but the eighth bit is always set. Therefore, the American English characters are assigned to the hexadecimal numbers between \$A0\$ and \$FF\$ on those systems. The international characters are assigned to the hexadecimal numbers from \$21\$ through \$7E\$. Refer to the system manuals and terminal operator guides for access to international characters on other systems.

Table B-3. ASCII Character Conversion Table (International)

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
1000	8	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
		<128>	<129>	<130>	<131>	<132>	<133>	<134>	<135>	<136>	<137>	<138>	<139>	<140>	<141>	<142>	<143>
1001	9	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
		<144>	<145>	<146>	<147>	<148>	<149>	<150>	<151>	<152>	<153>	<154>	<155>	<156>	<157>	<158>	<159>
1010	A	SP	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ó
		<160>	<161>	<162>	<163>	<164>	<165>	<166>	<167>	<168>	<169>	<170>	<171>	<172>	<173>	<174>	<175>
1011	B	Ø	Ù	Ú	Û	Ü	Ý	Ç	Ñ	ã	ē	î	õ	ü	ï	ð	ß
		<176>	<177>	<178>	<179>	<180>	<181>	<182>	<183>	<184>	<185>	<186>	<187>	<188>	<189>	<190>	<191>
1100	C	à	â	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ó
		<192>	<193>	<194>	<195>	<196>	<197>	<198>	<199>	<200>	<201>	<202>	<203>	<204>	<205>	<206>	<207>
1101	D	ø	ù	ú	û	ü	ý	ç	ñ	í	î	ï	ð	é	ê	ë	ì
		<208>	<209>	<210>	<211>	<212>	<213>	<214>	<215>	<216>	<217>	<218>	<219>	<220>	<221>	<222>	<223>
1110	E	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó
		<224>	<225>	<226>	<227>	<228>	<229>	<230>	<231>	<232>	<233>	<234>	<235>	<236>	<237>	<238>	<239>
1111	F	Ö	Γ	Τ	τ	†	+	+	×	¥	μ	©	®	À	™	Ω	DEL
		<240>	<241>	<242>	<243>	<244>	<245>	<246>	<247>	<248>	<249>	<250>	<251>	<252>	<253>	<254>	<255>

OVERVIEW

Preceding and following each BASIC program is a header that contains information about the program, such as length, DEF tables, Auto-End, flags, etc. The following is a description of the 33 bytes that are fixed in the format.

FORMAT

<u>BYTES</u>	<u>CONTENTS</u>
0-3	- Business BASIC revision level in 7-bit ASCII
4-7	- Length of program compiled code
8-11	- Miscellaneous flags
12-15	- Miscellaneous for encrypted programs
16-17	- 6: length of DEF area
18-19	- 2: length of DEF statement number table
	DEF statement number table goes here
20-21	- 2: length of DEF id table
	DEF id table goes here
	Compiled code goes here, between the DEF tables and the 65535 END statement.
22-23	- 5: length of 65535 END statement
24-25	- 65535
26	- hex 43: opcode for END
27-28	- 6: length of hash field
29-32	- hash field (not used)
	Variable tables go here

There are two DEF tables. The DEF statement number table consists of a two-byte length field followed by two-byte entries which are statement numbers corresponding to the user-defined functions in the program. The DEF id table consists of a two-byte length field followed by id entries which correspond to the user-defined functions in the program. An id entry consists of a one-byte length field, followed by a flag which is 1 if the function has an associated DEF statement and 3 if the function does not, followed by the id (function name) itself.

VARIABLE
TABLES

One of the key differences between Level 4 and Level 7 BASIC is in the use of variables. Level 4 BASIC uses a coding system that identifies the variable precisely. In reading the compiled format, you could interpret the variable labels. For example, a compiled statement that referenced "E101" indicated a numeric variable called "A". Level 7 compiled BASIC uses the same variable labels, but they no longer identify the variable. The label is used only to indicate the variable type and the order in which the variables were created. E0001 is the first numeric variable; EA0001 is the first string/numeric array variable.

This enhancement is largely transparent and presents no compatibility problems unless you are using the CPL and LST functions on statements from a program other than the main program you are running. The system has no way of identifying the variable names from the compiled code labels unless you provide the variable table for that program.

CPL/LST

To allow you to accomplish the task of compiling and listing statements from other program files, the CPL and LST functions may take a second parameter. You may use the functions with only the program statement, but the variables used are from the main program. The second parameter identifies the variables explicitly, and the variables used in the statement being compiled must be in the ID table.

Four errors are provided to identify problems: ERROR 68 is issued when the ID table is not in the correct format. ERROR 69 is issued if there is no variable table, or the variables in the statement being compiled are not in the table. Error 70 is issued if the third argument is not an active variable. Error 71 is issued if the third argument is the wrong length. These are explained more fully in Chapter 9.

When you want to use the listed form of a compiled statement, the variable name is taken directly from the ID table. How the variable is described in the ID table determines how it is listed.

FORMAT FOR
VARIABLE TABLES

There are eight variable tables. The tables must start on an even boundary, so if the hash field ends on an odd boundary, a 1-byte hole 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 the entire table.

The variable tables give information about the variables in the current BASIC environment. This information is kept in a form which:

1. Allows fast access to variables when a BASIC program is executing or LISTing.
2. Allows a fast LOAD or RUN of another BASIC program, when the variables in the incoming BASIC program must be merged with the variables in the present environment.

The eight variable 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 number 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 numeric id table, used for strings and numeric arrays.
5. String sort table: similar to numeric sort table, used for strings and numeric arrays.
6. String offset table: similar to 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.

INFORMATION DELETED

INFORMATION DELETED

VARIABLE TABLE
UTILITIES

There are three utility subroutines provided for you to utilize the variable tables on program files:

- /util/OPNTBL
- /util/CLSTBL
- /util/GRWTBL

OPNTBL

The utility that accesses the variable table for a program file is called "OPNTBL". It requires three parameters:

- the program file name for the table you want
- a logical unit number
- a string variable

For example, the program file "MYPROG" is opened on logical unit 2, and the variable table placed into the string variable T\$:

```
CALL "/util/OPNTBL","MYPROG",2,T$
```

CLSTBL

If you are creating a program file from a BASIC program without using the directives "PROGRAM" or "SAVE", the variable table for that program must be written out to that file. Two parameters are required:

- the logical unit number
- the variable table

For example,

```
CALL "/util/CLSTBL",2,T$
```

writes the variable table contained in T\$ to the file open on logical unit 2 just before it is closed. This procedure is required only if a program is being created by actually writing records to the file.

GRWTBL

When using the CPL function it is possible that modifications to the string being compiled would cause a new variable to be created. If so, an ERROR 698 is generated. To solve the problem, you must enlarge the table and then retry the CPL function. The routine "GRWTBL" is used to enlarge the table. Only one parameter is required: the string variable containing the table. For example,

```
500 IF ERR=698 THEN CALL "/util/GRWTBL",T$;RETRY
```

RELATIVE
BRANCH
ADDRESSES

The other major difference between Level 4 and Level 7 program files is relative branch addresses. Each statement containing a reference to another statement has a two-byte relative branch address to improve performance when branching occurs. A search through the entire program need not be performed. Instead, the branch goes directly to the address of the destination statement. The statements affected are:

- DOM= • GOSUB • ON/GOTO • SETESC
- END= • GOTO • SETCTL • TBL
- ERR= • IOL= • SETERR • TBL=
- EXITTO • ON/GOSUB
- DELETE (in program mode)
- INPUT (with branch list)
- LIST (in program mode)

When a SAVE is performed, the addresses are set and the "fast" flag is set. If a program is modified, say as the result of an error that drops into console mode, the "slow" flag is set, and a search for the destination statement is performed for each branch. This "slow mode" occurs because the addresses are relative memory addresses based on where they occur relative to the beginning of the program. The fast mode will not resume until the program is re-saved.

APPENDIX D - ALPHABETICAL SUMMARY OF
DIRECTIVES, FUNCTIONS,
VARIABLES AND I/O OPTIONS

<u>TASK*</u>	<u>TYPE</u>	<u>DESCRIPTION</u>
!	Directive	Executes BOSS/IX command
ABS	Function	Return absolute value
ADDR/ADDE	Directive	Add program to shared memory
AND	Function	Combine the bits of two strings
ASC	Function	Convert string to decimal
ATH	Function	Convert ASCII to hexadecimal
BEGIN	Directive	Reset system; clear variables and close all OPEN files and devices
BIN	Function	Return binary value
CALL	Directive	Transfer program control to another pgm
CHR	Function	Convert numeric expression to ASCII
CLASS=	I/O Option	Specify the spooler class file
CLEAR	Directive	Reset system; clear variables; does not close any OPEN files or devices
CLOSE	Directive	Release file or device
CPL	Function	Compile BASIC statements
CRC	Function	Check for data integrity
CTL	Variable	Return field terminator last used
DAY	Variable	Return system date
DEC	Function	Convert binary to signed decimal
DEF FNx(\$)	Directive	Define arithmetic operation or string expression
DELETE	Directive	Remove statement(s) from a program
DIM	Directive	Define an array
DIRECT	Directive	Define a Direct file
DOM=	I/O Option	Transfer program control if duplicate or missing key
DROP	Directive	Remove program from shared memory
DSZ	Variable	Return unused bytes in user task memory area
EDIT	Directive	Add, replace or delete characters in a statement
ENCRYPT	Directive	Encode program statements
END	Directive	Terminate a program
END=	I/O Option	Branch at end of file
ENDTRACE	Directive	Terminate SETTRACE listing
ENTER	Directive	Pass values from Calling to Called program and back

*Keywords or partial keywords (i.e., EXCEPT, AND, OR, THEN, ELSE, ON, TO) in this column may not be used as data names or variable names in program statements.

<u>TASK</u>	<u>TYPE</u>	<u>DESCRIPTION</u>
EPT	Function	Return exponent of expression
ERASE	Directive	Delete entry from Disk Directory
ERR	Variable	Return last occurring error
ERR=	I/O Option	Branch on error
ERROR	Directive	Display BASIC and file system error
ESCAPE	Directive	Interrupt program
EXECUTE	Directive	Generate or modify statements from within a program
EXIT	Directive	Return control to Calling program an error occurs
EXITTO	Directive	Transfer program control to specified statement
EXTRACT	Directive	Read data field from a file into variable field in a statement
EXTRACT RECORD	Directive	READ a full record from a file or device
FID	Function	Return file or device information
FILE	Directive	Define file type
FIND	Directive	Read data from a file into a variable
FIND RECORD	Directive	Read a full record from a file or device
FLOATING POINT	Directive	Initiate Floating Point Mode
FNx	Function	Define function
FOR/NEXT	Directive	Begin looping
FPT	Function	Return fractional part of expression
GAP	Function	Generate odd-parity, byte-for-byte
GOSUB	Directive	Transfer program control to internal subroutine
GOTO	Directive	Transfer program control to a statement
HSH	Function	HASH; perform data integrity check
HTA	Function	Convert hexadecimal to ASCII
IF/AND/OR/THEN/ ELSE	Directive	Conditionally execute a statement
IND	Function	Return index of next record
IND=	I/O Option	Specify index of record to be accessed
INDEXED	Directive	Define Indexed file
INPUT	Directive	Used for communication between operator and/or device and program
INPUT RECORD	Directive	Read a full record from a file
INT	Function	Return integer of expression
IOL=	I/O Option	Branch to IOLIST statement
IOLIST	Directive	Define list of variables
IOR	Function	Combine the bits of two strings
ISZ=	I/O Option	Access file as if indexed
KEY	Function	Return key of next record
KEY=	I/O Option	Specify key to be accessed

<u>TASK</u>	<u>TYPE</u>	<u>DESCRIPTION</u>
LEN	Function	Return length of string expression
LEN=	I/O Option	Specify length range of variable
LET	Directive	Assign value to a variable
LIB	Directive	Transfers control to a C program
LIST	Directive	Print statement(s)
LOAD	Directive	Bring a program into memory
LOCK	Directive	Protects a file from access by other users
LRC	Function	Check for data integrity
LST	Function	Convert BASIC statement to list format
LVL	Function	Return level of BASIC and file system
MERGE	Directive	Combine two programs
MOD	Function	Divide integers, return the remainder
NEXT	Directive	Used with FOR to create looping
NOT	Function	Return inverse of string
NUM	Function	Return numeric value of characters in a string
ON/GOTO	Directive	Conditionally transfers program control to a statement
ON/GOSUB	Directive	Conditionally transfers program control to a subroutine
OPEN	Directive	Access a file or reserve a device
OPTS=	I/O Option	Specify print job attributes
PFX	Variable	Return current prefix list
PGM	Function	Return compiled format for statement
PNM	Variable	Return name of program currently executing
POS	Function	Return character position
PRC	Variable	Returns current precision
PRECISION	Directive	Set number of places of rounding
PREFIX	Directive	Set the current prefix list
PRINT	Directive	Print to a file or device
PRINT RECORD	Directive	Write a full record to a file
PSZ	Variable	Return bytes used by a program, not including data
PUB	Function	Return list of programs in shared memory
READ	Directive	Read data into a variable
READ RECORD	Directive	Read a full record into a variable
RELEASE	Directive	Close files and release task
REM	Directive	Insert a comment
REMOVE	Directive	Delete the key of an existing record in a keyed file
RENAME	Directive	Rename a file or program
RESET	Directive	Reset system
RETRY	Directive	Transfer program control to the statement where the last error occurred
RETURN	Directive	Transfer program control to the statement following the GOSUB
RUN	Directive	Execute a program

<u>TASK</u>	<u>TYPE</u>	<u>DESCRIPTION</u>
SAVE	Directive	Copy program from user memory to program file on disk
SEQ=	I/O Option	Positions 1/2 inch tape on OPEN
SERIAL	Directive	Define a Serial file
SETCTL	Directive	Branch to specified statement number on CTL-Y
SETDAY	Directive	Set value of DAY variable
SETERR	Directive	Branch to error routine
SETESC	Directive	Branch when ESCAPE is pressed
SETTIME	Directive	Set value of TIM (time) variable
SETTRACE	Directive	List statements as they execute
SGN	Function	Return sign of numeric expression
SIZ=	I/O Option	Set maximum allowable characters for input
SORT	Directive	Define a Sort file
SSN	Variable	Return the system serial number
START	Directive	Reset system, start tasks
STOP	Directive	Terminate program before the physical end of the program
STR	Function	Convert numeric expression to string
STRING	Directive	Define a string file
SYS	Variable	Return operating system level
SYSTEM	Directive	Executes a Command Language command from BASIC
TABLE	Directive	Define values to translate characters to another code during I/O operation
TBL	Function	Translates a string using the specified TABLE statement
TBL=	I/O Option	Specify number of TABLE statement to be used
TCB	Variable	Return task information
TIM	Variable	Return current system time
TIM=	I/O Option	Specify seconds allowed for input
TSK	Function	Display configured devices
UNLOCK	Directive	Free a file locked from access
UNT	Variable	Return lowest unused unit number
VMERGE	Directive	Merge programs
WAIT	Directive	Specify length of pause in seconds
WRITE	Directive	Write to a file or device, then append a line feed (LF) to each variable in the list
WRITE RECORD	Directive	Write a full record to file or device
XOR	Function	Combine the bits of two strings

OVERVIEW

The BASIC program file format has been changed for level 7.2 Business BASIC. Appendix C describes the new format.

In order for programs written under 7.1 Business BASIC to be run under level 7.2, they must be converted to the new program format. A single command, "csave", is supplied to perform the file conversion.

During the system software upgrade procedure, the option is provided to convert BASIC program files. The command described here can be used to convert any files not converted during the upgrade procedure.

PROGRAM FILE
CONVERSION

The "csave" command is run under the control of the command interpreter.

FORMAT

```
@>csave {-h} {-b} {-l} {-n} {-r} {-s}
        {-f filelist} file1 {... fileN}
```

where:

- h Help. Displays help message.
- b Backup. Makes a backup copy of the old (7.1) file if possible.
- l Long. Displays the sizes (in both bytes and blocks) of the 7.1 file and the 7.2 file. The report also includes the number of errors encountered during the conversion.
- n No Save. Does not save the new (7.2) file. Used with the "-l" option, this option tells you how much larger your program will be, without putting the converted file on disk. This option invalidates the "-b" option.
- r Recursive. If a file in the list is a directory, all program files in the directory are converted.
- s Silent. No report is displayed.
- f Filelist. This option specifies a filelist name.

EXAMPLES

1. @>csave -l -b filename

This command converts the specified file producing the long format report and creating a backup copy of the 7.1 file.

2. @>csave -b -r /

This command converts every BASIC program file on the filesystem, creating a backup copy of each 7.1 file.

INDEX

10 - 10 Pitch, 8-5

16 - 16 Pitch, 8-5

8L - 8 Lines Per Inch, 8-5

@(x) - Horizontal Position, 8-3, 8-5

@(x,y)-Horiz & Vert Position, 8-3, A-14

! Directive, 4-1A

ABS (Absolute Value), 5-2

Absolute Value Function, 5-2

Access File as if Indexed Option, 7-8

ADDE, 4-2

ADDR, 4-2, 4-5, 4-19

add alloc (file growth allocation), 1-4

AND, 5-3

Argument List, 1-4

Arithmetic Expressions, 2-3

Arrays, 2-3, 4-13

ASC, 5-4

ASCII functions, 5-1

ASCII to Hex Function, 5-5

ATH, 5-5

Auto-end, 4-27

Available User Memory Variable, 6-4

Background, A-14

Background Mode Start Mnemonic, 8-9

Backspace Mnemonic, 8-3

BASIC Features

 Overview, A-1

B1 - Bin 1. 8-5

B2 - Bin 2. 8-5

BB - Begin Blink, 8-8

BE - Begin Echo, 8-11

BEGIN, 4-4

Begin Blink Mnemonic, 8-8

Begin Reverse Video Mnemonic, 8-8

Begin Underline Mnemonic, 8-8

BG - Begin Generating ERR 29, 8-11

BI - Begin Input Transparency, 8-11

BIN, 5-6

Binary to Decimal Function, 5-10

Binary Value Function, 5-6

BO - Begin Output Transparency, 8-11

Boolean algebra, 5-1

BR - Begin Reverse Video, 8-8

Braces, Use of, 1-3

Brackets, Use of, 1-3

Branch at End of File Option, 7-4

Branching

 At end of File (END=), 7-4

 Duplicate or Missing Key (DOM=), 7-3

Branching (Con't)

- On error occurrence (ERR=), 7-5
- SETCTL, 4-100
- SETERR, 4-102
- SETESC, 4-103
- Branching Statements, A-14
- BS - Backspace, 8-3
- BT - Begin Input Buffering, 8-11
- BU - Begin Underline, 8-8
- CALL, 4-5
- Caret (^), Use of, 2-2
- Carriage Return Mnemonic, 8-3, 8-5
- Catastrophic Errors
 - Error 100 Diagnostic, 9-2
- CE - Clear to End of Page, 8-8
- CF - Clear Foreground, 8-8
- CH - Cursor Home, 8-3
- Channels
 - devno, 1-4
 - diskno, 1-4
- CHR, 5-7
- CI - Clear Input Buffer, 8-11
- CL - Clear Line, 8-8
- CLASS=
 - /etc/class file, A-9
 - /etc/defaults, A-9
 - Class File, A-9
 - Class File Format, A-9
 - Options List, A-8
 - Priority, A-9
- Class File
 - Format, A-9
- CLEAR, 4-8
- Clear Foreground Mnemonic, 8-8
- Clear Line Mnemonic, 8-8
- Clear Screen Mnemonic, 8-8
- Clear to End of Page Mnemonic, 8-8
- CLOSE, 4-9
- CLSTBL, C-6
- Combine Strings Function, 5-3
- Command Execution, 2-13
- Command Line Options
 - c= (C compiler), 10-2
 - Command Field, 10-1
 - e (European format mask), 10-2
 - help, 10-1
 - k= (aux. lister), 10-2
 - nr (no return), 10-1
 - pgm (BASIC program), 10-2
 - q (quiet mode), 10-2

Command Line Options (Con't)

- s= (start size), 10-2
- tsk= (task ID), 10-2
- Compile Function, 5-8
- Compound Statements
 - Format
 - Console Mode, 3-1
 - Program Mode, 3-1
- Console Mode, 2-1
- Constants
 - Arithmetic Expressions
 - Priority, 2-3
 - Types, 2-3
 - Hexadecimal Strings, 2-4
 - Numbers, 2-2
 - String Constants, 2-4
- Contents Description, 1-2
- Control Bars, 1-6
- Control Key Use, A-16
- Control Variable, 6-2
- Control Variable Value Table, 6-2
- Conventions
 - Input Terminators, 1-6
 - Parameters, 1-4
 - Symbols, 1-3
- CPL, 5-8, C-2
- CR - Carriage Return, 8-3, 8-5
- CRC, 5-9
- CS - Clear Screen, 8-8
- CTL, 6-2
- Cursor Down Mnemonic, 8-3
- Cursor Home Mnemonic, 8-3
- Cursor Left Mnemonic, 8-3
- Cursor Position Read Mnemonic, 8-4
- Cursor Right Mnemonic, 8-4
- Cursor Up Mnemonic, 8-4
- Cyclic Redundancy Function, 5-9
- Date Variable, 6-3
- DAY, 4-101, 6-3
- Day Variable, 4-101
- DC - Delete Character, 8-8
- Debugging, 4-47, 4-59, 4-105
- DEC, 5-10
- DEF FNx\$ (Define Function String), 4-10
- DEF FNx (Define Function), 4-10
- Define Function, 5-14
- Delay Execution (See WAIT)
- DELETE, 4-12
- Delete Character Mnemonic, 8-8
- Devices
 - Access, 4-71
 - Maximum Open, 4-72

Devices, Logical, 1-4
devno (logical device number), 1-4
DIM, 2-4
DIM array, 4-13
DIM String, 4-15
DIRECT, 4-16, 4-17
Directives
 !, 4-1A
 Access Modes, 2-1
 ADDE, 4-2
 ADDR, 4-2
 BEGIN, 4-4
 CALL, 4-5
 CLEAR, 4-8
 CLOSE, 4-9
 DEF FNx, 4-10
 DEF FNx\$, 4-10
 DELETE, 4-12
 DIM array, 4-13
 DIM String, 4-15
 DIRECT, 4-16
 DROP, 4-19
 EDIT
 Line Editor, 4-20
 Screen Editor, 4-23
 ENCRYPT, 4-26
 END, 4-27
 ENDTRACE, 4-28
 ENTER, 4-29
 ERASE, 4-31
 ERROR, 4-32
 ESCAPE, 4-33
 EXECUTE, 2-13, 4-34
 EXIT, 4-35
 EXITTO, 4-36
 EXTRACT, 4-37
 EXTRACT RECORD, 4-38
 FILE, 4-39
 FIND, 4-40
 FIND RECORD, 4-41
 FLOATINGPOINT, 4-42
 FOR/NEXT, 4-43
 GOSUB, 3-3, 4-46
 GOTO, 4-47
 IF/THEN/ELSE, 4-48
 INDEXED, 4-51
 INPUT, 4-53
 INPUT RECORD, 4-58
 IOLIST, 4-59
 LET, 4-60
 LIB, 4-60A
 LIST, 4-61
 LOAD, 4-63
 LOCK, 4-64

Directives (Con't)

MERGE, 4-65
NEXT, 4-68
ON/GOSUB, 4-68A
ON/GOTO, 4-69
OPEN, 4-71
Overview, 4-1
PRECISION, 4-73
PREFIX, 4-75
PRINT, 4-76
PRINT RECORD, 4-77
PROGRAM, 4-78
READ, 4-80
READ RECORD, 4-86
RELEASE, 4-87
REM, 3-3, 4-88
REMOVE, 4-89
RENAME, 4-90
RESET, 4-91
RETRY, 3-3, 4-92
RETURN, 3-3, 4-93
RUN, 4-94
SAVE, 4-95
SERIAL, 4-98
SETCTL, 4-100
SETDAY, 4-101
SETERR, 4-102
SETESC, 4-103
SETTIME, 4-104
SETTRACE, 4-105
SORT, 4-106
START, 4-108
STOP, 4-110
STRING, 4-111
TABLE, 4-112
UNLOCK, 4-116
VMERGE, 4-117
WAIT, 4-119
WRITE, 4-120
WRITE RECORD, 4-121

Directory

Definition, 1-4

Directory Access Methods

PREFIX, 4-75
Prefix List, 6-6

Display Configured Devices Function, 5-36

DN - Cursor Down, 8-3

DOM=, 7-3

DROP, 4-19

DSZ, 6-4

Duplicate or Missing Key Option, 7-3

EB - End Blink, 8-8

Echo Begin Mnemonic, 8-11

Echo End Mnemonic, 8-11

EDIT

 Description, 4-20

 Line Editor, 4-20

 Screen Editor, 4-23

EE - End Echo, 8-11

EG - End Generating ERR 29, 8-12

EL - End Load, 8-5

ENCRYPT, 4-26

END, 4-27

END=, 7-4

End Blink Mnemonic, 8-8

End Load Mnemonic, 8-5

End Reverse Video Mnemonic, 8-8

ENDTRACE, 4-28

End Underline Mnemonic, 8-8

ENTER, 4-29

Entering BASIC, 2-13

EO - End Output Transparency, 8-12

EP - Expanded Print, 8-5

EPT, 5-11

ER - End Reverse Video, 8-8

ERASE, 4-31

ERR, 6-5

ERR=, 4-102, 7-5

ERROR, 4-32

Error 29 Generation Begin Mnemonic, 8-11

Error 29 Generation End Mnemonic, 8-12

Error Codes

 Catastrophic, 9-2

 Error Message, 9-1

 Non-Catastrophic, 9-1

 Numerical Listing of, 9-1

 Overview, 9-1

Error Exit Option, 7-5

Error Variable, 6-5

ES - Escape, 8-12

ESCAPE, 4-33

Escape Handling, 4-5

Escape Mnemonic, 8-12

ET - End Input Buffering, 8-12

EU - End Underline, 8-8

Exclusive Or Function, 5-38

EXECUTE, 2-13, 4-34, 4-46

EXIT, 4-35

EXITTO, 4-36, 4-45, 4-46

Expanded Print Mnemonic, 8-5
Exponent Function, 5-11
Exponentiation, 2-11
EXTRACT, 4-37
EXTRACT RECORD, 4-38

FF - Form Feed, 8-5
FID, 4-39, 5-12
FID Format Table, 5-13
Field Protection, A-14
File

Access

Key, 4-81
Sequential, 4-81

File Information Function, 5-12

Files

Access

By ISZ=+, 4-85
By Key, 4-17
Directory Access Methods, A-6
IND=, 5-19
Key, 4-83, 4-120, 7-9
Remove Key, 4-89
Sequential, 4-99, 5-19, 7-6, 7-8
Write Record, 4-121

Define

Direct, 4-16
Indexed, 4-51
Program, 4-78, 4-95
Serial, 4-98
Sort, 4-106
String, 4-111

Information About

Program File Name (PNM, 6-7
Program File Size (PSZ, 6-8

Information About (FID), 5-12

Sequential, 4-85

File System

Access

Sequential, 4-17

By key, 4-17

Channels, 1-5

Directory, 1-4

Directory Access, 1-4

Errors

Catastrophic, 9-2

Extract Record, 4-38

File Access

Closing, 4-9
Erasing Files, 4-31
Extract File, 4-37
Find, 4-40
Find Compared to Read, 4-40
Find Record, 4-41

File System, File Access (Con't)

- Lock Files, 4-64
- Maximum Open, 4-72
- Open Files, 4-71
- Prefix, 4-75
- READRECORD, 4-86
- Remove Key, 4-89
- Serial, 4-64
- Unlock Files, 4-116
- Write to Files, 4-120
- file ID, 1-5
- File Names, 1-5
- fileno, 1-5
- Files, 4-17
- File Types
 - Direct, 4-16
 - Indexed, 4-51
 - Program, 4-78
 - Sort, 4-106
 - String, 4-111
- Growth Allocation, 1-4, 1-5
- I/O Directives, 2-12
- Keys, 1-5
- Path Names
 - Full, 1-5
 - Partial, 1-5
- Prefix, 1-4
- Program Names, 1-5
- READ, 4-80
- Saving Programs, 4-95
- Serial, 4-98
- Working Directory, 1-4
- FIND, 4-40
- FIND RECORD, 4-41
- FLOATINGPOINT, 2-11, 4-42
- FNX, 5-14
- FOR/NEXT, 4-36
- FOR/NEXT (LOOP), 4-43
- Foreground, A-14
- Foreground Mode Start Mnemonic, 8-9
- Format Mask
 - Errors, 2-10
 - European, 10-2
 - Mask Options, 2-10
- Formatting Output, 2-7
- Form Feed Mnemonic, 8-5
- FPT, 5-15
- Fractional Numbers, 2-11
- Fractional Part Function, 5-15

Functions

- Absolute Value (ABS), 5-2
- ASCII to Hex (ATH), 5-5
- Binary to Decimal (DEC), 5-10
- Binary Value (BIN), 5-6
- Combine Strings (AND), 5-3
- Compile (CPL), 5-8
- Cyclic Redundancy (CRC), 5-9
- Define Function (FNX), 5-14
- Display Devices (TSK), 5-36
- Exclusive Or (XOR), 5-38
- Exponent (EPT), 5-11
- File Information (FID), 5-12
- Fractional Part (FPT), 5-15
- Hash (HSH), 5-17
- Hex to ASCII (HTA), 5-18
- Index (IND), 5-19
- Integer (INT), 5-20
- Inverse String (NOT), 5-29
- Length (LEN), 5-23
- Level (LVL), 5-26
- List (LST), 5-25,C-2
- Logical Or (IOR), 5-21
- Longitudinal Redundancy (LRC), 5-24
- Modulo (MOD), 5-27
- Next Record Key (KEY), 5-22
- Numeric to ASCII (CHR), 5-7
- Numeric Value (NUM), 5-30
- Odd Parity (GAP), 5-16
- Overview, 5-1
- Position (POS), 5-32
- Program (PGM), 5-31
- Public Programs (PUB), 5-33
- Sign (SGN), 5-34
- String (STR), 5-35
- String to Decimal (ASC), 5-4
- User Defined, 4-10, 5-14

GAP, 5-16

Ghost Tasks

- Communicating with Ghosts, A-2
- Release Task, A-1
- Restrictions on Ghosts, A-2
- Start Task, A-1

GOSUB, 3-3, 4-36, 4-46, 4-93

GOTO, 4-47

GRWTBL, C-6

Hash Function, 5-17

Hexadecimal Numbers, 2-4

Hexadecimal Strings, 2-4

Hexadecimal to ASCII Function, 5-18

Horizontal & Vertical Position Mnemonic, 8-3

Horizontal Position Mnemonic, 8-3, 8-5

HSH, 5-17

HTA, 5-18

- I/O Options
 - Record Index (IND=), 7-6
- I/O Directives
 - Allowable Devices, 2-12
 - PRINT, 4-76, 4-77
 - TABLE, 4-112
 - WRITE, 4-120
 - Write Record, 4-121
- I/O Options
 - Access File as if Indexed (ISZ=), 7-8
 - Branch at End of File (END=), 7-4
 - Duplicate or Missing Key (dom=), 7-3
 - Error Exit (ERR=), 7-5
 - Input Size (SIZ=), 7-12
 - IOLIST Statement Branch (IOL=), 7-7
 - Length of Variable (LEN=), 7-10
 - Overview, 7-1
 - Printer Attributes (CLASS=), 7-2
 - Print Job Options(OPTS=), 7-11
 - Record Key (KEY=), 7-9
 - Set Time Out (TIM=), 7-14
 - Translation Table (TBL=), 7-13
- IC - Insert Character, 8-9
- IF/THEN, 2-6
- IF/THEN/ELSE, 4-48
- IND, 4-17, 5-19
- IND=, 7-6
- INDEXED
 - Record Size, 4-52
- Index Function, 5-19
- INPUT
 - Description, 4-53
 - Display Inhibition, 4-54
 - Format, 4-53
 - Terminators, 4-53
 - Verification
 - Numeric, 4-55
 - String, 4-56
- Input/Output Options (See I/O Options)
- Input Buffer Clear Mnemonic, 8-11
- Input Buffering
 - Clear the Buffer, A-11
 - Effect of Escaping, A-11
 - Error Processing, A-12
 - Programming Pointers, A-12
 - Starting and Stopping, A-11
 - Table Processing, A-11
- Input Buffering Begin Mnemonic, 8-11
- Input Buffering End Mnemonic, 8-12
- INPUT RECORD
 - Compared to READ RECORD, 4-58
- Input Size Option, 7-12
- Input Terminators, 1-6
- Input Transparency Begin Mnemonic, 8-11

- Input Verification
 - Numeric, 4-55
 - String
 - Branching, 4-56
 - Length Test, 4-56
- Insert Character Mnemonic, 8-9
- INT, 5-20
- Integer Function, 5-20
- Introduction
 - Level 3/4 Compatibility, 1-1
 - Overview, 1-1
 - Scope, 1-1
- Inverse String Function, 5-29
- IOL=, 7-7
- IOLIST, 4-59
- IOLIST Statements Branch Option, 7-7
- IOR, 5-21
- ISZ=, 7-8

- KEY, 5-22
- KEY=, 7-9

- LD - Line Delete, 8-9
- LEN, 5-23
- LEN=, 7-10
- Length Function, 5-23
- Length of Variable Option, 7-10
- LET, 4-60
- Level Function, 5-26
- LF - Line Feed, 8-5, 8-9
- LI - Line Insert, 8-9
- Line Delete Mnemonic, 8-9
- Line Feed Mnemonic, 8-5, 8-9
- Line Insert Mnemonic, 8-9
- LIST, 4-61
- List Function, 5-25
- LOAD, 4-63
- LOCK, 4-64, 4-99
- Logical Expressions, 1-5
- Logical Operations
 - Relational Operators, 2-6
- Logical Or Function, 5-21
- Logical Units
 - devno, 1-4
 - diskno, 1-4
 - Lowest Unit Variable (UNT), 6-15
 - Maximum Open, 4-72
- Longitudinal Redundancy Check, 5-24

- Loops (See FOR/NEXT)
- Lowest Available Unit Variable, 6-15
- LRC, 5-24
- LST, 5-25,C-2
- LT - Left Cursor, 8-3
- LVL, 5-26

- Matrices, 4-13
- MERGE, 4-27, 4-65
- Mnemonics
 - Alphabetical Listing, 8-13
 - Cursor Control, 8-3
 - OS Control, 8-11
 - Overview, 8-1
 - Printer Control, 8-5
 - Supported Terminals Table, 8-15
 - VDT Controls, 8-8
- MOD, 5-27
- Modes
 - Console, 2-1
 - Program, 2-1
- Modulo Function, 5-27
- Motor Bars (See Control Bars)
- Multi-tasking, A-1

- New Line Mnemonic, 8-5
- NEXT, 4-68
- Next Record Key Function, 5-22
- NL - New Line, 8-5
- NOT, 5-29
- Null Output, A-16
- NUM, 5-30
- Numbers
 - Floating Point, 4-42
 - Precision, 4-73
 - Range, 2-2
- Numeric Editing
 - Formatted, 2-8
 - Non-formatted, 2-11
 - Numeric Editing Options, 2-9
- Numeric Editing Options, 2-9
- Numeric to ASCII Function, 5-7
- Numeric Value Function, 5-30

- Odd Parity Function, 5-16
- ON/GOTO, 4-69
- ON ERR/GOTO (See ON/GOTO)
- OP - Overprint, 8-5
- OPEN, 4-71
- OPENTBL, C-6
- Operating System Level Variable, 6-11
- OPTS=
 - Format, A-10
 - Option List, A-10
 - Procedure, A-10

- Output Format
 - Numeric Editing, 2-8
 - Positioning, 2-7
- Output Positioning, 2-7
- Output Transparency Begin Mnemonic, 8-11
- Output Transparency End Mnemonic, 8-12
- Overprint Mnemonic, 8-5

- Parameters, 3-2
- Parentheses, Use of, 1-3
- Passing Arguments, 4-5
- Path Names
 - Full, 4-75
 - Partial, 4-75
- PE - Protect End, 8-9, A-14
- PFX, 6-6
- PGM, 5-31
- Plot Mode Mnemonic, 8-6
- PM - Plot Mode, 8-6
- PNM, 6-7
- POS, 5-32
- Position Function, 5-32
- PRECISION, 4-73
- PREFIX, 4-75
- Prefix (Working Directory), 1-4
- Prefix List, 1-5, 4-75, 6-6
- Prefix List Variable, 6-6
- PRINT, 4-76
- Print Job Attributes Option, 7-2, 7-11
- Print Jobs, 7-2
- PRINT RECORD, 4-77
- PROGRAM, 4-78
- Program Files
 - Saving, 4-95
- Program Function, 5-31
- Program Header Format, C-1
- Program Mode, 2-1
- Program Name Variable, 6-7
- Programs
 - Run, 2-1
 - Start, 2-1
- Program Size Variable, 6-8
- Protect Mode, A-14
- Protect Mode End Mnemonic, 8-9
- Protect Mode Start Mnemonic, 8-9
- PS - Protect Start, 8-9
- PS - Start Protect, A-14
- PSZ, 6-8
- PUB, 5-33
- PUB (0) Format Table, 5-33
- Public Programming
 - ADD E, 4-2
 - ADD R, 4-2, A-4

Public Programming (Con't)

- Control Block Entries, A-4
- DROP, A-5
- Restrictions on, A-5
- Trace Flag, A-5
- Public Programs Function, 5-33
- Quotation Marks, Use of, 1-3
- RB - Ring Bell, 8-9
- RC - Read Cursor, 8-4
- READ
 - Description, 4-80
 - From Devices, 4-85
 - From Direct Files, 4-81
 - From Indexed Files, 4-85
 - From Serial Files, 4-85
 - From Sort Files, 4-83
 - From String Files, 4-85
- READ RECORD, 4-71, 4-86
- Record Index Option, 7-6
- Record Key Identifier Option, 7-9
- Records
 - Number of, 1-5
 - Size of, 1-5
- Relational Operators, 2-6
- RELEASE, 4-87
- REM, 3-3, 4-88
- REMOVE, 4-89
- RENAME, 4-90
- RESET, 4-91
- RETRY, 3-3, 4-92, 4-102
- RETURN, 3-3, 4-36, 4-46, 4-93
- Ring Bell Mnemonic, 8-9
- RT - Right Cursor, 8-4
- RUN, 4-94
- SAVE
 - Already Defined Program, 4-96
 - Compared to Level 8, 4-96
 - Undefined Program, 4-95
- SB - Start Background, 8-9, A-14
- Screen Narrow Mnemonic, 8-9
- Screen Wide Mnemonic, 8-9
- Sectors (secno), 1-5
- Semicolon, Use of, 3-1
- SERIAL, 4-98
- SETCTL, 4-93, 4-100

- SETDAY, 4-101
- SETERR, 4-70, 4-102
- SETESC, 4-5, 4-93, 4-103
- SETTIME, 4-104
- Set Time Out Option, 7-14
- SETTRACE, 4-105
- SF - Start Foreground, 8-9, A-14
- SGN, 5-34
- Shared Memory (See Public Programming)
- Sign Function, 5-34
- Sixteen Pitch Mnemonic, 8-5
- SIZ=, 7-12
- SL - Start Load, 8-6
- Slew (definition), 8-7
- Slew 2 - 8 Mnemonics, 8-6
- SN - Screen Narrow, 8-9
- SORT, 4-106
- SP - Superscript, 8-6
- Special Key Controls, A-16
- Specifying Print Job Attributes
 - CLASS=, A-8
 - etc/class file, A-8
 - OPTS=, A-10
- Spooling, 7-2, A-8
- SS - Subscript, 8-6
- SSN, 6-10
- START, 2-1, 4-108
- Start Load Mnemonic, 8-6
- Statement Format
 - Auto-sort, 3-1
 - Compound Statements, 3-1, 3-2
 - Delete, 3-1
 - Directives, 3-2
 - I/O Options, 3-2
 - Overview, 3-1
 - Parameters, 3-2
 - Statement Numbers, 3-1
- Statement Numbers
 - Execution, 3-1
 - Highest Available, 3-1
 - Order, 3-1
- Statements
 - Conventions, 1-3, 1-4
 - Parameters, 1-3, 1-4
- STOP
 - Compared to END, 4-110
 - Description, 4-110
 - Effects on MERGE, 4-110
- STR, 5-35
- STRING, 4-111
- String Function, 5-35
- String to Decimal Function, 5-4
- Subscripting String Variables, 2-4
- Subscript Mnemonic, 8-6

Superscript Mnemonic, 8-6
SW - Screen Wide, 8-9
Symbols, 1-3
SYS, 6-11
System Serial Number Variable, 6-10
System Variables
 Available User Memory (DSZ), 6-4
 Control (CTL), 6-2
 Date (DAY), 6-3
 Error (ERR), 6-5
 Lowest Unit (UNT), 6-15
 OS Level (SYS), 6-11
 Overview, 6-1
 Prefix List (PFX), 6-6
 Program Name (PNM), 6-7
 Program Size (PSZ), 6-8
 Selected Disk (SEL), 6-9
 Serial No. (SSN), 6-10
 Task Ctl Blk (TCB), 6-12
 Time of Day (TIM), 6-14

TABLE, 4-112
Task Control Block Format Table, 6-13
Task Control Block Variable, 6-12
TBL=, 7-13
TCB, 6-12
Ten Pitch Mnemonic, 8-5
TIM, 4-104, 6-14
TIM=, 7-14
Time of Day Variable, 6-14
Time Variable, 4-104
TR - Transmit Screen, 8-10
Translation table Option, 7-13
Transmit Screen Mnemonic, 8-10
TSK, 5-36
TSK (0) Format Table, 5-37

UNLOCK, 4-116
UNT, 6-15
UP - Up Cursor, 8-4

Variables

Conversion

 Numeric-to-String, 5-35
 String-to-Numeric, 5-30

DIM, 2-3, 2-4

IOLIST Option, 7-7

Modification during Call, 4-7

Modification Table, 4-7

Numeric, 1-5, 2-2

Overhead, 2-5

String

 Length Function, 5-23

 Length Option, 7-10

 Position function, 5-32

Variables (Con't)

- String Comparison, 2-5
- String Expressions, 2-5
- Subscripted Numeric, 2-3
- Subscripted String, 2-4

Variable Tables

- CPL/LST, C-2
- Relative Branch Address, C-7
- Variable Table Utilities
 - OPENTBL, C-6
 - CLSTBL, C-6
 - GRWTBL, C-6

Vertical Tab Mnemonic, 8-6

VMERGE, 4-117

VT - Vertical Tab, 8-6

WAIT, 4-119

Working Directory (See Prefix)

WRITE, 4-120

WRITE RECORD, 4-71, 4-121

XOR, 5-38

Business BASIC Level 7

Quick Reference Card

#6253B

November, 1986
006253-020

SYSTEM VARIABLES

FORMAT	OPERATION
CTL	Field terminator last used
DAY	Current system date
DSZ	Unused bytes in user task memory
ERR(errno1,...,errnon)	Error that occurred last
PFX	Current prefix list
PNM	Current program in memory
PRC	Current precision
PSZ	Bytes used by program
SSN	System Serial Number
SYS	Level of BASIC
TCB (numeric expr 0-12)	Task information
TIM	Current system time
UNT	Lowest unused logical unit number

INPUT / OUTPUT OPTIONS

FORMAT	OPERATION
CLASS = "class name"	Specify print job class name
DOM = stno	Duplicate or missing key
END = stno	Branch at end of file
ERR = stno	Branch on error
IND = numeric expr	Access record index
IOL = stno	Branch to IOLIST stno
ISZ = numeric expr	Access file as if indexed
KEY = string expr	Key of record to be accessed
LEN = minimum, maximum	Length range of variable
OPTS = "string expression"	Specify print job options
SEQ = numeric expr	Positions 1/2 tape at filemark
SIZ = numeric expr	Maximum characters to be input
TBL = stno	Branch to TABLE stno
TIM = numeric expr	Seconds allowed for input

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

MISCELLANEOUS

BASIC COMMAND LINE ARGUMENTS

@> basic {-help} {-nr} {pgm = } {s = } {-e} {-q} {-x}
{lib = } {trans = } {'command string'}

PRINT JOB OPTION STRING FORMAT (OPTS = "string expression")

This is the format required in the string expression used to specify print job options in the OPTS = I/O option:

OPTS = "alias = class = copies = -delete form = list = -off -notify
priority = -requeue -spool -vfu -wait -raw time = "

FUNCTIONS

FORMAT	OPERATION
ABS (integer expression)	Absolute value
AND (string expr, string expr)	Combine Strings
ASC (string expr {,ERR = stno})	String to ASCII decimal value
ATH (string expr {,ERR = stno})	ASCII chars to hex value
BIN (numeric expr, length)	Binary value
CHR (numeric expr {,ERR = stno})	Numeric expr to ASCII
CPL (string expression {,string expr, string variable} {,ERR = stno})	Compile a string
CRC (string expression {,2-byte string})	Checksum for string variable; data integrity check
DEC (string expr {, ERR = stno})	Binary to signed decimal
EPT (numeric expr)	Exponent of expr
FID (fileno {, ERR = stno})	File information
FNx{\$} (argument list)	Definable functions
FPT (numeric expr)	Fractional part of expr
GAP (string variable or literal)	Odd parity, byte-for-byte
HSH (string expr {,2-byte string})	HASH; data integrity check
HTA (string expr)	Hex value to ASCII chars
IND (fileno {,ERR = stno})	Index of record
INT (numeric expr)	Integer of expr
IOR (string expr, string expr)	Inclusive ORing
KEY (fileno {,ERR = stno} {,END = stno} {,IND = recno})	Key of next record
LEN (string expr)	Length of string expr
LRC (string variable or constant)	Longitudinal Redundancy check; data integrity check
LST (string expr {,string expr} {,ERR = stno})	Return program statement in listed format
LVL (numeric expression 0 - 15)	BASIC for file system level information
MOD (numeric expr a, numeric expr b)	Modulo function
NOT (string expr)	Inverse of string, bit-by-bit
NUM (string expr {,ERR = stno})	String expression to numeric value
PGM (stno)	Return compiled format; pgm name if stno = 0
POS (scan string relational operator target string {,step value})	Character position
PUB (numeric expression from 0 - 15)	Public program information
SGN (numeric expr)	Sign of expr
STR (numeric expr {,:mask})	Numeric expr to string
TBL (string expr, stno)	Translates string expression
TSK (numeric expression from 0 - 15)	Display configured devices
XOR (string expr, string expr)	Exclusive ORing

ERRORS

ERROR NUMBER CODE	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	SECTOR POINTER OUT OF RANGE
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 MEMORY IS FULL
17	INVALID PARAMETER/NON-CONFIGURED DISK
18	ILLEGAL CONTROL OPERATION
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	UNDEFINED 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
68	BAD SECOND ARGUMENT TO CPL OR LIST
69	MISSING VARIABLE ID
70	THIRD ARG IN CPL OR LST NOT ACTIVE
71	THIRD ARG IN CPL NOT LONG ENOUGH
95	LAN ERROR
98	SPOOLER ERROR
99	COMM ERROR
103	CATASTROPHIC READ FAILURE/FILE POINTERS DAMAGED
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
'8L'	8 lines per inch
'10'	10 pitch
'12'	12 pitch
'16'	16 pitch
'B1'	Bin 1
'B2'	Bin 2
'BB'	Begin blinking
'BE'	Begin echo
'BG'	Begin to generate ERROR 29
'BI'	Begin input transparency
'BO'	Begin output transparency
'BR'	Set reverse video mode
'BS'	Backspace
'BT'	Begin Input Buffering
'BU'	Begin underline mode
'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
'EB'	End blinking
'EE'	End echo
'EG'	End generation of ERROR 29
'EI'	End input transparency
'EL'	End Load (VFU)
'EO'	End output transparency
'EP'	Expanded print
'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
'NL'	New line
'OP'	Overprint
'PE'	End protect mode
'PG'	Print screen
'PM'	Begin plot mode
'PS'	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
'SB'	Start background mode
'SF'	Start foreground mode
'SL'	Start load (VFU)
'SN'	Screen narrow
'SP'	Superscript
'SS'	Subscript
'SW'	Screen wide
'TL'	Transmit line
'TP'	Transmit line protected
'TR'	Transmit terminal screen
'TS'	Transmit screen protected
'UP'	Cursor up
'VT'	Vertical tab

Directives

```
! command string

ADDE "prog ID" {,ERR = stno}
ADDR "prog ID" {,ERR = stno}

BEGIN { {EXCEPT} var1 ... varn }

CALL "prog ID" {,ERR = stno} {,argument list}
CLEAR { {EXCEPT} var1 ... varn }
CLOSE (fileno/devno {,ERR = stno} {,IND = index value})

DEF FNx{$} (argument list)= str expr
DELETE {first stno} {,} {last stno}
DIM array name (length of first dimension {,length of second {,length of third})
DIM string variable name (length {,str expr})
DIRECT "file name", keysz, recho, recsz {, diskno, secno, init_alloc, add_alloc} {,ERR = stno}
DROP "prog ID" {,ERR = stno}

EDIT stno {{C[copy through value]} {D[delete through value]} {R[replace value]} {[insert value]}}
EDIT stno {-HELP}
ENCRYPT "Old prog ID", "New prog ID"
END
ENDTRACE
ENTER argument list
ERASE "file ID" {,ERR = stno}
ERROR
ESCAPE
EXECUTE string argument
EXIT {numeric variable}
EXITTO stno
EXTRACT (fileno {,ERR = stno} {,END = stno} {,DOM = stno} {,IND = index value} {,KEY = key value}
    {,TBL = stno} {,SIZ = size} {,TIM = interval}) {argument list} {,IOL = stno}
EXTRACT RECORD (fileno {,ERR = stno} {,END = stno} {,DOM = stno} {,IND = index value} {,KEY = key value} {,TBL = stno}
    {,SIZ = size} {,TIM = interval}) {string variable}

FILE string
FIND (fileno {,ERR = stno} {,END = stno} {,DOM = stno} {,KEY = key value} {,TBL = stno} {,SIZ = size}) {variable} {,IOL = stno}
FIND RECORD (fileno {,ERR = stno} {,END = stno} {,DOM = stno} {,KEY = key value} {,TBL = stno} {,SIZ = size}) {variable}
FLOATINGPOINT
FOR ctrl variable = start expression TO end expression {STEP expr}

GOSUB stno
GOTO stno

IF logical expr {THEN statement} {ELSE statement} {FI}

INDEXED "file name", recho, recsz {, diskno, secno, init_alloc, add_alloc} {,ERR = stno}
INPUT {(fileno/devno {,ERR = stno} {,END = stno} {,DOM = stno} {,IND = index value} {,KEY = key value} {,TBL = stno}
    {,TIM = interval} {,SIZ = size})} {,@(expr{,expr})} {string constant} {,mnemonic} {,variable} {,IOL = stno}
INPUT RECORD (fileno/devno {,ERR = stno} {,END = stno} {,DOM = stno} {,IND = index value} {,KEY = key value} {,SIZ = size}) {string variable}
IOLIST argument list {,IOL = }

LIB "string expression" {,ERR = stno} {,arg list} {,RET = var/array elem}
{LET} {numeric variable = numeric expr} {,} {string variable = str expr} {,...}
LIST {(devno {,ERR = stno} {,TBL = stno})} {first stno} {,} {last stno}
LOAD "prog name"
LOCK (fileno {,ERR = stno})

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

NEXT control variable

ON expr GOSUB stno {, stno, stno...}
ON expr GOTO stno {,stno, stno...}
OPEN {INPUT} (fileno/devno {,ERR = stno} {,ISZ = recsz} {,CLASS = "printer class"} {,OPTS = "str expr"} {,SEQ = seq no}) "file/device ID"

PRECISION expr
PREFIX "directory/path1 ... directory/pathn"
PRINT {(fileno/devno {,ERR = stno} {,END = stno} {,IND = index value} {,KEY = key value} {,DOM = stno} {,TBL = stno})} {@(expr {,expr})}
    {list} {,IOL = stno} {,}
PRINT RECORD (fileno/devno {,ERR = stno} {,SIZ = size limit}) {,END = stno} {,DOM = stno} {,IND = index value} {,KEY = key value}
    {,TBL = stno}) {string variable}
PROGRAM "file ID", prog size {, diskno, secno, init_alloc, add_alloc} {,ERR = stno}

READ {(fileno/devno {,ERR = stno} {,END = stno} {,IND = index value} {,KEY = key value} {,TBL = stno} {,DOM = stno} {,SIZ = size limit}
    {,TIM = interval})} {@(expr {,expr})} {variable list} {,IOL = stno}
READ RECORD (fileno/devno {,ERR = stno} {,END = stno} {,IND = index value} {,KEY = key value} {,TBL = stno} {,DOM = stno}
    {,TIM = } {,SIZ = } {string variable}
RELEASE {"ghost task ID"}
REM {""} {remark} {""}
REMOVE (fileno, KEY = key value {,DOM = stno} {,ERR = stno})
RENAME "old file name", "new file name"
RESET
RETRY
RETURN
RUN {"prog name"}

SAVE {"file name"} {, prog size, diskno, secno, init_alloc, add_alloc}
SERIAL "file name", average recho, average recsz {, discno, secno, init_alloc, add_alloc} {,ERR = stno}
SETCTL stno
SETDAY "string expr"
SETERR stno
SETESC stno
SETTIME numeric expr
SETTRACE {(fileno/devno)}
SORT "file name", keysz, recho {, diskno, secno, init_alloc, add_alloc} {,ERR = stno}
START {pages} {,ERR = stno} {"prog ID"} {"task ID"} {"device"}
STOP
STRING "file ID" {, diskno} {, ERR = stno}
SYSTEM "command string"

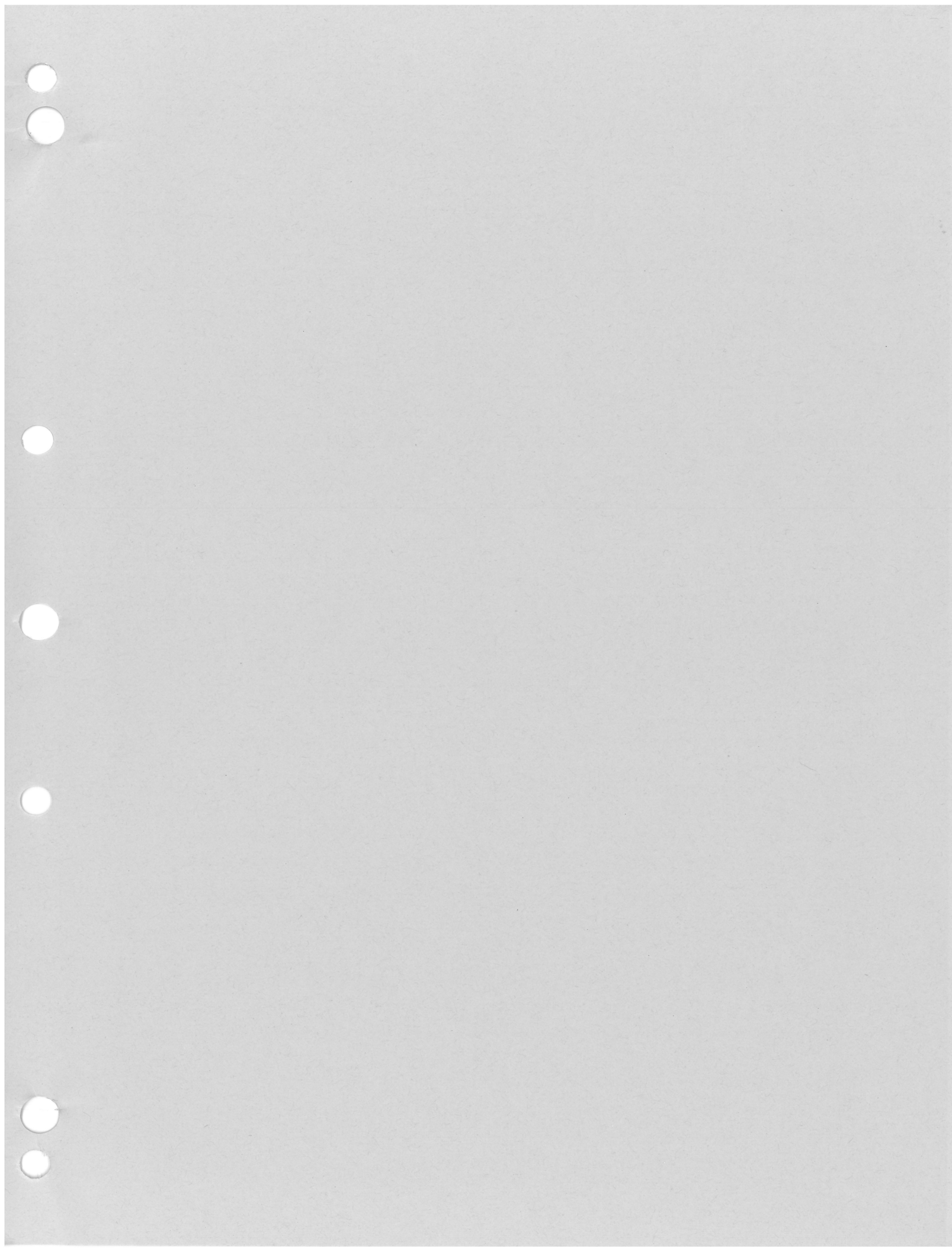
TABLE hexadecimal string

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

VMERGE "file ID"

WAIT number of seconds

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



MA/BasicFour[®]
Computers That Mean Business
