CET BASIC
Language Reference Manual

CET BASIC Language Reference Manual

Published and printed in the United States of America.

First Edition	1986
Second Edition	1987
Third Edition	1988
Fourth Edition	1994
Revision A	February, 1995
Revision B	March, 1996
Fifth Edition	July, 1996

TABLE OF CONTENTS

CHAPTER 1

Features of the CET BASIC Language

Introduction

CET BASIC is a modern, compiled language designed for developing professional, business applications for use on single-user, multi-user and network operating systems. The BASIC system can detect when a network or multi-user system is in use so that record and file locking takes place transparently to your program.

CET BASIC is a full-featured implementation of the BASIC language that far exceeds the current ANSI BASIC standards. More than a hundred built-in functions are provided for string handling, data conversion, screen control, output formatting and mathematics.

The system not only supports the IF-THEN construct found in the original Dartmouth and minimal ANSI BASIC, but it also supports the IF-THEN-ELSE construct and the multiple line IF, THEN and ELSE statements. WHILE-WEND and SELECT-CASE-CEND statements are also available for conditional processing.

The string manipulation functions illustrate the breadth of its capabilities: LEFT, RIGHT, and MID functions are provided along with the ability to perform substring operations by column position and extract, insert, replace, and delete the specified characters. The ability to overlay a string is also available.

The following sections describe some of the more powerful and unique features found in CET BASIC. Each of these features will be covered in detail later in this manual.

Data Files

CET BASIC supports three types of files: sequential, direct and indexed. These files may contain ASCII or binary data, depending upon how the individual records were created.

Records created with the PRINT statement are in an ASCII format and may be read with the INPUT or LINPUT statements. Records created with the WRITE statement are in a binary format and may be accessed with READ, READPREV or READNEXT statements.

Sequential Files

Sequential files consist of records which vary in length depending on the amount of data that has been entered. Records in sequential files are accessed in order, starting from the beginning of the file. That means that before a specific record can be read, all preceding records must be read first.

Direct Files

Direct files contain records of a fixed length which is determined when the file is created. The size of the file is dynamic and grows when necessary.

This type of file allows direct, random access to a record very quickly by using the record number which is relative to its position in the file.

Indexed Files

Indexed files are similar to direct files in that they contain fixed length records and must be created with a specific record length before they can be accessed. The file size is also dynamic.

Records in an indexed file may be accessed randomly by specifying the alphanumeric *key* or *index* value associated with the record or sequentially in sorted ASCII order by key.

CET BASIC data files are covered in more detail in a separate chapter.

Terminal Independence

In an operating system such as UNIX, where many types of terminals can be attached to the same computer, it is important that programs are able to function independently of the characteristics of the terminal in use. CET BASIC uses the UNIX /**etc/termcap** file to get information about many types of terminals so that the various functions will operate as expected.

For information on supporting and accessing a terminal's video and cursor attributes, see the *CET BASIC UNIX User's Guide.*

Chaining and Linking Program Modules

CET BASIC is unique in the DOS and UNIX environments in that it produces executables that may be chained or linked to other executables. This powerful feature allows you to move between individual program modules, maintaining open files and passing common variables as needed. Any memory (data space) that it not shared between the two modules will be deallocated so that it may be used by the current program.

Since the BASIC system treats chained and linked modules as one big program, CET BASIC allows you to write structured programs that far exceed the size of *normal* programs.

Subroutines in Languages other than CET BASIC

CET BASIC provides easy access to externally compiled C, assembler and BASIC (in the UNIX and Windows product) language subroutines. By using a CALL statement, arguments may also be passed to and from the external subroutine.

Many C language functions are included with the CET BASIC product to provide you with a wide variety of features. These functions are covered in the *CET BASIC User's Guide* for the product you are using. If you wish to write your own functions, refer to the *C Language Interface Guide* for specific information.

Commercial Development Libraries

CET BASIC is enhanced in various application areas by a number of libraries that can be linked to CET BASIC programs to perform operations which would be difficult or inefficient to code using CET BASIC alone.

The CET-POS Window System is an optional package that provides basic and multi-window support with overlapping displays and complete control over colors, borders and display areas. Advanced functions include pop-up choice list windows with built-in scrolling, searching and selection marking. This feature is described in a separate manual.

Additional libraries are provided and documented in the *CET BASIC Library Manual:*

Bar Code Library	Provides support for generating printed bar codes in over 12 standard formats.
DOS Communications Library	Provides device-level control over serial port ports for high-speed, reliable data transfer.
DOS Video Library	Provides direct control over the PC video system.
File Access Library	Provides access to CET BASIC data files from a C language subroutine.

Copy Protection

CET BASIC makes it possible for you to serialize your application programs so that they will run with only one CET BASIC Runtime System. This feature along with the CET KeyPlug provides you with the highest degree of copy protection possible.

For more information about copy protection, refer to the description of SERIAL under the OPTION statement in Chapter 5.

International Support

CET BASIC is used by application developers throughout the world since it provides programmers with:

- International date, time, and numeric format support.
- International character support.
- Easy-to-translate, ASCII formatted error message files.

Other Features

CET BASIC provides many other features that are not normally found in the other BASIC languages available for the micro-computer. Some of these features are:

- Multiple statements on one line.
- Line length of up to 800 characters.
- Long variable names of up to 64 characters.
- Multiple-line, user-defined functions (DEF FN-FNEND).
- Line labels with up to 64 characters.
- Error trapping with the ON ERROR GOTO statement.

- Special key trapping with the ON KEY statement.
- Record and/or file locking.
- Record lock detection.
- Special locked record lock handling with the ON LOCK statement.
- Complex IF-THEN-ELSE statements.
- Multiple-line IF-IFEND structure.
- Multiple-line WHILE-WEND structure.
- Multiple-line SELECT-CASE-OTHERWISE-CEND structure.
- String handling with strings of up to 32,767 characters in length.
- One and two-dimensional string arrays.
- Matrix (array) input/output and assignment.
- Formatted output with the PRINT USING statement.
- Mechanism for external mapping of printer escape character sequences.
- Input editing with LINPUT USING.
- Interface to externally compiled subroutines with the CALL statement.
- Execute operating system commands from within BASIC with the CSI statement or CSH function.
- Bit-manipulating, logical functions.
- 13-digit precision BCD (binary coded decimal) arithmetic.
- Floating point values in the range 10^{126} to 10^{-126}.
- IEEE format long and short precision binary floating point arithmetic.
- Integer arithmetic with values between –32,767 and +32,767.
- Statement numbers not required in program source files.
- Error messages which can be easily modified or translated to another language.
- Extensive set of string-handling functions.
- Extensive set of numeric and trigonometric functions.
- True native compiler.
- A standard interface for console cursor control.
- Spooled printing support.

- Full control of screen colors with the COLOR statement or the CRT$ function.
- Executable dimension (DIM) statements.
- Executable COMMON statements.

CHAPTER 2

CET BASIC Programs

Structuring a BASIC Program

A CET BASIC program consists of a set of statements constructed with the language elements and syntax described in the following chapters. The source code for the program must be in an ASCII file created by an text editor in a format recognized by the operating system in use.

Refer to your *CET BASIC User's Guide* for information on compiling and running BASIC programs in a specific environment.

Character Set

CET BASIC uses the full ASCII character set which includes the following. (Refer to the *Appendix* for a complete listing of the ASCII code values.)

- Letters A through Z and a through z.
- Digits 0 through 9.
- Special characters.

CET BASIC will automatically convert all lower-case letters to upper case except those occurring in string constants, the names of external programs specified with the CSI/CSH and CALL statements and in comments. A BASIC program does not distinguish between the variable balancedue, BALANCEDUE, BalanceDue and balanceDUE.

Line Format

The general format of a program line is as follows:

		——Statement——	
Statement Number	**Label**	**Verb**	**Operand**
1010	LABEL:	PRINT	SQR(X^+Y^2)

A line in a BASIC program may begin with or without a line or statement number. Not only is this optional, but some program lines may be numbered while others are not. The CET Brenum utility is available anytime you wish to renumber the lines in a program.

If numbers are used, the number must be a positive integer. When consecutive lines are numbered, the numbers must also be in ascending order. Any leading zeros or leading and trailing spaces will be ignored.

For readability, several lines may be considered a single, logical line by terminating each (physical) line with a tilde character (~):

```
100  PRINT A * (B + ~
          C – D~
          + 10)
```

This would be equivalent to entering either of the following lines.

```
100  PRINT A * (B + C – D + 10)
100  PRINT A*(B+C–D+10)
```

Line Labels

Lines may have a symbolic label in addition to or instead of a number. BASIC statements such as GOTO and GOSUB reference other lines of code by specifying either a label or statement number. In those cases, labels are generally recommended over statement numbers as they make programs more structured and therefore, more readable.

A line label may be a string of up to 64 characters consisting of one or more letters, digits or periods, but the first character must be a letter. When a line label is defined, it must precede any statement on the line and be separated from the first statement by a colon character (:).

The following lines show valid uses of line labels:

```
900       MAINLINE: WHILE CONTROL = 0
          GOSUB INPUT.ROUTINE
          IF INPUT$ = "" GOTO ERRORS
```

```
2000      INPUT.ROUTINE: REM subroutine to accept input...
ERRORS:
REM subroutine to process a null entry...
```

A character string can only be used as a line label once, but it can also be the name of a variable or keyword. Although this usage is valid, we suggest that you avoid this practice for the sake of clarity.

Statements

Each BASIC statement has a specific syntax. Detailed information on the statements and the elements that may be used is provided in separate chapters in this manual.

For now, note that each part of the statement must be separated by a space or TAB character. In the following example, PRINT is a keyword and CUR.DATE$ is a variable.

Acceptable: PRINT CUR.DATE$
Unacceptable: PRINTCUR.DATE$

Extra spaces or tabs may be used to format the statement and improve readability. Since the compiler ignores extra embedded spaces, they will not affect the execution of the program.

Each BASIC statement must start with a keyword which uniquely identifies the statement or operation to be performed. The one exception to this rule is the LET statement where the use of the keyword is optional.

Keywords are reserved and can not be used as variable names. For instance, the following is unacceptable as an assignment statement because DATA is a keyword.

```
LET DATA = 14
```

Although keywords may be used as line labels, this practice is not advised as it makes a program more difficult to read.

Multi-Statement Lines

More than one statement may be entered on a line if the statements are separated with a backslash. For example, the following line contains three complete PRINT statements:

```
1000 PRINT A$;B; \ PRINT CUR.DATE$ \ PRINT "Total =";TOTAL
```

The statement number and line label (if any) always refer to the first statement in a line. Therefore, in the previous example, you could not

perform a “GOTO 1000” and just execute the statement “PRINT CUR.DATE$" without executing PRINT A$;B; and PRINT "Total =";TOTAL at the same time.

Any valid statement can appear in a multi-statement line as long as:

- Only the first statement has a statement number and/or line label.
- Successive statements are separated with a backslash.

Including External Source Files

Sections of commonly used BASIC code such as declarations, user-defined functions and error processing routines (plus modifications unique to a particular program) may be stored in separate files and included when the program is compiled.

This feature makes programs easier to maintain because you can make a change in one external subroutine and then recompile every program that uses the subroutine. This feature also eliminates having to reenter the same code in many different programs, thus saving time and disk space.

Typically, the external files consist of unnumbered lines of BASIC code so you do not have to worry about whether or not the line numbers in the compiled program will be in numerical order. To include code that is stored in a separate file, make sure that the first character on the line (other than the statement number) is a pound sign (#). The format of an include line is:

```
#INCLUDE filename
```

The keyword INCLUDE may be entered in any case mode, but the *filename* must be in a format recognized by the operating system. Surrounding the *filename* in double quotes (") is optional, but recommended. For example, both of the following are valid:

```
#include include\poscode\helpwin
#INCLUDE  "../mysub/errcode"
```

The current directory is the default directory for #include files. The CET BASIC compiler will first search there for the specified *filename,* and then in the path specified by the include flag (-I).

Included files may also contain #include lines to reference other code.

Documentation Techniques

REMARK or REM statements may be used to document a BASIC program. The system will treat a statement that begins with 'REM' or 'rem' as a comment and ignore everything else in the line. Therefore, a REM statement must always be the last statement on the line. For example:

```
LET A=B \ REM Variable A receives current value of B
```

When adding remarks, note that:

1. The semicolon character (;) may be used as a substitute for the REM keyword. Although this is a handy feature, it does make a program less readable when you are scanning the listing.
2. The statement number or line label associated with a REM statement may be referenced by another (GOSUB) statement.
3. After a program is compiled, a remark has no impact on program execution.

CHAPTER 3

Elements of the BASIC Language

Introduction

This chapter describes the common elements that are used in CET BASIC programs. These elements include:

Constants	Numeric and alphanumeric constants that never change.
Variables	Numeric and alphanumeric fields whose values can change during the execution of the program.
Functions	A predefined or user-defined procedure whose value is determined when the program is executed.
Operators	Symbols that indicate the operation to be performed.
Expressions	A combination of one or more constants, variables, functions and/or operators.

Constants

A constant is an element whose value is specified in the source code. These values remain constant and cannot be changed during program execution.

There are two types of constants:

- Numeric constants such as 5.302 and –401.
- String constants such as “New York”.

Numeric Constants

A numeric constant consists of one or more digits, with or without a sign, a decimal point, or an exponential. The following are all valid constants:

253456 3.14159 –1234.01

Numeric constants cannot contain embedded spaces or special characters. All of the following will be considered invalid:

123 456	Contains a space
$1.45	Contains the '$' symbol
1.355CM	Contains units
1,234,555	Contains commas

BASIC accepts and maintains numeric constants with up to 13 significant decimal digits. When more than 13 digits are specified, the excess or least significant digits to the right will be truncated.

Whole numbers (without a decimal point) between −32768 and +32767 are considered *integer* constants and are given a shorter and more efficient internal representation. All other constants are called *floating point* or *real* constants.

Exponential Notation

Numbers with more than 13 digits may be written by using scientific or exponential notation to express the number as a constant multiplied by a power of ten. Exponential numbers contain an ‘E’ as in the following:

```
1234E5      9.8765432E−4
```

The constant “1234E5” represents 123400000 while “9.8765432E-4” would be .00098765432. When the exponent is positive, the decimal point is shifted to the right by the specified number of places, and when it is negative, to the left by that many places.

The allowable range for an exponential number depends on the internal representation used, but is generally between +126 and −126. (See the *Binary Real Numbers* section.)

Hexadecimal Integer Format

Integer constants may be entered in decimal format (base 10) or in hexadecimal format (base 16). A hexadecimal value must begin with a digit, consist of no more than four significant digits and be terminated with the letter H. Valid hexadecimal digits are 0–9 and the letters A–F which are used to represent the digits 10–15.

The following are valid hexadecimal constants:

```
1234H   0ABH    245H    0FFFFH  −1234H
```

The following will be considered invalid:

12AB	Missing "H"
0FFGH	G is not a valid hexadecimal character
1.24H	Not an integer

String Constants

A string constant is also referred to as a string literal. The constant may consist of nothing (a null) or a group of alphanumeric and/or special characters enclosed in a pair of single or double quotation marks. Whichever mark is used, both delimiters must be of the same type.

String constants may contain any printable or non-printable ASCII character except for a line terminator. CET BASIC will maintain every character between the delimiters exactly as it was entered into the source program without performing its usual lower-to-upper case conversion.

Strings are printed without the delimiting quotation marks. To include quotation marks in the string, either enter the opposite type that was used to delimit the string (i.e. single within double, double within single) or double the embedded quotation mark. For example:

String Constant	**Internal and Printed Representation**
"This is a string constant"	This is a string constant
'This is also a string constant'	This is also a string constant
"Look at Spot's spots."	Look at Spot's spots.
'Look at Spot"s spots.'	Look at Spot's spots.
"He said, ""Open the book.""""	He said, "Open the book."

Data Types

Numeric and string constants represent specific types of data that are stored in a data space in memory. CET BASIC recognizes the following five data formats:

Numeric Data Types

Numerics may be stored as:

- Decimal Real numbers
- Integers
- Short Binary Real numbers
- Long Binary Real numbers

Decimal real numbers, also referred to as *floating point* numbers, may be represented as 13 decimal digits, a sign and a signed exponential. They

occupy 8 bytes (characters) of memory. Integers are represented as two bytes of signed binary information.

The last two data types (binary real numbers) are seldom used in a business application. If you are developing scientific applications under UNIX, refer to a section at the end of this chapter for information on these data types.

String Data Type

Character string variables vary in length. The first two bytes are used to store the length of the string. The subsequent bytes are the characters of the string. Only the memory (data space) required to store the string is used.

Variables

Variables are symbolic names that are assigned to the specific locations in memory which contain one of the five data types described in the previous section.

A variable is different from a constant because its value may be changed during the execution of the program. Whenever a variable is referenced in the program, it is replaced by the last value it was given.

The name of a variable may be up to 64 characters long, not including an ending % or $ symbol. The rules for assigning variable names are:

- First character must be a letter (A to Z).
- Subsequent characters are optional and may consist of letters (A to Z), digits (0 to 9) or periods.
- A variable name may not include a space character.
- A reserved word may not be used unless it ends with a % or $ symbol. (This form is not recommended as it tends to make a program more difficult to read.)
- The first two characters cannot be "FN" or "fn" since these are reserved for the names of user-defined functions.

The variable name determines the type of data that may be stored in the variable. The possibilities are:

Decimal Real Numbers	no special terminating character
Integer Numbers	% terminating character
Character Strings	$ terminating character

Short Binary Real Numbers	! terminating character
Long Binary Real Numbers	!! terminating characters

The type-specifying character(s) at the end of a variable name are part of the name and make the variable unique. For example, the following names all refer to different variables:

CUSTOMER	Decimal real variable
CUSTOMER%	Integer variable
CUSTOMER$	String variable
CUSTOMER!	Short binary real variable
CUSTOMER!!	Long binary real variable

Note that variable names that do not end with a special character are also referred to as *numeric variables*. It is assumed that a decimal real number is to be stored in these variables, but a short or long binary real number may also be stored using the OPTION DEFAULT statement. (See the *Binary Real Numbers* section at the end of this chapter and the OPTION statement in Chapter 5.)

All lower case letters within a variable name are converted internally to upper case. The following are all considered valid names:

```
TOTAL   Sub.Total%   CUST.NO$   SUM!!   Desc1$   AMT!
```

The following names are not valid:

123A	Must start with a letter
A$ONE	A period is the only special character allowed
PRINT	Reserved word
TOOMUCH!!!	Too many '!' characters at the end of the name

Assigning Values to a Variable

When a program is executed, BASIC automatically initializes all numeric and integer variables to zero and all string variables to null. Changing the value of a variable is done through *assignment* or by an *input operation*. For example, variables may be entered by the user with the INPUT statement:

```
INPUT CUST.NO$
```

Variables may also be assigned a value with the LET statement. Note that the LET keyword is implied and may be omitted in the following examples.

```
LET BAL = INV.AMT - PAID
```

All the variables in the example above are of the same data type. The following statement would be invalid.

```
LET BAL = CURR$
```

The VAL function could have been used in the previous example to convert the value of the string variable to a numeric so that it can be assigned to BAL:

```
LET BAL = VAL(CURR$)
```

During assignment, all variables are treated like constants and the same rules apply. For example, the following statements illustrate what happens when an attempt is made to assign a integer value that is beyond the acceptable range (-32768 and +32767). The program will detect an arithmetic overflow condition (error #3) and abort when a value greater than or equal to 64K is encountered.

	Result stored in BAL%
LET BAL% = “32678”	32678
LET BAL% = “65533”	-3
LET BAL% = “65534”	-2
LET BAL% = “65535”	-1
LET BAL% = “65536”	Error #3 is detected.

It is important to note this behavior when porting applications created under THEOS. In that environment, all numbers greater than 32767 are displayed as 32767 and stored as –32768.

Regardless of which platform you are using, we strongly recommend that you check for valid numbers and correct any problems programmatically. This way you can avoid getting a fatal error and make sure that the results stored in your database are correct.

Array Variables

Simple variables allow one data value or field to be stored per variable. CET BASIC provides a special type of variable referred to as an array so that large amounts of data may be manipulated without having an equivalent (large) number of variables.

Instead of reserving one location to store data, an array variable reserves a set of memory locations to store a table of numeric or string items.

Array names have the same format as the variable names described in the last section. The name specifies which of the five data types can be stored in the array. For example, PRICE can name an array of decimal real numbers, QUANTITY% an array of integers and DESCRIPTION$ an array of strings.

An array may be thought of as a grid of rows and columns with one data element per cell. If the array is one-dimensional, then the grid will have only one row. Two-dimensional arrays have two or more rows.

To refer to a particular item in an array, specify the array name followed by the cell number(s) or subscript(s) enclosed in parentheses. For example:

PRICE(I)	A one-dimensional array
QUANTITY%(3, J)	A two-dimensional array

The valid range of subscripts is determined by the number of dimensions in the array. Subscripts may be arbitrary expressions and are placed in parentheses after the array name. If there are two subscripts (for two-dimensional arrays), they must be separated by a comma.

The following grid shows a two-dimensional array of 3 rows and 3 columns. The coordinates to reference the data in each location are noted.

	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

All elements in an array have the same minimum value. By default, this is zero which means that the first row and column is numbered zero. Even though there are 3 columns and 3 rows, the largest subscript in this array is SAMPLE$(2,2). Using a subscript beyond the upper and lower limits of the array generates an out of range error which is trappable at runtime. (Refer to the ON ERROR statement in Chapter 5.)

Since it may be awkward to use arrays numbered in BASE 0, CET BASIC provides an OPTION BASE 1 statement which may be used so that arrays will start in row 1 and column 1. The following grid shows the previous array using BASE 1.

	1	2	3
1	(1,1)	(1,2)	(1,3)
2	(2,1)	(2,2)	(2,3)
3	(3,1)	(3,2)	(3,3)

Creating Array Variables

Arrays may be dynamically created with a DIM statement that assigns a name and size to the array to avoid wasting data storage space as in:

```
DIM TEXT$(5,8)
```

The COMMON statement may also be used to dimension an array whose values need to be passed to other program modules.

Both DIM and COMMON are executable statements that allow you to specify the upper limits of the array.

```
DIM A$(20), B%(3,5)
```

In the above example, A$ is a one-dimensional array of strings whose upper limit is 20. A$ will consist of table entries A$(0), A$(1), ..., A$(20) when the default BASE 0 is used. B% is a two-dimensional array whose maximum first subscript is 3 and second subscript is 5. The entries for the B% array would be B%(0,0), B%(0,1), ...B%(0,5), B%(1,0), ... B%(1,5), ..., B%(3,5).

In this example, the program could also refer to simple (un-subscripted) variables A$ and B%, which would have values entirely independent of the arrays A$ and B%. (Although this feature is allowed for compatibility purposes, we do not recommend that you use it as it makes a program much more difficult to read.)

If an array is used before it is defined with a DIM or COMMON statement, the default dimensions are assigned when the statement is executed. One-dimensional arrays are dimensioned to 10 (elements). If two subscripts are specified, the array is automatically dimensioned to 10, 10.

Allowing the system to dimension arrays to a default value is acceptable, but not recommended. Not only is data storage allocated more efficiently, but it is better programming practice to create arrays explicitly using DIM or COMMON statements and avoid problems that may occur later. Because arrays are dynamically dimensioned, BASIC will always ignore a DIM statement that attempts to allocate the array to a smaller size later in the program.

When a numeric array is created, all of its entries are initialized to zero. String arrays are set to null.

Subscripts may be arbitrary numeric expressions. When an array reference is evaluated, the subscript is rounded to the nearest integer value, if necessary. This behavior differs from many BASIC languages which truncate the value.

For instance, suppose that AR is a numeric array whose maximum subscript is 20 and that AR(0) = 0.1, AR(1) = 1.1, AR(2) = 2.1, ..., AR(20) = 20.1. Also suppose that I = 5 and J = 13. In that case:

```
AR(I + 5) = 10.1
AR(I + J) = 18.1
AR(J / I) = 3.1      (note that J / I = 2.6, which is rounded to 3)
```

Although the above forms are valid, we recommend using integer variables and expressions for subscripts whenever possible, as they are more efficient.

Functions

A BASIC function acts as an abbreviation for a predefined or user-defined series of operations. The syntax for a function is similar to that of an array variable, except that instead of one or two subscripts the function has zero or more *arguments* which it uses to perform the desired operation.

There are two types of BASIC functions:

- Built-in or intrinsic functions such as SQR, LOG, and SIN.
- User-defined functions with names prefixed by the characters “FN”.

The following are valid function names:

```
SQR(25)          Intrinsic - returns the square root of 25
INT(TOTAL)       Intrinsic - returns the integer value of TOTAL
FNA$(A1$,B2)     User-defined function
```

Built-in Functions

CET BASIC provides a full set of built-in or intrinsic functions which may be used to perform a wide variety of trigonometric, algebraic, and general numeric and string operations, including some to handle screen operations.

The names of the built-in functions are considered reserved words and may not be used as variable names.

A list of the reserved words is included in the *Appendix* for your convenience. For a detailed description of the built-in functions, refer to a separate chapter later in this manual.

User-Defined Functions

When a set of operations needs to be executed a number of times, a user-defined function may be created to perform the operations. Note that this

type of function must be defined in each program in which it is used. In addition, the name of the function must always start with the letters FN.

For information on user-defined functions, refer to the description of the DEF statement in Chapter 5.

Expressions

Expressions are used extensively within a BASIC program to specify a series of operations to be performed on variables, constants and functions. The result of an expression is a single value.

There are several types of expressions. They are:

- Arithmetic expressions
- String expressions
- Relational expressions
- Boolean expressions

The type of expression is not only determined by the kind of operations it contains, but also by the type of constants, variables or functions that are being operated on.

An expression may contain several subexpressions of different types. Generally, the type of the subexpression determines the order in which it is evaluated. However, parentheses may be used to alter this order. (See the *Expression Evaluation* section later in this chapter.)

The following are examples of expressions:

2.3456	Arithmetic expression
A*SQR(GIRTH%)	Arithmetic expression
NAME$[1:10]&"ABCDEF"	String expression
"Name:"&SPACE$(1+4)&NAME$	String expression with arithmetic subexpression
A OR B	Boolean expression
NAME1$ > NAME2$	Relational expression
CAT <= BIRD AND DOG	Boolean expression with relational subexpression
CAT <= (BIRD AND DOG)	Relational expression with Boolean subexpression

Operators that affect only one item or term in an expression are called *unary* operators. The substring and NOT operators are always unary operators. The

substring operator follows the affected term, while NOT precedes the term to be modified.

Operators that use two terms are called *binary* operators. Binary operators are placed between the terms they use as operands. Plus and minus signs can be either binary or unary operators.

Parentheses are referred to as a grouping operator since they may be used to specify the logical groupings in an expression.

Each type of expression is described separately in the following sections.

Arithmetic Expressions

The arithmetic expression is the type most commonly used. It has a numeric value and is defined as either:

term* binary-operator *term

unary-operator *term*

An arithmetic term may consist of any of the following:

- Numeric or integer constants
- Numeric or integer variables
- Numeric or integer array variables
- Numeric or integer functions
- Logical expressions
- Binary expressions
- Relational expressions
- Other arithmetic expressions

The arithmetic operators are:

Operator	Function	Meaning
^	Exponentiation	Raises a number to a power
*	Multiplication	Multiplies two values
/	Division	Divides one value by another
+	Addition	Adds two values (for a unary positive)
-	Subtraction	Subtracts two values (for unary negative)

When an expression contains multiple operators, the order in which the expression is evaluated is determined by the *precedence* of the operators: exponentiation is performed first, then multiplication and division, and finally addition and subtraction. (See the *Expression Evaluation* section.)

An arithmetic expression whose terms have different data types is evaluated in the following sequence:

- If any term is a long or short binary real, all items are forced to long binary real and evaluation takes place according to that data type.
- If any term contains a decimal real number, all items are forced to a decimal real type and evaluation takes place accordingly.
- All items are converted to integers and evaluated.

This evaluation process may be surprising to programmers familiar with other BASIC languages where the result of the expression "5 / 4" is 1.25. With CET BASIC, integer division is used and the result will be 1. In order to get the floating point result, you will need to specify 5.0 / 4.0 or 5 / 4.0.

Two arithmetic operators cannot be used together unless the second operator is a unary minus or unary plus.

The following are examples of valid arithmetic expressions:

A%	Integer result
A% + 23	Integer result
SUB.TOTAL + CURRENT * UNIT.PRICE	Decimal Real result
ONE% * THREE	Decimal Real result
1/-4	Integer result
PI * RADIUS!^+2	Long Binary Real result
3 * 4 / (PI * R^2)	Decimal Real result

String Expressions

String expressions may contain only string terms. The syntax is either:

term* operator *term

***term* substring-operator**

String terms may consist of any of the following:

- String constants
- String variables
- String array elements
- String functions
- String expressions

The following are examples of string terms:

String	String Type

"AbcdefgH"	string constant
NAME$	string variable
M$(I+1), A$(3,5)	string array elements
SPACE$(14), LEFT(A$, 4)	string functions
(A$ & "abcde" & C$[3:I-2])	string expression

A special string function called STR$ may be used to convert a numeric term to a string term so that NAME$ & STR$(NUM%) will also be acceptable.

There are two string operators.

Operator	Function	Meaning
&	Concatenation	Joins two or more strings
[m:n]	Substring	Extracts part of the string

The concatenation operator combines two strings to make one longer string. This operator may also be written as a plus (+). For example:

Expression	Result
"alpha" & "bet"	"alphabet"
"John" + " " + "Doe"	"John Doe"

The substring operator extracts a short string from a longer one. An operator of the form [m:n] extracts characters starting from character m through character n. The length of the substring will always be n - m + 1.

The numbering of characters always begins at 1. Therefore, if m is less than 1, 1 is used instead of m. Conversely, if n is greater than the length of the string term, the length of the term is used instead of n. The substring will be empty or null if m is greater than n. For example:

Expression	Result
"alphabet" [3:5]	"pha"
"alphabet" [-10:50]	"alphabet"
"alphabet" [5:4]	"" (a null string)
("alphabet" [3:6]) [2:4]	"hab"
("alpha" & "bet") [2:7]	"lphabe"
"alpha" & "bet" [2:7]	"alphabet"

Note that substrings may be used on the left side of a string assignment statement with a somewhat different meaning. For information on this use, refer to the description of the LET statement in Chapter 5.

Relational Expressions

A relational expression operates on numeric or string terms and produces a sixteen bit integer value of -1 (*true*) or 0 (*false*). A relational expression is defined as either:

numeric-expression* relation *numeric-expression

string-expression* relation *string-expression

A relation may be any of the following. Note that the operators >=, <= and <> are treated as single tokens.

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
=	Equal to
<>	Unequal to

The following are all considered valid relational expressions:

Expression	Type
STRING$ > "HELLO"	String expression
NUM1 <= NUM2!	Numeric expression
NUMBER% <> 225*(5-ONE)	Numeric expression with an arithmetic subexpression
539 = ONE	Numeric expression (not assignment)

The following are not acceptable:

"Goodbye" <> 25	Can't mix strings with numerics without using the STR$ function to convert the numeric value 25
5 < NUM <= 18	Must use (5 < NUM) AND (NUM <= 18)
FIRST$ OR SECOND$	Logical operators cannot be used with string terms

Boolean Expressions

A Boolean expression is an integer expression that produces an integer value. A Boolean expression is defined as either:

term binary*-operator *term

NOT *term*

Each term may consist of:

- Integer constants

- Integer variables
- Integer arrays
- Integer functions
- Integer arithmetic expressions
- Binary expressions
- Relational expressions

Boolean expressions use one or more binary operators to compare, combine or negate the actual *bits* of the affected integer terms. Each integer is composed of 16 bits where each bit is either a zero (0) or a one (1). Numeric values are converted to integers before expressions are evaluated.

A decimal integer is expressed in bits by converting the number to base two (actually, *two's complement*) notation and adding leading binary zeros, if necessary. The following is a list of some equivalent values:

Decimal	Binary Bits
0	00000000 00000000
1	00000000 00000001
100	00000000 01100100
32767	01111111 11111111
-32768	10000000 00000000
-1	11111111 11111111

The following binary operators are recognized:

Operator	Function
AND	Result is 1 if both operand bits are 1. Otherwise, the bit in the result is 0.
OR	Result is 1 if either one or both of the operand bits are 1, otherwise the bit in the result is 0.
XOR	Result is 1 if either (but not both) operand bits are 1. Otherwise, the resulting bit is 0. (eXclusive OR)
IMP	Result is 1 unless left operand bit is 1 and the right operand is 0. Otherwise, the bit in the result is the same as the bit in the second term. (IMPlication)
EQV	Result is 1 if both bits are the same (either both 1 or both 0). If they are different, the result is 0. (EQuiValence)
NOT	Result is 1 if the operand bit is 0, otherwise it is 1. In other words, each bit in the affected term is reversed.

Boolean operators have a precedence which determines order of evaluation. This precedence is covered in the *Expression Evaluation* section.

The following tables, called truth tables, illustrate the results of the Boolean operations for all possible bit combinations. Note that this the default interpretation in a CET BASIC program. Under some platforms, the OPTION LOGICAL statement may be used if AND, OR and NOT need to be interpreted as logical operators. (Refer to the OPTION statement in the *CET BASIC Statements* chapter.)

The AND and OR operators are defined by:

AND

A%	B%	A% AND B%
0	0	0
0	1	0
1	0	0
1	1	1

OR

A%	B%	A% OR B%
0	0	0
0	1	1
1	0	1
1	1	1

The XOR (eXclusive OR) operator is defined by:

XOR

A%	B%	A% XOR B%
0	0	0
0	1	1
1	0	1
1	1	0

The IMP (IMPlication) and EQV (EQuiValence) operators are defined by:

IMP

A%	B%	A% IMP B%
0	0	1
0	1	1
1	0	0
1	1	1

EQV

A%	B%	A% EQV B%
0	0	1
0	1	0
1	0	0
1	1	1

The NOT operator is defined as:

NOT

A%	NOT A%
0	1
1	0

The NOT operator performs a bit by bit operation, not a logical operation on its whole operand. Therefore, it is possible that "IF A AND B" and "IF NOT(A AND B)" will *both* evaluate to true. This happens when A AND B is not all 0's or all 1's. Some examples of valid Boolean expressions are:

```
NUM1% OR NOT NUM2%
I% AND 23
I% AND (NUMBER XOR TOTAL) IMP TEST%
(A AND B) OR (A AND C)
STRING$ >= "A" AND STRING$ <= "Z"
```

The following are unacceptable:

```
STRING$ OR "HELP"       Must be arithmetic terms
NUM1% AND OR NUM2       Binary operators cannot be adjacent
```

Boolean expressions are normally used to evaluate terms that are the result of relational expressions in which the bits are either all on or all off. However, since the expression compares all sixteen bits of each term, there are many other uses for logical expressions. The *bit switches* or binary coded

information used in some of the arguments to the Window System functions illustrate how this feature may be used. The following examples illustrate how the logical operators work on non-relational values:

```
15 AND 14          0000000000001111 (15)
      AND          0000000000001110 (14)

                   ----------------
                   0000000000001110 (14) (True)

10 OR 23           0000000000001010 (10)
      OR           0000000000010111 (23)

                   ----------------
                   0000000000011111 (31) (True)

25 XOR 13          0000000000011001 (25)
      XOR          0000000000001101 (13)

                   ----------------
                   0000000000010100 (20) (True)

29 XOR 29          0000000000011101 (29)
      XOR          0000000000011101 (29)

                   ----------------
                   0000000000000000 (0) (False)

234 EQV 3429       0000000011101010 (234)
      EQV          0000110101100101 (3429)

                   ----------------
                   1111001001110000 (-3472) (True)

56 IMP 720         0000000000111000 (56)
      IMP          0000001011010000 (720)

                   ----------------
                   1111111111010111 (-41) (True)

NOT 720  NOT       0000001011010000 (720) (True)

                   ----------------
                   1111110100101111 (-721) (Also True!!)
```

Setting and extracting bit fields in integers may also be done using the built-in shifting functions LRL, LRR, LSL and LSR that are covered in the *Built-in Functions* chapter later in this manual.

Expression Evaluation

BASIC evaluates expressions according to an operator precedence. Each arithmetic, string, relational and Boolean operator in an expression has a predetermined position in the hierarchy of operators.

Unless altered by parentheses, the operators with the highest precedence number are evaluated first. If there are operators with the same precedence, they are evaluated left to right. The following table lists all of the operators in order by their precedence:

Operators	Description	Precedence
()	Grouping	12
function	Predefined functions	12
^	Exponentiation	11
& +	String concatenation	10
[m : n]	Substring	9
+ –	Unary positive and negative	9
* /	Multiplication and division	8
+ –	Addition and subtraction	7
= < <= = <>	Relational operators	6
NOT	Binary NOT	5
AND	Binary AND	4
OR	Binary OR	3
XOR	Binary exclusive OR	2
EQV	Binary equivalence	1
IMP	Binary implication	0

For example:

Expression	Correct Evaluation	Incorrect Evaluation
3 - 4 + 5	(3 - 4) + 5 = 4	3 - (4 + 5) = -6
3.0 / 4.0 * 5.0	(3.0 / 4.0) * 5.0=3.75	3.0 / (4.0*5.0) = .15

BASIC evaluates the following expression in five steps:

```
A = 15 ^ 2 + 12 ^ 2 - 35 * 8

1. 5^2 = 225          Exponentiation (leftmost)
2. 12^2 = 144         Exponentiation (next)
3. 35*8 = 280      Multiplication
4. 225+144 = 369      Addition (leftmost)
5. 369-280 = 89       Subtraction
```

Parentheses may be used to change the order of evaluation since BASIC always evaluates expressions inside parentheses first. Even if they are not required, parentheses may always be used to clarify an expression.

Consider how parentheses effect the result of the following expressions:

```
25^2 + 30^2 / 2       (25^2 + 30^2) / 2

1. 25^2 = 625         1. 25^2 = 625
2. 30^2 = 900         2. 30^2 = 900
```

```
3. 900/2 = 450          3. 625+900 = 1525
                        4. 625+450 = 1075     4. 1525/2 = 762.5

  Result is 1075          Result is 762.5
```

As shown in the precedence table, the relational operators have precedence over the Boolean operators. For example, in the following:

```
"A" > "B" OR "A" <= "D"

1. "A">"B" = 0          0000000000000000 (false)
2. "A"<="D" = -1        1111111111111111 (true)
3. 0 OR -1 = -1         1111111111111111 (true)
```

Binary Real Data Types

Real variables and expressions normally default to the CET BASIC decimal floating point format. Binary real arithmetic is also supported.

If your calculations are primarily financial, you can skip this section. Binary real arithmetic is generally considered unsuitable in business applications since many exact dollar amounts are not exact binary numbers. For instance, 2.20 expressed in binary is 10.001100110011 and rounding this to any finite length will give a slightly inaccurate result. It is possible to circumvent this problem to some extent by using PRINT USING and the ROUND function, but this is tricky and generally not worth the effort.

Therefore, this topic is normally of interest only to those doing scientific and engineering calculations where the speed of computation, the range of intermediate results and/or the degree of precision is important.

Because of its specific use, this feature is currently only supported in the CET BASIC product for UNIX. Check with your CET distributor about availability if you need to use this feature in your application.

The advantages of binary real arithmetic include:

Speed of Computation

CET BASIC provides for use of both short and long IEEE formatted binary real numbers. On systems with a math coprocessor, all arithmetic operations and intrinsic (built-in) functions will execute from 10 to 30 times faster when using binary instead of decimal real values.

Precision

CET BASIC decimal real numbers always maintain exactly 13 decimal digits, whereas binary real numbers can represent approximately 6 or 16 decimal digits of precision, depending upon whether single or double precision values are used.

Short binary real numbers are four bytes long with about 6 decimal digits of precision, and an exponent range from +38 to –38. Long binary real numbers are 8 bytes long with about 15 decimal digits of precision, and have an exponent range from +308 to –308. All computations take place internally in the longer format, so the only gain from using the shorter form is in reduced storage requirements.

Binary Real Variables

Normally, variable names which are followed by no special characters are interpreted as decimal real or floating point variables. Decimal real variables and constants retain 13 decimal digits of precision and range in value from 10^{-128} to 10^{+128}.

Binary real variables are recognized by names of the form *name*! and *name*!!, where *name* is any variable name and the exclamation point(s) indicates that the variable is a binary real.

A single exclamation point is used for single-precision variables. These variables are interpreted as IEEE standard, 32-bit binary floating point numbers with approximately 6 decimal digits of precision that range from 10^{-38} to 10^{+38}.

A double exclamation point is used to a designate double-precision. These variables are interpreted as IEEE standard, 64-bit floating point numbers with approximately 16 decimal digits of precision and a range from approximately 10^{-308} to 10^{+308}. For example:

X	a decimal real variable
X!	a 32-bit binary real variable
X!!	a 64-bit binary real variable

Binary real array variables are also supported. This feature, coupled with the executable nature of the DIM statement, provides a powerful method for generalizing many mathematical algorithms whose array dimensions are not known until execution time.

Binary Real Intrinsic Functions

CET BASIC intrinsic or built-in functions which accept numeric arguments may be invoked with binary real values as arguments.

Using binary real arguments in some functions will cause the computation to be carried out in binary real arithmetic. This feature is present in any intrinsic function for which additional precision may be desired, including the trigonometric (e.g., SIN), logarithmic (e.g., LOG2) and other transcendental functions.

CET BASIC also supports binary real forms of some functions that do not have arguments. These include:

INF! and INF!!	Binary versions of INF
EPS! and EPS!!	Binary versions of EPS
PI! and PI!!	Binary versions of PI

Specific functions such as HEXOF and LRL require arguments of limited range. The intrinsic function BFLOAT forces its argument to a 64-bit binary floating point type.

Note that using binary real expressions as arguments to many intrinsic functions will have no affect on the result. For example, the following lines will execute the same.

```
N!! = 10 \ PRINT RPT$(N!!,"*")
N = 10 \ PRINT RPT$(N,"*")
```

Binary Real User-Defined Functions

User-defined function names may also be suffixed by a single or double exclamation point to indicate that the value they deliver is in binary floating point form. For example:

```
DEF FN.BR!!(A%,B!,C$)
    IF LEN(C$) < 10
        FN.BR!! = B! ^ A%
    ELSE
        FN.BR!! = B! ^ -A%
    IFEND
FNEND
```

Binary Real Arithmetic

A binary real variable or expression may be used anywhere a decimal real variable is allowed. Conversion between the various numeric data types is automatic. All binary floating point computation takes place in the IEEE standard 8-bit temporary format. Parameters are passed and function values are returned in a 64-bit format.

The only difference between variable names with a single and double exclamation point is that the former have less precision and require half as much data storage.

OPTION DEFAULT

To make it easier to write programs that use binary real arithmetic, the keyword DEFAULT may be used with the OPTION statement to indicate how variables with no special trailing character(s) should be interpreted.

The syntax of the OPTION DEFAULT statement is as follows. The suffix x may be null, 4 or 8.

```
OPTION DEFAULT REALx
```

From the point in the program that an OPTION DEFAULT statement occurs, all newly defined variables with no special trailing characters will assume the data type specified with OPTION DEFAULT.

For example, if (at compilation time) OPTION DEFAULT REAL4 was the most recently scanned DEFAULT statement, all new variables which have no special terminating characters will become short binary reals, and all floating point constants will be stored internally as short binary reals.

The DEFAULT REAL value may be changed during the execution of a program. To change to a long binary use the OPTION DEFAULT REAL8 statement. Similarly, the program can be returned to the normal default (decimal real), by executing:

```
OPTION DEFAULT REAL
```

OPTION DEFAULT also affects the meanings of certain built-in functions. PI, INF, EPS, and FLOAT will all assume values in the current default data type, and constants will be stored internally as 64-bit binary floating points.

Certain other mathematical functions, such as SQR, LOG, EXP, and the trigonometric functions, are *generic* in the sense that they produce a value

which is the same type as their argument. In fact, different routines are used internally for decimal and binary SQR.

For example, the following code computes the square roots of numbers from 2 to 1000. It uses decimal floating point variables exclusively.

```
FOR I = 2 TO 1000
      S = .5*(1 + I)
      WHILE (ABS(S*S/I - 1) > 1E-6)
            S = .5*(S + I/S)
      WEND
NEXT I
```

A binary floating point version of the sample above may be created by prefixing the code with the statement:

```
OPTION DEFAULT REAL8
```

This version will execute approximately 33 times faster than the first one when a math coprocessor is present.

The next example uses decimal real arithmetic:

```
FOR I = .001 TO 1 STEP .001
      S = SIN(I)^2 + COS(I)^2
NEXT I
```

The same sample code using binary real arithmetic:

```
FOR I!! = .001 TO 1 STEP .001
      S!! = SIN(I!!)^2 + COS(I!!)^2
NEXT I!!
```

Disadvantages and Caveats

There exist some potential disadvantages in the use of binary real arithmetic. Consider the following:

Implementation Restrictions

Floating point arithmetic may not be supported on all math coprocessors.

Accuracy

Binary real variables can only represent decimal fractions within the precision that they retain (16 digits in 64-bit reals), whereas decimal

arithmetic always maintains exact results for decimal fractions within the bounds of its precision (13 digits).

For this reason, decimal real arithmetic has conventionally been the choice of developers who must perform arithmetic on (dollar) values which contain decimal fractions (cents). Often, identical results can be obtained by scaling dollar-denominated values by 100, and performing all calculations with scaled data. The ROUND function may also be used to greatly reduce the error introduced by using binary real arithmetic.

It may be important to note that if the usual formula for compound interest is used (P*(1+I)^N), the results are liable to differ slightly from what banks obtain using floating point values. Banks normally perform the calculation by executing the following statement *n* times:

```
P = ROUND(P*(1+I), 2)
```

PRINT and PRINT USING Limitations

The PRINT and PRINT USING statements may not support the extended range of double precision binary real expressions and expressions smaller than 10^{-128} or larger 10^{+128} may yield unpredictable results.

Limits on Expression Complexity

Very complicated expressions requiring deep nesting and/or function calls can cause the math coprocessor stack to overflow. This condition can be detected only at runtime.

Please contact your CET distributor is you have questions about using binary real data or the level of support that is available in the specific product you are using.

CHAPTER 4

CET BASIC Data Files

Introduction

This chapter is designed to give you general information on the various types of CET BASIC data files. For other file-related information refer to:

Topic	Section
Creating files using Bcreate	Development Utilities and Built-in Functions chapters in the *CET BASIC User's Guide*
File-related environment variables	CET BASIC Compiler and Runtime Systems chapter in the *CET BASIC User's Guide*
Maintaining a CET database	related statements in chapter 5 of this manual.
Multi-user file access	the last section in this chapter

File Names

CET BASIC programs recognize files specified with the complete path name or a name relative to the current directory. File names may be in a format native to the operating system in use. For instance, under DOS you can reference **c:\pos\cp\data\custname**, **\pos\cp\data\custname** or **.\custname**.

For compatibility purposes, the CET DOS/Networking and W/32 Windows products recognize file names in either a DOS or UNIX format. For example, the CET BASIC will automatically convert any "/" characters in the name to "\". Since UNIX is case sensitive, a UNIX formatted name must be specified in order to access a file with a name in lower case letters.

File names may be also be specified in a THEOS format. Any name that contains a period, but does not begin with a drive code, a slash (/ or \) or a dot slash (./ or .\) is considered a THEOS name, and the default *filetype* translation method will be used to determine which file should be accessed. Drive codes are ignored unless the B_?DRIVE variable is set.

THEOS File Name	DOS File Name	UNIX File Name
CUST	CUST	CUST
CUST.DATA	DATA\CUST	DATA/CUST
CP.DATA.CUST	CP\DATA\CUST	CP/DATA/CUST

File-related environment variables may be set to affect how files are accessed. For example, B_LIBFNTYP may be used so that CET BASIC will read a name formatted as **a.b.c** and open **.\b\a\c** in the current directory. Please refer to the *CET BASIC User's Guide* for more information on the variables that are available for the specific product you are using.

Accessing Files

The OPEN statement is used to open a file on a specific I/O *channel.* The *channel* number is then used in all further references to the file.

The number of files which may be open at the same time depends upon the CET BASIC product in use and the type of the files themselves. The number is limited because each open file requires buffer and table space, but is generally at least sixteen indexed files. Additional direct and sequential files may be opened at the same time.

Valid channel numbers are integers from 1 to 24. (More channels are available in environments other than DOS.) The console is considered as an open file, and in some statements is assumed to be on channel #0.

The sequence of statements in a BASIC program that use a file is:

```
OPEN
INPUT, LINPUT, PRINT, READ, WRITE, ...
CLOSE
```

OPEN

Before a file can be accessed, an OPEN statement must be performed to specify the name of the file, the associated channel number, the access mode and method and any options that are to be used with the file while it is open.

INPUT, LINPUT, PRINT, READ, WRITE. . .

A variety of statements may be used to perform input and output operations on the file. Which statement to use depends upon the access mode specified in the OPEN statement and the format of the file itself.

CLOSE

The CLOSE statement indicates that the operations on the file that was opened on the specified channel are now complete, and the channel should be closed. Afterwards, the channel may be used to open another file, while the first file may be reopened on a different channel. When the program ends, any files which are still open are closed automatically.

Refer to the *CET BASIC Statements* chapter for specific information on these and the other BASIC statements which may be used to access a file.

Access Mode

Files may be accessed in three ways. The access mode to use must be specified with the OPEN statement. A file may be opened for:

INPUT

This mode indicates that the file is to be used only for input operations such as those performed with the READ and INPUT statements. When this mode is in effect, BASIC will not allow data to be output to the specified file channel so that the user cannot accidentally change the contents of the file.

OUTPUT

This mode indicates that the file is to be opened only for output operations such as those performed with the WRITE and PRINT statements. BASIC statements that perform input type operations on the specified channel will not be allowed. This mode is normally used when a file is being created, or on an output-only device such as a printer.

UPDATE

This mode allows both input and output operations to be performed on the file, and is the one most commonly used in *data entry* programs.

When running under a network or multi-user system, CET BASIC will automatically handle all file and record locking when this mode in effect. Since the locking/unlocking operations obviously take time, we recommend

using the UPDATE mode only when needed in order to optimize performance (especially on a DOS-based network).

File Access Methods

The OPEN statement also requires that you specify which method should be used to access the file. This must be the same method that was used to create the file.

SEQUENTIAL

Records in sequential files are read or written in sequential order, one after the other, starting at the beginning of the file. Since a program always has to read each record in the file before it can access the one desired, this method is not normally used when records need to be retrieved quickly.

Sequential files may contain variable length records. The record length is determined by the amount of data that is entered. Since the records are created one after another, with no gaps or empty spaces between them, this type of file uses disk space very efficiently.

Typically, this type of file is used to store daily transactions where existing records need to be read, but not updated. New records will be added to the end of the file.

Sequential files are normally created by opening a file for OUTPUT with the option SEQUENTIAL. If a file with the specified name already exists, it will be deleted before the new file is created. Therefore, subsequent PRINT or WRITE statements always add records to the beginning of the file. The only way to add to an existing file is to open it with the EXTEND option, which causes records to be added at the end.

PRINT will output data in an ASCII format. The WRITE statement outputs binary data.

Under DOS and Windows, a carriage-return+linefeed is used as the end-of-record indicator. UNIX uses a carriage-return unless OPTION NEWLINE is in effect in which case a linefeed is used.

DIRECT

Direct files allow direct, random access to a record by using the key to the record. The key is a number indicating its position in the file. The first record entered has a key of 1, the second has a key of 2, etc. Typically, direct files

are used when the records have sequentially numbered keys such as an invoice or order numbers.

Access to this type of file is very fast since any record may be accessed directly, without having to read any other record.

Direct files are accessed by opening a file with the option DIRECT. Each record has a fixed length which is determined when the file is created. Note that direct files must be created with the CET Bcreate utility or the Bcreate function before they can be accessed. Refer to your *CET BASIC User's Guide* for information on how this is done.

PRINT or WRITE statements may be used to add records to an existing file. PRINT will output data in an ASCII format, while WRITE outputs binary data.

```
WRITE #channel, record-no: data-list..
PRINT #channel, record-no: data-list...
```

Each WRITE or PRINT statement normally updates the contents of an entire record. Therefore, a direct record can only be modified by reading in a record, changing some or all of it, and writing it back out.

Each record in a CET BASIC direct file is organized as follows:

Number	Length	Contents
0	*hlen*	Information describing the file.
1	*reclen*	Data written by the program, if any.
2 (and so on)	*reclen*	Data written by the program, if any.

The variable *hlen* is the length of the header record, and *reclen* is the value of the RECLEN parameter specified when the file was created.

Note that record zero is not accessible by a BASIC program. This record is referred to as the header record because it contains information associated with the file. The format of this record is:

Byte Offset	0	2	4	6
Byte Value	MagNo	OffSet	RecLen	Filler

The first two bytes (MagNo) of the header record contain a *magic* number which identifies this file as a direct file. Traditionally, this has been the hexadecimal number E5DA.

The next two bytes (OffSet) contain the number of bytes of offset into the file for the first data record. This field actually represents the length of the

header record in bytes. It is included so that future versions of CET BASIC can properly access existing direct files, even if a change is made in the structure of the header record.

The fourth and fifth bytes (RecLen) contain the record length with which the direct file was created. This value, together with the header OffSet value, is used by the CET BASIC direct access routines to determine the byte offset to the desired record.

The remainder of the header record (Filler) contains zeros.

INDEXED

Indexed files are also referred to as ISAM files (indexed sequential access method). This access method is similar to the one used for direct files in that any record in the file may be accessed directly by using the record's key.

Records are maintained logically in alphabetic sequence by key. That means that your program may allow users to view their records in alphabetical order even though they were entered randomly.

This file format is the one most commonly used even though maintaining a sequential index for the records in the file means that updating this type of file is slightly slower than for a direct file.

Indexed files, like direct files, must be created with the CET Bcreate utility or the built-in Bcreate function. The record key is a fixed-length string associated with each record, whose length is also fixed when the file is created.

Indexed files are written by opening the file for OUTPUT or UPDATE with the option INDEXED. Records are added by executing a statement in the following format:

WRITE *#channel, key: data-list...*
PRINT *#channel, key: data-list...*

Each WRITE or PRINT statement normally updates an entire record in the indexed file. WRITE outputs binary data and PRINT outputs ASCII data.

CET BASIC normally uses the C-ISAM indexed file structure, a commercial multi-user access method from Informix, Inc. C-ISAM creates and maintains two files for every CET BASIC indexed file. (When C-ISAM is not available for a particular platform, a C-ISAM compatible product is used.)

Accessing a file such as **data\orders** generates a reference to a pair of files named **data\orders.dat** and **data\orders.idx**. The former represents the set

of data records provided by a CET BASIC program, while the latter is the index file maintained by C-ISAM.

Each data record in the **.dat** file actually contains both the data and the key to the record. The key portion is a copy of the key associated with the record in the **.idx** file, and is included in the data record to provide sufficient redundancy for file restoration in the event of disk failure or corruption.

Record Allocation Requirements

Since indexed and direct files must be created with a fixed record length, the maximum record size required must be calculated for each file. This is done by determining which types of fields will be written and allocating the number of required bytes accordingly.

Field Type	Length
Decimal Real Number	9 bytes
Long Binary Real Number	9 bytes
Short Binary Real Number	5 bytes
Integer Number	3 bytes
String Type	2 or 3 bytes

For strings shorter than 255 bytes, add 2 bytes plus the length of the longest string, otherwise add 3 bytes plus the string length.

Each of the sizes noted above includes the length of the data and a one-byte type code which is stored with the field. The total record length must be at least the sum of these lengths, plus one byte for the end-of-record indicator. For example, a direct file must be created with a record length of 33 bytes in order to contain the following data:

```
WRITE #1,N: "RECORD",1,2,A,B
```

When porting files from UX BASIC, note that the lengths required for each record are slightly larger. You must add one extra byte for each numeric field (so the lengths are 10, 10, 6, and 4, respectively) and two bytes for the end-of-record indicator. Therefore, in the example above, the record would have to contain at least 38 bytes. Also note that UX BASIC direct and indexed files must be created using the UXCREATE utility instead of the CET Bcreate utility.

Record Buffer Limitations

In general, any PRINT or WRITE statement will cause an identical record to be written, regardless of whether the destination is an indexed, direct or sequential file. However, when the record to be written to a direct or indexed file exceeds the default record buffer length or the value of the environment variable B_BUFSIZ, a truncation error will be detected. Sequential files have no such limit as all output is performed immediately without buffering.

Data Representation in Files

CET BASIC statements may be used to transmit or record data in three formats: binary, ASCII or raw.

ASCII FILES

ASCII formatted files are similar to those created with a standard text editor. This type of file may be created with a BASIC program with PRINT or PRINT USING statements.

The various PRINT options that affect the format of the print line include:

- Use of TAB functions to position the next character output.
- Use of commas to print data in 21-character columns, or the semicolon to suppress the addition of extra spaces.
- Opening the file with the QUOTE option to place commas between individual data items and enclose strings within quotation marks.

More advanced formatting is available with the PRINT USING statement.

ASCII formatted files can only be read by the INPUT, LINPUT (USING) or GET statements.

- INPUT statements may be used to read any stream of data separated by commas or semicolons when the file is opened with the option COMMA. (This record format is created using PRINT statements with files opened with the option QUOTE.)
- LINPUT statements may be used to read an entire ASCII record into a string variable. LINPUT USING is a similar statement that provides additional length control by using a mask which may be edited.
- GET may be used to read a raw, unformatted stream of bytes from an input channel or the console.

Variable Length

When using a PRINT statement terminated with a semicolon, each successive PRINT appends output to the same record. The length of an ASCII record is limited only by the size of the I/O buffer, approximately 32 Kbytes.

Under DOS and Windows, the end of record delimiter is the carriage-return+linefeed. The CET UNIX-based products use a carriage-return as the default delimiter.

To conform to UNIX standards, a file may be opened with the option NEWLINE if the linefeed delimiter is used. The environment variable B_OPENNL may also be set to avoid having to modify code ported from another platform that uses carriage-return delimiters. For compatibility purposes, both of these features are recognized under DOS and Windows, but they will have no effect.

Pure ASCII

ASCII files may be read with any editor. Normally, these files do not contain any special codes or control characters. To write a non-printable character to an ASCII file you must:

1. PRINT a string which has embedded, non-printable characters. This includes printing the value of AT$ or CRT$ functions to a device other than a console.
2. OPEN the file without the FORMAT option and the (optional) record delimiter will be either the carriage-return or the linefeed character.
3. OPEN the file with option FORMAT and the carriage-return, linefeed and formfeed (decimal 12, hexadecimal 0C) characters will be automatically written at the end of the record.

BINARY FILES

Binary files contain compressed data with enough internal information about the individual data items to provide some trappable error detection.

Records are created with the WRITE statement, and can be accessed by the READ statement. Consider the following WRITE Statement:

```
WRITE #1: A, B%, C$
```

The format of this record will be:

Byte Offset	0	1	9	10	12	13	15	lenC(C$)+15
Byte								

Value	2	A	1	B%	4	len	C$	0

Notice that each data item written to a binary file is preceded by a 1-byte code for the data type.

Data Type	Type Code	Length in Bytes
Integer	1	2
Real	2	8
String	4	*len* variable
End-of-record	0	N/A

String values are preceded by the string data type code (4) and their length (*len* in the illustration above).

The default record length for binary files is 1200 under DOS and 2056 under the UNIX-based products. The environment variable B_BUFESIZ may be used to increase the default length.

Raw Files

Raw, unformatted files are created with the PUT statement which records or transmits a byte value to the designated channel. No data type conversion is performed. Neither the data type codes nor the record delimiter characters are placed in the output stream.

The GET statement may be used to get unformatted data directly from an open channel one byte at a time. Although GET may be used to access a file of unknown structure, it is not normally used to get data from direct, indexed or sequential files.

Multi-User File Protections

When several programs running on a multi-user system attempt to access the same file, a problem known as a *race* condition may occur regardless of whether or not a BASIC program is involved. CET BASIC provides you with some powerful features to protect users against the consequences of such conditions.

For example, imagine having two programs attempt to update the BALANCE field in the same record (in an indexed or direct file). Program A tries to add 500 to the field while program B tries to subtract 200. If both programs are executed simultaneously, the following condition may occur:

Program A reads record and gets BALANCE.
Program B reads record and gets the same BALANCE.
Program B computes BALANCE -200 and writes it back to the record
Program A computes BALANCE +500 and writes it back to the record.

When that happens, the final value will be BALANCE+500 instead of the correct BALANCE+300. Depending on how the programs are interleaved, the final result could also be BALANCE+300, BALANCE+500 or BALANCE-200.

If both of these programs are CET BASIC programs, this problem will never occur. By default, the BASIC system will *lock* the record when it is read by Program A so that it can not be read or written by any other BASIC program until Program A has done the corresponding write or the file is closed.

If Program B attempts to read the locked record, the program is suspended, with periodic retries until the record is unlocked unless there is an ON ERROR or ON LOCK routine in effect. These statements may be used to specify other behavior for a locked record condition.

If the B_EMULATE variable is set, a locked record is not considered a trappable error. If you wish to use THEOS emulation and special error handling, set the B_THLOCK variable off (to null).

Note that if an ON ERROR routine exists, but does contain any code to handle a locked record (error #48), the program will abort with an appropriate error message.

It is also important to note that the environment variable B_OPTLOCK must be set so that the file pointer is not moved after the RESUME statement and the READNEXT or READPREV statements will attempt to read the desired record again. If this variable is not set, BASIC will move to the next or previous record, by default.

In some applications it may be necessary to update several records of a file in such a way that if they were accessed by another program before the entire operation was completed, the results could be invalid.

To guarantee exclusive access to a file, use OPEN with the LOCK option. If the file is already being used by another program when the OPEN request is made, the default action is to terminate. If that is undesirable, code may be written to perform a SLEEP and try again until the file is available.

For further information on record and file locking, refer to the various BASIC statements that handle file I/O and the *CET BASIC Error Handling*

System chapter. The environment variables B_EMULATE, B_THLOCK and B_OPTLOCK are covered in the *CET BASIC User's Guide*.

CHAPTER 5

CET BASIC Statements

Introduction

This chapter describes the statements which are available in CET BASIC. For easy reference, each statement is covered in a separate section in alphabetical order. For a quick overview, the statements have been grouped by function and listed below.

Control and/or Branching Statements

END	Exits program.
QUIT	Exits program.
STOP	Exits program.
FOR	Defines and starts a program loop.
BREAK	Exits from FOR loop.
NEXT	Marks end of FOR statement.
GOSUB	Executes a subroutine and returns.
ON ... GOSUB	Conditionally executes a subroutine.
RETURN	Exits from a subroutine.
GOTO	Unconditionally branches to another section of code.
ON ... GOTO	Conditionally selects branch.
IF	Tests a condition and executes statements depending on the result.
THEN	Defines the true path of execution for an IF statement.
ELSE	Defines the action to be taken if the result of the IF statement is false.

BREAK	Aborts the program loop.
IFEND	Ends a multi-line IF statement.
ON ERROR	Specifies error handling routines.
ON INTERRUPT	Specifies console interrupt key handling.
ON KEY	Specifies special key handling.
ON LOCK	Specifies handling of locked file or record.
ON MOUSE	Specifies handling of mouse clicks (under DOS).
RESUME	Exits error handling or interrupt processing routines and returns to the execution of the program.
SELECT	Specifies the value that determines which statements will be executed.
CASE	Specifies one possible case for the SELECT statement.
OTHERWISE	Specifies the SELECT default case.
CEND	Marks end of SELECT statement.
SLEEP	Suspends processing for a period of time.
WAIT	Suspends processing until a specific event occurs.
WHILE	Executes statements while a specified condition is true.
BREAK	Exits from WHILE-WEND loop.
WEND	Marks end of WHILE-WEND loop.

Assignment and Declaration Statements

DIM	Allocates memory for array variables.
COMMON	Declares variables to be passed between program modules.
CLEAR	Reclaims array space and removes variables from COMMON.
DATA	Defines the data constants to be used.
RESTORE	Resets the data pointer so DATA statements may be reused.
DEF	Defines a sequence of statements as a user-defined function.
FNEND	Marks the end of a multi-line user-defined function.
LET	Assigns a value to a variable or array.

MAT	Assigns one array to another or initializes an array to a specific value.
MAT SORT	Creates a sort-order array for a one-dimensional array.
SWAP	Exchanges the values of two variables.

File and Terminal Input and Output Statements

OPEN	Opens an I/O channel.
CLOSE	Closes an open I/O channel.
UNLOCK	Releases a locked record without updating it.
DELETE	Erases a record from the file.
GET	Gets unformatted data from an I/O channel.
UNGET	Puts characters back into the input buffer.
PUT	Outputs unformatted bytes of data to an I/O channel.
INPUT	Accepts ASCII data from the terminal or a file.
LINPUT	Accepts lines or records of ASCII data from the terminal or a file.
LINPUT USING	Accepts a specified number of characters of ASCII data and allows them to be edited.
PRINT	Outputs ASCII data to the terminal or an open file.
PRINT USING	Edits and formats ASCII data prior to output.
MAT INPUT	Places ASCII data into an array.
MAT PRINT	Outputs one or more arrays of ASCII data to a file or terminal.
MAT READ	Reads binary data from a file into an array.
MAT READNEXT	Reads the binary data from the next record in an indexed or direct file into an array.
MAT READPREV	Reads data from the previous record in an indexed file into an array.
MAT WRITE	Writes the contents of an array to a record in a file.
READ	Accepts data from a DATA statement or an open I/O channel.
READNEXT	Accepts data from the next record in an indexed or direct file.
READPRIOR/	Accepts data from the previous record **READPREV** in an indexed file.

WRITE	Outputs binary, formatted data to a file.
WAIT	Waits for data to be available on an open I/O channel.

Program Linkage Statements

CALL	Loads and transfers control to an external subroutine.
CHAIN	Ends the current program module starts another one without the COMMON variables. All I/O channels will be closed.
LINK	Operates similar to CHAIN except all open channels will be closed.
RUN	Ends the current program, closes all open channels, and starts another program.
CSH / CSI	Executes an operating system command and then continues execution of the BASIC program.
CSI.RETURN.CODE	Returns code from a CSH / CSI statement.

Other Statements

OPTION	Sets various parameters to affect program execution.
RANDOMIZE	Resets the pseudo-random number generator.
REM	Marks comments to be ignored by the BASIC program.

Statement Reference Guide

A complete description of each of the CET BASIC statements is included in the following section. For quick reference, they are covered in alphabetical order.

BREAK

The BREAK statement is provided as a means of exiting the current program structure.

Syntax BREAK

Description The BREAK statement may be used to exit from any FOR-NEXT or WHILE-WEND loop. Execution will be transferred to the first statement following the one which terminates the structure (NEXT or WEND).

The compiler will detect an error if a BREAK is detected outside a recognized structure. Since BREAK performs an unconditional jump to another statement, it is normally used in an IF statement.

Examples	**Explanation**
FOR I% = 1 TO 100 IF A(I) = X THEN BREAK NEXT ENDLOOP:	BREAK is equivalent to GOTO ENDLOOP.

FOR I%= 1 TO 200
  BREAK
  WHILE A(I) < X
  ...computations...
  IF B < 0 THEN BREAK
  …computations...
  WEND
  L1:
NEXT
L2:
	Here, BREAK goes to L2, not L1.
	
	Here, BREAK goes to L1, not L2.

See Also FOR, WHILE

CALL

The CALL statement provides a means of interfacing to externally compiled subroutines. When executed, control will be transferred to the specified subroutine, optionally passing values to and from the called program.

The CET DOS/Network product allows calls to be made to compiled C or assembly language subroutines. All other CET BASIC products also permit calls to externally compiled BASIC programs.

Syntax

1 CALL *program-name*

2 CALL *program-name* [(*arg-list*)]

program-name= *name of the external program*
arg-list = *arg* [, *arg-list*]
arg = *expression*
= ADDROF (*variable*)
= MATADDROF (array)

Description

The *program-name* will be used exactly as specified in terms of upper and lower-case letters, and must match the case used in the name of the external subroutine.

Mode 1: In this mode, the CALL statement transfers control to the specified program, but no arguments are passed to or from the called program.

Mode 2: This mode transfers control and passes arguments to and from the specified program.

Sometimes, arguments are passed as values only ("call by value"). That means that the arguments are evaluated and their values are passed to the subroutine, but nothing is passed back to the calling program.

If the subroutine must return a value to the BASIC program, the value must be passed as an argument of the ADDROF or MATADDROF function. These functions cause the memory address of the variable to be passed instead of the value itself ("call by reference").

The CALL statement recognizes the following types of arguments:

Argument Type	Example
Integer	QTY% or I%+2 or ADDROF(QTY%)
Numeric	NBR or NBR+3 or ADDROF(NBR)
String	NAME$ or A$&"AB" or ADDROF(A$)
Integer Array	MATADDROF(QTY%)
Numeric Array	MATADDROF(NBR)
String Array	MATADDROF(A$)

Subscripts may contain arbitrary expressions such as A.MAT(L+3,J-1). If a variable such as "BAL" is the name of both an array and a simple variable, ADDROF(BAL) will pass the address of the simple variable and MATADDROF(BAL) will pass the address of the array.

After an argument is evaluated, its value is placed in the argument list followed by an integer indicating its data type. When the ADDROF(*variable*) is specified, the type code used is that of the *variable* plus 8000H. When MATADDROF is used, C000H is added to the value.

The specific type codes are as follows:

Parameter Type	Type Code
Integer	1
Float	2
String	4
Short Binary Float	5
Long Binary Float	6

Caution The number and type of arguments used in the CALL statement must exactly match the number and type of arguments the subroutine is expecting. If they do not, the results are unpredictable. (The compiler detects a syntax error if an improper call is made to one of the C routines built into the CET product.)

Most C compilers expect parameters in the reverse of the order in which they are actually sent. (See the following example.)

Example	Explanation
CALL MyCsub(addrof(X%),S$)	Calls the C subroutine MyCsub (see below), giving it the memory location of X% and a string parameter. MyCsub will return the length of the string in X% as control is transferred back to the BASIC program..

```
MyCsub(t2, str, t1, xptr)
int t1, t2, *xptr;
char *str;
{
     if (t1 == 0x8001)
     if (t2 == 4)
     if (str) *xptr = *str & 0xff;
     else  *xptr = 0;
     else *xptr = 0; /* error */
   else     ; /* error */
}
```

Points to string itself, not S$

Returns length of string if parameters are correctly specified.

Incorrect Examples	Explanation
CALL MyCsub(X%, S$)	ADDROF(X%) needed.
CALL MyCsub(ADDROF(X%))	Wrong number of parameters.
CALL Another(ADDROF(X+1))	Can't take ADDROF (expression).
CALL Another(1+ADDROF(X))	Can't use ADDROF as part of an expression.

See Also GOSUB, ADDROF and MATADDROF functions

For information on writing C functions refer to the *C Language Interface Manual*.

CASE

The CASE statement is used to conditionally execute statements in a SELECT-CASE-CEND structure.

Syntax

1 CASE *relational-expression*

2 CASE *expression*

Description The form and function of the CASE statement depends upon the format of the SELECT statement.

Mode 1: When the SELECT statement does not specify an *expression*, the CASE statement must define the complete *relational-expression* needed for the true/false test. The statements following the CASE statement will execute only when the result of the test is true (non-zero).

Example	**Explanation**
SELECT	
PRINT CONTROL$	Statements before the first CASE won't be executed.
CASE CONTROL$=""	If CONTROL$ is null, execute following. If not null, go to next CASE statement.
PRINT CONTROL$	Only executed when case is true.
GOSUB NULL.ROUT	" " " " " "
CASE CONTROL$="HELP"	If CONTROL$ is "HELP", execute following.
GOSUB HELP.ROUT	Only executed when case is true.
QUIT	" " " " " "
OTHERWISE	Execute if no previous CASE was true.
PRINT "Invalid input"	
CEND	End of SELECT structure.

Mode 2: This form is used when the SELECT statement includes the first part of the *relational-expression.* If the *expression* in the CASE statement is equal to the one specified by SELECT, then the statements following the CASE are executed.

Example	**Explanation**
INPUT CONTROL$	
SELECT CONTROL$	
CASE ""	if CONTROL$ is null execute following else go to next CASE statement.
PRINT CONTROL$	Only executed when CONTROL$ is empty.
GOSUB NULL.ROUT	" " " " " " " "
CASE "HELP"	If control is "HELP" execute following else skip to next CASE statement.
GOSUB HELP.ROUT	Only executed when CONTROL$="HELP".
QUIT	" " " " " " " "
OTHERWISE	Execute if no previous CASE is true.
PRINT "Invalid input";	

CEND	End of SELECT structure.

Notes Both the CASE and SELECT expressions must be of the same type (string, integer, numeric) in order for the structure to operate correctly.

In either mode, if the result of a CASE statement is false (zero), the statements following CASE are bypassed until another CASE, OTHERWISE or CEND is encountered. When all the CASE statements have been evaluated as false, the statements following the OTHERWISE, if any, are performed. Afterwards, control is transferred to the CEND statement.

When the result of a CASE statement is true, the statements following the CASE are executed, in order, until another CASE, OTHERWISE or CEND occurs at the same nesting level. Execution will resume with the CEND statement.

Restrictions This statement is part of a programming control structure. It is suggested that the program enter at the top of the structure with SELECT and exit at the bottom with CEND. While a GOTO statement is allowed, a GOSUB is recommended. The program should not branch out of the structure via a RETURN or RESUME statement.

See Also CEND, OTHERWISE, SELECT

CEND

The CEND statement marks the logical and physical end of the SELECT-CASE-CEND program structure.

Syntax CEND

Description Every SELECT must have a corresponding CEND statement. After the CASE or OTHERWISE statements have been conditionally executed, control is transferred to CEND. Normal program execution is then resumed.

Examples	**Explanation**
SELECT OPTION$	Using variable OPTION$
CASE "HELP"	
GOSUB DISPLAY.HELP	Perform if OPTION$="HELP"
CASE "INIT"	
SELECT	Perform if OPTION$="INIT"
CASE PROG%=1	
GOSUB PROG1	Perform if OPTION$="INIT" and PROG% = 1
CASE PROG%=2 OR PROG%=3	
GOSUB PROG2.3	Perform if OPTION$="INIT" and PROG% = 2 or 3
OTHERWISE	
GOSUB PROG4	Perform if OPTION$="INIT" and PROG% <> 1, 2, or 3
CEND	
GOSUB INIT.VAR	Perform if OPTION$="INIT"
GOSUB INIT.FILE	" " " " " "
CASE "PRINT"	
DEVICE.NUM% = 16	Perform if OPTION$="PRINT"
CASE "TYPE"	
DEVICE.NUM% = 15	Perform if OPTION$="TYPE"
CEND	
RETURN	Perform always.

Restrictions The program should enter at the top of the structure with SELECT and exit at the bottom with CEND. Do not branch out of the structure via a RETURN or RESUME statement. While a GOTO is allowed, the use of GOSUB statements to perform conditional operations is recommended.

See Also CASE, OTHERWISE, SELECT

CHAIN

The CHAIN statement transfers execution from one BASIC program to another.

Syntax CHAIN *program-name-expr*

program-name-expr = any string-expression that evaluates to a valid program name

Description The CHAIN statement terminates execution of the current program, closes all opened files, discards all variables not declared as common, loads the program specified by the *program-name-expr* and continues execution at the beginning of the called program.

CHAIN operates similar to other program execution statements. The following chart indicates the important differences between the statements. Note that CHAIN does not close files while RUN closes files and discards any variables that were used.

	CHAIN	LINK	RUN	CSI	QUIT	END
Close all files	X		X		X	X
Clear all COMMON data			X		X	X
Clear all non-COMMON data	X	X	X		X	X
Clear current ON ERROR trap	X	X	X		X	X
Clear all ON KEY traps	X	X	X		X	X
Exit open program structures	X	X	X		X	X
Execute another program	X	X	X	X	X	
Return to calling program				X		
Reset all OPTIONs			X	X	X	X

All active programming control structures in the program issuing the CHAIN will be terminated: FOR-NEXT, WHILE-WEND, IF-THEN-ELSE, IF-IFEND, SELECT-CASE-CEND, DEF-FNEND, and GOSUB-RETURN. The CHAIN statement also disables any active ON ERROR, ON LOCK, ON KEY and ON INTERRUPT statements.

Note To transfer control to a program at a point other than the top, define a control variable as COMMON and then test it with an ON GOTO statement at the start of the target program.

Also note that the *string-expression* must evaluate to a valid program name for the operating system in use. A search is made for the program in the current directory, and then along the path list specified by the B_LKPATH environment variable, if any. When the specified program cannot be found, a trappable error will occur.

Limitations Only use CHAIN to execute another BASIC program. QUIT should be used to transfer to a non-BASIC program.

Examples	Explanation
CHAIN "SEGM01"	Program named 'SEGM01' will be loaded, all files will be closed, & control will pass to the first statement of 'SEGM01'.
CHAIN "SEGM0" & STR$(I)	When STR$(I) equals 1, this statement is the same as the example above. If it's equal to 3, program 'SEGM03' will be executed, and so on.

See Also CLOSE, COMMON, LINK, RUN

CLEAR

The CLEAR statement clears the contents of variables and arrays.

Syntax CLEAR [*variable-list*]

variable-list = variable-name [*, variable-list*]

Description The CLEAR statement releases the data storage used for arrays and reinitializes simple numeric variables to zero and string variables to null. Variables defined as common are also removed from common. Execution of this statement makes it possible to redimension the cleared arrays using a DIM or COMMON statement. (DIM and COMMON are dynamically executed at runtime.)

If a list of variable names is given, the names must be simple variables. Both the array and the simple variable with the specified name will be cleared. Because of the nature of CET BASIC as a compiled language, both simple and array variables will be created, if necessary, and cleared. Therefore, the practice of using CLEAR to set simple variables to null or 0 is not recommended.

If a variable list is not specified, all the arrays that have not been declared by a COMMON statement will be cleared.

Restriction CLEAR may not be used to clear the individual elements of an array.

Examples	**Explanation**
CLEAR A, B%, C$	Sets A and B% to 0 and C$ to null, and releases storage for A(), B%(), C$() arrays, so they can be reDIMensioned. If any of these variables have been declared as COMMON, they will also be removed.
CLEAR	Releases storage space for all arrays first created implicitly, by use in an expression, or explicitly by a DIM statement , but NOT those that have been declared COMMON.
Incorrect Example	**Explanation**
CLEAR A(3), B%(), “ABC”	Must be simple variable names.

See Also COMMON, DIM

CLOSE

The CLOSE statement closes an I/O channel. Once closed, both the I/O channel and the associated physical device or file may be reopened on another channel.

Syntax CLOSE #*channel*

channel = integer-expression

Description CLOSE causes any file operation in progress to be completed, unlocks any locked records in the file and releases the I/O channel so that the program may use it to open another file or device.

Execution of a CHAIN, END, QUIT or RUN statement automatically closes all open files. Performing a CSI operation does not close files unless the environment variable B_CSHFCLOSE is set.

Limitations The *channel* must have been specified with an OPEN statement.

Examples	**Explanation**
CLOSE #1	File opened on channel #1 is closed.
CLOSE #INPUT%	File opened on channel corresponding to value of INPUT% is closed.

Incorrect Examples	**Explanation**
CLOSE "IVY.MASTER:A"	File names are not allowed.
CLOSE #4,#5,#6,#10	Multiple channels not allowed.
CLOSE #99	Channel number is too large.

See Also CHAIN, END, OPEN, QUIT, RUN

COLOR

The COLOR statement defines the colors for normal and reverse video. It will have no effect on devices that do not support color.

Syntax COLOR *fg*, [*bg*, [*rvfg, rvbg*]]

fg = *foreground color for normal text*
bg = *background color for normal text*
rvfg = *foreground color for text in reverse video*
rvbg = *background color for text in reverse video*

Description The COLOR statement recognizes the integers from 0 to 7 and displays:

COLOR	CODE	COLOR	CODE
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

Notes The COLOR statement provides eight primary colors. If the CRT$ function is also used to turn on underline or half-intensity, any text will automatically be displayed in another color. (Up to 16 colors are available using the CRT$ function covered in the chapter on *Color and Line Drawing Features*.)

To change the background color of the entire screen, a clear screen (e.g. PRINT CRT$("clear")) must be executed after the COLOR statement.

If the background color is changed, the new color will automatically be used in any subsequent chained or linked program. Although another COLOR statement is performed in a program started via a CSI/CSH statement, the screen will revert to black momentarily when the CSI/CSH is executed. This *flash* does not occur when a program is called with CHAIN or LINK.

Examples	**Explanation**
COLOR 7,4 PRINT AT$(1,22); ERR$ COLOR 7,0	Defines white on red for normal text. Only the message appears in the new colors. Resets color to white on black.
COLOR 7,1 PRINT CRT$("CLEAR") PRINT "Hello" CHAIN "newprog"	Defines white on blue for normal text. Clears screen using blue as the background color. The text will appear in white letters. NEWPROG will use white on blue.

See Also CRT$ function, B_DFLTFGC and B_DFLTBGC environment variables.

COMMON

The COMMON statement defines the variables that may be shared between chained or linked program modules. This statement may also be used to dimension common arrays.

Syntax COMMON *variable-list*

variable-list = *variable-name* [, *variable-list*]
= *array* [, *variable-list*]
variable-name = *integer-variable-name*
= *numeric-variable-name*
= *string-variable-name*
array = *array-name* (*rows* [, *cols*])

Description The COMMON statement is *executable*. When the program is executed, the variables specified in the *variable-list* are added to the current list of common variables. If an *array* is specified, but memory has not been allocated for it with a DIM statement, then COMMON will also allocate storage for the array with the specified dimensions and set all the elements to zero or null. Otherwise, the existing array is merely added to common.

It is not necessary to specify the variables in a COMMON statement in any particular sequence. Common variables are accessed by name, not by location or sequence.

When a CHAIN or LINK statement is executed, the specified program is scanned for any COMMON statements. All variables which are declared as common in both programs will retain the value they had in the program which executed the CHAIN or LINK. All other variables will be cleared, even those declared as common in one of the programs.

The dimensions of a common array are retained when chaining or linking to another program. Redimensioning common arrays will have no effect.

Common variables may be removed using the CLEAR statement and reused as ordinary variables or added to the list again with another COMMON statement.

Notes Since COMMON (and DIM) statements are executable, the common list for the current program has no variables in it until a COMMON is executed. COMMON statements may appear anywhere in the program as long as they are executed at least once prior to performing a CHAIN or LINK, at which time the common list is passed along with the values for the variables and arrays.

Notice that the simple variable STORE is never declared as common in the following examples.

Examples	Explanation
COMMON A,B,A%	The simple variables A, B, and A% are defined for use in another program.
COMMON A1%,ARRAY$(5,22)	Similar to above but also dimensions ARRAY$.
DIM STORE(20,30)	Creates the STORE array but does not put it in COMMON.
COMMON STORE(5, 10)	Places the STORE array in COMMON, but does not redimension it (i. e., it is still a 20 by 30 array) or alter its values.
CLEAR STORE	Releases storage for the STORE array and removes it from COMMON. Sets the simple variable STORE to 0.
COMMON STORE(5, 10)	Creates the STORE array as a 5 by 10 array, zeroes all of its elements, and places it in COMMON.

See Also CHAIN, CLEAR, DIM, LINK, OPTION, RUN

CSI / CSH

The CSI statement is used to execute a command and return to the calling program after the operation is completed. CSH is a synonym to CSI. For brevity, only the CSI statement is referred to in this section.

Syntax CSI *command-line*

Description The CSI statement may be used to execute any valid command for the operating system in use. Executing a CSI statement suspends the operation of the current BASIC program and starts up the new process. Upon return, the program is restored to its state immediately prior to executing the statement.

All arguments and switches required by the command are recognized. Any unquoted or single-quoted arguments will be folded to upper case before executing the command. Double-quote arguments to maintain the case mode.

CSI does not automatically close open I/O channels. To close files, set the B_CSHFCLOSE environment variable. (A DOS program may perform a call to the Bputenv function to set this variable at runtime.)

Note Use the CSH function if you want to reference the return code of the command in the calling program.

CET provides a set C functions to perform file-related operations. We recommend that you use these to eliminate have to leave the BASIC environment to perform a similar operation with the CSI statement.

Restrictions Because the calling program does not quit, the two processes are active at the same time. To free up enough memory for the command to operate under DOS, set the variable B_CSHSWAP so that most of the CET BASIC runtime system and all of your program are *swapped out* during the execution of the CSI statement. If extended memory is available, swapping is directed to it. Otherwise, BASIC will look for the presence of expanded memory.

Normally, performing a CSI does not cause a problem, but it is best to avoid leaving the BASIC environment, if possible.

Examples	Explanation
CSI "Bcreate work.fil clear"	CET utility is invoked to clear the file under DOS.
CSI "Bcreate ./work.file clear"	The equivalent UNIX command.
CSI "lpr " & FILE.NAME$	The UNIX print spooler is invoked to print the contents of the file called FILE.NAME$.

See Also B_CSHFCLOSE and B_CSHSWAP environment variables, Built-in functions (see the *CET BASIC User's Guide*), CSI.RETURN.CODE, CSH function

CSI.RETURN.CODE

CSI.RETURN.CODE passes back a return code from the command that was executed by the last CSI/CSH statement.

This statement has been implemented for THEOS compatibility purposes only. If you need to execute a command from within a BASIC program, we recommend using the CSH function. This function is designed to execute a command like the CSI statement does except that, after execution, the return code from the command becomes the value of the CSH function.

Syntax CSI.RETURN.CODE

Description The return code from the last command executed may be used to determine the success or failure of the operation. A return code of zero normally indicates that the program executed successfully. A non-zero return code indicates an error condition.

Examples	**Explanation**
CSH "create work.fil clear"	CET CREATE utility is invoked to clear the DOS file.
SELECT CSI.RETURN.CODE	Start the SELECT-CEND structure.
CASE 0	Operation successful condition.
do something…	
OTHERWISE	Operation was not successful.
do something else…	
CEND	End of structure.

See Also CSH/CSI, CSH function

DATA

The DATA statement defines the static information to be read using a READ statement.

Syntax DATA *data-list*

data-list = *data-element* [,*data-list*]
data-element = *numeric-constant*
= *quoted-string-constant*
= *unquoted-string-constant*

Description The DATA statement defines an area in memory for static data storage and places the elements in the *data-list* into this area. The elements in multiple DATA statements are used sequentially, in the same order as they appear in the program. The RESTORE statement may be used if it is necessary to alter this sequence.

The type of data in the *data-list* does not need to be the same as the corresponding variable being loaded by the READ statement. CET BASIC will attempt to do a data conversion. Character strings which are read as numerics will be scanned starting from the left and converted until a non-numeric character is encountered. Numeric elements read as string variables will be converted using the STR$ function.

When a data element contains leading or trailing spaces, embedded commas, or a leading digit, decimal point, or sign, the element must be defined as a quoted string constant.

Note Refer to the MAT READ and READ statements for details on how the data may be used to define the initial contents of arrays and simple variables.

Restrictions When a DATA statement is part of a multi-statement line, it must be specified last.

Examples	**Explanation**
DATA 1.23, 2.34, 3RD LITERAL, ANOTHER LITERAL	
DATA " Leading space, trailing space, and commas "	
DATA 11.11, 1, 0, 1, 1, 5, 1, 16, 1, -1	
READ A, B%, C, D$	Sets A=1.23, B%=2, C=3, and D="ANOTHER LITERAL"
READ A$, B$	Sets A$=" Leading space, trailing space, and commas " and sets B$ ="11.11".

See Also MAT READ, READ, RESTORE

DEF

The DEF statement is used to define a sequence of statements as a user-defined function. This feature is handy when the same operation needs to be performed in a number of different places in the program.

Syntax

1 DEF FN*variable* = *expression*

2 DEF FN*variable* (*argument -list*) = expression

3 DEF FN*variable*

4 DEF FN*variable* (*argument-list*)

variable = *simple-variable-name*
expression = *any valid expression*
argument-list = *simple-variable-name* [*, arguments*]

Description The DEF statement has two basic formats. Modes 1 and 2 are single-line functions where the entire statement must fit on one line. Modes 3 and 4 define multi-line functions that use an FNEND statement to mark the end of the program structure.

Mode 1: Defines a single-line function with no arguments. The *variable* and *expression* must agree in type (both strings or numerics). The *expression* is evaluated and returned as the value of the function. Any *variable* referenced in the function will have a value defined by the main program.

Mode 2: Defines a single-line function with one or more arguments. As in mode 1, the value of the *expression* is returned as the value of the function. However, the variables in the function may either refer to variables in the main program or to variables in the *argument-list*.

Variables in the *argument-list* are considered local variables that exist only within the specific function. Upon execution, local variables are replaced with the value of the corresponding argument expression. These values may be reassigned within the function, but any new value will be lost when the function ends.

The program cannot reference or alter local variables since they are completely independent of any variable that is used outside the function definition even if it has the same name.

If the *expression* uses a variable that is not on the *argument-list,* it is considered a global variable whose value is determined outside the function.

Mode 3: Defines a multi-line, user-defined function without arguments. The statements following the DEF are executed until an FNEND is encountered.

The value of the function is then returned, and execution resumes at the point where the function was referenced.

As in mode 1, the *variables* referenced by the function have values defined by the main program.

Mode 4: Defines a multi-line version of mode 2.

In both modes 2 and 4, the value of the function is assigned by a LET statement preceding the FNEND: LET FN*variable* = *expression*. The use of the LET keyword is optional.

There may be more than one assignment statement in a definition, but the last one executed will be the only one that is returned. If a multi-line function definition does not assign a value to the function, its value will be zero or null. No error is detected.

Notes

The DEF statement may be placed anywhere in the program. The statements within a DEF-FNEND structure will be executed when the function is called, not when the function is defined.

Function names must start with FN. They may consist of one or more letters, digits, periods, or $ characters such as in FNCALC, FN.ORD% and FNchg2$.

An expression may include any valid element. A single-line function should not reference itself as this will cause the program to go into an infinite loop. If the FN*variable* appears in an *expression* in a multi-line function, it is assumed to be a recursive call to the function rather than a reference to the function's current value.

Local variables in the *argument-list* must be simple variable names. Array references are invalid, although references to array elements may be used as arguments when calling the function.

Restrictions

If a function is used before it is defined, no automatic type conversion is performed. The arguments must have exactly the same type as the arguments in the function definition.

DEF functions cannot be redefined.

Function definitions cannot be nested. There can be only one FNEND statement for each function.

Most statements can be used within the function definition (between the DEF and FNEND statements). However, transfers into or out of the definition via GOTO or GOSUB statements are not recommended as the values of the arguments may be changed with unpredictable results.

The CET BASIC Runtime System will check to insure that any GOSUB executed within a function is followed by a RETURN before FNEND is encountered. Conversely every RETURN within the function must match a GOSUB which was executed in the same function call.

Examples	**Explanation**

DEF FNX%(A, B)
IF B <= 0
  FNX% = A
ELSE
  FNX%=2*FNX%(A+1,B–1)-1
IFEND
FNEND
	A and B are arguments.
	
	Recursive call to function.

PRINT FNX%(2.6, 1.5)	Prints 13.
DEF FNA(X) =SQR(X^2+Y^2)-SQR(X)	X is the local variable. The value of Y is taken from the global variable Y.

Incorrect Examples	**Explanation**
DEF FNA(S^2) = 2 * S+S	Argument must be a simple variable.
DEF FNA$(B) = 2 * B	Function name must match expression in type (string or numeric).
A = 2 + FNB(3) DEF FNB(X) = X^2.5	The call FNB(3) passes the integer 3 as CET BASIC doesn't know that FNB expects an argument. This error is detected at compile time. It can also be avoided in this instance by making the function call FNB(3.0)
DEF FNA(X, Y) DEF FNB(Z) = Z^2 FNA = X - Y FNEND	Can't nest function definitions.
DEF FNCT(A) FNCT = FNCT + 1	FNCT on right-hand side must be a function call, and, in this case, have parameters.

See Also FNEND, LET

DELETE

The DELETE statement removes a record from an open disk file.

Syntax DELETE #*channel* , *key*

channel = *integer-expression*
key = *expression*

Description The DELETE statement removes the record with the specified key from a disk file that was opened for UPDATE on the specified I/O channel.

To delete a record from an indexed file, the *key* must be a string expression. For direct files, a numeric *key* is required.

No error is detected if the record to be deleted does not exist.

Examples	**Explanation**
OPEN #1: FILE$, UPDATE INDEXED	
OPEN #2: FILE1$, UPDATE DIRECT	
DELETE #1, "0001"	Record with key "0001" is deleted.
DELETE #2, 30	Record number 30 is deleted.

See Also OPEN

DIM

The DIM statement dynamically defines and allocates memory for string and numeric array variables.

Syntax DIM *array-variable-list*

array-variable-list = *array-variable* [, *array-variable-list*]
array-variable = *array-name* (*row* [, *col*])
array-name = *simple-variable-name*
size = *integer-expression*

Description The DIM statement is *executable*. When executed, it declares and allocates storage space for the arrays on the *array-variable-list*. The elements of the array to zero or null.

The use of a subscripted variable prior to defining it with a DIM statement will cause the variable to be implicitly dimensioned as an array with the default size of 10 if one subscript is used. When two subscripts are used, the array is dimensioned to 10,10.

Notes If an array has already been dimensioned in either the current program or the calling program (and passed in common), it will not be redimensioned to a different size. The CLEAR statement must be executed in order to change the dimensions of an array.

Because the DIM statement is executable, the following code will generate a subscript error.

```
I% =15
DIM A (5)
if I% < 10 and A (I%)+5 THEN GOTO 100
```

Some BASIC Languages may execute the GOTO statement, even though the reference A(I%) is invalid. That may happen because the AND and OR operators stop when sufficient information is acquired to determine the truth value of the expression. CET BASIC does not support these *short-circuit* relational operators.

Numeric or string array variables may have one or two dimensions. Arrays are considered one-dimensional if only one subscript is used to define the number of elements. For a two-dimensional array, one subscript is used to indicate the number of rows in the array and another to indicate the number of columns.

The lower bounds of subscripts are 0 in each dimension unless OPTION BASE 1 is in effect. For instance, if the array X is dimensioned as X(5), there will be six elements with subscripts of 0, 1, 2, 3, 4, and 5. (Refer to the *Elements of the BASIC Language* chapter for more information on arrays.)

The maximum value for each dimension depends upon the platform in use. For instance, under DOS, the maximum is about 10,000 since this is a 16-bit operating system with limited resources.

Any reference to an array with subscripts outside the allocated size will cause a "Subscript out of range" error which is trappable.

CET BASIC allows the same variable name to be used to define both a simple and an array variable. Although this feature is supported, it is not recommended as it affects the readability of a program.

Examples	**Explanation**
DIM X(20),A$(5,5),Z(N%)	Array X has 21 elements, string array A$ has 36 elements, and array Z has N%+1 elements.
DIM X(15)	Does not redimension X because X has not been CLEARed.
CLEAR X	Deallocates memory allocated to X.
DIM X(15)	Resizes X and initializes all of its elements to zero.

Incorrect Examples	**Explanation**
DIM X(2,2,2)	Can have only 2 dimensions.
DIM Y(99999)	Maximum dimension is 32767.

See Also CLEAR, COMMON, OPTION

ELSE

ELSE defines the statements to be executed when the result of a multi-line IF statement is false.

Syntax

1 ELSE

2 ELSE *statement*

3 ELSE *line-number*

statement = *any* BASIC *statement*
line number = *any line number in the program*

Description **Mode 1**: When on a line by itself, ELSE indicates the start of the statements to be executed if the result of the IF condition is false. The end of the structure is marked with an IFEND statement.

Mode 2: This mode is most commonly used when a single statement is to be executed when the IF statement is false.

Mode 3: This is a special case of mode 2 where ELSE *line-number* is equivalent to the statement ELSE GOTO *line-number*.

Examples	**Explanation**
IF LNO	Test LNO for non zero
GOSUB NON.ZERO	Perform if LNO not equals 0
ELSE	
PRINT USING "###",PNO	Perform if LNO = 0
GOSUB PRINT.HEADER	" " " "
IFEND	End of conditional execution.
IF VAL > CONTROL	Test for VAL > CONTROL
IF VAL > LIMIT	Perform if VAL > CONTROL
GOSUB ERROR	If val > control and val > limit gosub error
ELSE IF ERR.NUM < ERR.LIMIT THEN QUIT	QUIT if val > control and val <= limit and err.num < err.limit
IFEND	End of multi-line IF statement.
ELSE CALL PROCESS	Perform if VAL <= CONTROL
IFEND	End IF VAL > CONTROL

Incorrect Examples	**Explanation**
IF CONTROL<LIMIT THEN WAIT ELSE PRINT "ERROR"	Not in multi-line IF statement.
IF ERROR < EPS GOTO DONE ELSE REPROCESS	ELSE *label* is not allowed. Use ELSE GOTO *label*.

See Also IF, IFEND, THEN

END

The END statement is used to terminate the execution of a program.

Syntax END

Description The END statement terminates the execution of a program and closes all open files. Control is transferred back to the calling environment.

We recommend that END be used to mark the physical end of a program. The QUIT statement should be used if execution should be terminated before then.

See Also CLOSE, STOP, QUIT

FNEND

The FNEND statement is used to mark the end of a multi-line, user-defined function.

Syntax FNEND

Description FNEND is used to terminate a user-defined function that consists of more than one line. After exiting, control is transferred back to the statement which called the function.

The statements between and including the DEF and FNEND statements are part of the function definition and are not executed unless the function is called.

Restrictions A multi-line function may have only one FNEND statement.

Example	**Explanation**
DEF FNTEST(A) FNTEST = PI*A*A FNEND	Start of function definition. End of function definition.

DEF FNCENTER$(strg$,LNG%)
  IF LNG% = 0
  FNCENTER$ = ""
  ELSE
  IF LEN(STRG$)=0
  FNCENTER$=SPACE$(LNG%)
  ELSE
  FILL% = LNG-LEN(STRG$)
  C$ = SPACE$(FILL%/2) & STRG$ & SPACE(FILL%/2)
  IF MOD(FILL%,2) = 0 THEN C$ = C$ & " "
  FNCENTER$ = C$
  IFEND
  IFEND
FNEND
	Start of function definition that calculates the ‘center’ of the string.
	
	End of function definition.

See Also DEF

FOR

The FOR statement defines the starting point and conditions of execution for a FOR-NEXT program loop.

Syntax

1 FOR *num-index* = *start* TO *finish* [STEP *increment*]

2 FOR *index* = *expression-list*

num-index = *simple-numeric-variable*
start / finish = *numeric-expression*
limit = *numeric-expression*
increment = *numeric-expression*
index = *simple-variable*
expression-list = *expression* [, *expression-list*]

Description

Mode 1: The program loop is executed with a numeric or integer index variable (*num-index*) set to an initial value (*start*). Using an *increment* of 1, the variable takes on a sequence of values as the statements between FOR and the corresponding NEXT statement are executed.

After each iteration of the loop, the *num-index* is tested against the *finish* value. Execution is terminated when this value is exceeded and *num-index* = *finish* +1, the first value that exceeded the limit.

This mode is the standard form of the statement. If the STEP clause is not specified, it is equivalent to STEP 1. The *increment* may be a positive or negative value. The looping construct is as follows:

```
FOR var = start TO finish STEP step
        ...statements...
    NEXT var
```

The above statements are equivalent to:

```
var = start
for.start:
    ...statements...
    var = var + step
IF var is not 'beyond' finish THEN GOTO for.start
```

The meaning of *beyond* depends on the sign of STEP. If the value is positive, then the loop terminates when *var* is greater than *finish.* If negative, the loop ends when STEP is less than *finish.*

Mode 2: This is a special, non-standard form of the FOR statement in which the successive values of the *index* are listed explicitly in the statement itself.

In this mode, the index variable may also be a string. Before the loop is executed, all expressions in the *expression-list* are evaluated to confirm that they all have the same type as the index variable, either all numeric or all string.

The NEXT statement assigns the elements in the *expression-list* to the index variable for each iteration of the loop. When the last element of the *expression-list* is used, the loop is executed with that *index* value. Normal program execution resumes after the NEXT statement.

This mode is not recommended if you plan on recompiling with other BASIC language because it may translate the final index value for *beyond* finish differently.

Notes

Expressions for *start*, *finish*, and *increment* are computed only once, before the loop begins. If *increment* is a constant, then the loop termination condition is optimized at compile time.

FOR-NEXT loops may be nested to any depth. When two structures overlap, one must be completely contained inside the other.

You may exit a FOR-NEXT loop with a GOTO or BREAK statement. This will leave the loop *open* so that it can be reentered up until another FOR-NEXT loop is executed with the same index variable name.

Using GOTO or BREAK to jump into the middle of a FOR-NEXT structure which is not open is an undetected error and can cause unpredictable results.

Examples	**Explanation**
FOR I=1 TO 10 NEXT I	Loop will execute 10 times unless exited with a GOTO or BREAK.
FOR I%=C+3 TO R*2 STEP .2	Initial value is 3 plus the current value of C. Limiting value is current value of R times 2. If limit is less than initial value, the NEXT I% loop is not executed and control will pass to the line after NEXT I%. If variables C or R are changed within the loop, initial value and limiting value will not be affected as they are evaluated only once.

FOR I% . . .
  FOR J%
  .
  .
  NEXT J%
  FOR J%
  .
	This illustrates a correct form of nesting.

```
        .
    NEXT J%
    NEXT I%
```

```
FOR INDEX$="A","B","ABCD"&SPACE$(2)    Loop will execute 3 times.
NEXT INDEX$                            Variable name in NEXT must match
                                       FOR index variable, if used.
```

```
FOR I%= 1,2,3,6,8,22,99          Will execute 7 times.
  FOR A$="A","B",X$              Will execute 3 times for each of the seven outer
                                 loops.
    NEXT A$                      Terminate inner loop.
  NEXT I%                        Terminate outer loop.
```

```
FOR J = 5 TO 1 STEP 2            Loop will not execute at all.
```

```
FOR K = 1 TO 4 STEP –5           “  “     “     “     “
```

Incorrect Examples **Explanation**

```
FOR 1 TO 50 STEP 2               Index variable missing.
```

```
FOR X = 1 STEP –3                Limiting expression is missing.
```

```
FOR I . . .                      Illegal nesting.
    FOR J
        .
        .
    NEXT I                       Must be NEXT J.
NEXT J
```

```
FOR I . . .                      Illegal nesting.
    FOR I . . .                  The compiler will not catch the error here.
    NEXT I
NEXT I
```

```
FOR I . . .                      Illegal nesting.
    IF . . .
        .
        .
        NEXT I                   FOR-NEXT pairs are determined at compile time not
    ELSE                         dynamically at run time.
        .
        .
        NEXT I
    IFEND
```

See Also NEXT, WHILE, BREAK

GET

The GET statement is used to read a raw stream of bytes or characters from an open input channel or device. Any special formatting of the data is ignored.

GET is the only means of reading unconverted characters from the console or from non-text files that were not created in a CET BASIC format.

Syntax

1 GET *variable-list*

2 GET *#channel: variable-list*

channel = *numeric-expression*
variable-list = *variable* [*, variable-list*]

Description

Mode 1: This mode is used to read unformatted bytes of data from the console, one character at a time. The characters are not echoed, and the usual BASIC conversion of special characters is not performed.

Each character in the stream file is evaluated and assigned, in order, to the variables on the *variable-list*. If the variable is numeric, it receives an 8-bit integer. Only one character is assigned to string variables.

The mode is equivalent to GET #0: *variable-list*.

Mode 2: The GET statement gets one or more characters from the specified channel, normally a file opened for INPUT SEQUENTIAL or UPDATE SEQUENTIAL. Otherwise, this mode operates the same as the first one.

Notes

GET does not test to see if the input channel is ready before accepting the input. When the device is not ready, and there is no data, numeric variables are set to zero and string variables are set to null.

In DOS, zero is a legitimate value that may be returned from the keyboard so string variables must be used to distinguish between a legitimate zero value and a no data condition.

Examples	**Explanation**
A%=1	
WHILE A%	Repeat until A%=0.
GET A%	Clear (UNIX) console type ahead buffer.
WEND	
FOR I%=1 TO 8	Repeat for eight characters.
WAIT #0	WAIT for a character at the console.
GET A$	
WORD$[I%:I%]=A$	Save as part of password.

```
    PUT "*"                     Echo asterisk.
NEXT                            I% assumed (not recommended usage).

WAIT #0 \ GET A$, B$, C$        A$ is never ""; B$ = C$ = "" if an ordinary character
                                was entered.
IF A$ = CHR$(27) ...            The ESC character, the lead-in to an ANSI control
                                key sequence. B$ and C$ can be examined to find
                                out which one.
```

See Also CLOSE, INPUT, OPEN, PUT, READ, WAIT

GOSUB

The GOSUB statement calls an unstructured subroutine in the current program. An unstructured subroutine is a sequence of BASIC statements whose execution terminates with a RETURN statement.

This is a handy feature that allows you to group frequently used code in a subroutine, and then invoke the subroutine with a single GOSUB statement.

Syntax GOSUB *line-reference*

line-reference = line-number or line-label

Description The GOSUB statement saves the location of the next statement and jumps to the subroutine specified by the *line-reference*.

When a RETURN is executed, control is passed back to the prior level until the first subroutine is reached. At that point, execution will continue at the statement following the first GOSUB that was performed.

Notes Subroutines may be nested. In other words, a subroutine may contain other GOSUB statements. By default, GOSUBs may be nested up to 32 levels deep before an error is detected. Although GOSUBs nested to a deeper level are normally considered to be a *logic* problem, the environment variable B_GOSUBMAX may be used to increase the limit, if necessary.

An error is detected when either a GOSUB or RETURN is executed without the corresponding statement.

Example	**Explanation**
GOSUB LAB \ A = A+1	Subroutine starting at the line-label “LAB” will be executed. Upon return, A will be incremented
LAB: PRINT A RETURN	Control will be transferred to the statement following the GOSUB that called this subroutine.

Incorrect Example	**Explanation**
DEF FNA(X, Y) . . GOSUB END.LAB . . END.LAB: FNEND	Must RETURN from GOSUB before terminating function. This will be detected as a runtime error.

See Also CALL, ON GOSUB, RETURN

GOTO

The GOTO statement unconditionally transfers control to another location in the current program. The keyword may also be written as GO TO.

Syntax GOTO *line-reference*

line-reference = line-number or line-label

Description When the GOTO statement is executed, control is transferred to another location in the current program. When the *line-reference* is a non-executable statement such as a REM, control will pass to the first executable statement following that line.

Notes A GOTO statement should not be used to jump into the middle of a FOR-NEXT loop because the loop variables will not be initialized properly. Similarly, a GOTO should not be used transfer control to the middle of a subroutine, a multi-line DEF-FNEND function, or IF-IFEND, WHILE-WEND or SELECT-CEND structures. These types of program structures should only be entered at the top, otherwise the results may be unpredictable.

From a structured programming viewpoint, the use of GOTO statements are not recommended. Whenever possible, GOSUB and BREAK statements should be used instead.

Examples	**Explanation**
GOTO 1000	Control is unconditionally transferred to line 1000.
GOTO BEGIN	Control is unconditionally transferred to the line with the label BEGIN.

Incorrect Example	**Explanation**
20 GOTO 20	Infinite loop. This is not detected by BASIC.

See Also GOSUB, ON GOTO, ON ERROR, ON INTERRUPT, ON KEY and ON LOCK

IF

The IF statement conditionally executes one or more statements.

Syntax

1 IF *expression* THEN *statement*(s) [ELSE *statement*(s)]

2 IF *expression* THEN *statement*(s)

3 IF *expression*

Description The IF statement evaluates the *expression* and is considered true when the result is non-zero and false when the result is zero.

Mode 1: This format is referred to as a single-line IF-THEN-ELSE statement since it contains all the operations to be performed whether the result of the IF test is true or false.

When the *expression* is true, the THEN *statement(s)* are executed until a matching ELSE is encountered. If the *expression* is false, control is transferred to the statements in the ELSE clause.

Execution terminates at the end of the IF-THEN-ELSE line and control is transferred to the next line unless a GOTO was performed.

Mode 2: This is a single-line, IF-THEN statement. If the *expression* is true, the *statement(s)* following the THEN keyword are executed. No operation is performed when the result of the test is false.

Mode 3: This format begins a multi-line IF-IFEND statement with the following structure:

```
IF  expression
      THEN  true-statements
      ELSE
            false-statements
IFEND
```

When the *expression* is true, the *statements* following the IF are executed until the matching ELSE or IFEND statement is encountered. Otherwise, the *statements* between the matching ELSE keyword (optional) and the IFEND statement are executed.

Execution will always resume at the statement following the matching IFEND unless a GOTO was executed. In that case, THEN line-number and ELSE *line-number* are equivalent, respectively, to THEN GOTO *line-number* and ELSE GOTO *line-number*.

Notes

IF-THEN-ELSE statements may be nested. When THEN, ELSE or IFEND is encountered during the analysis of an IF statement, it is considered to match the most recently executed IF statement that is still incomplete.

Note that the following syntax is allowed, but the wrong ERL value will be generated if an error occurs on line 40. The ELSE statement must be on a line by itself in order for the ERL function to operate as expected.

```
10  IF    expression
20        THEN OPEN #1: "filename"…
30  ELSE OPEN #2: "filename"…
40  IFEND
```

Examples	**Explanation**
IF A=1 THEN PRINT "A = 1"	When A is equal to 1, the literal 'A = 1' is printed, otherwise control passes to the following line.
if a=1 then print 'OK' else 30	When variable A equals 1, the literal 'OK' printed, otherwise GOTO 30 is executed.
If inp then gosub spec.char	When the value of the function INP is non-zero, the subroutine SPEC.CHAR is executed, otherwise control is passed to the following line.
IF VALUE>0 THEN PRINT "POSITIVE" ELSE PRINT "NEGATIVE" IFEND	If VALUE is greater than zero, THEN "POSITIVE" is displayed ELSE "NEGATIVE" is displayed. The end of the IF-IFEND structure.
IF LINE.NO > LIMIT GOSUB TOP.OF.PAGE PRINT USING "###", PAGE.NO LINE.NO = 0	Perform if LINE.NO > LIMIT " " " " " " " " "
IFEND	End of conditional execution.
IF VAL > CONTROL IF VAL > LIMIT GOSUB ERROR	Test for VAL > CONTROL Perform if VAL > CONTROL Perform if VAL > CONTROL AND VAL > LIMIT GOSUB EXIT
ELSE IF ERR.NUM < ERR.LIMIT THEN QUIT	QUIT if VAL > CONTROL and VAL <= LIMIT and ERR.NUM < ERR.LIMIT
IFEND ELSE CALL PROCESS IFEND	End IF VAL > LIMIT Perform if VAL <= CONTROL End IF VAL > CONTROL

Incorrect Examples	**Explanation**
IF Q=50 ELSE X=10 \ PRINT Y	The 'THEN' keyword is missing.
IF I >1 THEN 200 \ I = I+1	Statement I = I+1 will never be executed. Error undetected by BASIC.

See Also ELSE, IFEND, THEN

IFEND

The IFEND statement is used to mark the end of a multi-line IF-THEN-ELSE structure.

Syntax IFEND

Description IFEND terminates an IF statement and marks the end of the conditionally executed THEN and ELSE statements. If multi-line IF-THEN-ELSE structures are nested, one IFEND is required for each structure.

Notes For examples, see the discussion of the IF statement.

See Also ELSE, IF, THEN

INPUT

The INPUT statement accepts ASCII data from an operator or an open file.

Syntax

1 INPUT *variable-list*

2 INPUT [*prompt* ,] *variable-list*

3 INPUT #*channel*: *variable-list*

4 INPUT #*channel* [, *key*]: *variable-list*

variable-list = *variable* [, *var-list*]
prompt = *string-literal-expression*
channel = *numeric-expression*
key = *integer-expression*

Description

Mode 1: Accepts a line of ASCII data from the console and extracts one or more fields. The program displays the question mark (at the current cursor position) as the default prompt character and waits until data is entered. (The prompt character may be changed with the OPTION PROMPT statement).

As characters are accepted from the keyboard they are echoed back to the console, but no characters are transmitted until a carriage return is entered.

Mode 2: Accepts a line of data in the same manner as in mode 1 except that the specified *prompt* is displayed on the screen immediately before the current prompt character.

Mode 3: Accepts a data record in the same manner as in mode 1 except that no prompting occurs and characters are accepted from the sequential file opened for INPUT or UPDATE on the specified channel. This file must be an ASCII formatted file created with a text editor or with the BASIC PRINT statement. No characters are echoed to the screen during this process.

Mode 4: Accepts a data record in the same manner as in mode 3 except that the open file must be a direct or indexed file.

If an indexed file is opened, the *key* must be a string expression. To access a direct file, the *key* must be a numeric. In either case, the record for the specified *key* must have been written by the BASIC PRINT statement.

Notes

When specifying a prompt in mode 2, only a *string-literal-expression* may be used so that CET BASIC is able to recognize the prompt character(s). Otherwise, the prompt will be interpreted as an input variable. For example, a *string-literal-expression* might be:

```
""&A$ or "Hello, "&NAME$&", how are you".
```

If the input is from the console (modes 1 and 2), the following editing keys may be used to modify the data before it is entered.

Key	Operation
Back Space	Deletes the character to the left of the cursor.
Left Arrow	Deletes the character to the left of the cursor.
CTRL/X	Deletes the entire line.

Input from the console is normally terminated by pressing Return. If the first character entered is a control key such as Esc, input is terminated immediately and the INP function is set to the value of the control character. Otherwise, the INP function is set to zero.

The CRT$ function may be used to determine a particular INP value. For example, the down-arrow will return ASC (CRT$("DOWN")). Refer to the *Character Codes* section in the *Appendix* for the actual codes produced.

When the end of line or record is detected, the data is scanned for field separators. The normal separator is the comma unless OPTION COMMA has been set to use a semicolon.

BASIC will edit the input fields according to the type specified by the corresponding variable on the *variable-list*. If the variable is numeric, the field is terminated by the first non-numeric character (a character other than a digit, period, comma or a sign).

String fields are trimmed of any leading, trailing or extra embedded spaces unless the input field is enclosed in quotation marks. When quoted, the quotes are stripped from the data, but the extra spaces are left as is.

The data fields are read in the order they are input and assigned to the variables in the *variable-list.* If there are more input fields entered than there are variables in the *variable-list*, the extra fields are discarded. When fewer fields are input, the fields are assigned to the respective variables and the remaining variables are set to zero or null depending upon their type.

CET BASIC does not prompt for further input if the input line is insufficient, nor does it display an error if the input line contains too many items.

Under DOS and Windows, the INPUT and MAT INPUT statements will automatically read a carriage-return+linefeed character as the record delimiter.

By default, a carriage-return will be read as the delimiter under UNIX. If the file has been opened with the option NEWLINE (or the variable B_OPENNL

has been set), the linefeed delimiter will be used. For compatibility purposes, option NEWLINE and the variable B_OPENNL are recognized under DOS and Windows, but they will have no effect.

Note that there does exist the potential for program error when using the linefeed as a record delimiter. The problem occurs when the CRT$ ("DOWN") string is included in the record. Because the *token* used to represent the ↓ key is the same as a linefeed character (decimal 10), the key will also be interpreted as a delimiter.

Examples	**Explanation**
INPUT N	A question mark is displayed and the program is suspended until the operator types a return or enters a control key as the first character
OPTION PROMPT "" INPUT "NAME: ", CUST$, A, B	The prompt "NAME: " is displayed and three fields are accepted, one string and two numeric.
OPEN #1: "CONSOLE", INPUT SEQUENTIAL INPUT #1: A$, B, C	Again, three fields are accepted from the operator, one string and two numeric. No prompt will be displayed and no control characters are allowed.
OPEN #2: "DATA\FILE", UPDATE DIRECT INPUT #2, 13: RECORD$	The 13th record in the file is read into the variable RECORD$.

Incorrect Examples	**Explanation**
INPUT A, B, 2.3, D	Only variables may be INPUT.
INPUT	At least one variable must be specified.
INPUT "FLD1", F1, "FLD2", F2	Only one prompt is allowed.
OPEN #1: "./DATA/FILE", INPUT DIRECT INPUT #1, "ABCDE": A$, B$	Must use numeric key for direct access.
OPEN #2: "DATA.FILE2", OUTPUT INDEXED INPUT #2, "ABCDE": A$, B$	Access must be opened INPUT or UPDATE.

See Also CLOSE, LINPUT, MAT INPUT, MAT READ, MAT READNEXT, OPEN, OPTION, READ, READNEXT

LET

The LET statement assigns a value to a simple variable or an array.

Syntax

1 [LET] *variable* = *expression*

2 [LET] *string-variable* [*from:through*] = *string-expression*

3 [LET] *FNvariable* = *expression*

4 [LET] ERR = *numeric expression*

Description LET is the only BASIC statement where the keyword is optional. In all forms of the statement, the right side of the *expression* is evaluated and assigned to the *variable* on the left of the first equal sign. Since the previous contents of the *variable* are replaced after the expression has been evaluated, the *variable* may be an element in the expression as in LET X = X + 1.

Mode 1: This is the standard form of the assignment statement. The type of *expression* must match the type of the *variable* and may be either a string, numeric or an integer. Variables may be either simple or array variables.

Mode 2: The second format of the LET statement provides a powerful method of modifying string variables and array elements using the substring operator (covered in the *Elements of the BASIC Language* chapter).

In this case, the substring operator identifies the portion of the *variable* to be modified. The general format of the statement is *string-variable* [*from* : *through*] and is interpreted as follows:

Replacement: When the *from* position is less than or equal to the *through* position, the characters that are within that range are replaced with those in the *string-expression.* The *string-expression* value will be either padded with spaces on the right or truncated to the same length as the substring + 1.

Deletion: When the value of *from* is greater than *through*, the characters in the *from* position up to, but not including, those in the *through* column position are deleted from the *variable.* The characters in the *string-expression* are then inserted into the *variable* starting with the *through* column position.

Insertion: When *from* is zero, the characters in the *string-expression* are inserted into the *variable* immediately after the *through* position.

Assuming that A$ contains the string ABCDEFGHIJ, then the following examples should make these rules clear.

```
A$[4:6] = "123"        ABC123GHIJ
A$[6:4] = ""           ABCDGHIJ
```

A$[6:4] = "01234"	ABCD01234GHIJ
A$[0:6] = "0123"	ABCDEF0123GHIJ
A$[0:0] = "0123456"	0123456ABCDEFGHIJ

Mode 3: This mode assigns a value to a multi-line, user-defined function. Refer to the section on the DEF statement for information on its use.

Mode 4: This mode of the LET statement not only provides the ability to test the error handling routines, but it also allows the program to define its own trappable errors. When this statement is executed, BASIC acts as though an error with the value of *numeric-expression* has occurred and assigns the value to the ERR function. Control is then transferred to the error handling routine defined by the ON ERROR statement.

If all or some of the statement lines in your program are unnumbered, we recommend that you enter the LET ERR statement on the same (numbered) line as the one that may cause an error. This way, the ERL function can return meaningful information. (See the -# compiler flag description for an alternative method.)

Examples	**Explanation**
LET A = 1.23	The constant 1.23 is assigned to the variable A.
A = 1.23	Same as previous example.
LET A = A + 1	The current value of A is incremented by 1.
LET A$ = "ABCDEF"	The string variable A$ is set to 'ABCDEF'.
LET A$ = A$ & "GHIJ"	The string variable A$ is concatenated with the string 'GHIJ' and afterwards will contain 'ABCDEFGHIJ'.
50 ERR = 30 \ OPEN #1: F$,INPUT SEQUENTIAL	If an ON ERROR statement has been used, then ERR = 30 and ERL = 50 will be returned.

Incorrect Examples	**Explanation**
LET A$ = B + 2	Cannot mix string and numeric expressions.
LET A = B = C	This is syntactically correct but probably does not do what you intend. It computes the value of the expression B = C, -1. If B does equal C, 0 if B does not equal C, and assigns that value to A. The value of B does not change!
LET +1 = A	Cannot assign a value to a constant.

See Also MAT, ON ERROR

LINK

This statement transfers execution from one BASIC program to another. During the process, LINK:

- Clears all the variables that are not declared as COMMON.
- Loads the specified program.
- Continues execution at the top of the called program.

Syntax LINK *program-name-expr*

program-name-expr = string-expression

Description The *program-name-expr* must refer to another BASIC program since only BASIC can handle common variables and open channels. CALL, CSI or QUIT statements may be used to transfer control to a non-BASIC program.

LINK is similar to CHAIN, except that open files are not closed. The following chart illustrates the differences between the various program execution statements.

	CHAIN	LINK	RUN	CSI	QUIT	END
Close all files	X		X		X	X
Clear all COMMON data			X		X	X
Clear all non-COMMON data	X	X	X		X	X
Clear current ON ERROR trap	X	X	X		X	X
Clear all ON KEY traps	X	X	X		X	X
Exit open program structures	X	X	X		X	X
Execute another program	X	X	X	X	X	
Return to calling program				X		
Reset all OPTIONs			X	X	X	X

When a LINK statement is executed, the current directory is searched for the specified program, followed by the path defined by the environment variable B_LKPATH. A trappable error (30) is detected if the file cannot be found.

Example	**Explanation**
LINK "orders"	Link to the program named 'orders'.

See Also CHAIN, CLOSE, COMMON, CSI, QUIT, RUN

LINPUT

The LINPUT statement accepts an entire line or record of ASCII formatted data from an operator, an open file or an input device.

Syntax

1 LINPUT *string-variable*

2 LINPUT *prompt, string-variable*

3 LINPUT *#channel: string-variable*

4 LINPUT *#channel , key : string-variable*

prompt = *string-literal-expression*
channel = *numeric-expression*
key = *expression*

Description The LINPUT statement operates similar to INPUT with the following exceptions:

- Only one variable may be specified for LINPUT.
- Only one string field is accepted. Embedded commas do not terminate the field.
- Leading, trailing or extra embedded spaces are not trimmed. Quotation marks are not removed.

LINPUT also allows any of the following keys to be used to modify the data before it is entered.

Key	Function
Back Space	Deletes character to the left of the cursor.
Left Arrow	Deletes character to the left of the cursor.
CTRL/X	Deletes the entire line.

Input from the console is normally terminated by pressing the Return key. If the first character entered is a control key such as Esc, input is terminated immediately and the INP function is set to the value of the control character. (INP values are listed in the *Character Codes* section in the *Appendix*.)

Notes LINPUT may be used as a line-oriented version of GET to accept ASCII data from serial devices such as an open COM (communications) channel. In that case, a Return must be entered to terminate the line.

LINPUT and INPUT statements may be used to read ASCII data from a file created with a text editor or with the BASIC PRINT statement.

Under DOS and Windows, the LINPUT (USING) and INPUT statements will automatically read a carriage-return+linefeed as the record delimiter.

By default, a carriage-return will be read as the delimiter under UNIX. If the file has been opened with the option NEWLINE (or the environment variable B_OPENNL has been set), the linefeed delimiter will be used to conform to the UNIX convention.

Note that there does exist the potential for program error when using the linefeed as a record delimiter. The problem occurs when the CRT$ (“DOWN”) string is included in the record. Because the *token* used to represent the ↓ key is the same as a linefeed (decimal 10), the key will also be interpreted as a delimiter.

For compatibility purposes, option NEWLINE and the variable B_OPENNL are recognized under DOS and Windows, but they will have no effect.

Binary data created with WRITE or MAT WRITE must be read with a READ, READNEXT, READPREV, MAT READ, MAT READNEXT or MAT READPREV statement.

Examples	**Explanation**
LINPUT A$	The default prompt string is displayed at the current cursor position and execution is suspended until the operator terminates the input with a carriage return or a control key in the first character position. In the former case, the entire line is assigned to A$. In the latter, the ASCII value of the control key is assigned to the INP function.
LINPUT "NAME",A$	The literal "NAME" followed by the current default prompt is displayed. If, for instance, the initial default prompt, "? " is still in effect, the string "NAME? " is displayed. Execution is suspended until the operator terminates input.
OPTION PROMPT "" LINPUT "NAME: ", A$	The literal "NAME: " is displayed and then a line of input is accepted as above.
OPEN #1: "CONSOLE", INPUT SEQUENTIAL LINPUT #1: STRING$	Similar to first example, but no prompt nor INP capabilities. Characters are echoed to the screen since the open file is the console.
OPEN #2: "DATA.FILE", INPUT DIRECT LINPUT #1,5: RECORD$	Record number 5 of file is read into variable RECORD$.
OPEN #2: “COM”, INPUT SEQUENTIAL LINPUT #2: DEV$	A line of ASCII data is read from the serial device on COM 2 and stored in DEV$.

Incorrect Examples	Explanation
LINPUT A1	Must be a string variable.
LINPUT A$, B$, C$	Only one variable is allowed.
LINPUT RPT$(3, ">") & " ", A$	The first string expression will not be recognized as a string literal expression. The compiler can be 'fooled' into handling this properly by using "" & RPT$(...

See Also INPUT, LINPUT USING, MAT INPUT, MAT READ, MAT READNEXT,MAT READPREV, READ, READNEXT, READPREV, INP function.

LINPUT USING

The LINPUT USING statement accepts a string of ASCII data up to the specified length from the console. Editing capabilities are provided.

Syntax

1 LINPUT [*prompt*,] USING *literal-mask, string-variable*

2 LINPUT [*prompt*,] USING *mask, string-variable*

literal-mask = *string-literal-expression*
mask = *string-expression*
string-literal-expression = *string-literal* [*&string-expression*]
prompt = *string-literal-expression*

Description

Typically, a program may set the default prompt to "" with the OPTION PROMPT statement and use a prompt string like "" & AT$(3,5) & "NAME: ". In that case, the initial "" indicates that a *string-literal-expression* follows. If this mode is used, the comma following the *prompt* is required.

Mode 1: Displays the (optional) *prompt* text and the current prompt string defined by the OPTION PROMPT statement, loads the *string-variable* with the *literal-mask*, and accepts input from the operator. (Note that the *string-variable* should be cleared first.)

In this mode, the *literal-mask* is used to determine the maximum length of the input line. For example, both "" & SPACE$(10) and "!" & SPACE$(9) limit the line to 10 characters.

If the first character in the *literal-mask* is an exclamation mark, BASIC will automatically terminate input when the last character in the field is entered so that a carriage return is not needed. If the *literal-mask* is "!", LINPUT USING returns immediately after the first character is input.

Mode 2: This mode is identical to the first one except that the *string-variable* is loaded with the *mask* (a string that does not begin with a quote mark) and displayed with the cursor in the first position so editing may be performed. After the allowed number of characters have been entered, the bell is rung every time a non-editing key is pressed. Pressing the Return key will terminate input.

Notes

The LINPUT USING statement is similar to LINPUT except that it

- Limits the length of the input field to the number of characters in the *input-mask.*
- Allows a default input string to be specified.
- Allows the input string to be edited.

- Does no further editing on the user's input. For example, trailing spaces are not trimmed.
- Performs no carriage-return+linefeed under DOS and Windows or carriage-return under UNIX when the operation is terminated. BASIC will move the cursor to the end of the input field except in mode 1 when a mask of "!" is used or when a special control key is entered.

In both modes, LINPUT USING allows extensive editing of the input field. The following keys are recognized:

Key	Function
—>	Moves right. (non-destructive)
<—	Moves left. (non-destructive)
Delete or CTRL/Z	Deletes character under cursor.
Back Space	Deletes character left of cursor.
Insert	Toggles between insert/replace mode.

Entering a control key other than the editing keys listed above will terminate input and set the INP function to the value of the control key. Otherwise, INP is set to zero when a Return key is entered or when "!" is used to perform automatic entry (mode 1).

Examples	**Explanation**
LINPUT USING "abcdefg", A$	The default prompt is displayed, but "abcdefg" is not. A maximum of 7 characters are accepted for the value of A$.
LINPUT USING "!abcdef", A$	Works like the previous example except that when the seventh character is entered, input is terminated immediately
LINPUT "Sex (M or F)", USING "!", SEX$	Useful for 1-character responses.
A$ = "abcdefg" LINPUT USING A$, A$	The default prompt is displayed, followed by the string "abcdefg". The new value of A$ will be the user-edited version of this string, and will be at most 7 characters long.
A$ = "!abcdef" LINPUT USING A$, A$	This works exactly like the previous example except that now the initial value for A$ displayed will be "!abcdef". In this form, input does not terminate on entry of the last character.
LINPUT USING "!", CMD	Code for a screen-editor type application.

```
SELECT CMD
    CASE 0
        GOSUB PROCESS.LETTER
    CASE ASC(CRT$("UP"))
        PRINT CRT$("UP");
    CASE ASC(CRT$("DOWN"))
        PRINT CRT$("DOWN");
        etc.
```

Incorrect Examples	**Explanation**
LINPUT "CMD> " USING A$, B$	No comma after prompt.
LINPUT AT$(1,24) & "CMD> ", USING A$, B$	AT$. will not be recognized as the start of a string-literal-expression prompt string. Use "" & AT$ ...

See Also INPUT, LINPUT, MAT INPUT, OPTION, CRT function, INP function.

MAT

The MAT statement may be used to assign a value to all elements of an array, to copy one array to another, to initialize an array or to perform one of the standard matrix operations on a pair of arrays.

Syntax

1 MAT *array-name* = *array-name*

2 MAT *array-name* = *(expression*)

3 MAT *array-name* = *array-name* + *array-name*

4 MAT *array-name* = *array-name* - *array-name*

5 MAT *array-name* = *array-name* & *array-name*

6 MAT *array-name* = *array-name* * *array-name*

Description **Mode 1**: The contents of the source *array-name* on the right of the equal sign is copied in linear order into the destination *array-name* on the left.

The arrays need not have the same dimensions or total number of elements. When the arrays are dimensioned differently, all the elements from the first row of the source array are copied before any elements in the second row. If the destination array has fewer columns, then the excess elements from the first row of the source will be copied to the second row of the destination array.

If the destination array is smaller than the source array, only the elements that will fit are copied. When the destination array is larger, any remaining elements are set to zero or null. If the destination array has not been dimensioned, it will be given the size of the source array.

Mode 2: Sets all elements of the array to the value of the *expression*. The *expression* must be enclosed in parentheses, and may be a constant, variable, or a complex expression. In all cases, its type must match that of the *array-name*.

The *expression* limits the size of the array that will be written. If the destination *array-name* has not been dimensioned, it will be created with the default dimensions (10, 10).

Since BASE 0 is the default, modes one and two operate on all elements of an array, even those where one or both subscripts are zero.

Mode 3: Adds the corresponding elements of the two source arrays and places the results in the destination array. If string arrays are specified, string concatenation is performed.

Mode 4: Subtracts corresponding elements of the two source arrays and places the results in the destination array. String operands are not allowed.

Mode 5: Performs string concatenation on corresponding elements of the two source arrays and places the results in the destination array. All arrays must be of string type. Note that in modes 3 through 5, all arrays must be of the same data type since no type conversion is performed.

The two source arrays must have the same dimensions. The destination array must either have the same dimensions or none (the array has not been defined with a DIM statement). In the latter case, an array equal in size to each of the two source arrays will allocated for the destination array.

Mode 6: Performs matrix multiplication on the two source arrays. All arrays must have the same numeric type. The source arrays must have dimensions which obey the usual requirements for matrix multiplication, and the destination array must have the proper dimensions to receive the product or have no storage allocated, in which case an array of the proper size will be dimensioned.

Restrictions Modes 3 through 6 are only available with the CET UNIX-based products.

Examples	**Explanation**
DIM A$(5),B$(5),C(20)	Defines sizes of arrays A$, B$, and C.
FOR I%=0 TO 5 B$(I%) = STR(I%) NEXT	Sets B$(0) = "0", B$(1) = "1", etc.
MAT A$ = ("")	Sets all 6 elements in A$ to be empty or null.
MAT A$ = B$	Copies values of B$ into A$. B$ is unchanged.
MAT C = (1)	Sets all 21 elements of C to be 1.
MAT D = C	Creates D as an array with maximum dimension 20 and copies the 21 elements (1's) of C into D.

Incorrect Examples	**Explanation**
DIM A$(5),B$(6),C$(5,2)	Defines size of arrays A$, B$, and C$.
MAT A$ = (1)	Expression must match array in type.
MAT A$ = B$	Arrays are of different size.
MAT B$ = C$	" " " " "

See Also DIM, COMMON, LET, OPTION, and other MAT statements

MAT INPUT

The MAT INPUT statement accepts data from the console or a file and assigns the input fields to the elements of an array.

Syntax

1 MAT INPUT *array-name*

2 MAT INPUT *#channel : array-name*

3 MAT INPUT *#channel , key : array-name*

array-name = *array-name* [(*subscript*)]
= *array-name* [(*subscript1, subscript2*)]

Description **Mode 1**: Accepts one line of data from the keyboard and assigns each field to an element in the *array-name* similar to the way INPUT operates:

```
DIM ANS$(3)
MAT INPUT ANS$
```

The following statement performs the same function:

```
INPUT ANS$(1),ANS$(2),ANS$(3)
```

Mode 2: Accepts ASCII formatted data from a file opened for INPUT or UPDATE SEQUENTIAL on the specified *channel.* A record is read from the file and the individual fields are assigned in consecutive order to the elements in the *array-name.*

Mode 3: Accepts ASCII formatted data from a direct or indexed file which is open for INPUT or UPDATE. To access a direct file, the *key* must be a numeric. For indexed files, the *key* must be a string expression. In either case, the file must have been written with PRINT or MAT PRINT statements.

Notes When MAT INPUT is used to read input from the console or a sequential file, one line of input is used to fill a one-dimensional array and one line per value of the first subscript (i.e. per row) to fill a two-dimensional array.

Only one record of a direct or indexed file will be input for each MAT INPUT statement. If there are fewer fields in the record than there are data elements in the array, the remaining elements will be set to zero or null. If there are extra fields, they will be ignored.

By default, no assignment is made to any elements of an array with the subscript zero (0). To cause all array elements to be assigned, the statement OPTION MATIO 0 should be executed prior to the MAT INPUT statement. (An example is given below.)

If the *array-name* has not dimensioned previously, it will be allocated a default size (10, 10).

Under DOS and Windows, the INPUT and MAT INPUT statements will automatically read a carriage-return+linefeed character as the record delimiter.

By default, a carriage-return will be read as the delimiter under UNIX. If the file has been opened with the option NEWLINE (or the environment variable B_OPENNL has been set), the linefeed delimiter will be used to conform to the UNIX convention.

For compatibility purposes, option NEWLINE and the variable B_OPENNL are supported under DOS and Windows, but they will have no effect.

Examples	**Explanation**
DIM AR(4)	Defines elements AR(0), ..., AR(4)
MAT INPUT AR	Accepts values for AR(1), ..., AR(4)
INPUT AR(1),AR(2),AR(3),AR(4)	Identical effect to MAT INPUT AR
DIM BR(3,5)	Defines elements BR(0,0), ..., BR(3,5)
MAT INPUT BR	Accepts values for BR(1,1), BR(1,2), ..., BR(1,5), BR(2,1), ..., BR(2,5), ..., BR(3,5)
OPTION MATIO 0	
MAT INPUT AR	Accepts values for AR(0), AR(1), ..., AR(4)

See Also COMMON, DIM, INPUT, LINPUT, LINPUT USING, OPTION, MAT, MAT READ, READ

MAT PRINT

The MAT PRINT statement outputs the contents of one or more arrays to the console or an open output file.

Syntax

1 MAT PRINT *array-name-list ...*

2 MAT PRINT *#channel : array-name-list ...*

3 MAT PRINT *#channel , key: array-name-list*

array-name-list = *array-name* [*punct* [*array-name-list*]]
array-name = *array-name* [(*subscript*)]
= *array-name* [(*subscript1, subscript2*)]
punct = *,*
= *;*

Description **Mode 1**: Displays the contents of one or more arrays on the console. The format in which each array is printed is controlled by the punctuation following the *array-name*. If there is none, each element in the array is displayed on a separate line. If either a comma or a semicolon is used, the elements in the array are displayed as a matrix, in multiple columns and rows. A one-dimensional array is printed on a single line while a two-dimensional array uses one line for each value of the first subscript.

Using a comma after the *array-name* will print the elements one per print zone (21 characters). If a semicolon is used, the elements are printed one right after the other. Only numeric items will be separated with spaces.

Mode 2: Prints one or more arrays to a sequential file or printer opened on the specified *channel*. When the output is directed to a printer, it will be formatted as in mode 1. Otherwise, the entire output from the MAT PRINT statement is placed in a single record with commas separating the individual data elements in the array.

Mode 3: Outputs one or more arrays to a record in a direct or indexed file. When specifying the *key*, use a numeric or integer for direct files, and a string expression to access indexed files. The entire output from the MAT PRINT statement is written to the record specified by the *key*.

Notes By default, only the elements of the array with subscript(s) 1 or larger are output. However, if the OPTION MATIO 0 statement is in effect, all entries will be output.

When the output is directed to a device other than the console, the channel must be open for OUTPUT or UPDATE with the correct access mode. Since

indexed and direct files are created with a fixed record length, any excess data will be truncated.

The MAT PRINT statement outputs ASCII data. To access this information use the MAT INPUT, INPUT or LINPUT statements.

Under DOS, the PRINT and MAT PRINT statements will automatically output a carriage-return+linefeed character as the record delimiter.

By default, a carriage-return will be output under UNIX. If the file has been opened with the option NEWLINE (or the environment variable B_OPENNL has been set), the linefeed delimiter will be used to conform to the UNIX convention.

For compatibility purposes, option NEWLINE and the variable B_OPENNL are supported under DOS and Windows, but they will have no effect.

Examples	**Explanation**
DIM A(5), B(2, 3) MAT A = (5) MAT PRINT A; MAT PRINT A, MAT PRINT A	Produces output: 5 5 5 5 5 5 5 5 ... 5 5 5 etc. The same result could be obtained with the statement MAT PRINT A; A, A
MAT B = (1) MAT PRINT B;	Produces output: 1 1 1 1 1 1
OPTION MATIO 0 MAT PRINT B;	Produces output: 1 1 1 1 1 1 1 1 1 1 1 1

See Also DIM, MAT WRITE , PRINT, PRINT USING, OPEN, OPTION

MAT READ

The MAT READ statement reads data from a file or from DATA statements into an array.

Syntax

1 MAT READ *array-name*

2 MAT READ *#channel: array-name*

3 MAT READ *#channel, key: array-name*

array-name = *array-name*
= *array-name* (*subscript*)
= *array-name* (*subscript1, subscript2*)

Description **Mode 1**: Reads the DATA statements in the current program and copies the data into the elements of the specified array. If insufficient data is available, a trappable error ("Out of data") occurs. If there are more DATA statements than array elements, the excess statements will not be read until the next MAT READ or READ is executed.

Mode 2: Reads a record from a binary sequential file opened for INPUT or UPDATE on the specified *channel*. The record must have been written by a WRITE or MAT WRITE statement.

Each field in the record is assigned in consecutive order to the elements in the array. If there are fewer fields than there are array elements, the remaining elements will be set to zero or null, depending upon the array type. Any extra data fields in the record will be ignored.

Mode 3: Reads a record with the specified *key* from the direct or indexed file. Use a numeric key to access a direct file and a string key to read an indexed file. Otherwise, this mode operates identically to mode 2.

Notes By default, only the elements of the array with subscript(s) 1 or larger are read with this statement. Use the OPTION MATIO 0 statement if all of the array elements should be read.

The MAT READ statement reads elements into an array. Each field read is assigned to the next higher subscript of the array. When the array is two dimensional, the second subscript varies first.

Both the MAT READ and READ statements are designed to access binary data. When they are used to read data that was written with PRINT or MAT PRINT, all elements in the array will be set to zero or null. This data is in an ASCII format and must be read with MAT INPUT, INPUT or LINPUT.

Reading a record in an indexed or direct file opened for UPDATE will automatically *lock* the record. If another program attempts to access the record, that program is suspended, with periodic retries until the record is unlocked unless there is an ON ERROR or ON LOCK routine in effect. These statements may be used to specify other behavior for a locked record condition.

If the B_EMULATE variable is set, a locked record is not considered a trappable error. If you wish to use THEOS emulation and special error handling, set the B_THLOCK variable off (to null).

Note that if an ON ERROR routine exists, but does contain any code to handle a locked record (error #48), the program will abort with an appropriate error message.

For further information on record and file locking, refer to the *CET BASIC Error Handling System* chapter. The environment variables B_EMULATE and B_THLOCK are covered in the *CET BASIC User's Guide.*

Examples	**Explanation**
DIM AR(4),A$(2,5) MAT READ AR DATA 1.23, 45, 6.7, 3.3, .987 DATA TWO, 3.33, etc.	Sets AR(1) = 1.23, AR(2) = 45, etc.
MAT READ A$	Sets A$(1,1) = ".987", A$(1,2) = "TWO", A$(1,3) = "3.33", etc.
OPTION MATIO 0 \ RESTORE MAT READ AR	Sets AR(0) = 1.23, AR(1) = 45, etc.
DIM STATE$(51) MAT READ STATE$ DATA "ALABAMA" DATA "ALASKA" DATA "ARIZONA"...	Add 1 to number of DATA statements.

See Also COMMON, DATA, DIM, MAT READNEXT, MAT READPREV, OPEN, OPTION, READ, RESTORE

MAT READNEXT

The MAT READNEXT statement reads the next record in an indexed or direct file and copies the data fields into the elements of an array.

Syntax MAT READNEXT #*channel , key : array-name*

array-name = *array-name*
= *array-name* (*subscript*)
= *array-name* (*subscript1, subscript2*)

Description The next record in the direct or indexed file is read and the data is copied into the elements of the array. The key of that record is then stored in the specified *key*.

Notes If the file is being accessed for the first time since it was opened, the MAT READNEXT statement

- Reads the first record in the file.
- Sets the *key* to null and the EOF indicator to true if there are no more records to be read.

Refer to the description of mode 3 of the MAT READ statement for general information since these two statements operate the same. Record locking is also covered in that section.

Note that when using an ON ERROR or ON LOCK routine to handle record locks, the environment variable B_OPTLOCK must be set so that the file pointer is not moved after the RESUME statement. This way MAT READNEXT will attempt to read the desired record again. If this variable is not set, BASIC will move to the next record, by default.

Example	**Explanation**

```
DIM ARRAY(4)                                    Program prints the entire file contents.
OPEN #1: "TRANS", INPUT INDEXED
WHILE NOT EOF(1)
    MAT READNEXT #1, KEY$: ARRAY
    IF NOT EOF(1) THEN PRINT KEY$;'-- ';\MAT PRINT ARRAY;
WEND
CLOSE #1
```

See Also COMMON, DIM,MAT READ, MAT READPREV, OPEN, OPTION, READ, READNEXT, READPREV

MAT READPREV
MAT READPRIOR

The MAT READPREV statement or its synonym MAT READPRIOR may be used to read the prior record and copy the data fields into the elements of an array. For brevity, only MAT READPREV will be referred to in this section.

Syntax MAT READPREV #*channel , key : array-name*

array-name = *array-name*
= *array-name* (*subscript*)
= *array-name* (*subscript1, subscript2*)

Description The previous record in the direct or indexed file is read and the data is copied into the elements of the array. The key to that record is then stored in the specified *key*.

Notes If the file is being accessed for the first time since it was opened, the MAT READPREV statement

- Assumes the current position is at the beginning of the file.
- Sets the *key* to null and the EOF indicator to true.

Refer to the description of mode 3 of the MAT READ statement for general information since these two statements operate the same. Record locking is also covered in that section.

Note that when using an ON ERROR or ON LOCK routine to handle record locks, the environment variable B_OPTLOCK must be set so that the file pointer is not moved after the RESUME statement. This way MAT READPREV will attempt to read the desired record again. If this variable is not set, BASIC will move to the previous record, by default.

The following example illustrates how a program might read the last record in an indexed file. A *key* of CHR$(255) is used to position the pointer to the end of the file since no character has a value greater than 255. After reading the *record*, MAT READPREV may then be used to read backwards through the file, displaying the contents of each record on the screen.

Example	**Explanation**
DIM ARRAY(4)	
OPEN #1: "DATA", INPUT INDEXED	
READ #1, CHR$(255): ARRAY	Reads the record at end of file.
WHILE NOT EOF(1)	
MAT READPRIOR #1, key$: array	Prints entire file contents backwards.
IF NOT EOF(1) THEN PRINT KEY$;'-- ';\MAT PRINT ARRAY;	

```
WEND
CLOSE #1
```

See Also COMMON, DIM, MAT READ, MAT READNEXT, OPEN, OPTION, READ, READNEXT, READPREV

MAT SORT

The MAT SORT statement may be used to create an array that contains the sorted indices of another array. This feature is useful for sorting items *in memory*.

Syntax MAT *index-array* = SORT (*source-array*)

index-array = *array-name*
= *array-name* (*subscript*)
= *array-name* (*subscript1, subscript2*)

Description The index values in the *source-array* are sorted in ascending order and stored in the *index-array.* The actual contents of the *source-array* are not changed.

Both arrays should have been previously dimensioned. If the *source-array* is a two-dimensioned array, only the first elements will be used in the sort operation. Any remaining elements will be set to zero.

The *index-array* must be numeric, but the *source-array* may contain either numeric or string data.

Notes Both arrays should be the same size in order to index the entire source-array. If the *index-array* is smaller, only that number of elements will be sorted. If the *index-array* is larger, the extra elements are set to zero.

Example	**Explanation**
DIM SOURCE$(5),INDEX%(5)	Arrays are dimensioned to 5.
SOURCE$(1) = “Smith”	Loads the SOURCE$ array. SOURCE$(2) = “Jones” SOURCE$(3) = “Laurel” SOURCE$(4) = “Hardy” SOURCE$(5) = “Adams”
MAT INDEX% = SORT (SOURCE$)	Array is initialized and sorted.
FOR I% = 1 TO 5 PRINT SOURCE$(INDEX%(I%)) NEXT I%	Prints the array in sorted order: Adams Hardy Jones Laurel Smith

See Also COMMON, DIM

MAT WRITE

The MAT WRITE statement writes the contents of an array to the specified file.

Syntax

1 MAT WRITE *#channel: array-name*

2 MAT WRITE *#channel, key: array-name*

array-array = *array-name*
= *array-name* (*subscript*)
= *array-name* (*subscript1, subscript2*)

Description **Mode 1**: Writes a formatted record to a sequential file opened for OUTPUT.

Mode 2: Writes a formatted record to a direct or indexed file. The *key* must be a numeric for direct access, and a string expression for indexed files. In either case, the file must be open for OUTPUT or UPDATE.

Only one record will be written to the file. If the record is not long enough to hold all the data, the excess data will be truncated. A trappable error #47 is generated.

Notes By default, the zero elements of an array are not written to the file unless the OPTION MATIO 0 statement is in effect. In that case, the elements where one or both subscripts are zero will be written.

Examples	Explanation
OPTION BASE 1 OPEN #1: "FILE\TEST", OUTPUT SEQUENTIAL, EXTEND DIM B(4),A$(2,5) MAT WRITE #1: B	Outputs 4 fields to the file on channel 1
WRITE #1: B(1),B(2),B(3),B(4)	This statement is identical in function to: MAT WRITE #1: B
OPEN #2: "DATA\CUST", OUTPUT DIRECT MAT WRITE #2,1: A$	Ten elements will be written to the first record in DATA\CUST.
write #2,1:a$(1,1),a$(1,2),a$(1,3),a$(1,4),a$(1,5),a$(2,1),a$(2,2),a$(2,3),a$(2,4),a$(2,5)	This statement is identical to MAT WRITE #2,1:A$.

See Also COMMON, DIM, MAT PRINT, OPEN, OPTION, WRITE

MOUNT

The MOUNT statement performs no function in a CET BASIC program. It is recognized to maintain compatibility with THEOS BASIC programs that may use it to inform the operating system that a new diskette may have been inserted into the disk drive.

Syntax MOUNT *string-expr*

NEXT

The NEXT statement marks the end of a FOR-NEXT loop structure, and causes the loop to be repeated if the limit specified in the FOR statement has not been reached.

Syntax

1 NEXT

2 NEXT [*variable-name*]

Description

Mode 1: NEXT marks the end of the current FOR-NEXT loop, passes control to the matching FOR statement, and gives the index variable its next value. If the specified limit has not been reached, the FOR-NEXT loop will be repeated. This is the recommended mode for the statement.

Mode 2: NEXT passes control to the most recent FOR-NEXT structure that uses the *variable-name* as its index variable. This mode must be used with caution if multiple FOR-NEXT loops are open so that control is not passed to the wrong loop.

Notes

When a BREAK statement occurs within a FOR-NEXT loop, control is passed to the first statement following the NEXT statement.

Examples	Explanation

FOR I%=1 TO 5
  PRINT I%
  NEXT I%
	Repeats following instructions five times.
	
	This marks the end and causes repeat.

FOR I%=1 TO 5 STEP 1
  PRINT I%
  NEXT I%
	Same as above.

FOR S$="A","B","C"
  PRINT S$
  FOR I%=1 TO 5
 	
  PRINT I%
  NEXT
NEXT
	Performs loop 3 times.
	
	Performs this loop 5 times for each of the 3 major loops.
	
	Marks end of FOR I% loop.
	Marks end of FOR S$ loop.

See Also

BREAK, FOR

ON ERROR

The ON ERROR statement specifies the location of user-defined, error-handling routines.

Normally, BASIC will display an error message and abort the program when an error is detected. ON ERROR may be used to detect the error and perform an operation to correct the problem.

Since error handling is such an important issue in an application, it is covered separately in the *CET BASIC Error Handling System* chapter later in this manual. A list of error codes has been included in the *Appendix* for your convenience.

Syntax

1 ON ERROR GOTO *line-reference*

2 ON ERROR GOTO 0

line-reference = *line-number*
= *line-label*

Description

Mode 1: Transfers control to the specified line in the program whenever a trappable error is detected.

Mode 2: Disables any user-defined error handling and directs BASIC to use its standard default processing.

Notes

ON ERROR will process the interrupt key and file/record lock errors if other ON INTERRUPT and ON LOCK routines are not currently in effect.

ON ERROR processing may be enabled at different times and directed to different locations within the same program. It may also be disabled.

When an error occurs and ON ERROR processing has been specified, execution of the statement in error is completed, ignoring any further errors which may occur, before control is passed to *line-reference*.

The error-handling routine must be terminated with a RESUME statement. Normally, RESUME transfers control back to the statement which caused the error.

An error handling routine may be tested with the LET ERR statement to invoke the routine. Information about the error may be determined by using the functions ERR, ERF and ERL to return the error number, the channel on which the last I/O operation was performed, and the line in the program which caused the error. (See the description of the -# compiler flag if some or all of the program lines are unnumbered.)

The ON ERROR statement may be used within an error-handling routine.

ON ERROR processing is automatically disabled by RUN, CHAIN or LINK statements. Each program must define its own error handling routine.

It is easy to add error routines to existing programs. Simply write the code you want to use and store it in an external file with unnumbered lines. Then, create a program to open your BASIC source files (with a **.b** extension) and write the line #include "*filename*" at the top of each file. Then, recompile all of your program files. The code stored in the specified *filename* will be inserted in the program during the compilation process.

```
Example                                   Explanation

ON ERROR GOTO ERR.ROUTINE          Routine to handle trappable errors.

ERR.ROUTINE:
    SELECT ERR                            Using error function, select error routine.
    CASE 1                                Perform if ERR=1 (INTERRUPT key).
        IF ERL<1000 OR ERL>1999 THEN RESUME
                                          Ignore if not of interest else do the
                                          following.
        GOSUB CLOSE.REPORT
        RESUME  MENU
    CASE 36                               Perform if ERR=36 (out of data).
        IF ERL=990 THEN RESUME 991
        RESUME
    CASE 30                               Perform if ERR=30 (file not found).
        PRINT AT$(1,24);"Invalid file name";CHR$(7);
        LINPUT " Type <return> to continue: ",USING "!",ANSWER$
        RESUME
    CEND
RESUME 0
```

See Also LET, ON INTERRUPT, ON LOCK, RESUME and the ERF, ERL, ERR functions.

ON GOSUB

The ON GOSUB statement provides a means of calling one or more subroutines based upon the result of an expression. This keyword may also be entered as ON GO SUB.

Syntax ON *numeric-expression* GOSUB *line-references*

line-references = *line-number* [, *line-references*]
= *line-label* [, *line-references*]

Description The *numeric-expression* is evaluated and converted to an integer value *n,* if necessary. Control is then transferred to the *nth reference* specified in the *line-reference*.

Notes If the value of *n* is less than one o*r* greater than the number of the *line-references*, none of the items on the line are invoked. When that happens, execution continues with the statement following the ON GOSUB.

There is no limit to the number of *line-references* in the list, other than the maximum number of characters on a line must not exceed 800.

When control is transferred by an ON GOSUB statement, BASIC operates as though a GOSUB has been executed and expects a corresponding RETURN statement.

For more specific information on the ON GOSUB statement, refer to the description of GOSUB and RETURN.

Examples	**Explanation**
on num% gosub 100, 200,err.rout	A GOSUB to 100, GOSUB to 200 and GOSUB to ERR.ROUT is executed when NUM% = 1, 2 or 3 respectively. After the selected subroutine has finished, control is passed back to the statement on the line following the ON GOSUB statement.

See Also ON GOTO, GOSUB, RETURN

ON GOTO

The ON GOTO statement provides a means of calling one or more subroutines based upon the result of an expression. This keyword may also be entered as ON GO TO.

Syntax ON *numeric-expression* GOTO *line-references*

line-references = *line-number* [, *line-references*]
= *line-label* [*, line-reference* s]

Description The *numeric-expression* is evaluated and converted to an integer value *n,* if necessary. Control is then transferred to the *nth reference* specified in the *line-references*.

Notes If the value of *n* is less than one or greater than the number of *line-references*, none of the items on the line are invoked. When that happens, execution continues with the statement following the ON GOTO.

There is no limit to the number of *line-references* in the list, other than the maximum line length may not exceed 800 characters.

For more specific information on how ON GOTO operates, refer to the description of the GOTO statement.

Examples	Explanation
ON I GOTO 100,110,100,130	When I = 1 or I = 3, control passes to line 100, I = 2 to line 110, I = 4 to line 130. If I <= 0 or I > 4, control passes to the next line.
on index-4 goto line1, , line3	When INDEX = 5 control passes to LINE1; When 7, control passes to LINE 3; When INDEX <= 4, INDEX = 6, or INDEX > 7, control passes to the next line.

See Also GOSUB, GOTO, ON GOSUB

ON INTERRUPT

The ON INTERRUPT statement instructs BASIC that a user-defined, interrupt-handling routine exists at a specified location. This feature allows a program to process the system interrupt key differently than a trappable error.

Since error handling is such an important issue in an application, it is covered separately in the *CET BASIC Error Handling System* chapter later in this manual. A list of error codes has been included in the *Appendix* for your convenience.

Syntax

1 ON INTERRUPT GOTO *line-reference*

2 ON INTERRUPT GOTO 0

line-reference = *line-number*
= *line-label*

Description **Mode 1**: Enables interrupt key processing and transfers control to the routine at the specified *line-reference*.

Mode 2: Disables user-defined, interrupt processing and returns either to the specified ON ERROR processing (if any) or to the default BASIC error handling mechanism.

In both modes, GO TO may be used as a substitute for GOTO.

Notes BASIC processes the interrupt key as follows:

- If ON INTERRUPT processing is in effect, the code at the specified *line-reference* is used.
- If ON ERROR processing is in effect, the specified routine is executed.
- If no user-defined processing is specified, BASIC aborts the program and displays an appropriate message.

When an interrupt key is detected and is trapped by either an ON ERROR or ON INTERRUPT routine, any other program errors will be ignored until a RESUME statement is executed.

By default, the RESUME statement resumes execution at the line after the one in which the interrupt key was detected. However, RESUME may be used to continue execution at a different location in the program.

The current ON INTERRUPT processing is automatically disabled by RUN, CHAIN or LINK statements. Each program segment must define its own interrupt-handling routine.

An interrupt-handling routine may use the ERR or ERL functions to determine which error occurred and where, and the LET ERR statement to invoke the appropriate action.

Example	Explanation
ON INTERRUPT GOTO INTERRUPTED	Pressing the interrupt key will transfer control to INTERRUPTED.

```
INTERRUPTED:
    PRINT "Do you really want to Quit ?";
    IF YESNO$ = "Y" THEN RESUME 0 ELSE RESUME
```

See Also LET ERR, ON ERROR, ON LOCK, RESUME, the ERL and ERR functions

ON KEY

The ON KEY statement may be used to specify the location of user-defined routines for special key processing.

Note that this statement may not be supported in all of the CET BASIC products. Please check with your CET distributor about its availability under the product you are using.

Syntax

1 ON KEY *key-value* GOTO *line-reference*

2 ON KEY ON

3 ON KEY OFF

line-reference = *line-number*
= *line-label*

Description

Mode 1: Enables the special processing at the specified *line-reference* whenever the key defined by the *key-value* is detected. Normal execution is interrupted until a RESUME statement is performed. Note that a *line-reference* of 0 is a special case which removes the indicated *key-value* from the list of values that generate an ON KEY condition.

Mode 2: Enables all ON KEY detection. Since this feature is initially set off, the ON KEY ON statement must be executed at least once before any key processing can occur.

Mode 3: Disables ON KEY processing. This is the default condition. If ON KEY ON has been previously executed, no further action will be taken until another ON KEY ON statement is executed.

Notes

The *key-value* may be any arithmetic expression that matches the *translated* value of a keystroke. This translated value is defined as either:

- the ASCII value of the keystroke if it is in the range of printable characters (those that would not set INP in an INPUT, LINPUT or GET statement).
- the INP value of the keystroke if it is not in the range of printable characters (those that would set INP in an INPUT, LINPUT or GET statement).

The following illustrates how the *key-value* of ”q” may be specified using the ASC function (covered in the chapter on *Built-in Functions*).

```
ON KEY (ASC(“q”)) GOTO TRAPKEY
```

To trap for the occurrence of a letter such as “q” in both upper and lower-case:

```
ON KEY(81) GOTO TRAPKEY
ON KEY(113) GOTO TRAPKEY
```

Note that when B_EMULATE is set to THEOS, the INP value of a keystroke changes according to the dictates of THEOS emulation. For example, the following statement would always trap on Ctrl+W and the F1 key when B_EMULATE is set:

```
ON KEY(23) GOTO TRAPKEY
```

All ON KEY statements are active *concurrently* until one or more is deactivated by executing an ”ON KEY *key-value* GOTO 0” statement. The maximum number of concurrent ON KEY statements is eight. This default limit may be increased by setting the environment variable B_ONKEYMAX.

It is not necessary that each ON KEY statement point to the same routine. In fact, it can be useful to have multiple routines that process the *key-value* according to when it was detected.

ON KEY detection takes priority over other forms of input statements. If the application is executing an INPUT or LINPUT and one of the ON KEY character values is detected, the INPUT or LINPUT statement is interrupted and the ON KEY action is taken.

If a WAIT #0 or GET statement will accept the entry of an ON KEY character, the statement is interrupted and the ON KEY action is taken. Afterwards, the WAIT or GET must be executed again to wait for another entry.

Restrictions When a special key is being processed, trapping for another key is suspended until a RESUME statement is executed or another ON KEY is executed for the same key value.

See Also RESUME

ON LOCK

The ON LOCK statement allows a program to define special processing to handle a locked file or record condition in an indexed or direct file.

Since error handling is such an important issue in an application, it is covered separately in the *CET BASIC Error Handling System* chapter. A list of the error codes has been included in the *Appendix* for your convenience.

Syntax

1 ON LOCK GOTO *line-reference*

2 ON LOCK GOTO 0

line-reference = *line-number*
= *line-label*

Description The ON LOCK statement instructs BASIC that a user-defined, error-handling routine exists at *line-reference* or that the default BASIC system error-handling routine is to be used.

Mode 1: Enables processing of lock errors using the code located at *line-reference*. This is the recommended mode.

Mode 2: Disables special lock processing and turns it over to the active ON ERROR routine, if any, or to the standard default processing.

When an attempt is made to access a file or record locked by another user, by default BASIC will

- Perform ON LOCK processing, if any.
- Perform ON ERROR processing, if any.
- Wait and then attempt to access the file or record again. Note that if there is an ON ERROR routine, but it does not handle a record lock condition, the program will abort with an error #48.)

If ON LOCK is in effect, the lock-handling routine is started at *line-reference* when a lock condition is detected and is logically terminated by the execution of a RESUME statement.

It is important to note that the environment variable B_OPTLOCK must be set so that the file pointer is not moved after RESUME and the (MAT) READNEXT or (MAT) READPREV statements will attempt to read the desired record again. If this variable is not set, BASIC will move to the next or previous record, by default.

The ON LOCK routine can get information on how to handle the error by interpreting the functions ERR, ERF and ERL which return the error number, the file channel on which the last I/O operation was performed and the line number on which the error occurred, respectively.

When a lock condition is being processed, any additional program errors will be ignored until a RESUME statement is executed.

Notes The current ON LOCK processing is automatically disabled by RUN, CHAIN or LINK statements. Each program must define its own error-handling routine.

The following is a simple example of what might be done when a locked file is encountered. You may wish to set a accumulator that gets incremented each time the subroutine is entered. When the accumulator reaches a certain value, you could then give the operator an alternative action to take.

Example	**Explanation**
ON LOCK GOTO LOCKED	Locked files or records will be handled by LOCKED.
LOCKED:	
IF ERR=49 THEN PRINT “Lock on file number “, ERF	
SLEEP 1	Wait for file to be unlocked.
RESUME	

See Also LET ERR, ON ERROR, ON INTERRUPT, RESUME, and ERF, ERL and ERR functions

ON MOUSE

The ON MOUSE statement instructs BASIC that a user-defined, mouse handling routine exists at a specified location. To use this feature, you must load a mouse driver and set the environment variable B_MOUSESUP.

The statement is only available in the CET DOS and Networking product. The CET W/32 Application Builder for the Windows platforms provides its own method of mouse support.

Syntax ON MOUSE GOTO *line-reference*

line-reference = *line-number* [, *line-reference*]
= *line-label* [, *line-reference*]

Description Enables mouse click processing and transfers control to the routine at the specified *line-reference*.

Notes A mouse click will terminate input from WAIT, GET, INPUT, LINPUT or LINPUT USING statements. This feature permits the operator to use the mouse to enter a response or select an item. For instance, a program might set MOUSE.CLICK% to FALSE%, and then after a LINPUT, determine if the statement ended with a value for the input string, an INP value or a mouse click.

Consider the following code segment. When a mouse click is detected by the ON MOUSE statement, control is transferred to the mouselab subroutine. Upon returning from mouselab, your program could test for MOUSE.CLICK%=TRUE% and determine what response is associated with the indicated row and column.

```
ON MOUSE GOTO mouselab
.
.
.
mouselab:
MOUSE.CLICK%=TRUE%
LAST.BUTTON%=ERM_BUTTON
LAST.ROW%=ERM_ROW
LAST.COL%=ERM_COL
RESUME
```

Note the use of the following variables:

ERM_BUTTON Returns 0 for a left button click and 1 for a right click

ERM_ROW	Returns the row position of the last button click (the button release).
ERM_COL	Returns the column position of the last button click (the button release).

See Also B_MOUSESUP variable, *DOS Mouse Subroutines* chapter in the *CET BASIC DOS User's Guide*

OPEN

The OPEN statement is used to open an input/output (I/O) channel to a disk file or a physical device such as a printer.

Syntax

1 OPEN *#channel: file , mode, method* [, *options*]

2 OPEN *#channel: device, mode*, SEQUENTIAL [, *options*]

channel = *integer-expression*
file = *string-expression*
mode = *INPUT*
= *OUTPUT*
= *UPDATE*
method = *SEQUENTIAL*
= *DIRECT*
= *INDEXED*
options = *option* [, *options*]
= *EXTEND*
= *QUOTE*
= *FORMAT*
= *LOCK*
= *NEWLINE*
= *RAWMODE*
= *UX*
= *DOSEOF*

Description **Mode 1**: This is the most common form of the OPEN statement. It is used to open a disk file and specify how it should be accessed and processed. Note that indexed and direct files must exist before they before they can be accessed with an OPEN statement. That is not the case with sequential files which will be created automatically when a statement attempts to PRINT or WRITE a record. (See the *CET BASIC Data Files* chapter for details.)

Mode 2: This form is used to open a physical device such as the console (CON), printer (PRT), communications port (COM) or a tape (TAP). The only options that may be used in this mode are FORMAT and QUOTE.

Each of the elements in the statement are described as follows:

Channel

The file I/O *channel* is an integer expression whose value is between 1 and 24. The channel number is used in all references to the file or device such as PRINT #1 where #1 is the channel. The console is considered an open file, and in some statements is assumed to be channel #0.

The number of channels and files which may be open at the same time depends upon the CET BASIC product in use and the type of files themselves. This number is limited because each open file requires buffer and table space, but is generally at least sixteen indexed files. Additional direct and sequential files may also be opened at the same time.

File Names

CET BASIC programs recognize files specified with the complete path name or a name relative to the current directory. File names may be in a format native to the operating system in use. For instance, under DOS you could have **c:\pos\cp\data\custname**, **\pos\cp\data\custname** or **.\custname**.

For compatibility purposes, the CET DOS/Networking and W/32 Windows products recognize file names in either a DOS or UNIX format. For example, the CET BASIC will automatically convert any "/" characters in the name to "\". Since UNIX is case sensitive, a UNIX formatted name must be specified in order to access a file with a name in lower case letters.

File names may be also be specified in a THEOS format. Any name that contains a period, but does not begin with a drive code, a slash (/ or \) or a dot slash (./ or .\) is considered a THEOS name, and the default *filetype* translation method will be used to determine which file should be accessed. Drive codes are ignored unless the B_?DRIVE variable is set.

THEOS File Name	DOS File Name	UNIX File Name
CUST	CUST	CUST
CUST.DATA	DATA\CUST	DATA/CUST
CP.DATA.CUST	CP\DATA\CUST	CP/DATA/CUST

File-related environment variables may be set to affect how files are accessed. For example, B_LIBFNTYP may be used so that CET BASIC will read a name formatted as **a.b.c** and open **.\b\a\c** in the current directory.

Unless the complete path name (or the B_LIBFNTYP variable) is specified, BASIC will search the current directory for any indexed or direct file to be opened, and then along the path specified in the B_FPATH variable. If the file can still not be found, then an error 30 is detected. (BASIC always expects to find a sequential file in the current working directory.)

Please refer to the *CET BASIC User's Guide* for more information on the variables that are available for the specific product you are using.

Physical Device Names

The OPEN statement expects a logical name for the device to be used for input or output. The following names are recognized:

CONSOLE
Specifies that the user's terminal is to be used for I/O operations. The device acts like a pair of files, one for output and another for input. When the standard input file is not redirected, the console will receive the output from PRINT and PRINT USING as well as from INPUT, LINPUT, and LINPUT USING statements.

In some cases, you may need to open the console explicitly. (See the FORMAT option.) The device may be opened in any mode, but only for SEQUENTIAL access. CON is recognized as a synonym. For example:

```
OPEN #1: "CON", INPUT SEQUENTIAL
```

Since the console is opened on channel zero automatically, do not use channel #0 in the OPEN statement. The console should not be opened on more than one channel at a time, or the results may be unpredictable.

PRINTER*n*
Specifies one of the system printers. The device can be opened only for OUTPUT SEQUENTIAL. PTR*n* is recognized as a synonym.

Keyword	Output Device
PRT or PRINTER	**/dev/lp** under UNIX - PRN under DOS
PRT*n* or PRINTER*n*	**/dev/lp*n*** under UNIX - LPT*n* under DOS

Printing operations are done according to the operating system in use. Please refer to the *CET BASIC User's Guide* for information.

Note that under DOS, the built-in Bprtchk function may be used to determine the status of the printer and avoid critical errors that can cause a program to abort.

COM*n*
Specifies one of the available communications ports. For example, to access a COM port under DOS or Windows:

```
OPEN #1; "COM1", input sequential
```

Under UNIX, you must specify the device name as in:

```
OPEN #1; "/dev/ty8", input sequential
```

For reliable, high-speed data transfer under DOS, we recommend using the DOS *Communication Library* functions documented in the CET BASIC *Library Manual.*

TAP
TAP, TAPE and TAPE1 may all be used to specify a tape device after a device driver has been installed.

Mode

The mode specifies the type of operation to be performed on the device or file. The valid modes are:

INPUT
Data can only be read from a file or device. If an indexed or direct file is to be opened, it must already exist. The permissible operations on files open for INPUT are INPUT, LINPUT, READ, GET and their variants. No record locking is performed.

Under UNIX, option NEWLINE will have to be specified in order to read records with linefeed delimiters instead of the default carriage-return which is normally used to indicate the end of a record.

OUTPUT
Data can only be written to a file or device. If the access method is SEQUENTIAL and option EXTEND is not specified, any previously existing file with this name will be deleted and a new one created. Otherwise, the (indexed or direct) file must already exist.

The permissible operations on files opened for OUTPUT are PRINT, WRITE, PUT and their variants. No record locking is performed.

UPDATE

Data may be read from or written to the file or device. The access method is either DIRECT or INDEXED.

Record locking will be performed by default to ensure reliable update operations in a multi-user environment. Record locks are removed by writing the record, using the UNLOCK statement or closing the file.

The ON LOCK statement is provided so that you may define special handling to be performed when a locked condition is detected.

For more information on record locking, please refer to the *Multiuser File Protection* section in the *CET BASIC Data Files* chapter. The handling of lock conditions is covered in the *CET BASIC Error Handling System* chapter.

Method

This is a keyword specifying the method to use to access the file or device. The access method must be the same as the one used to create the file. For specific information on the different types of files refer to the *CET BASIC Data Files* chapter.

SEQUENTIAL
Specifies that records should be accessed in the order in which they are stored in the file.

DIRECT
Specifies that records should be accessed by a numeric key value relative to the record's position in the file.

INDEXED
Specifies that records should be accessed by a specific alphanumeric key or in sequential order by the key value using READNEXT and READREV statements.

To maintain compatibility with THEOS BASIC, KEYED may be used as a synonym for INDEXED.

Options

The options that may be used while accessing the file or device are:

DOSEOF
Specifies that a Ctrl+Z (01AH) should be used when necessary to detect the end of a file that was not created with a CET BASIC program.

EXTEND
Causes a file opened for SEQUENTIAL OUTPUT to be positioned to the last record prior to performing any operations. The result of subsequent WRITE and PRINT statements will be appended to the end of the file.

The EXTEND option has no effect on files opened for INDEXED or DIRECT access. Using EXTEND with INPUT mode will cause a syntax error.

FORMAT
Specifies that files opened for SEQUENTIAL access will have a leading ANSI forms control character supplied by each PRINT or PRINT USING statement.

The ANSI forms control characters are:

Character	Action
+	Skip no lines before printing (used to overstrike).
0	Skip two lines before printing.
–	Skip three lines before printing.
1	Form feed (to top of next page) before printing.
" "	Skip to the next line before printing.

End-of-line termination is suppressed for FORMAT files, and any trailing comma or semicolon character is ignored.

When the output device is the console, the FORMAT option is implemented by skipping lines from the current cursor position as dictated by the forms control character. The character "1" will send the equivalent of CRT$("clear") to the console when the device has been opened explicitly. The "+" is treated as a null and causes PRINT to skip one line.

The control characters that are available when outputting to a file are:

Character	Action
+	Prints a <carriage-return> preceding the record.
0	Prints a <carriage-return>/<linefeed> preceding the record.
–	Prints a <formfeed> preceding the record.

LOCK
Specifies that the entire file is open for the exclusive use of the operator. If the file is already open, the program will normally wait until it is closed.

If a program successfully opens a file with OPTION LOCK, a trappable error #49 ("File locked") will be generated any time another program attempts to open the file.

Detecting and handling lock conditions is covered in the *CET Basic Error Handling System* chapter.

NEWLINE
Specifies that the linefeed character should be used as the end-of-record indicator instead of the carriage-return which is the default under UNIX. The B_OPENNL environment variable may be set with the same result.

This option should always be used when reading files created by non-BASIC programs such as a text editor.

For compatibility purposes, option NEWLINE is recognized under DOS and Windows, but it will have no effect.

QUOTE
Specifies that string fields output with the PRINT or MAT PRINT statements are to be enclosed in quotes if the string contains any embedded quotes, commas or spaces. Files written with this option can be read using INPUT.

A comma will always be output between fields unless the OPTION COMMA has been specified. In that case, a semicolon will be used as the field separator unless the B_COMMASEP variable has been set. Currently, this variable only exists under DOS. Contact your CET distributor if you need this feature to be supported in another CET product.

RAWMODE
Specifies that the stream of characters should not be modified. When used, the automatic carriage-return+linefeed substitution for carriage-return characters is suppressed.

RAWMODE is a special option designed to be used when printing bar codes generated with the CET Bar Code Library functions.

UX
Specifies that the file has been written or is to be written in a UX-BASIC format. This option also implies that NEWLINE is to be used under UNIX.

Note that the first record in a UX-formatted direct file is number 0, as opposed to CET BASIC direct files which begin with record number 1.

Examples

```
OPEN #1: "MAST.DATA:A", INPUT SEQUENTIAL
OPEN #5: "./test.data", OUTPUT DIRECT
OPEN #J: "PRINTER", OUTPUT SEQUENTIAL, FORMAT
OPEN #8: F$, UPDATE INDEXED, LOCK
OPEN #15: "./usr/me/checks.c1h", OUTPUT SEQUENTIAL
OPEN #4: "PRINTER.FILE:S", OUTPUT SEQUENTIAL, EXTEND, FORMAT
```

See Also CLOSE, DELETE, GET, INPUT, LINPUT, MAT INPUT, MAT PRINT, MAT READ, MAT READNEXT, MAT READPRIOR, MAT WRITE, MAT PRINT, PRINT USING, PUT, READ, READNEXT, READPRIOR, UNLOCK, WAIT and WRITE

OPTION

The OPTION statement sets or resets the global parameters that affect program execution. In particular, note the SERIAL option which may be used to serialize BASIC programs so that they will only work with a runtime system that has the same serial number.

Syntax OPTION *option-list*

option-list	=	*option* [, *option-list*]
option	=	BASE *base*
	=	BOOLEAN
	=	CASE *case-mode*
	=	COMMA
	=	DATEFORM *dateform*
	=	DEGREE
	=	LOCK *numeric variable or constant*
	=	LOGICAL
	=	MATIO *mat_io_origin*
	=	PROMPT *prompt*
	=	RADIAN
	=	SERIAL *serial-number*
base	=	0 or 1
case-mode	=	*string-expression*
dateform	=	1, 2, or 3
default-type	=	REAL
mat_io_origin	=	0 or 1
prompt	=	*string-expression*
serial-number	=	*integer-constant*

Description The operating mode of the program is affected by the parameters in the OPTION statement. Separate multiple options with a comma. For example:

```
OPTION SERIAL 123, BASE 1
```

This statement is equivalent to:

```
OPTION SERIAL 123
OPTION BASE 1
```

The options BASE and SERIAL may be placed anywhere in a program. The parameters they set are constant throughout the entire program, even before the OPTION statement is executed. When a CHAIN or LINK is performed, all programs must use the same BASE option. SERIAL is examined only in the first program module.

Except for BASE and SERIAL, all of the other options are executable and may be used to alter program execution as often as desired.

BASE
Sets the lower limit of all array subscript references throughout the program. When the option BASE is not used, the default is zero.

The OPTION BASE 0 statement is supported for documentation purposes and for compatibility with THEOS BASIC. It has no effect since zero is the default value.

OPTION BASE 1 is used mainly in programs that perform calculations with arrays that do not use subscript 0. This avoids having to allocate the small amount of extra storage the subscript 0 entries would use.

If OPTION BASE 1 is specified in a BASIC program, it must also be specified in all programs which LINK or CHAIN to it and in the programs which it LINKs or CHAINs to. Otherwise, an error will be detected.

BOOLEAN
Specifies that the AND, OR and NOT operators should be interpreted as binary operators. This is the default behavior.

Note that currently this option is only available in the CET DOS product.

CASE
Specifies the case mode for all characters input from the console. CASE may be specified with either an upper or lower-case letter as follows:

Case Mode	Function
"U"	Forces all letters that are input to upper case. This is the default mode.
"M"	No conversion is to be done on console input. Characters are echoed and transmitted in the same case as they are entered.
"L"	Converts all alphabetic characters that are input from the console to the opposite case (upper to lower and vice-versa) before they are echoed back to the console. The characters will also be transmitted in this (converted) way.

COMMA
Specifies that numbers should be input and displayed in the European format instead of the American format which is the default. This option is passed to any chained or linked programs.

Setting this option also the following effect:

	American Format	European Format
Decimal fractions	Periods are used to display decimal fractions such as 123.45.	Commas are used to display decimal fractions such as 123,45.
Field delimiters	Commas are used to separate input fields.	Semicolons are used as field separators.
Thousands	Commas are used to delineate thousands in output fields.	Periods are used to delineate thousands as in the number 4.123,99.

The environment variable B_COMMASEP is provided for our international developers who need to use the European format, but want to use commas as field delimiters instead of semicolons. (Currently, this variable is available only under DOS.)

Note that the standard American format must be used in the BASIC program itself and in masks for PRINT USING and the FORMAT$ function.

When option COMMA is executed, any further ASCII I/O operations such as PRINT and INPUT, and the FORMAT$, NBR, STR$ and VAL functions are affected.

DATEFORM
Specifies the date format to be used by the DATE$, DTE$, and DAY functions. A number from 1 to 3 will format dates as follows. Any other value will be ignored.

Refer to your *CET BASIC User's Guide* for information on the B_4DYEAR variable which may be set so that the DATE$ and DTE$ functions return a 4-digit year. The variable B_YR2000 may also be used to indicate which year should be the first one to be interpreted as 20XX.

DATEFORM	Format
1	American dates as mm/dd/yy - the default
2	European dates as dd/mm/yy
3	International dates as yy/mm/dd

DEFAULT
Specifies the type of subsequently declared or occurring variables with no type-specifying suffix (!, !!, %, or $) and of floating point constants. OPTION DEFAULT REAL is the default, and indicates that variables such as

ABC and X will be decimal real numbers and constants like 3.1415 will be stored as decimal.

OPTION DEFAULT REAL4 and REAL8 will cause the subsequent declarations of variables and constants to be short and long binary real numbers, respectively. Use caution if you change the DEFAULT option in the course of a program. (For more information on Binary real numbers, refer to the chapter *Elements of the BASIC Language*.)

Setting this option will have no effect in the CET BASIC DOS and W/32 products. It is supported under UNIX where binary real arithmetic is of interest to developers doing scientific and engineering calculations and the speed of computation, range of intermediate results and/or degree of precision is important.

DEGREE
Indicates that subsequent arguments to (inverse) trigonometric functions and their return values will be in degrees. (See option RADIAN.)

LOCK
Causes a suspension of the program for a specified number of seconds when an attempt is made to access a file or record that is locked by another user. If the file or record is still locked at the end of this interval, the program will either terminate with a trappable error or invoke the code associated with a current ON ERROR or ON LOCK routine.

The default lock time is 0, which means that the program will wait indefinitely until the file or record is released.

The syntax is:

OPTION LOCK *numeric-variable* | *constant*

The OPTION LOCK statement is *executable*. If multiple statements are used in a program, each execution of OPTION LOCK will change the lock interval.

Currently, OPTION LOCK is only available with the CET DOS/Networking product. For compatibility purposes, the statement will be recognized in the other CET products, but it will have no effect. In any case, we recommend that you use ON LOCK to define special handling of locked conditions.

LOGICAL
Specifies that all subsequent executions of the binary operators OR, AND and the unary operator NOT in a numeric expression will be performed with the

operators interpreted as logical operators. (Currently, this option is only available with the CET DOS/Networking product.)

Since OPTION LOGICAL is evaluated during compilation (as is OPTION BOOLEAN), its placement in the program dictates how the OR, AND and NOT operators are parsed. To ensure that these operators are interpreted correctly, the statements where they are used must be executed after an OPTION LOGICAL (or OPTION BOOLEAN) statement.

Refer to the *Elements of the BASIC Language* chapter to see how Boolean expressions are handled. The following tables, called truth tables illustrate the results of AND, OR and NOT operations when OPTION LOGICAL is set.

AND

A%	B%	A% AND B%
0	0	0
0	non-zero	0
non-zero	0	0
non-zero	non-zero	-1

OR

A%	B%	A% OR B%
0	0	0
0	non-zero	-1
non-zero	0	-1
non-zero	non-zero	-1

NOT

A%	NOT A%
0	-1
1	0

MATIO

Determines how all MAT I/O operations will be performed. OPTION MATIO 0 indicates that all subsequent operations will use the entire array, including subscript 0. The MAT PRINT or MAT WRITE statements will output any zero elements (row 0, column 0). Input statements will begin reading with element zero. This option will have no effect when OPTION BASE 1 is used.

For compatibility with UX-BASIC, a CET BASIC program will execute with this option in effect when it has been compiled with the -U switch.

OPTION MATIO 1 will cause all subsequent MAT input and output operations to ignore any zero elements. This behavior conforms to the ANSI standard. Although this the default setting, OPTION BASE 1 may be used with the same effect.

PROMPT
Specifies the prompt character displayed when an INPUT or LINPUT statement is executed. The default is to display a "? ". To change the prompt to a null string, enter OPTION PROMPT "".

RADIAN
Indicates that subsequent arguments to (inverse) trigonometric functions and their return values will be in radians. This is the default unless OPTION DEGREE is specified.

SERIAL
Provides a means of serializing a program so that it will only work with the CET BASIC Runtime System that has the same serial number stored in the Key Plug. By default, a CET application will work with any CET Key Plug

Note that OPTION SERIAL need only be specified in the first program of a CHAIN or LINK sequence. That means that only one program needs to be recompiled in order to use this feature. Other programs may be serialized to a non-existent serial number such as 1 to prevent a user from bypassing the serialization mechanism.

Example	Explanation
OPTION BASE 1	Set index base for arrays to 1.
OPTION CASE "M"	Accept input with no translation.
OPTION PROMPT CHR$(0)	No default prompt.
OPTION PROMPT ">> ", case "u"	The input prompting literal changed to the characters: >> followed by a space. Fold all input to upper case.
OPTION PROMPT "What? "	Prompt literal changed to What?

OTHERWISE

The OTHERWISE statement is used in a SELECT-CASE-CEND structure to mark the end of the CASE statements and specify the action to take when none of the cases are true.

Syntax OTHERWISE

Description The OTHERWISE statement is similar to CASE except that there is no expression specified. If none of the preceding CASE statements were true, then the statements following the OTHERWISE will be executed. If one CASE was true, control is transferred to the CEND statement which closes the SELECT structure.

The OTHERWISE statement follows the last CASE statement. No CASE statements will be evaluated after the OTHERWISE is encountered.

There should only be one OTHERWISE statement in any particular SELECT structure. Any additional statements will be ignored.

Restrictions A program should enter a SELECT-CASE-CEND structure at the top with SELECT and exit at the bottom with CEND. It should not branch out of the structure via a RETURN or RESUME statement, otherwise the results will be unpredictable. Although GOTO statements are allowed, a GOSUB is recommended when performing conditional operations.

Example

```
SELECT RAD*2.*PI
    CASE 0
        SELECT SUBVALUE%
            CASE 20
                ...
            CASE 32
                ...
            OTHERWISE
        CEND
    CASE I - 14
        ...     Perform only if RAD*2.*PI=I-14
    CASE J%
        ...     Perform only if RAD*2.*PI=J%
CEND
```

Explanation

```
                Perform only if RAD*2.*PI=0.
                Perform only if RAD*2.*PI=0 and SUBVALUE%=20.
                Perform only if RAD*2.*PI=0 and SUBVALUE%=32.
                Perform if RAD*2.*PI = 0 and neither of the above
                cases is true.
                End of inner SELECT structure.
                End of outer SELECT structure.
```

See Also CASE, CEND, SELECT

PRINT

The PRINT statement outputs ASCII data to a disk file or a physical device such as a console or printer.

Since formatting output is a major concern in most applications, this topic is covered in detail in a separate chapter later in this manual.

Syntax

1 PRINT [comma] [*expression punct*]...

2 PRINT *#channel*: [comma] [*expression punct*]...

3 PRINT *#channel, key*: [comma] [*expression punct*]...

channel = *numeric-expression*
key = *expression*
punct = *,*
= *;*

Description The PRINT statement evaluates expressions, converts their values to an ASCII text format and outputs the values to an open file or device.

Mode 1: Displays data on the console at the current cursor position.

Mode 2: Prints data to a sequential file or device opened for OUTPUT or UPDATE. A trappable error #35 (“Invalid access mode”) will be detected if this mode is used to print to an indexed or direct file.

Mode 3: Prints data to a record in a direct or indexed file opened for OUTPUT or UPDATE. In this case, the *key* to the record must be specified. Use a numeric key to access a direct file and a string expression with an indexed file. An error #35 is detected if this mode is used to print to a sequential file.

Note that the formatting operations described in this section only apply to a file or device opened for SEQUENTIAL access. When the access method is DIRECT or INDEXED, subsequent PRINT statements will overwrite the existing data, no matter what punctuation is used with PRINT.

Notes An output record is considered to be divided into *print zones* of twenty-one characters each. Punctuation characters may be used in the PRINT statement will determine whether extra space characters will be output.

If an expression is followed by a comma, enough spaces will be added to the data to move the cursor (or *print-head*) to the beginning of the next print zone. If the list of *expressions* includes an empty field between commas, a

whole print zone will be skipped and filled with spaces. A leading comma will have the same effect.

If the punctuation character is a semicolon, no extra space is added and the cursor is not moved.

Terminating the statement with either a comma or semicolon will cause the next PRINT statement to output text on the same line. If no punctuation is used, BASIC will output a line terminator character at the end of the printed record. Output from the next PRINT will be displayed on the following line.

For example:

```
PRINT “This text will appear on the first line.”
PRINT “This text will appear on the second line”;
PRINT “followed by this…”
```

displays:

```
This text will appear on the first line.
This text will appear on the second line followed by this…
```

Under DOS, the PRINT and MAT PRINT statements will automatically output a carriage-return+linefeed character as the record delimiter. By default, a carriage-return will be output under UNIX. If the file has been opened with the option NEWLINE (or the environment variable B_OPENNL has been set), the linefeed delimiter will be used to conform to the UNIX convention.

Note that there does exist the potential for program error when using the linefeed as a record delimiter. The problem occurs when the CRT$ (“DOWN”) string is included in the output. Because the *token* used to represent the ↓ key is the same as a linefeed (decimal 10), the key will also be interpreted as a record delimiter.

For compatibility purposes, option NEWLINE and the variable B_OPENNL are supported under DOS, but they will have no effect.

When mode 1 is used to PRINT to the console, the AT$ function may be used to position the cursor to a specific location of the screen. A semicolon is normally used to keep the cursor at that position. For example:

```
PRINT AT$(10,5);”This will appear starting in column 10, row 5.”
```

If the AT$ function is used to print to a device other than the console, the column and row values will be ignored.

The CLS$ and CRT$("clear") functions operate identically and may be used to clear the screen and position the cursor in column 1, row 1. In both cases, a semicolon should be used to terminate the expression and keep the cursor at that location.

The TAB function may be used with modes 1 and 2 to move the cursor (or *print-head*) to the specified column position. Note that use of TAB is generally meaningless unless it is followed by a semicolon.

TAB expects a numeric expression. The expression is evaluated and rounded to the nearest integer. Spaces are then added to the current line so that the next character printed will appear in the specified position. If the value of the TAB expression is less than the current position, a line-terminator character will be output and the cursor or *print-head* will be moved to the specified column on the new line. For example:

	Displays:
PRINT "123456789012345"	123456789012345
PRINT "A";TAB(5);"B";TAB(10);"C"	A B C
PRINT "A";TAB(5);"B";TAB(5);"C"	A B C

In general, the output rules for numeric expressions printed with this statement are:

- Leading and trailing zeros are suppressed.
- The decimal point is suppressed if the number can be represented as an integer.
- A maximum of thirteen digits are displayed.
- Numeric fields are output with a leading space or minus sign and a trailing space.

For example:

```
PRINT "12345678901234567890"
PRINT 1;-5;12.50;15.00
PRINT 1,-5,12.50,15.00
```

displays:

```
12345678901234567890
 1  -3   5   12.5  15
 1                -3                    5…
```

When mode 1 is used to PRINT to a printer, the CRT$ function may be used to send special control sequences. This topic is addressed at the end of the *Formatted Output* chapter.

Using modes 2 or 3 with a file or device that was opened with option QUOTE causes each expression (field) to be separated by a comma or a semicolon if OPTION COMMA is in effect. All string fields will be quoted if they contain leading, trailing or embedded spaces, commas, quotation marks or multiple spaces. The QUOTE option causes punctuation within the expression list to be ignored.

If the PRINT statement is directed to a device (such as a printer) opened with the option FORMAT, the ANSI forms control character must be specified at the start of each output line. This character is never printed. Any end of line punctuation will also be ignored.

The valid control characters are:

Character	Function
1	Starts a new page.
+	Stays on the same line (and over prints it).
0	Advances two lines.
-	Advances three lines.

The examples provided below are followed by the output they produce. Since formatting output is a major concern in most applications, this topic is covered in more detail in the *Formatted Output* chapter later in this manual.

Examples

```
LET A = 1.23 \ B = 34.56 \ C = 345.678
LET A$ = "ABCDEFG" \ B$ = "HIJKLMN" \ C$ = A$ + B$
PRINT "A =";A;"B =";B;"C =";C \ PRINT "A =";A;"B =";B;"C =";C
PRINT A+B+C,A*B,C*A,B/A,A/B \ PRINT A,,B
PRINT A;TAB(10);B \ PRINT A,TAB(10),B
PRINT A$;B$;C$; \ PRINT " ";B$; \ PRINT A$
OPEN #2: "DATA.FILE:A",OUTPUT DIRECT,QUOTE
PRINT #2,5: A$,B$,A,B
```

The output to the console is:

```
A = 1.23 B = 34.56 C = 345.678
A = 1.23 B = 34.56 C = 345.678
381.467999998 42.5087999999 425.183939999    28.0975609756
3.55902777778E-02
1.23           34.56
```

```
 1.23   34.56
 1.23
                  34.56
```

(The first comma causes a TAB to beyond column 10; so the output goes to column 10 of the next line. The next comma moves to the second zone.)

```
ABCDEFGHIJKLMNABCDEFGHIJKLMN HIJKLMNABCDEFG
```

The fifth record of the file is:

```
ABCDEFG,HIJKLMN, 1.23 , 34.56
```

See Also MAT PRINT, MAT WRITE, OPEN, PRINT USING, WRITE, Tab function. Also refer to the *Formatted Output* chapter for more information and examples.

PRINT USING

The PRINT USING statement writes formatted ASCII data to a disk file, a terminal or a device such as a printer.

Since formatting output is a major concern in most applications, this topic is covered in detail in *Formatted Output* chapter later in this manual.

Syntax

1 PRINT USING *mask* [, *expression-list*] [;]

2 PRINT #*channel*: USING *mask* [, *expression-list*] [;]

3 PRINT #*channel, key*: USING *mask* [, *expression-list*]

mask = *string expression*
expression list = *expression* [, expression-list]

Description **Mode 1**: Displays formatted output on the console starting at the current cursor position.

Mode 2: Prints formatted ASCII data to a sequential file or device opened for OUTPUT or UPDATE. A trappable error #35 ("Invalid access mode") will be detected if this mode is used to print to an indexed or direct file.

Mode 3: Prints formatted ASCII data to a record in a direct or indexed file opened for OUTPUT or UPDATE. In this case, the *key* to the record must be specified. Use a numeric key to access a direct file and a string expression with an indexed file. An error #35 is detected if this mode is used to print to a sequential file.

Notes The PRINT USING statement is similar to PRINT except that an explicit format *mask* is given for each expression. This feature makes it possible to output numeric data with a fixed number of digits after the decimal point (as for dollar amounts), floating dollar signs, etc. String fields may be justified, truncated or padded.

The *expressions* will be displayed in the order that they are listed and in the format specified in the *mask*. An *expression* may contain string or numeric functions as long as they correspond to the type specified in the *mask*. For details on the mask specifications refer to the *Formatted Output* chapter later in this manual.

All *expressions* must be separated by commas. A semicolon may be used as the terminating punctuation to suppress the output of a line terminator character.

Note that using the option QUOTE with the OPEN statement has no effect on the PRINT USING output. However, option FORMAT has the same effect as with PRINT.

Examples	Explanation
PRINT USING "$$$.##", X	Prints to the console.
PRINT #3: USING MASK$, A, B	Prints to sequential file 3 using a variable mask.
print #5, 14: using m$ & "##", a	Prints to direct file on channel #5, record 14 using a mask computed by a string expression.
PRINT #6, "ABC": USING "TAX = ###.##", RATE * COST;	Prints to indexed file on channel 6, record ABC, with no line separator at the end of the record.

See Also MAT PRINT, OPEN, PRINT, FORMAT$ function, and *Formatted Output* chapter

PUT

The PUT statement writes bytes of raw data to the console or an open sequential file or device.

Syntax

1 PUT *expression- list*

2 PUT *#channel : expression-list*

expression-list = *expression* [*, expression-list*]

Description **Mode 1**: Writes bytes of data to the console. When no *channel* is specified, PUT #0 is assumed.

Mode 2: Writes bytes of data to the sequential file or device open on the specified channel. PUT #0 will output to the console as in mode 1.

Notes Data is written as a stream of characters or bytes with no output conversion taking place. Disk files opened for OUTPUT or UPDATE SEQUENTIAL may be accessed with PUT. In fact, this is the only way that CET BASIC can write non-BASIC formatted files.

The actual output is determined by the type of the expression. If numeric, it will be converted to integer type if necessary and the least significant byte will be written. String expressions are output in their entirety.

Example	Explanation

FOR I%=1 TO LEN(STRING$)
  PUT STRING$[I%:I%]
  NEXT
	Repeat for length of string.
	Output one char of string.

PUT 13,10	Output carriage return, line feed to screen.
PUT 27, "]", "3", "7", "m"	Output ANSI control sequence to screen.

See Also GET, OPEN, PRINT

QUIT

The QUIT statement terminates a program and returns to the calling environment. All open channels will be closed.

Syntax QUIT [*expression*]

Description When the QUIT statement is used without specifying an *expression*, the return code is set to 0 as the program is terminated. This is equivalent to the statement QUIT 0.

If a *numeric-expression* is specified, the program returns to the calling environment with the return code set to the value of the expression. This must be a number in the range of 0 to 255.

When QUIT is executed with a *string-expression*, control does not return to the calling environment. Instead, the *string-expression* is interpreted as a command and is invoked. BASIC will remember the current (calling) environment and return to it after exiting from the command specified by QUIT.

Allowable commands depend upon the operating system in use. Any error detected by the operating system cannot be trapped and passed back to the BASIC program.

Note that BASIC is able to implement this QUIT "*command*" feature by starting up a new process. Because the first program does not terminate until the QUIT is completed, two processes are actually active at the same time. Normally, this does not cause a problem, but it is best to avoid using this procedure, if possible.

Examples	Explanation
QUIT	Exits BASIC with return code 0.
QUIT 3	Exits BASIC with return code 3.
QUIT "ls -l"	BASIC is exited and "ls -l" executed.
QUIT "dir memo.txt"	BASIC is exited and "dir memo.txt" executed.

See Also CSI/CSH, END, RUN, STOP

RANDOMIZE

The RANDOMIZE statement causes the pseudo-random number generator to be reseeded and start with a new sequence of numbers.

Syntax RANDOMIZE

Description The RANDOMIZE statement may be used so that the RND function will return a different set of *random* numbers each time the program is executed.

Note that the RND function does not produce true random numbers. By default, it starts from a *seed* value and uses each subsequent value to compute the next number. The RANDOMIZE statement causes this *seed* to start at a fairly random value each time the program is run.

RANDOMIZE should be performed only once (in the initialization routine). Because it starts the RND function based on the time clock, successive applications of RANDOMIZE in the same program will not be random with respect to each other.

Typically, programs are tested without using RANDOMIZE so that results will be consistent.

Example	**Explanation**
RANDOMIZE	Choose a random starting point.

Incorrect Example	**Explanation**
RANDOMISE	Misspelled.
RANDOMIZE I	No operands allowed.

READ

The READ statement reads data from DATA statements or from a file created with WRITE or one of its variants.

Syntax

1 READ *variable-list*

2 READ #*channel : variable-list*

3 READ #*channel , key : variable-list*

variable-list = *variable* [, *variable-list*]
channel = *numeric-expression*
key = *expression*

Description

Mode 1: Data is read from the DATA statements into the variables specified in the *variable-list.* If there are fewer DATA statements than there are variables, a trappable error #36 (Out of data) is detected. If there are more DATA statements after assigning the variables, they will be used by the next READ statement.

Mode 2: Reads a record from a sequential file. Each field in the record is assigned in consecutive order to the variables in the *variable-list.*

Mode 3: Reads a record from a direct or indexed file and assigns each field in the record, in consecutive order, to the variables in the *variable-list.*

Notes

The elements of the DATA statements are treated as if they were all defined in one long statement. When an element is read from the data *pool*, a pointer is advanced so that the next time an element is to be read it will be the next item in the data pool.

It is possible to reset the data pointer to the beginning or to any point in the pool by using the RESTORE statement.

If a *channel* is specified, each execution of a READ statement will read one entire record from the file (or device) opened for INPUT or UPDATE. If no *key* is specified, the channel must be opened for SEQUENTIAL access.

When a file is opened for DIRECT access, the *key* will be interpreted as the number of the record to be read. This number must be a positive integer. If the specified record does not exist, the variables in the variable list will be assigned null or zero values according to their data type. The EOF function will return a *true* value when reporting information about the channel.

When the file is opened for INDEXED access, the *key* is interpreted as the key of the record to be read and must be a string. If there is no record in the

file with this key, the variables in the variable list will be assigned null or zero values and the EOF function will return *true* for the channel. At this point, a READNEXT statement may be used to return the next available record. This feature allows you to position the pointer for a sequential reading.

When the READ statement is used to read a file, it expects the records to be in a binary format written with a WRITE statement. ASCII records written with PRINT or a variant may not be read. Attempting to do so will cause the variable in the *variable-list* to be cleared to nulls.

The data fields in the record and the variables in the *variable list* should be of the same type, both strings, integers or numerics. When there is a mismatch, the following conversions are performed:

Field Type	Variable Type	Conversion
Nonnumeric string	Integer	Set to zero
Nonnumeric string	Numeric	Set to zero
Numeric string	Integer	String to integer
Numeric string	Numeric	String to float
Integer	String	Integer to string (STR$)
Integer	Numeric	Integer to float (FLOAT)
Numeric	String	Float to string (STR$)
Numeric	Integer	Float to integer (FIX)

Reading a record in an indexed or direct file opened for UPDATE will automatically *lock* the record. If another program attempts to access the record, that program is suspended, with periodic retries until the record is unlocked unless there is an ON ERROR or ON LOCK routine in effect. These statements may be used to specify other behavior for a locked record condition.

If the B_EMULATE variable is set, a locked record is not considered a trappable error. If you wish to use THEOS emulation and special error handling, set the B_THLOCK variable off (to null).

Note that if an ON ERROR routine exists, but does contain any code to handle a locked record (error #48), the program will abort with an appropriate error message.

For further information on record and file locking, refer to the *CET BASIC Error Handling System* chapter. The environment variables B_EMULATE and B_THLOCK are covered in the *CET BASIC User's Guide.*

Examples	**Explanation**
100 DATA 1.23, 2.34	
READ A	A is set to 1.23.
READ B, C	B is set to 2.34 and C is set to 3.45.
RESTORE 900	Take DATA starting on line 100.
READ A$	The string "1.23" is assigned to A$.
RESTORE 800	Next data element will come from line 800.
READ B$, C$	B$ is set to "3.45" and C$ is set to "LITERAL".
800 DATA 3.45, LITERAL, 2ND LITERAL	
OPEN #1: "DATA\FILE", OUTPUT DIRECT	
WRITE #1, 18: "123AB", 1.23, 12	
OPEN #1: "", INPUT DIRECT	
READ #1,18: A, B%, C$, D$	A set to 123, B% to 1, C$ to "1.23", and D$ to "".

See Also DATA, INPUT, MAT READ, OPEN, READNEXT, READPREV, RESTORE

READNEXT

The READNEXT statement reads the next record from an indexed file.

Syntax READNEXT #*channel, key : variable-list*

Description The READNEXT statement accesses indexed files by the key to the record. This statement is similar to READ except that it only operates on a file opened for INDEXED access and the *key* must be a string variable, not an expression. Type conversion takes place as in the READ statement.

Refer to the READ section for information on record locking. In addition, note that when using an ON ERROR or ON LOCK routine to handle record locks, the environment variable B_OPTLOCK must be set so that the file pointer is not moved after the RESUME statement. This way, READNEXT will attempt to read the desired record again.

Notes When READNEXT is executed, the record read is the one whose key is the next (in sequence) one after the last key specified by a READ statement or the last key returned by a READNEXT. If the first access to the file is via a READNEXT statement, the first logical record in the file will be read.

The key of the record read is returned in the *key* variable. If no record exists, *key* will be set to a null string and the EOF function will return *true*. A subsequent READPREV statement will read the last record in the file.

READNEXT can only be used to access files created with WRITE or one of its variants. Attempting to read ASCII formatted records created by PRINT or MAT PRINT will set the variables in the *variable-list* to null.

Example **Explanation**

If an indexed file contains records with the following keys:

000100
001001
003234
003235

then the following statements will print the string "003234"

```
READ #1, "002000": A$          Position between records 001001 and 003234.
READNEXT #1, KEY$: A$          READs record 0032340
PRINT KEY$                     Displays 0032340
```

See Also INPUT, MAT READ, MAT READNEXT, MAT READPREV, OPEN, READ, READPREV, and EOF function

READPRIOR
READPREV

The READPREV and READPRIOR statements are synonyms. Either statement may be used to read the previous record from an indexed file. For brevity, only READPREV will mentioned here.

Syntax READPREV *#channel , key : variable-list*

Description READPREV accesses indexed files sequentially in reverse order by key. This statement is similar to READ, however it only operates on a file opened for INDEXED access and the *key* must be a string variable, not an expression. Type conversion takes place as in the READ statement.

Refer to the READ section for information on record locking. In addition, note that when using an ON ERROR or ON LOCK routine to handle record locks, the environment variable B_OPTLOCK must be set so that the file pointer is not moved after the RESUME statement. This way, READPREV will attempt to read the desired record again. If this variable is not set, BASIC will move on to the preceding record, by default.

Notes When READPREV is executed, the record read is the one whose key is the one prior (in sequence) to the last key specified in a READ statement or the last key returned by READPREV. If the first access to the file is via a READPREV statement, the last logical record in the file will be read.

The key of the record read is returned in the *key* variable. If there is no previous record, then *key* will be set to a null string and the EOF function for this channel will return *true*. A subsequent READNEXT statement will read the first record in the file.

READPREV can only be used to access files created with the WRITE statement or a variant. Attempting to read ASCII formatted records created by PRINT or MAT PRINT will set the variables in the *variable-list* to null.

Example **Explanation**

If an indexed file contains records with the following keys:

000100
001001
003234
004000

then the following statements will print the string "001001":

```
READ #1, "003200": A$          Position between records 001001 and 003234.
READPRIOR #1, KEY$: A$         READ record 001001
PRINT KEY$                     Displays 001001

READ #2, CHR$(127): DUMMY$
READPRIOR #2, LAST$: A$        Reads the last record in the file.
```

See Also MAT READ, MAT READNEXT, MAT READPREV, READ, READNEXT

REM

The REM statement may be used to insert a remark into a BASIC program.

Syntax REM [*unquoted-string-literal*]

Description REM statements may be used anywhere that a BASIC statement is allowed. When encountered, control will be passed to the next line in the program.

If the first three characters of a statement are REM, it is considered a remark. Therefore, REM, REMARK and REMITTANCE all serve to introduce comments if they begin a statement.

A semicolon character may be substituted for the REM keyword when it is the first character in the statement.

Since everything following the REM statement is ignored, it must always be the last statement on a line. A blank line is also treated as a remark and is ignored.

Restrictions A REM statement may not be entered on a line with a DATA statement since BASIC has no way of distinguishing the text in the remark from the data that should be read.

Examples	**Explanation**
REMARK: This is a remark REM: This is a REMARK	
LET A = B \ rem This is a remark	Recommended syntax for using a REM on the same line as a statement.

Incorrect Examples	**Explanation**
DATA 1,2,3,4,5 \REM ABCDEF	The REM will be treated as a DATA element.
LET A=B REM This is a remark	Statement separator missing.
REMIT = 0.05 * TAX	Programmer probably didn't intend this to be a remark.

RESTORE

The RESTORE statement may be used to reset the position of the pointer in the DATA statement *pool*.

Syntax RESTORE [*line-number*]

Description The RESTORE statement is used to reset the read pointer to the top of program or to a line greater than or equal to the specified *line-number*. This statement makes it possible to reuse or skip elements in DATA statements.

Initially the DATA read pointer is set to the top of the program. With each READ or MAT READ statement, the pointer is reset to point to the next available DATA statement. When there are no more statements, the next time a READ or MAT READ is executed an "Out of Data" error will be detected.

The RESTORE statement may be used to reset the DATA pointer so that READ and MAT READ statements can reread the data items.

Example	Explanation
RESTORE	The next READ will read the first data element of the first DATA statement in the program.
RESTORE 9000	The next READ will read the first data element of the first DATA statement whose line number is 9000 or larger.

See Also DATA, MAT READ, READ

RESUME

The RESUME statement terminates a user-defined error or key-handling routine and specifies what to do next.

Syntax

1 RESUME

2 RESUME 0

3 RESUME *line-reference*

Description

Mode 1: Control is passed back to the statement that was being executed when the error was detected. That statement is then reexecuted except for when an ON ERROR or ON INTERRUPT routine has been used to process the interrupt key. Since the interrupted statement will already have finished executing, RESUME will transfer control to the next statement in the program.

Mode 2: Control is transferred to the standard error-processing mechanism. BASIC will display the appropriate error message and terminate as if no error-handling routine had been specified.

Mode 3: The RESUME statement terminates error processing and passes control to the *line-reference*.

Notes

RESUME should only be used to exit from a routine that handles special processing for ON ERROR, ON INTERRUPT, ON LOCK, ON MOUSE or ON KEY statements. Otherwise, the results will be unpredictable.

Until the RESUME statement is executed, BASIC will ignore any other errors or events that may occur in the program

Examples	**Explanation**
ERROR.ROUTINE:	
SELECT ERR	
CASE 1	
RESUME INTRPT.ROUT	Process Interrupt Error.
CASE 22	
RESUME	Retry the statement that caused the error.
OTHERWISE	
RESUME 0	Give up. Let BASIC handle the error and abort.
CEND	

See Also

ON ERROR, ON INTERRUPT, ON LOCK, ON KEY

RETURN

The RETURN statement returns after performing a GOSUB or ON GOSUB statement.

Syntax

1 RETURN
2 RETURN *line-reference*

Description

Mode 1: Returns control to the statement following the GOSUB or ON GOSUB statement that called the subroutine.

Mode 2: Control is transferred to the specified *line-reference.*

Notes

A subroutine is an unstructured sequence of BASIC statements that should be entered at the top and exited at the bottom with a RETURN statement. Although BASIC will not detect an error if a subroutine contains multiple RETURN statements, it is not recommended.

GOSUB and RETURN statements are paired at runtime. A RETURN corresponds to the most recently performed GOSUB for which no RETURN has been executed. If no statement exists, a trappable error is generated.

Restrictions

Mode 1 should always be used to RETURN from a subroutine invoked by a programming structure such as a SELECT-CASE-CEND. Using a *line-reference* to RETURN to some other place will corrupt the program stack.

The stack of return points from successively executed GOSUB statements is not maintained when a CHAIN, LINK or RUN is executed.

A runtime error is detected if GOSUB statements are nested more than 32 levels. It is assumed that any deeper nesting is due to a programming error.

Examples	**Explanation**
GOSUB ROUTINE	Execute subroutine at label ROUTINE.
PRINT A$ \ GOTO 9000	Statements executed after RETURN.
ROUTINE:	Beginning of subroutine.
PRINT "I am here"	Body of subroutine.
RETURN	Exit subroutine.
GOSUB IN.ROUT	Execute subroutine a label IN.ROUT.
IN.ROUT: Input subroutine	Beginning of subroutine.
IF invalid data found	
RETURN CLOSE.UP	Error exit.
ELSE	
RETURN	Regular return.
IFEND	

See Also

GOSUB, ON GOSUB

RUN

The RUN statement terminates execution of the current program and starts executing another one.

Syntax RUN "*program-name*"

Description The RUN statement closes all open files, clears all variables, and executes the program specified in the *program-name*. The following chart illustrates the differences between the program execution statements:

	CHAIN	LINK	RUN	CSI	QUIT	END
Close all files	X		X		X	X
Clear all COMMON data			X		X	X
Clear all non-COMMON data	X	X	X		X	X
Clear current ON ERROR traps	X	X	X		X	X
Clear all ON KEY traps	X	X	X		X	X
Exit open program structures	X	X	X		X	X
Execute another program	X	X	X	X	X	
Return to calling program				X		
Reset all OPTIONs			X	X	X	X

The RUN statement is virtually identical to terminating the execution of the current program and running a new program from the command line. The only difference is that RUN will maintain current display characteristics and minimize setup time for the new program.

Restrictions The *program-name* must be a valid name in a form recognized by the operating system in use. The name should be quoted. If the program cannot be found, a trappable error is generated.

Only use RUN to execute a BASIC program, otherwise the console may be left improperly set. Use QUIT to transfer to a non-BASIC program.

Example	**Explanation**
RUN "JOE"	Execute program named "JOE".
Incorrect Example	**Explanation**
RUN PROGRAM	Program name must be an expression.
RUN "PROGRAM" LABEL	Line labels not allowed - must start at the beginning.
RUN	Program name must be specified.

See Also CHAIN, CSH/CSI, LINK, QUIT

SELECT

The SELECT statement defines and starts a SELECT-CASE-CEND programming structure to perform the conditional execution of statements.

Syntax

1 SELECT

2 SELECT *expression*

Description

The SELECT-CASE-CEND structure may be used to replace multiple IF-THEN-ELSE statements or ON GOTO or ON GOSUB statements to improve program readability.

Mode 1: Starts a multi-branching structure where each of the associated CASE statements specifies the complete relational expression to be tested. The statements following CASE will be executed only when the result of the test is true (non-zero). The general form of the structure is:

```
SELECT
    CASE relational-expression
        statements
    CASE relational-expression
        statements
    .
    .
    [OTHERWISE
        statements ]
CEND
```

Mode 2: Starts a multi-branching structure where the SELECT statement includes the left part of the relational *expression.* If the *expression* in the CASE statement is equal to the one specified by SELECT, then the statements following the CASE are executed. For example:

```
SELECT var%
    CASE 1
        statements
    CASE 2
        statements
    .
    .
    [OTHERWISE
        statements]
CEND
```

Notes

In either mode, if the result of a CASE statement is false (zero), the statements following the CASE are skipped until another CASE, OTHERWISE or CEND is encountered.

When the CASE is true, the statements following the CASE are executed, in order until another CASE, OTHERWISE or CEND statement occurs at the same nesting level.

If no CASE evaluates to true, the statements following OTHERWISE, if any, are performed.

Every SELECT statement must have one (and only one) closing CEND statement where normal program execution is resumed.

Restrictions This statement is part of a programming control structure. The program should enter at the top with SELECT and exit at the bottom with CEND. Although a GOTO statement may be used to branch out of a CASE, a GOSUB is recommended since it provides more structure and readability to a program.

In mode 2, both the SELECT and CASE expressions must be of the same data type (string or numeric) or a syntax error will be reported.

Any statements inserted between SELECT and the first CASE statement will never be executed.

SELECT structures may be nested and may contain or be contained in other complex structures such as an IF-THEN-ELSE or WHILE-WEND. As usual, proper nesting rules must be followed or the compiler will detect an error.

Examples	**Explanation**
SELECT PI * RADIUS^2	Define VALUE.
CASE 0	
SELECT	Perform only if VALUE=0.
CASE SUBVALUE% = 20	
...	Perform only if VALUE=0 and SUBVALUE%=20.
CASE SUBVALUE% > 32	
...	Perform only if VALUE=0 and SUBVALUE%>32.
CEND	End of inner SELECT structure.
	Go here if VALUE=0 and SUBVALUE% not = 20 and SUBVALUE% <32.
CASE I-14	
...	Perform only if VALUE=I-14.
OTHERWISE	
	Perform only if VALUE not=0 and not = I-14.
CEND	End of outer SELECT structure.

See Also CASE, CEND, OTHERWISE

SLEEP

The SLEEP statement suspends program execution for a specified amount of time. This feature is useful when it is necessary to wait for a short interval between screen displays or program events.

Syntax SLEEP *numeric-expression*

Description When the SLEEP statement is encountered, the *numeric-expression* is evaluated and execution is suspended until the specified time has elapsed or the program is aborted.

The *numeric expression* must be a positive value, otherwise the statement will be ignored. Decimal fractions are allowed.

The smallest usable time period depends on the operating system and hardware in use, but it is seldom less than 10 to 20 milliseconds.

Example	Explanation
SLEEP 10	Suspend processing for 10 seconds.
SLEEP X/4	Wait for one fourth of X value.

See Also WAIT

STOP

The STOP statement terminates program execution. This feature is available for testing and debugging purposes.

Syntax

1 STOP

2 STOP *expression*

Description The STOP statement terminates execution of the program similar to mode 2 of the QUIT statement. The return code is set to 253.

Mode 1: Stops program execution and displays the message “Stop”.

Mode 2: Stops program execution and displays the message “Stop” along with the value of the *expression*.

Notes When testing a program with multiple STOP statements, use mode 2 to indicate the location or the value of some critical variable when the program stopped.

STOP is normally used for abnormal program termination. QUIT or END should be used for normal termination.

Example	Explanation
STOP	Program stops execution.
STOP A$	Program stops execution and displays the current value of A$.
STOP "Test Complete"	Program stops execution and prints the string "Test Complete".

See Also END, QUIT

SWAP

The SWAP statement may be used to exchange the values of two variables.

Syntax SWAP *variable1, variable2*

Description The SWAP statement exchanges the value of *variable1* with the value of *variable2.*

SWAP performs the same operation that is executed with the following statements:

```
TEMP.VAR$ = FIRST$
FIRST$ = SECOND$
SECOND$ = TEMP.VAR$
```

Both variables must be of the same type; strings, integers or numerics. They may be simple variables or array elements.

Example	Explanation
SWAP FIRST$,SECOND$	Will yield the same result as the statements above.

See Also LET

THEN

The THEN statement is used to specify the action(s) to take when an IF condition is true.

Syntax

1 THEN

2 THEN *statement*

3 THEN *line-number*

Description When the result of an IF statement is true (non-zero), the statement(s) following THEN will be executed. If the result, the THEN statements will be skipped.

Mode 1: Indicates the beginning of the lines that are conditionally executed as part of a multi-line IF-IFEND programming structure. When the IF statement is true (non-zero), the statement(s) following THEN will be executed until a matching ELSE or IFEND is encountered. If the result of the IF condition is false, the THEN statements will be skipped.

The keyword THEN is optional, but should be used to improve readability.

Mode 2: Indicates that control should be transferred to the specified *statement* when the IF condition is true. As in mode 1, the use of THEN is optional, but should be used to improve readability.

Mode 3: Indicates that control should be transferred to the specified *line-number* when the IF condition is true. This statement is equivalent to THEN GOTO *line-number*. In this mode, THEN is required.

Examples	**Explanation**

IF A
  THEN
  GOSUB NON.ZERO
  ELSE
  PRINT USING "###",A
IFEND
	Used alone on a line for readability.

IF VALUE > CONTROL
  THEN IF TOL > LIMIT THEN GOSUB ERROR
  ELSE QUIT
IFEND
	THEN is also used here for readability.
	
	This ELSE goes with IF VALUE...

IF X < 14
  THEN 200
  ELSE
  GOSUB BIG.SUB
  PRINT MSG$
IFEND
	THEN required here.

IF SEL$="" THEN GOSUB GET.SEL	THEN used for readability. No ELSE is performed.

Incorrect Examples	**Explanation**
IF VALUE>5 THEN 100 THEN PRINT "XYZ"	Not in a multi-line IF statement.
IF ERROR > TOL THEN ERR.ROUTINE	Label not allowed. Must use line number or THEN GOTO ERR.ROUTINE

See Also ELSE, IF, IFEND

UNGET

The UNGET statement may be used to put characters into the console's input stream buffer.

Currently, this feature has not been implemented in all the CET products. Please check with your CET distributor about its future availability.

Syntax

1 UNGET *expression-list*

2 UNGET *#channel: expression-list*

Description

Mode 1: Puts one character for each expression in the *expression-list* into the console's input buffer.

Mode 2: Puts one character for each expression in the *expression-list* into the input buffer for the specified file *channel*. This file must be open for INPUT SEQUENTIAL.

At least one character will be placed into the input buffer. Normally, the console can UNGET from 1 to 8 characters, depending upon the operating system in use. Disk files may be able to UNGET from 1 to 32,767 characters while the buffer for a communications device may only allow one character.

Notes

Until a character is read from the buffer, another UNGET may not have any effect on the contents of the input buffer.

Example	**Explanation**
GET A$	
IF ASC(A$) > 32	Check for valid input.
UNGET A$	Put the first character back.
LINPUT B$	Obtain the entire string.
ELSE	
PRINT "Invalid Entry"	
IFEND	

See Also

GET

UNLOCK

The UNLOCK statement releases the lock on any records in the file open on the specified channel.

Syntax UNLOCK *#channel*

Description All records locked on the specified file channel are released.

Notes Record locking is performed automatically whenever records are read from a indexed or direct file opened for UPDATE. Statements such as INPUT, LINPUT and READ will lock the record being read to prevent other users from using the same record. Record locking is removed when the file (channel) is closed.

The UNLOCK statement may be used if the program *knows* that the record is no longer needed and/or another program needs to access the record for read-only purposes.

If a file has been opened with the LOCK option, this statement will not release the lock on the file. That may only be done by closing the file.

Example	**Explanation**
READ #1, KEY$: BAL	Reads the KEY$ record and locks it.
IF BAL > 1000	
WRITE #1, KEY$: BAL+BONUS	Releases the record by rewriting it.
ELSE	
UNLOCK #1	Releases the record without rewriting.
IFEND	

See Also INPUT, LINPUT, MAT INPUT, MAT PRINT, MAT READ, MAT READNEXT, MAT READPREV, MAT WRITE, OPEN, PRINT, PRINT USING, READ, READNEXT, WRITE

WAIT

The WAIT statement waits for a key to be entered by the operator or a specific character to be available on the input channel.

Syntax

1 WAIT

2 WAIT #*channel*

Description

Mode 1: Performs a *page wait* on the console. The cursor is displayed in the lower-left corner of the screen on top of a carat symbol (^). Execution is suspended until a character is entered. Typically, this is done in order to give the operator time to read what is on the screen.

Mode 2: Suspends execution until one or more bytes of data are available for input on the specified *channel*. Although this mode is valid for all devices, it is normally used on a console and not disk files.

WAIT #0 is equivalent to WAIT except that the carat is not displayed. The program merely waits for a character to be input from the console.

Notes

The file on the specified channel must be open for INPUT SEQUENTIAL or UPDATE SEQUENTIAL. The console is always assumed to be open on channel #0.

Example	**Explanation**
OPEN #1: "CON", INPUT SEQUENTIAL	Opens the console for input.
WAIT #1:	Waits for first character.
LINPUT #1: RECORD$	Gets line of data.
FOR I%=1 TO PAGE(0) - 1 PRINT A.LINE$(I%) NEXT I%	Displays a screen full of data.
WAIT	Waits for operator to release the page.
WAIT #0	Wait for next character from console...
GET A%	and GET it.

See Also

GET

WEND

The WEND statement marks the end of a WHILE-WEND control structure.

Syntax WEND

Description The WEND statement is part of a looping structure that begins with WHILE and ends with WEND.

When the program encounters a WEND statement, the condition in the corresponding WHILE statement is evaluated. If the condition is true, the statements following the WHILE are executed. If the condition is false, execution continues at the statement following the WEND.

The BREAK statement may be used to exit a WHILE-WEND loop and transfer control to the statement following the WEND in the current structure.

WHILE-WEND structures may be nested.

For more information and examples, refer to the description of the WHILE statement.

See Also BREAK, WHILE

WHILE

The WHILE statement marks the beginning and sets the conditions for a WHILE-WEND programming structure.

Syntax WHILE *expression*

Description The WHILE statement begins a WHILE-WEND structure. During compilation, each WHILE is paired with a unique WEND statement which must physically follow it in the program.

When a WHILE statement is encountered, the *expression* is evaluated. If it is true (non-zero), the following statements are executed, up to the corresponding WEND. Otherwise, execution continues at the statement following the WEND.

Execution of the WEND statement returns control to the WHILE statement where the *expression* is reevaluated. Therefore, it is important that the statements between WHILE and WEND change the value of the *expression* or there is some other means of exiting the loop. Otherwise, an infinite loop will result.

The BREAK statement may be used to exit a WHILE-WEND loop and transfer control to the statement following the current WEND.

WHILE-WEND structures may be nested.

Examples	**Explanation**
WHILE NOT EOF(3)	
READ #3: A	
IF A = -99 THEN BREAK	End-of-data sentinel.
WHILE A > B	Won't be done at all if A initially <= B.
A = A - 1	
WEND	End inner loop.
WEND	End outer loop.

Incorrect Example	**Explanation**
WHILE A > .0003	
IF X < 12 THEN WEND	Structures not nested properly. WHILE can have
A = A / 2	only one WEND, which is determined at compilation.
WEND	

See Also BREAK, WEND

WRITE

The WRITE statement outputs binary formatted records to an open disk file.

Syntax

1 WRITE *#channel: expression-list*

2 WRITE *#channel, key: expression-list*

channel = *numeric-expression*
expression-list = *expression* [, *expression-list*]
key = *expression*

Description

Mode 1: Writes the values in the *expression-list* to a sequential disk or tape file. Records are always appended to the end of the file.

Mode 2: Writes the values in the *expression-list* to a direct or indexed disk file. The record is written in the position indicated by the *key*.

Notes

The WRITE statement outputs fields in a special BASIC format. Each field has a preceding code that specifies its data type so that READ statements can perform the proper input conversion and access the data.

Code	Data Type	Length
1	Integer value	3
2	Numeric value	9
3	String value	2+string length
4	End of record	1

Restrictions

The *channel* must refer to a file opened for OUTPUT or UPDATE. For information on opening files refer to the OPEN statement. The types of files that can be accessed are covered in the *CET BASIC Data Files* chapter.

When the file is opened for DIRECT access, the *key* will be interpreted as the number of the record to be written. This number must be a positive integer. If the *key* value is greater than the currently allocated number of records, the file will be extended so that the record can be written. When the file is opened for INDEXED access, the *key* is interpreted as the key of the record to be written. It must be a string expression.

Each execution of a WRITE statement will write one entire record to the file. If a record with the specified *key* already exists, it will automatically be replaced.

Direct and indexed files have fixed record lengths that are determined when the files are created. Attempting to write a record longer than the allocated length will result in the trappable error #47 (“Truncated record on Write”).

The WRITE statement automatically unlocks any record which was locked by the program unless the file was opened with option LOCK.

Since the WRITE statement writes binary formatted records, only the READ statement and its variants may be used to access them. Attempting to use an INPUT or LINPUT statement will cause an error.

Examples

```
OPEN #1: "DATA.FILE", OUTPUT SEQUENTIAL
OPEN #2: "CUSTOMER.MASTER", UPDATE INDEXED
OPEN #3: "AREA.CODES", UPDATE DIRECT
WRITE #1: DATA1, DATA2, STRING$, 1*34+5
WRITE #2, NAME$: ADDR$, CITY$, STATE$, FORMAT$(ZIP,"99999"), BALANCE
WRITE #3, 212: "New York City"
```

Incorrect Examples	Explanation
WRITE #1,23: A,B,C,D	Not valid for sequential access.
WRITE #2: A$,BETA$,C	Indexed access requires a string key.
WRITE #3,"REC"&STR(I%): A,B	Direct access requires a numeric key.

See Also MAT READ, MAT READNEXT, MAT READPREV, MAT WRITE, OPEN, READ, READNEXT, READPREV

CHAPTER 6

Built-In Functions

Introduction

CET BASIC provides a wide variety of built-in or intrinsic functions that may be used for numeric computation, string manipulation, data type conversion, display control, and error detection.

In addition to the built-in functions described in this chapter, there are some functions that have been designed to retrieve specific system, file and program related information. Since these functions depend upon the operating system in use, they are covered in the *CET BASIC User's Guide.*

A variety of video, bar code, and communication features are also available with the external libraries that are documented in the *CET BASIC Library Manual.* (These libraries may be optional add-on products under some platforms.)

The Window System is an optional package that includes a complete set of functions that handle the screen display so you can quickly add visual appeal and increased functionality to your application. Contact your CET distributor if you need more information about the Window System or any of the CET BASIC Libraries.

The built-in CET functions differ from any of the other functions in that they do not require the CALL statement to be invoked. Some guidelines for using the built-in functions are:

- The names of the functions are reserved words and may not be used as variable names.
- The terminating '$' character in the name of a string function is optional. For example, SPACE(5) is treated the same as SPACE$(5).

- The operation of trigonometric functions is determined by the setting of OPTION RADIAN (the default) or OPTION DEGREE.

Alphabetical List of Built-in Functions

Each of the CET BASIC functions are described in this chapter. Where appropriate, one or more short examples are given.

For clarity, the following conventions have been used:

- The term *num-exp* stands for an arbitrary numeric expression.
- The term *string-exp* stands for an arbitrarily complex string expression.
- The string A$ in the examples is assumed to have the value "ABCDEFGHIJKLMNOPQRSTUVWXYZ", unless defined otherwise

A chart listing all of the built-in functions along with a brief description has been included at the end of this chapter for your convenience.

ABS(*num-exp*)

The ABS function evaluates the *num-exp* and returns its absolute or unsigned value. The *num-exp* may be either an integer or a numeric, but the returned value is always a numeric.

Example:

```
PRINT ABS(23);ABS(-23)

23    23
```

See Also: SGN

ACOS(*num-exp*)

The ACOS function returns the arccosine or inverse cosine of the *num-exp*. The *num-exp* is evaluated and interpreted as the cosine of an angle. The angle is computed and returned. If OPTION DEGREE is in effect, the value is in degrees. Otherwise, it is in radians.

Arccosines have values from 0 to π radians or 0 to 180 degrees.

ACOT(*num-exp*)

ACOT returns the arccotangent or inverse cotangent of the *num-exp*. The *num-exp* is evaluated and interpreted as the cotangent of an angle. The angle

is computed and returned. If OPTION DEGREE is in effect, the value is in degrees. Otherwise, it is in radians.

Arccotangents have values from 0 to π radians or 0 to 180 degrees.

ACSC(*num-exp*)

ACSC returns the arccosecant or inverse cosecant of the *num-exp*. The *num-exp* is evaluated and interpreted as the cosecant of an angle. The angle is computed and returned. If OPTION DEGREE is in effect, the value is in degrees. Otherwise, it is in radians.

Arccosecants have values from 0 to π radians or 0 to 180 degrees.

ADDROF(*variable*)

The *address of* function is used with the CALL statement to pass the memory address of a variable, so that the value *variable* can be modified by the subroutine. The variable may be the name of a simple string or numeric variable or a specific array element.

A subroutine evaluates the address of the variable and assigns it a new value. This value is then returned to the calling BASIC program.

Examples:

```
CALL BW.OPEN(ADDROF(WIN%),5,10,65,10,1)
CALL BW.SELECT(WIN%)
```

These lines of code call functions available in the optional Window System package. The first CALL statement opens a new window called WIN% with a specified location, size, and frame type. The ADDROF function is used to return the value assigned to the memory address of WIN% so that it may be used to reference the window in other functions such as BW.SELECT.

See Also: MATADDROF function, CALL statement

ANGLE(*num-exp1,num-exp2*)

ANGLE returns the arctangent of the ratio between two tangents. The numeric expressions are evaluated and interpreted as the two legs of a right triangle. The arctangent of *num-exp1* is divided by *num-exp2* and returned. If OPTION DEGREE is in effect, the value is in degrees. Otherwise, it is in radians.

ASC(*string-exp* [,*byte*])

The string expression is evaluated and the ASCII value of the first character in the resulting string is returned as an integer. An optional second parameter may be used to indicate a byte number of the string.

Refer to the *Appendix* for a list of the ASCII code values.

Example:

```
A$ = “ABCDEFG” \ PRINT ASC(A$)
65

A$ = “abcde” \ PRINT ASC(A$) \ PRINT ASC(A$,1) \ PRINT ASC(A$,2)
97   97   98
```

In the above example, the first character of A$ is evaluated. 65 is the ASCII value of the first character of the string (“A”).

See Also: CHR$, ORD

ASEC(*num-exp*)

ASEC returns the arcsecant of the *num-exp*. The *num-exp* is evaluated and interpreted as the secant of an angle. The angle is computed and returned. If OPTION DEGREE is in effect, the value is in degrees. Otherwise, it is in radians.

Arcsecants have values from 0 to π radians or 0 to 180 degrees.

ASIN(*num-exp*)

ASIN returns the arcsine of the *num-exp*. The *num-exp* is evaluated and interpreted as the sine of an angle. The angle is computed and returned. If OPTION DEGREE is in effect, the value is in degrees. Otherwise, it is in radians. Arcsines have values from 0 to π radians or 0 to 180 degrees.

AT$(*num-exp1,num-exp2*)

Returns a string which, if output to the console, will position the cursor at column *num-exp1* and row *num-exp2*.

The value of the AT$ function is the actual cursor positioning escape sequence. The cursor position will be returned if the column and row numbers are valid for the console.

The upper left-hand corner of the screen has the coordinates 1,1. Any coordinate values less than one or greater than the maximum number of columns and rows will cause the function to return a null string.

The AT$ function will have no effect unless it is used in a PRINT statement.

Example:

```
PRINT AT$(LINE(0)/2, PAGE(0)/2);
```

This example will position the cursor in the center of the console screen.

See Also: CRT$

ATAN(*num-exp*)

ATAN returns the arctangent of the *num-exp*. The *num-exp* is evaluated as the tangent of an angle. The angle is computed and returned. If OPTION DEGREE is in effect, the value is in degrees. Otherwise, it is in radians.

Arctangents have values from 0 to π radians or 0 to 180 degrees.

ATN(*num-exp*)

This is a synonym for the ATAN function.

BIN(*string-exp*)

The BIN function analyzes a string expression that contains a binary number of up to 16 digits and returns its numeric value as an integer from –32768 to +32767. BIN is the complement of the BINOF$ function.

Example:

```
PRINT BIN("0101010101010101");BIN("0000111100001111")
```

```
21845      3855
```

BINOF$(*num-exp*)

BINOF$ is the inverse of the BIN function. It evaluates the *num-exp* and converts it to an integer, if necessary. The value is then translated into a string of characters representing its value in binary. A 16-character string of binary digits (0 and 1) is always returned.

Example*:*

```
PRINT BINOF$(123);"   ";BINOF$(23129)
000000001111011   0101101001011001
```

BFLOAT(*num-exp*)

BFLOAT computes the value of the *num-exp* and converts it to a (long) binary real. Binary reals are special data types that are covered in detail in the *Elements of the BASIC Language* chapter.

CEIL(*num-exp*)

The CEIL function evaluates a *num-exp* and returns the smallest integer that is not less than the value of the expression.

Example:

```
PRINT CEIL(PI);CEIL(-PI);CEIL(12345.0001);CEIL(4)
4     -3   12346   4
```

The ceiling of a whole number is the same number. Positive fractional numbers are rounded up. Negative fractional numbers are truncated to a whole number.

See Also: FLOOR

CHR$(*num-exp*)

CHR$ generates a 1-character string which is the ASCII value of the specified *num-exp.* The *num-exp* should be a positive number from 1 to 255.

This function is useful for printing and/or testing for control characters. Refer to the *Appendix* for a listing of the ASCII values.

Note that under THEOS emulation, printing the value of CHR$(X), where X is the integer returned for the value of CRT$(C$), will generate the same result as printing CRT$(C$). In that case, C$ must be one of the valid CRT function parameters such as “RVON”.

Example:

```
PRINT CHR$(65)
A
```

See Also: ASC, ORD

CLS$

This function returns a character string that, when printed to the console, will clear the screen. The string that is output is actually the form-feed character. If a CLS$ is directed to a printer, it will cause a page eject.

This is identical to executing a CRT$("CLEAR"). The function must be used in a PRINT statement or it will have no effect.

Example:

```
PRINT CLS$;"This line will be the first and only text on the screen";
```

See Also: CRT$

CMDARG$(*num-exp*)

CMDARG$ returns the command line argument which is a string of arbitrary length that corresponding to the value of the *num-exp*. The value of the expression is treated as an index into the arguments specified on the command line when the program was invoked. CMDARG$(0) is the first parameter and returns the program name.

Example:

If a program named TEST is invoked with the command:

```
TESTPGM one two three
```

The following statement will print:

```
FOR I%=0 TO 2 \ PRINT CMDARG$(I%);" "; \ NEXT I%
```

```
TESTPGM one two
```

Note that this function is defined differently than in older versions of THEOS BASIC where the command line elements are indexed from one.

COS(*num-exp*)

COS returns the cosine of the angle specified by the *num-exp*. The angle is expressed in degrees if OPTION DEGREE is in effect. Otherwise, it is in radians. Cosines have values from –1 to +1.

COSH(*num-exp*)

COSH computes and returns the hyperbolic cosine of the value specified by the *num-exp.*

COT(*num-exp*)

COT returns the cotangent of the angle specified by the *num-exp*. The angle is expressed in degrees if OPTION DEGREE is in effect. Otherwise, it is in radians.

COTH(*num-exp*)

COTH computes and returns the hyperbolic cotangent of the value specified by the *num-exp.*

CRT$(*string-exp*)

CRT$ returns a string of characters which perform a wide variety of control functions when output to the console or printer. The *string-exp* values in the following chart are recognized, although some operations may not be supported on some devices or in all CET products.

Note that the CRT$ functions that display color and graphic characters are covered separately in the chapter on *Color and Line Drawing Features*. Mapping the CRT$ strings (indicated with an asterisk) to the escape sequences which are to be sent to a printer is device dependent and covered in the *Formatted Output* chapter.

String	Effect on Console	Effect on Printer
BELL	Sounds the bell.	Sounds the bell.
BOFF	Sets blink mode off.	*Sets italics off.
BON	Sets blink mode on.	*Sets italics on.
CLEAR	Clears the screen.	Causes a page eject.
CR	Operates as a CLEAR.	Causes a page eject.
DC	Deletes the current character and shifts any characters to the right are moved left one position.	
DL	Deletes the current line. Any lines below are moved up.	
DOWN	Moves cursor to the same column in the	Sends a line-feed.

	next line.	
EOL	Erases characters from the cursor position to end of line.	*Sets double width print on.
EOS	Erases characters from the cursor position to the end of the screen.	*Sets double width print off.
ESC	Sends an escape character.	Sends an escape.
EU	Erases unprotected fields under THEOS emulation. Otherwise, erases high-intensity fields.	
FOFF	Sets format mode off.	*Sets compressed print on.
FON	Sets format mode on.	*Sets compressed print off.
HOME	Moves cursor to upper-left corner.	
IC	Inserts a space and shifts any characters to the right.	
IL	Inserts blank line. Moves lines below down one row.	
KOFF	Turns the cursor off.	*Sets double height print off.
KON	Turns the cursor on.	*Sets double height print on.
LEFT	Moves cursor one position to the left.	Moves printhead one character left.
MOFF	Sets monitor mode off.	
MON	Sets monitor mode on.	
POFF	Sets protect mode off.	
PON	Sets protect mode on. Characters are in high intensity and protected if CRT$(“EU”) is used. Under THEOS emulation, half-intensity is displayed.	
RIGHT	Moves cursor one position to the right.	Moves printhead one character right.
RVOFF	Turns reverse video mode off.	*Sets boldface off.
RVON	Turns on reverse video mode.	*Sets boldface on.
TAB	Moves cursor to next tab position.	*Moves printhead to next tab position.
UP	Moves cursor to same column, one line up.	
ULOFF	Turns underline mode off.	
ULON	Turns underline mode on. Characters	

	appear in yellow on color devices.	

The CRT$ function recognizes string expressions in either upper, lower or mixed case characters. The underlined portion of the string indicates the minimum number of characters that must be entered.

This function is similar to AT$ and CLS$ in that a value is simply returned and no operation will be performed unless the value is used with PRINT.

The value returned by CRT$ is a string of one or more unprintable ASCII characters. CET BASIC translates this value to an internal code. (See the *Appendix* for a list.) CET BASIC recognizes these internal codes during execution of the PRINT statement, and translates them into the desired action during output. Therefore, files containing CRT$ (and AT$) information may be sent without change to any device recognized by the operating system.

CET BASIC normally displays low-intensity text with is typical in the DOS and UNIX environments. Setting the B_EMULATE or B_THPON variables will ensure that a high-intensity foreground is used as under THEOS.

In developing applications (under UNIX), it is important to note that RVON may turn on reverse video from the specified position up until the end of the line or screen. Terminals which use a space character to display attributes may exhibit a flash during this process. To subdue this effect, execute RVOFF first. Then when you use RVON, only the areas between the two video fields will be affected.

In the CET DOS/Networking product, the CRT$("MON") statement may be used to force the display of the actual ASCII value of all characters printed to the screen until a CRT$("MOFF") is encountered. Under THEOS emulation, CRT$("MON") will return a string with the ASCII value of 128 and CRT$("MOFF") will return a string value of 129.

The behavior of the CRT$ function depends upon the actual device in use. For example, under UNIX the behavior of the console and terminals is defined by the **etc/termcap** file. If there are problems, this file may be modified to produce the desired results. Refer to the *CET BASIC UNIX User's Guide* for information on how this is done.

Example:

```
PRINT CRT$("MON");CHR$(24);CHR$(25);CRT$("MOFF")
↑↓
```

See Also: AT$, CHR$, CLS$

CSC(*num-exp*)

CSC returns the cosecant of the angle specified by the *num-exp*. The angle is expressed in degrees if OPTION DEGREE is in effect. Otherwise, it is in radians. Cosecants have values that range from –∞ to –1 and +1 to +∞.

CSCH(*num-exp*)

CSCH computes and returns the hyperbolic cosecant of the value specified by the *num-exp.*

CSH(*string-exp*)

This function is identical to the CSH/CSI statement, except that it also passes back the return code for the program it executes. The *string-exp* is evaluated and passed to the operating system for execution. The calling program then waits for the command to finish at which time normal program execution continues. The return code value of the command that was executed becomes the value of the CSH function.

For example, the UNIX command test may be used to check for the existence of files with certain properties. The command returns 0 if the required condition is true and non-zero if it is false (which is the opposite of the usual BASIC interpretation). The following code segment illustrates this feature.

Example:

```
If CSH("test -r afile") <> 0 \ REM if afile isn't readable
      CSH("Bcreate ./afile d recl 10") \ REM create it
IFEND
```

DATE$(*num-exp*)

DATE$ evaluates the *num-exp* as the number of days since January 1, 1900 and returns the corresponding date-string. The value 0 (zero) is interpreted as the current system date.

The date-string is in a normalized date format according to the currently set OPTION DATEFORM. The environment variable B_4DYEAR may be set to display the 4-digit year.

DATEFORM	Format

1	American dates as mm/dd/yy - the default
2	European dates as dd/mm/yy
3	International dates as yy/mm/dd

Example:

```
PRINT DATE$(10);"  ";DATE$(28262)
01/10/00  05/18/77
```

See Also: DAY, DTE$

DAY(*string-exp*)

DAY evaluates the *string-exp* and interprets it as a date field according to the currently set OPTION DATEFORM (see DATE$ above). Non-numeric characters in the *string-exp* are considered as separators between the month, day, and year.

The number of days between January 1, 1900 and the specified date is returned. Note that although the return value is always a whole number, it must be referenced as a numeric and not an integer. An invalid date *string-exp* will cause the function to return a –1.

Example:

```
PRINT DAY("5/17/77"),DAY("1-1-0")
28261    -1
```

See Also: DATE$, DTE$

DEG(*num-exp*)

The DEG function interprets the value of the *num-exp* in radians and returns the degree equivalent. The sign of the *num-exp* is retained as the sign of the function.

Example:

```
PRINT DEG(PI)
180
```

See Also: RAD

DEL$(*string-exp,num-exp1,num-exp2*)

The DEL$ function returns the *string-exp* after deleting the subfield *num-exp2* from the field *num-exp1*. When *num-exp2* = 0, all of field *num-exp1* is deleted. When the designated field does not exist, the string is returned unmodified.

String fields are formatted with fields and subfields using the INS$ function. The REP$ and DEL$ functions may be used to modify the subfields. EXT$ may be used to extract a subfield.

Example:

For B$ equaling "AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD"

```
DEL$(B$,2,0) equals  AAAA^C1C1C1]C2C3C3]C3C3C3^DDDD
DEL$(B$,3,3) equals  AAAA^BBBB^C1C1C1]C2C2C2^DDDD
DEL$(B$,3,0) equals  AAAA^BBBB^DDDD
```

The field and subfield delimiters are shown here as the ^ and] characters. The actual characters used have their parity bits set (CHR$(128 + ASC("^")) and CHR$(128 + ASC("]")). On some terminals these will appear as ^ and] while on others they will print as an assortment of *funny* marks.

See Also: EXT$, INS$, REP$

DTE$(*string-exp*)

DTE$ validates the *string-exp* for a valid date according to the currently set OPTION DATEFORM (covered with the DATE$ function). A date may use any non-numeric character as a delimiter between the month, day and year, except for a semicolon or a comma.

If the date is valid, a string in the current date format is created and returned as the value of the function. If the string is invalid, a null string is returned.

The environment variable B_4DYEAR may be set to display the 4-digit year. The variable B_YR2000 may also be set to the first year to be interpreted as 20XX. For information on how to use these variables, please refer to the *CET BASIC User's Guide* for the product you are using.

Example:

```
PRINT DTE$("7/6/76"), DTE$("2/30/76"), DTE$("112154")
07/06/76                11/21/54
```

See Also: DATE$, DAY

EOF(*num-exp*)

The EOF function evaluates the *num-exp* and checks the I/O channel corresponding to that value for an end-of-file condition. Typically, this function is used after a READ, READNEXT or READPREV function, or in a WHILE-WEND structure as in WHILE NOT EOF(1). If EOF is evaluated after the execution of a READPREV statement, a true value indicates the beginning-of-file. The end-of-file status is returned as either 0 (*false*) or -1 (*true*).

Channel #0 (the console) is never at end-of-file. A file opened for OUTPUT is always at end-of-file, regardless of the type of file.

Example:

```
READ #1, KEY$ : F1$, F2$
IF EOF(1) THEN PRINT “End of File”
```

EPS

EPS returns the smallest positive value that the system can maintain. This value is normally the smallest decimal real value, 1.0E–128. If OPTION DEFAULT has set the default real type to other than decimal real (under UNIX), EPS returns the smallest positive value for that type.

See Also: INF

EPS!

EPS! returns the smallest positive short binary real value that can be maintained. Both EPS! and EPS!! are only supported under UNIX.

EPS!!

EPS!! returns the smallest positive long binary real value.

ERF

This function returns the channel number associated with the file on which the last I/O operation was performed. Refer to the description of the ON ERROR statement to see how this function may be used.

ERL

When an error is detected, ERL normally returns the line number for the statement which caused the error. If the error occurred on a line that was not numbered, the returned value will refer to the most recently executed line that had a number. If the program was compiled with the line selection flag (–#), ERL returns the actual number of the line in the source code that caused the error.

Refer to the description of the ON ERROR statement to see how this function may be used to attempt to resolve the error condition.

ERR

ERR returns the number of the last error that occurred during the execution of a program. A value of zero is returned if no error was detected. Refer to the *Appendix* for a list of the error codes.

ERRMSG$(*string-exp*)

This function displays a message on the bottom row on the screen, waits for a response from the operator, and returns the single character response. After the operator responds, the message is cleared, and the response is returned as the value of the function.

See Also: YESNO$

EXP(*num-exp*)

EXP evaluates the *num-exp* and returns the value of $e^{\text{num-exp}}$. This is the complement of the LOG function.

EXT$(*string-exp,num-exp1,num-exp2*)

EXT$ extracts from the *string-exp* the subfield *num-exp2* of the field *num-exp1* and returns this as its value. If *num-exp2* = 0, all of the field specified by *num-exp1* is returned. Refer to the description of the DEL$ function for information on the field and subfield delimiters, shown here as ^ and].

Example:

For the string, B$ = AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

```
EXT$(B$,2,0) equals  BBBB
EXT$(B$,3,3) equals  C3C3C3
EXT$(B$,3,0) equals  C1C1C1]C2C2C2]C3C3C3
EXT$(EXT$(B$,3,0),1,2) equals  C2C2C2
```

See Also: DEL$, INS$, REP$

FIX(*num-exp*)

FIX truncates the value of *num-exp* and returns the integer portion. No rounding is performed. If the result is beyond –32768 and +32767, an error (#50) occurs. Use the IP function to return the value of larger numbers.

Example:

```
PRINT FIX(1.5);FIX(.5);FIX(5.5);FIX(-43.5)
```

See Also: BFLOAT, FLOAT, IP

FLOAT(*num-exp*)

FLOAT evaluates the *num-exp* and converts it, if necessary, to the default real type, which is usually decimal. No rounding is performed. (FLOAT is the complement of the FIX function.)

Example:

```
PRINT 1/4;1/FLOAT(4);1/4.;FLOAT(4.25)
0      .25  .25  4.25
```

See Also: BFLOAT, FIX

FLOOR(*num-exp*)

FLOOR evaluates the *num-exp* and returns the largest integer that is not greater than the value of the expression. The floor of a whole number is the same number. Positive fractional numbers are truncated to the whole number. Negative fractional numbers are equal to the *num-exp* – 1.

Example:

```
PRINT FLOOR(PI);FLOOR(-PI);FLOOR(12345.9999);FLOOR(4)
3      -2    12345    4
```

See Also: CEIL

FORMAT$(*num-exp,string-exp*)

FORMAT$ returns a formatted number of a specific length with the same capabilities as the PRINT USING statement. The two expressions are evaluated and the *num-exp* is formatted according to the masking characters in the *string-exp*. If the *num-exp* cannot be formatted in the specified length, the resultant string will start with a percent character. The following characters may be used in the *string-exp*:

Characters	Function
**	Replaces leading zeros with asterisks.
$$	Suppresses all leading zeros and displays a floating dollar sign.
DB	Displays negative values with a trailing 'DB'.
CR	Displays negative values with a trailing 'CR'.
>	Displays negative values surrounded with angle brackets.
#	Suppresses leading zeros except for the 1's position.
9	Displays number with leading zeros.
+	Displays a trailing sign when specified at the *string-exp*.
–	Displays a trailing sign for negative numbers only.
,	Displays numbers with comma or period separators depending upon whether OPTION COMMA is used.
^^	Uses exponential format with a single unsigned digit.
^^^	Uses exponential format with a single signed digit.
^^^^	Uses exponential format with a double signed digit.
^^^^^	Uses exponential format with a triple signed digit.

The masking characters +. –, DB, CR, and all the ^ variants must be specified at the end of the string as in 99.99+. The $ and * characters may only be used at the beginning.

Examples:

```
PRINT FORMAT$(23,"99999");FORMAT$(23,"#####")
00023      23

PRINT FORMAT$(23,"**###");" ";FORMAT$(123456.78,"$$#,#####.##")
***23 $123,456.78

PRINT FORMAT$(12345,"#.#####^^")
1.2345E4
```

```
PRINT FORMAT$(-12345.67,"#,######.##>")
<12,345.67>
```

See Also: STR$ function, PRINT USING statement, and the *Formatting Output* chapter.

FP(*num-exp*)

The FP function evaluates the *num-exp* and returns the fractional portion. The sign of the *num-exp* is retained as the sign of the returned value.

Example:

```
PRINT FP(PI);FP(1234.5678)
.141592653590      .5678
```

See Also: FIX, INT, IP

GCD(*num-exp1,num-exp2*)

GCD returns the Greatest Common Divisor of the two arguments. The sequence of the expressions is irrelevant.

Example:

```
PRINT GCD(171,42);GCD(100,81);GCD(3.5,4.5)
3      1     .5
```

HEX(*string-exp*)

The HEX function analyzes the value of the *string-exp* as hexadecimal digits (0–9 and A–F) and converts it to the equivalent 16-bit integer value, which is returned. The HEX function is the complement to HEXOF$.

Example:

```
PRINT HEX("0FF");HEX("100")
255          256
```

HEXOF$(*num-exp*)

HEXOF$ converts the value of the *num-exp* to an integer, if necessary, and translates it into a 4-character string representing the value in hexadecimal. This string is returned as the value of the function.

The HEXOF$ function is the complement to HEX.

Example:

```
PRINT HEXOF$(94);" ";HEXOF$(23129)
005E 5A59
```

INF

INF returns a constant representing the largest value that the system can maintain. This value is normally the largest decimal real value which is 9.999999999999E+126. If OPTION DEFAULT has set the default real type to other than decimal (under UNIX), INF returns the largest value for that type.

See Also: EPS

INF!

INF returns a constant representing the largest short binary real value that the system can maintain. Note that INF! and INF!! are only supported in the CET BASIC products for UNIX.

INF!!

INF returns a constant representing the largest long binary real value that the system can maintain.

INP

INP returns the ASCII integer value of the control character used to terminate the last input (via an INPUT or LINPUT statement). Only the first character will be considered as a control character.

For example, if the last input was Ctrl+Z, the INP value is 26. Terminating an entry by pressing the Return key will set the INP value to zero.

Refer to the *Character Codes* section in the *Appendix* for a list of INP values.

INS$(*string-exp,num-exp1,num-exp2,string-exp2*)

This function is the inverse of the EXT$ function. It inserts a substring into a string. The substring *string-exp2* will be inserted into the *string-exp* after the subfield *num-exp2* of field *num-exp1*.

When the subfield number is zero (*num-exp2* = 0), the substring (*string-exp2*) becomes the entire field (*num-exp1*). All prior fields or subfields of the *string-exp* remain, but may be shifted to new relative positions. To insert a string as the first subfield of a field, specify *num-exp2* = –1.

Example:

For the string B$ = AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

INS$(B$,2,0,"NEW") equals
AAAA^BBBB^NEW^C1C1C1]C2C2C2]C3C3C3^DDDD

INS$(B$,0,0,"NEW") equals
NEW^AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

INS$(B$,3,1,"NEW") equals
AAAA^BBBB^C1C1C1]NEW]C2C2C2]C3C3C3^DDDD

INS$(B$,3,-1,"NEW") equals
AAAA^BBBB^NEW]C1C1C1]C2C2C2]C3C3C3^DDDD

INS$(B$,7,2,"NEW") equals
AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD^^^^]]NEW

See Also: For a discussion of the field and subfield delimiters, shown here as ^ and] symbols, refer to the DEL$ function.

INT(*num-exp*)

The INT function returns the largest signed integer that is less than or equal to the value of the *num-exp*. The result of this function is an integer or real, depending upon the argument used.

This is synonymous with the operation of the FLOOR function. It differs from the IP function in that it does not truncate a negative fractional number. Since INT rounds negative decimal fractions, it might return a smaller value.

Example:

```
PRINT INT(1.5);INT(.5);INT(-4.6)
1      0     -5
```

See Also: FIX, FLOOR, IP

IP(*num-exp*)

IP truncates the value of the *num-exp* and returns the integer portion as a real number. This function differs from INT only when the *num-exp* is negative. IP will always truncate a negative fractional number. Since INT rounds negative decimal fractions, these two functions may return different values.

Example:

```
PRINT IP(PI);IP(-PI);IP(1234567.9876)
3      -3    1234567
```

See Also: CEIL, FLOOR, FP, INT

LCASE$(*string-exp*)

LCASE$ returns a string that is the same as its argument except that all upper case letters have been converted to lower case.

Example:

```
PRINT LCASE$("NOW IS THE TIME FOR ALL GOOD MEN ...")
now is the time for all good men ...
```

See Also: UCASE$

LEFT$(*string-exp,num-exp*)

LEFT$ returns the characters in the *string-exp* starting from the value of the *num-exp*. This function is equivalent to using the substring operator [1*:num-exp*]. Refer to the chapter on *Elements in the BASIC Language* for information on using substrings.

Example:

```
A$="ABC Stores, Inc."
PRINT LEFT$(A$,10)
ABC Stores
```

See Also: MID$, RIGHT$

LEN(*string-exp*)

LEN returns the current length of *string-exp* including any nulls and control characters.

Example:

```
PRINT LEN("ABCDEF");LEN("    X   ");LEN(SPACE$(10))
6      10   10
```

LINE(*num-exp*)

The LINE function returns the integer value of the line length of the device opened on the I/O channel specified by the *num-exp*. This is generally only meaningful when *num-exp* = 0 which refers to the console.

Example:

```
PRINT LINE(0) \ REM Console terminal
80 \ REM when B_EMULATE is NOT set to THEOS

PRINT LINE(0)
79 \ REM when B_EMULATE is set to THEOS
```

See Also: PAGE

LOG(*num-exp*)

LOG evaluates the *num-exp* and assigns the natural logarithm of that value to the function ($\log_e$*num-exp)* where 2.718281828459 is the value of *e*.

See Also: EXP, LOG2, LOG10

LOG2(*num-exp*)

LOG2 evaluates the *num-exp* and assigns the binary logarithm (logarithm base 2) of that value to the function ($\log_2$*num-exp*).

LOG10(*num-exp*)

LOG10 evaluates the *num-exp* and assigns the common logarithm (logarithm base 10) of that value to the function ($\log 10_{10}$*num-exp*).

LPAD$(*string-exp,num-exp*)

LPAD$ returns the *string-exp* with spaces added to the left, if necessary, to make it the length specified by the *num-exp*. If the length of the *string-exp* is equal to or greater than the *num-exp*, it is returned unmodified.

Example:

```
PRINT "#";LPAD$("1234",6);"#"
#  1234#

PRINT "#";LPAD$("1234",3);"#"
#1234#
```

See Also: RPAD$

LRL(*num-exp1,num-exp2*)

LRL returns an integer value after a logical rotate left is performed on *num-exp1* for *num-exp2* bit positions. Any bits rotated out of the left end of *num-*

exp1 will be reinserted at the right. If *num-exp2* is greater than 15 or less than 1, no rotation is performed.

num-exp1 is considered a signed number which will be converted to integer type, if necessary. This may have unforeseen results when, for instance, attempting to evaluate LRL(32768,1). (32768 as an unsigned integer has the same bit pattern as -32768 as a signed integer.) Since 32768 is too large to be a signed integer, it will be a real, and when LRL attempts to convert it, a trappable error will occur. If the program needs to use unsigned constants in shift instructions, hex constants should be used. (32768 = 8000H)

Example:

```
PRINT LRL(128,2);LRL(-32768,1)
512         1
```

See Also: LRR, LSL, LSR

LRR(*num-exp1,num-exp2*)

LRR returns an integer value after a logical rotate right is performed on *num-exp1 for num-exp2* bit positions. Any bits rotated out of the right end of *num-exp1* will be reinserted at the left. If *num-exp2* is greater than 15 or less than 1, no rotation is performed.

Example:

```
PRINT LRR(128,2);LRR(1,1);LRR(-32768,1)
32    -32768   16384
```

See Also: LRL, LSL, LSR

LSL(*num-exp1,num-exp2*)

LSL returns an integer value after a logical shift left is performed on *num-exp1* for *num-exp2* bit positions. Any bits shifted out of the left end of *num-exp1* will be discarded, and zeros will be inserted at the right. If *num-exp2* is greater than 15 or less than 1, no shift is performed.

Example:

```
PRINT LSL(128,2);LSL(-32768,1);LSL(16384,1)
512   0     -32768
```

Refer to the description of LRL for information on using unsigned numbers.

See Also: LRL, LRR, LSR

LSR(*num-exp1,num-exp2*)

LSR returns an integer value after a logical shift right is performed on *num-exp1* for *num-exp2* bit positions. Any bits shifted out of the right end of *num-exp1* will be discarded, and zeros will be inserted at the left end. If *num-exp2* is greater than 15 or less than 1, no shift is performed.

Example:

```
PRINT LSR(128,2);LSR(-32768,1);LSR(1,1)
32    16384    0
```

See Also: LRR, LSL and LRL (for information on using unsigned numbers).

LTRIM$(*string-exp*)

LTRIM$ returns the *string-exp* after removing any leading spaces.

Example:

```
PRINT "#";LTRIM$("  ABC  DEF  ");"#"
#ABC      DEF      #
```

See Also: RTRIM$, TRIM$

MATADDROF(*array*)

The MATADDROF function is used with the CALL statement to pass the memory address of an array variable, so that the value *array* can be modified by the subroutine. A subroutine evaluates the address of the array variable and assigns it a new value. This value is then returned to the calling BASIC program.

Examples:

```
DIM A$(25)
CALL MYPROG("INIT",MATADDROF(A$))
CALL BW.OPEN(ADDROF(WIN%),5,10,65,10,1)
CALL BW.SELECT(WIN%)
```

The first CALL statement uses the MATADDROF function to return the value of the array A$. The next one opens a window called WIN% with a specified location, size, and frame type. The ADDROF function is used to return the value assigned to the memory address of the simple variable WIN% so that it may be referenced in the CALL to the BW.SELECT function.

See Also: ADDROF function, CALL statement

MATCH(*string-exp1,string-exp2*)

This function compares *string-exp1* with the pattern mask specified by *string-exp2*. If it matches, -1 (true) is returned. A zero return value indicates that no match was made.

The pattern mask (*string-exp2*) may contain special characters as well as literals. The masking characters are interpreted as follows:

Character	Meaning
@	Any single letter or space character will match.
#	Any single numeric digit will match.
?	Any single character will match.
*@	Any number of letters or space characters will match.
*#	Any number of numeric digits will match.
*?	Any number of alphabetic or numeric characters will match.
%	A special escape character that can precede a masking character to specify that the character should be treated as a literal match.

All other characters are treated as literal match characters. The corresponding position in the first string must contain the specific character.

The following examples show sample masks along some matching and non-matching strings:

Mask: "ABC?"

Any four character string starting with the uppercase letters A, B, and C. For example:

Matching	**Not Matching**
"ABCX"	"abcx"
"ABC1"	“ABC”
"ABC-"	“ABCDE”

Mask: "ABC*?"

Any string of four or more characters starting with the uppercase letters A, B, and C.

Matching	**Not Matching**
"ABCDEF"	“A”
"ABC#*$$234"	"ABDXLKJ"

Mask: "ABC*?DEF"

Any string starting with ”ABC” ending with “DEF”. One or more characters between these letters are acceptable.

Matching	**Not Matching**
"ABCXDEF"	“ABCDE”
"ABC7.2.72.F"	“ADEF”
"ABC124ABCDEF"	"ABCDEF"

Mask: "###-##-####"

Any eleven character string containing three digits, a hyphen, two digits, a hyphen, and four digits such as a social security number.

Matching	**Not Matching**
"123-45-6789"	"123456789"
	"123/45/6789"

Mask: "@@@###"

Any six character string whose first three characters are letters or spaces and whose last three characters are digits.

Matching	**Not Matching**
"abc123"	“123ABC”
"ABC123"	“AB1234”
"Xyz002"	"ABCX123"

See Also: SCH

MAX(*num-exp1,num-exp2*)

MAX returns the value of the larger of the two numeric expressions. Their order is not important.

Example:

```
PRINT MAX(5,21);MAX(PI,3.14);MAX(1,1)
21    3.141592653590  1
```

See Also: MIN

MID(*string-exp,num-exp1,num-exp2*)

MID$ returns the middle portion of a string. Starting at the position specified by *num-exp1, num-exp2* characters are returned. This function is equivalent to using the following substring expression:

(*string-exp*)[*num-exp1 : num-exp1 + num-exp2* - 1]

Example:

```
PRINT MID$(A$,15,5)
OPQRS
```

See Also: LEFT$, RIGHT$

MIN(*num-exp1,num-exp2*)

MIN returns the value of the smaller of the two expressions. Their sequence is not important.

Example:

```
PRINT MIN(5,21);MIN(1,-1);MIN(3*23,70)
5      -1    69
```

See Also: MAX

MOD(*num-exp1,num-exp2*)

MOD returns the value of *num-exp1* modulo *num-exp2*. The MOD function divides *num-exp1* by the value of *num-exp2* and returns the remainder as the value of the function. The sign of *num-exp1* is retained as the sign of the return value.

Example:

```
PRINT MOD(11,4);MOD(2.2,.8)
3      .6
```

MSEC

MSEC returns the current system time expressed as the number of milliseconds since midnight.

See Also: SECOND, TIME$

NBR(*string-exp*)

NBR returns a value that indicates whether a string expression represents a valid number. The *string-exp* is analyzed to determine if it contains only:

- The digits 0 through 9
- One plus or minus sign
- One period (or comma if OPTION COMMA is in use)

- Leading or trailing spaces
- The letter E to signify an exponential number.

NBR returns a zero (*false*) when any invalid character is detected. The function returns –1 (*true*) when all the characters are valid, the string is empty, or it consists entirely of blanks and tabs.

Example:

```
PRINT NBR("123");NBR("0ABCH");NBR("1.23E23")
-1     0     -1
```

OCT(*string-exp*)

The OCT function analyzes the *string-exp* as an octal integer and returns its integer value. The valid octal digits are 0 through 7. Analysis stops if a non-octal digit is encountered, and the preceding string is returned.

OCT is the complement to the OCTOF$ function.

Example:

```
PRINT OCT("071");OCT("100")
57     64
```

OCTOF$(*num-exp*)

OCTOF$ evaluates the *num-exp*, converts it to an integer type if necessary, and translates it into the string of characters representing its value in octal. A six character string is always generated.

The OCTOF$ function is the complement to OCT.

Example:

```
PRINT OCTOF$(123);" ";OCTOF$(94)
000173          000136
```

ORD(*string-exp*[,*byte*])

ORD is a synonym to the ASC function. The *string-exp* is evaluated and the ASCII value of its first character is returned. An optional second parameter may be used to indicate a byte number of the string.

Refer to the description of ASC for examples and to the *Appendix* for a list of the ASCII code values.

See Also: ASC

OVR$(*string-exp1,num-exp1,num-exp2,string-exp2*)

OVR$ overlays one string onto or into another. This is done by making *string-exp2* exactly *num-exp2* characters long by either truncating it or adding spaces on the right. The returned value is the result of overlaying the modified *string-exp2* onto *string-exp1* starting at position *num-exp1*.

Example:

```
PRINT OVR$(A$,2,3,"0123456");
A012EFGHIJKLMNOPQRSTUVWXYZ

PRINT OVR$(A$,2,9,"0123456");
A0123456KLMNOPQRSTUVWXYZ
```

PAGE(*num-exp*)

PAGE returns the integer value of the page length for the device opened on the I/O channel specified by *num-exp*. This is generally only meaningful when *num-exp* = 0 is used to reference the console.

The actual return value will depend upon whether the environment variable B_EMULATE=THEOS is set. The B_SCRNOLINES may be set to alter the default page length for the console.

Example:

```
PRINT PAGE(0) \ REM Console terminal
24 \ REM when B_EMULATE is NOT set

PRINT PAGE(0)
23 \ REM when B_EMULATE is set to THEOS
```

See Also: LINE

PI

PI returns the decimal real value of π (3.14159265359). If OPTION DEFAULT has set the default real type to other than decimal real (under UNIX), PI returns the binary real value of π.

PI!

PI! returns the binary real value of π (3.14159265359) when OPTION DEFAULT has been set to REAL4 or REAL8. (This function is currently only supported under UNIX and UNIX-related environments.)

PI!!

PI!! operates like PI! and returns the binary real value of π.

POS(*num-exp*)

The POS function returns the current column position on the line being output to the device opened on the channel specified by the *num-exp*. Generally, this is only meaningful when *num-exp* = 0 is used to reference the console.

Example:

```
PRINT "123456";POS(0)
123456    6
```

RAD(*num-exp*)

RAD evaluates the *num-exp* as a number of degrees. The radian equivalent is returned as the value of the function. The sign of the *num-exp* is retained as the sign of the return value.

REP$(*string-exp1,num-exp1,num-exp2,string-exp2*)

REP$ operates similar to the INS$ function except that it replaces a subfield instead of inserting the subfield. *string-exp2* will replace the existing *string-exp1* at field *num-exp1*, subfield *num-exp2*.

When *num-exp2* = 0, *string-exp2* replaces the entire field. If the specified field or subfield does not exist, sufficient empty fields will be added to create it.

Example:

For the string B$ = AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

REP$(B$,6,0,"HERE") equals
AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD^^HERE

REP$(B$,3,2,"NEW") equals
AAAA^BBBB^C1C1C1]NEW]C3C3C3^DDDD

REP$(B$,2,2,"NEW") equals
AAAA^BBBB]NEW^C1C1C1]C2C2C2]C3C3C3^DDDD

Refer to the DEL$ function for a discussion of the field and subfield delimiters, shown here as ^ and].

See Also: DEL$, EXT, INS$

RIGHT$(*string-exp,num-exp*)

RIGHT$ returns the substring of the *string-exp* starting at the character position specified by *num-exp* until the end of the string. This function is equivalent to using the substring operator:

(*string-exp*)[*num-exp:255*]

Example:

PRINT RIGHT$(A$,20)
TUVWXYZ

See Also: LEFT$, MID$

RND

RND returns the value of the next pseudo-random number. The value is a decimal real number in the range of 0 < RND < 1. This function accepts no parameters. The RANDOMIZE statement may be used to randomly initialize a sequence of pseudo-random numbers returned from RND.

ROUND(*num-exp1,num-exp2*)

ROUND returns the result of rounding *num-exp1* to the number of places specified by *num-exp2*. The *num-exp2* may be positive or negative. If negative, the return value specifies a number of places to the *left* of the decimal point. Positive values indicate a number of places to the *right*.

Example:

PRINT ROUND(PI,4);ROUND(1234.567,-2)
3.1416 1200

PRINT ROUND(1.234567,4);ROUND(2.34,0)
1.2346 2

RPAD$(*string-exp,num-exp*)

RPAD$ returns the *string-exp*, padded on the right with spaces when necessary to make it at least *num-exp* characters long.

Example:

```
PRINT "#";RPAD$("1234",6);"#"
#1234      #

PRINT "#";RPAD$("1234",3);"#"
#1234#
```

See Also: LPAD$

RPT$(*num-exp,string-exp*)

RPT$ returns a string generated by repeating the *string-exp* for *num-exp* times.

Example:

```
PRINT RPT$(3,"ABCD")
ABCDABCDABCD
```

RTRIM$(*string-exp*)

RTRIM$ returns the *string-exp* after removing any trailing spaces.

Example:

```
PRINT "#";RTRIM$("  ABC  DEF  ");"#"
#      ABC      DEF#
```

See Also: LTRIM$, TRIM

SCH(*num-exp,string-exp1,string-exp2*)

SCH searches *string-exp1* starting at character position *num-exp* for the first occurrence of the substring specified as *string-exp2*. If *num-exp* is a negative value or zero, the search will start in position one.

If the substring is found, its starting character position is returned. Otherwise, zero is returned. The value of SCH is always zero when the starting position *num-exp* is greater than the length of the string specified by *string-exp1.*

When the substring *string-exp2* is null, *num-exp* is returned.

Example:

```
PRINT SCH(1,"ABCDEFGH","D");SCH(3,"ABCDEFGH","EFG")
4      5

PRINT SCH(1,"ABCDEFGH","X");SCH(2,"ABC","")
0      2
```

See Also: MATCH

SEC(*num-exp*)

SEC returns the secant of the angle specified by the *num-exp*. The angle is expressed in degrees if OPTION DEGREE is in effect. Otherwise, it is in radians. Secants have values that range from –∞ to –1 and +1 to +∞.

SECH(*num-exp*)

SECH computes and returns the hyperbolic secant of the *num-exp.*

SECOND(*string-exp*)

SECOND evaluates the *string-exp* as a normalized time of day (hh:mm:ss) and returns the number of seconds from midnight to that time. Any non-numeric character except for a semicolon or comma may be used as a delimiter between the hours, minutes, and seconds. For example, a valid *string-exp* could be: "1.1.1", "1-1+1", "1H1M1S", "1%1", "1", etc. To get the current time of day in seconds use SECOND(TIME$(0)).

Example:

```
PRINT SECOND("12:00:00");SECOND("01:05:08");SECOND("2:03:00")
43200      3908      7380
```

See Also: MSEC, TIME$

SGN(*num-exp*)

SGN evaluates the *num-exp* and returns –1, 0, or +1 depending upon whether the value is negative, equal to 0, or positive, respectively.

Example:

```
PRINT SGN(PI);SGN(-1.0/-2.0);SGN(-43);SGN(PI-PI)
1      1     -1    0
```

SIN(*num-exp*)

SIN returns the sine of the angle specified by the *num-exp*. The angle is expressed in degrees if OPTION DEGREE is in effect. Otherwise, it is in radians. Sines have values that range from –1 to +1.

SINH(*num-exp*)

SINH computes and returns the hyperbolic sine of the value of the *num-exp.*

SPACE$(*num-exp*)

SPACE$ returns a string of *num-exp* spaces.

Example:

```
PRINT LEFT$(A$,3)&SPACE$(4)&MID$(A$,4,5)
ABC  DEFGH
```

See Also: RPT$

SQR(*num-exp*)

SQR returns the square root of *num-exp*.

Example:

```
PRINT SQR(4);SQR(25);SQR(11);SQR(-4)
2      5     3.316624790355   0
```

STR$(*num-exp*)

STR$ returns a trimmed string of characters representing the value of *num-exp*. This is similar to the string that would be output by the PRINT statement except that there are no leading or trailing blanks. Note that the VAL(STR$(*num-exp*)) is always equal to the *num-exp*.

Example:

```
PRINT "ABC";STR$(1.23);"DEF"
ABC1.23DEF

PRINT "ABC";1.23;"DEF"
ABC        1.23       DEF
```

See Also: VAL

TAB(*num-exp*)

This function is only valid when used as one of the arguments of a PRINT statement. The print or cursor position is advanced to column *num-exp*. If the current print or cursor position is already beyond that column, it will be advanced to that column on the next line.

Example:

```
PRINT "Column 1";TAB(20);"Column 2"
Column 1        Column 2
```

See Also: POS, SPACE$

TAN(*num-exp*)

TAN returns the tangent of the angle specified by the *num-exp*. The angle is expressed in degrees if OPTION DEGREE is in force. Otherwise, it is in radians. Tangents have values that range from $-\infty$ to $+\infty$.

TANH(*num-exp*)

TANH computes and returns the hyperbolic tangent of the *num-exp.*

TIME$(*num-exp*)

TIME$ interprets the *num-exp* as the number of seconds since midnight, converts the value to the format hh:mm:ss, and returns the resulting string. TIME$(0) is interpreted as the current time of day. The maximum integer is 32676. If *num-exp* is out of range, the value "00:00:00" will be returned.

Example:

```
PRINT TIME$(7199),TIME$(0)
1:59:59      15:24:32
```

See Also: SECOND

TRIM$(*string-exp*)

The TRIM$ function returns the *string-exp* stripped of any leading or trailing spaces. Multiple embedded spaces are reduced to a single space. This feature is useful for deleting extra spaces that the operator entered accidentally.

Example:

```
PRINT "#";TRIM$("  ABC  DEF  HIJ  ");"#"
#ABC DEF HIJ#
```

See Also: LTRIM$, RTRIM$

UCASE$(*string-exp*)

UCASE$ returns the *string-exp* after all lower case letters have been converted to upper case.

Example:

```
PRINT UCASE$("Now is the time for all good men ...")
NOW IS THE TIME FOR ALL GOOD MEN ...
```

See Also: LCASE

VAL(*string-exp*)

VAL evaluates the *string-exp* as a numeric constant (see NBR function). If the *string-exp* is a valid number, then the value of that number is assigned to the function. VAL will always return a zero for null strings or strings that consist entirely of blanks and tabs.

If any invalid characters are encountered, a trappable error occurs, by default. If the variable B_EMULATE or B_THVAL is set, VAL returns a zero when an invalid number is detected. We recommend using the NBR function to test the *string-exp* for validity before applying VAL.

Example:

```
PRINT VAL("123");VAL("1.234E23");VAL("");VAL("   ")
123          1.234E+023        0           0

PRINT VAL("ABCD")
Illegal number
```

See Also: NBR

YESNO$

YESNO$ displays the message "(Y/N)" on the console at the current cursor position and waits for the operator to enter a Y or N response. Any response other than a Y, y, N, or n has no effect. Pressing Return is not required. After the operator responds correctly, the character is converted to upper case if necessary, displayed, and returned as the value of the function.

Example:

```
PRINT "Are you ready to proceed? ";
IF YESNO$ = "Y" THEN GOTO PROCEED
```

Note: The environment variable B_LANGUAGE supports this function so that the program can display the appropriate characters for the specified language. This variable is covered in the *CET BASIC User's Guide.*

Function Summary

Name	Return Value
ABS	Absolute value of a number
ACOS	Inverse cosine
ACOT	Inverse cotangent
ASCS	Inverse cosecant
ADDROF	Memory address for a variable
ANGLE	Angle in right triangle
ASC	ASCII value of a character
ASEC	Inverse secant
ASIN	Inverse sine
AT$	Cursor position
ATAN	Inverse tangent
ATN	Alternate name for ATAN
BIN	Integer value of a binary string
BINOF$	Binary string value
BFLOAT	Floating point value of a binary real number
CEIL	Smallest floating point integer
CHR$	Associated character for an ASCII value
CLS$	Form-feed to produce a screen clear or page eject
CMDARG$	Command line argument
COS	Cosine
COSH	Hyperbolic cosine
COT	Cotangent
COTH	Hyperbolic cotangent.
CRT$	Attribute setting

CSC	Cosecant
CSCH	Hyperbolic cosecant
CSH	Return code from the command which it executed
DATE$	Date string
DAY	Date value of a date string
DEG	Degree equivalent of a radian value
DEL$	String after a subfield was deleted
DTE$	Validated date string
EOF	End-of-file status of an open file
EPS	Smallest, positive value maintained by the system
ERF	Channel number for file on which last I/O operation was performed
ERL	Number of statement line which produced the error
ERR	Number of error that occurred
ERRMSG$	Input in response to message
EXP	Exponential number
EXT$	Subfield of a string
FIX	Integer value of a floating point number
FLOAT	Floating point value of an integer number
FLOOR	Largest floating point value
FORMAT$	Formatted string value
FP	Fractional part of a number
GCD	Greatest common denominator
HEX	Integer value of hexadecimal string
HEXOF$	Hexadecimal string value of an integer
INF	Largest value maintained by system
INP	Control character that terminated last input
INS$	String after a subfield was inserted
INT	Largest signed integer < or = to the value
IP	Whole number part of a number
LCASE$	Lowercase equivalent of a string
LEFT$	Beginning character(s) of a string

LEN	Length of a string
LINE	Maximum line length
LOG	Natural logarithm
LOG2	Binary logarithm
LOG10	Common logarithm
LPAD$	String with space(s) added to the left
LRL	Integer with bits rotated to the left
LRR	Integer with bits rotated to the right
LSL	Integer with bits shifted to the left
LSR	Integer with bits shifted to the right
LTRIM$	String with leading spaced removed
MATCH	True/false to indicate if string matched the mask
MAX	Larger of two values
MID$	Middle portion of a string
MIN	Smaller of two values
MOD	Remainder of one value divided by another
MSEC	Number of milliseconds since midnight
NBR	True/false to indicate if string is a valid number
OCT	Integer value of octal string
OCTOF$	Octal string for integer value
ORD	ASCII value of character
OVR$	String after another string was overlaid onto/into it
PAGE	Maximum page length
PI	Decimal real value of π
PI! or PI!!	Binary real value of π (under UNIX)
POS	Current column position
RAD	Radian equivalent of degrees
REP$	String after a subfield has been replaced
RIGHT$	Ending character(s) of a string
RND	Next pseudo-random number
ROUND	Rounded value

RPAD$	String with space(s) added to the right
RPT$	String with duplicated characters
RTRIM$	String with trailing characters removed
SCH	Position where string of characters was found
SEC	Secant
SECH	Hyperbolic value of secant
SECOND	Time in number of milliseconds since midnight
SGN	Sign of value
SIN	Sine
SINH	Hyperbolic value of sine
SPACE$	String of space characters
SQR	Square root of value
STR$	String corresponding to a number
TAB	Control character to position output to next tab stop
TAN	Tangent
TANH	Hyperbolic value of tangent
TIME$	Time string with number of seconds since midnight
TRIM$	String after all leading, trailing, and extra embedded spaces have been removed
UCASE$	Uppercase equivalent of string
VAL	Numeric value of string
YESNO$	Response from yes/no prompt

CHAPTER 7

Formatted Output

Introduction

CET BASIC provides you with some very powerful tools for formatting output. Since this is such an important issue in an application, this topic is covered here in more detail than in the chapters on statements and built-in functions.

A special feature is provided so that you may use the CRT$ function to send escape character sequences to control printer output on any of the popular printers in use today. (This feature is covered in a separate section at the end of this chapter.)

The PRINT USING statement and the FORMAT$ function may also be used to control the appearance of data, thus enabling the program to create formatted lists, tables, reports and forms. Since these are the most commonly used features, they will be covered first.

In the following example, two programs print a series of numbers. The first program uses the PRINT statement and the second uses PRINT USING.

Program #1:

```
PRINT 1
PRINT 10
PRINT 123.5
PRINT 100
PRINT .23433
PRINT 1000000
PRINT -3
```

Program #2:

```
MASK$ = "#,#######.##-"
PRINT USING MASK$,1
PRINT USING MASK$,10
PRINT USING MASK$,123.5
PRINT USING MASK$,100
PRINT USING MASK$,.23433
PRINT USING MASK$,1000000
PRINT USING MASK$,-3
```

Executing the programs will output:

```
0                       10.00
123.5                   123.50
100                     100.00
0.23433                 0.23
1000000                 1,000,000.00
-3                      3.00-
```

As the example illustrates, the PRINT statement left justifies numbers, performs no rounding and displays negative values with a leading, floating minus sign.

PRINT USING and the FORMAT$ function may be used to format numbers in several ways, making it easier to read and interpret the output. The numeric-formatting functions that may be specified are:

- Number of significant digits.
- Location of decimal point.
- Exponential format.
- Inclusion of special symbols (asterisk fill, dollar sign, commas, leading zeros).
- Alternate methods of indicating any negative values (trailing sign, < >, trailing DB or CR).

The following string-formatting functions may also be specified with the PRINT USING statement:

- Number of characters.
- Left-justified format.
- Right-justified format.
- Centered format.
- Extended format.

All of the formatting functions for the PRINT USING statement and FORMAT$ function are specified by using a mask that contains the formatting information.

The significant difference between these two features is that while FORMAT$ only allows one numeric field to be specified in the mask, PRINT USING allows you to format one or more string or numeric values at the same time.

The PRINT USING statement has been used in the examples in this chapter. Refer to the description of the FORMAT$ function for that syntax to use.

PRINT USING and FORMAT Masks

The PRINT USING and FORMAT$ masks are string expressions that contain the formatting and non-formatting characters to use to control the output.

A PRINT USING mask may contain the format information for more than one field, whereas a FORMAT$ mask may only be used to format one numeric field.

Non-formatting characters are treated as literals or field separators. These characters will be included as-is in the output field. In general, the non-formatting characters are any characters that are not specifically identified in this chapter. Also note that:

- Special formatting characters that operate in groups (\\ $$ ** DB CR ^^ ^^^ ^^^^ ^^^^^) will act as non-formatting characters when they are used separately ($ * D B C R ^).
- Formatting characters such as the comma, period, minus, DB, CR, +, and > act as non-formatting characters if they appear outside a numeric field specification.

For Example:

Execution of the following lines

```
PRINT USING "Example 1: ##",1
PRINT USING "This is a DB record 99",1
```

will output

```
Example 1: 1
This is a DB record 01
```

Numeric Field Masks

Numeric field masks may be used for both PRINT USING and FORMAT$. A numeric field mask requires a numeric input value. All numeric data types are converted to decimal reals before they are formatted.

The length of the output field will always be the length specified in the mask unless there is insufficient space. (See the *Field Specification Too Small*

section.) If necessary, the number will be rounded to the specified decimal precision.

For Example:

Execution of the following lines

```
PRINT USING "##",1.4,1.5,1.6   PRINT USING "##",-1.4,-1.5,-1.6
```

will output

```
1                              -1
2                              -2
2                              -2
```

Specifying Number of Digits

The simplest method of formatting numerics is where each digit in the printed result is represented in the mask by a # or a 9. The position of the decimal point, if any, is specified by a literal '.'. If any characters to the left of the decimal point are #'s, then leading zeros are suppressed. If any character is a 9, all leading zeros are printed.

When no sign is specified and a negative value is output, a leading, floating minus sign will be output in the first position where a leading zero would have been displayed.

For Example:

Execution of the following lines

```
PRINT USING "  #",1        PRINT USING "  9",1
PRINT USING "  ##",1       PRINT USING "  99",1
PRINT USING "#####",1      PRINT USING "99999",1
PRINT USING "9####",1      PRINT USING "#9#9#",1
PRINT USING "#####",-1     PRINT USING "99999-",-1
PRINT FORMAT$(23," ####") PRINT FORMAT$(23,"9####")
```

will output

```
1                            1
   1                         01
    1                     00001
00001                     00001
   -1                     00001-
   23                     00023
```

Decimal Point Specification

A decimal point may be entered in the mask to indicate the decimal precision. The digits to the right of the decimal point will be rounded or, when necessary, extra zeros will be added to display the required precision.

If one or more digits are specified to the left of the decimal point, at least one digit will be output, even if a zero is required to do so, unless that place is required for a minus sign.

Only one decimal point may be specified in a mask. A second decimal point will indicate the end of the mask and the start of another numeric field.

For Example:

Execution of the following lines

```
PRINT USING "   .##",0          PRINT USING " .99",0
PRINT USING "  #.##",1          PRINT USING " 9.99",1
PRINT USING "  ##.##",1.2345    PRINT USING "99.99-",-.2
PRINT USING "#####.####",1.24   PRINT USING " 9.999-",-.234569
```

will output

```
   .00                      .00
  1.00                      1.00
   1.23                    00.20-
    1.2400                  0.235-
```

Comma Specification

Commas may be inserted in the output field by using the comma character in any position to the left of the decimal point specification.

When a comma is used, the output field will be formatted with a comma appearing every third digit from the decimal point (or least significant digit if the decimal point specification is not used), working from right to left.

The comma character is also a digit specifier.

More than one comma may be specified to make the format mask easier to read. For example, #,######### has the same effect as ###,###,###.

For Example:

Execution of the following lines

```
PRINT USING "#,####",1          PRINT USING "#,####",12345
PRINT USING "#,###########",1E9 PRINT USING "#,###,###,###",1E9
PRINT USING "##,####.##",1234.5 PRINT USING "###,###.##,"1234.5
```

will output

```
     1                      12,345
1,000,000,000               1,000,000,000
 1,2344.50                  1,234.50
```

A number may be formatted with a floating dollar sign immediately before the most significant digit by entering two dollar sign characters. To format a number with a dollar sign in the first field position, use a single dollar sign character.

The double dollar sign characters must be at the start of the field unless asterisk fill is specified. In that case, the characters must immediately follow the asterisk fill specification.

The double dollar sign characters will use one position for the floating dollar sign and reserve one position for a digit. If the number to be formatted is negative, a sign specification must be used to avoid error.

The numeric field specification character 9 may not be used in a field with floating dollar sign specification. If it is used, the 9 will be interpreted as the end of the field and the start of the next numeric field.

Extra dollar sign characters may be used instead of the # character. For instance, $$$$$$$$ is the same as $$######.

For Example:

Execution of the following lines

```
PRINT USING "$$###.##",12      PRINT USING "   $$",1
PRINT USING "$$###.##",1234    PRINT USING "$$,#####.##",12345
PRINT USING "$$###.##-",-1234  PRINT USING "$$######.##-",-12345
```

will output

```
  $12.00                          $1
$1234.00                       $12,345.00
$1234.00-                       $12345.00-
```

Asterisk Fill Specification

To format a number with leading asterisks, enter two asterisk characters at the beginning of the field specification. The double asterisk characters indicate that any leading zeros are to be replaced with asterisks and that one position is to be reserved for digits.

If the number to be formatted is negative, sign specification must also be specified.

The numeric field specification character 9 may not be used in a field with asterisk fill specification. If it is, the 9 will be interpreted as the end of the field and the start of the next numeric field.

Extra asterisk characters may be used instead of the # character. For instance, ******** is the same as **######.

For Example:

Execution of the following lines

```
PRINT USING "**##.##",123     PRINT USING "****.##",123
PRINT USING "**##.##-",-123   PRINT USING "**$$.##",1
PRINT USING "**$$$$$-",-2     PRINT USING "****",.5678
```

will output

```
*123.00                       *123.00
*123.00-                      **$1.00
*****$2-                      ***1
```

Sign Specification

There are several methods of indicating how to display signed values. When the mask field does not explicitly specify how to format a negative value, a leading minus sign is generated, by default. Unless the format mask includes leading zeros (9), a floating dollar sign ($$) or asterisk fill (**), one of the sign specifications must be entered.

When sign specification characters are used, they must be entered at the end of the format field. If the characters appear at the beginning or middle of the field, they will be treated as non-formatting characters or field separators, respectively.

The following formatting specifications may be used to indicate how negative numbers should be displayed.

Trailing Sign

A plus sign character at the end of a format specification indicates that the sign of the field (+ or -) is to be output at the end of the number.

Trailing Minus Sign

A minus sign character at the end of a format specification indicates that negative numbers should be displayed with a minus sign at the end.

Trailing Debit Sign

Debit specification characters (DB) appearing at the end of a format specification will display negative numbers with the literal DB to the right.

Trailing Credit Sign

Credit specification characters (CR) appearing at the end of a format specification indicate that a literal CR is to be output at the end of the number when the value is negative.

Angle Bracket

An angle bracket character (>) at the end of a format specification indicates that negative numbers are is to be surrounded with angle brackets. The left angle bracket replaces the right-most (suppressed) leading zero digit.

Remember that this specification requires two display positions when calculating the maximum length of a field.

Also note that this sign specification may not be used with the numeric field specification characters 9, $$, or **.

Except for the angle bracket, negative value specifications may be used with any other numeric field formatting characters with the exception of exponential field specification.

For Example:

Execution of the following lines

```
PRINT USING "####+",123          PRINT USING "####+",-123
PRINT USING "####-",123          PRINT USING "####-",-123
PRINT USING "####DB",123         PRINT USING "####DB",-123
PRINT USING "####CR",123         PRINT USING "####CR",-123
PRINT USING "####>",123          PRINT USING "####>",-123
PRINT USING "######.##>",12      PRINT USING "######.##>",-12
```

will output

```
123+                    123-
123                     123-
123                     123DB
123                     123CR
123                     <123>
12.00                   <12.00>
```

Exponential Field Specification

BASIC normally prints a number in scientific notation (E format) only when its printed representation is longer than 13 digits. For example, 123456789000000 would be printed as 1.23456789E+014.

If PRINT USING or FORMAT$ is used, any number may be output in E format by entering the exponential field specification (^^^^^) at the end of the numeric mask. When an exponential number is to be formatted, only the characters specifying the number of digits (#), the decimal point position (.) or the E part (^) may be used.

The exponential field specification may be entered with less than five caret characters if it is known that the exponent will fit in a smaller specification.

Specification	Formats
^^^^^	Exponents from –126 to + 126
^^^^	Exponents from –99 to +99
^^^	Exponents from –9 to +9
^^	Exponents from 0 to 9

Execution of the following lines

```
PRINT USING "#.##^^^^^",124
PRINT USING "########^^^^^",123445
PRINT USING "#####.####^^^^",12345678
PRINT USING "#.#####^^",1234567
```

will output

```
 1.24E+002
 12344500E-002
 12345.6780E+03
 1.23456E6
```

Field Specification Too Small

When a field specification does not specify sufficient digits to display the number, a percent character (%) will be output followed by the unformatted number. This situation can occur for several reasons:

- Field is not large enough to include all the significant digits
  Example: mask = ### number = 1234
- Field is not large enough to include the commas specified
  Example: mask = #,### number = 12345
- Field is not large enough to include floating dollar sign
  Example: mask = $$### number = 12345
- Field is not large enough to include leading minus sign
  Example: mask = ### number = -123

In the following examples a double field mask is used to print two numbers. The first number will not fit in the mask, but the second number which is identical will fit in the second mask.

For Example:

Execution of the following lines

```
PRINT USING "### ####.##",1234,1234
PRINT USING "#,### #####",12345,12345
PRINT USING "$$### #####",12345,12345
PRINT USING "**### #####",12345,12345
PRINT USING "### ###-",-123,-123
PRINT USING "#.##^^^^^ ##.##^^^^^",-1,-1
PRINT USING "#.##^^ #.##^^^^",1E+12,1E+12
```

will output

```
% 1234              1234.00
% 12345             12345
% 12345             12345
% 12345             12345
%-123               123-
%-1                 -1.00E+000
% 1000000000000     1.00E+12
```

String Field Masks

String field masks may only be used with the PRINT USING statement, and not FORMAT$ since this type of mask requires a string value as input.

The length of the output will be the same length as the mask except when using extended fields. If a string value is longer than the string field mask, BASIC will normally print as much of the string that will fit and truncate the extra characters on the right.

Single Character

To specify that only the first character of the string value is to be printed, use the single quote character (‘) or the exclamation mark (!) as the string mask.

For Example:

Execution of the following line

```
PRINT USING "!","ABCDEFGH"     PRINT USING "'","XYAX"
```

will output

```
A                              X
```

Left-Justified Field

To specify a left-justified, string field, use the single quote character (') followed by one or more L characters in either upper or lower case. The number of L characters plus the lead-in quote indicate the length of the field.

Alternately the backslash character (\) may be used to mark the beginning and end of the string field. In this format, the number of spaces plus the two back slant characters specify the field length. With either method, the minimum string length is two.

BASIC prints a left-justified, string field starting at the left-most position. If there are any unused places, spaces are printed after the string. If there are more characters in the string value than in the mask, BASIC truncates the string value on the right.

For Example:

Execution of the following lines

```
PRINT USING "'L","ABCDEF"       PRINT USING "'LLLLL","1234567890"
PRINT USING "'LLL","1234567"    PRINT USING "'LLLLL","AB"
PRINT USING "\   \*","ABC"      PRINT USING "\   \*","ABCDEFG"
```

will output

```
 AB                             123456
 1234                           AB
 ABC *                          ABCDE*
```

Right-Justified Field

To specify a right-justified, string field use the single quote character (') followed by one or more R characters in either upper or lower case. The number of R's plus the lead-in quote should indicate the output field length.

BASIC prints a right-justified, string field so that the last character of the string is in the right-most position. If there are any unused places before the string, they are filled with spaces. If there are more characters in the string value than in the mask, BASIC truncates the string on the right.

For Example:

Execution of the following lines

```
PRINT USING "'RRRRRR","ABCD"
PRINT USING "'RRRRRR","AB"
PRINT USING "'RRRRRR","ABCDEF"
PRINT USING "'RRRRRR","ABCDEFGHIJKLMNOP"
```

will output

```
   ABCD
     AB
 ABCDEF
ABCDEFG
```

Centered Field

A centered string field is specified by using the single quote character (') followed by one or more C characters. The number of C characters (upper or lower case) plus the lead-in quote indicate the length of the field.

BASIC prints the string so that its center is in the center of the field. If the string cannot be centered, such as a two character string in a five character

field, BASIC prints the string one character off center to the left. If the length of the string is longer than the mask field, the string will be truncated.

For Example*:*

Execution of the following lines

```
PRINT USING "'CCCCCCCCCC","ABC"
PRINT USING "'CCCCCCCCCC","ABCDEF"
PRINT USING "'CCCCCCCCCC","A"
PRINT USING "'CCCCCCCCCC","ABCDE"
PRINT USING "'CCCCCCCCCC","ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

will output

```
      ABC
    ABCDEF
       A
     ABCDE
  ABCDEFGHIJK
```

Extended Field

The extended field specification is the only mask that may be used to print the entire string. BASIC left-justifies the string as it does for a left-justified field. If the string has more characters than are indicated in the mask, BASIC will extend the field and automatically print the entire string. Note that using this feature may cause other items to become misaligned.

To specify an extended field, use the single quote lead-in character (') followed by one or more E's in either upper or lower case. The number of E's plus the lead-in quote indicate the minimum length of the output field.

For Example:

Execution of the following lines

```
PRINT USING "'E-","ABCDEF"
PRINT USING "'EEEE-","ABCDEF"
PRINT USING "'EEEEEEEEEEEEEEEEEE-","ABCDEFGHIJKLMNOP"
```

will output

```
 ABCDEF-
 ABCDEF-
 ABCDEFGHIJKLMNOP-
```

Multiple Fields In One Mask

The PRINT USING statement allows multiple fields to be specified in one mask. When this is done, the values of the expressions in the statement are matched in a one-to-one relation with the fields in the mask. (The FORMAT$ function only allows one numeric field to be used in the mask. If a second field is specified, it will be used to mark the end of the mask.)

For Example:

Execution of the following lines

```
PRINT USING "### #### #### ##%",1,2,3,4
PRINT USING "999 9999 9999 99%",100,123,5,2
PRINT USING "'RRRRRRRRRR ### 'E","Item",23,"Description"
```

will output

```
 1     2  3   4%
100  0123 0005  02%
    Item 23Description
```

As mentioned earlier, any non-formatting characters in the mask field are treated as literal characters to be included in the output:

For Example:

Execution of the following lines

```
PRINT USING "ITEM 9999 Amount each: $$$$$$.##",23,15.40
```

will output

```
ITEM 0023 Amount each:   $15.40
```

Re-using Mask Fields

The PRINT USING statement will re-use the mask field if there are more input values specified than there are fields in the mask. BASIC will output an end-of-record delimiter each time the mask is used.

For Example:

Execution of the following lines

```
PRINT USING "$$$$$$,$$$.##",1,23.4,34,234,5467.2,1235.924
```

will output

```
     $1.00
    $23.40
    $34.00
   $234.00
 $5,467.20
 $1,235.92
```

CRT$ Function Mapping for Printers

CET BASIC provides the B_MAPPRT*n* environment variable as a mechanism to simplify the mapping of the CRT$ functions to the escape sequences that are to be sent to the specific printers that are used in your application. All the required information is stored in an external, text file.

For example, suppose you have defined the escape sequences for an HP Laserjet printer in the **hplaser.lpr** file. To use the map file on the printer opened as PRINTER1, set the B_MAPPRT1 environment variable as:

```
SET B_MAPPRT1=HPLASER.LPR
```

This statement will ensure that when PRINTER1 is opened, all output generated by the CRT$ functions will refer to the **hplaser.lpr** file (in the current directory) to determine which escape sequences should be sent.

A variety of printer map files may be defined for the commonly used printers. The syntax for the entries in a printer map file is:

CRT-*argument* = *string*

The CRT-*argument* must be one of the valid CRT$ functions such as HOME and CLEAR.

The *string* value must be an ASCII string containing the characters that you wish to have sent to the printer by the CRT-*argument*.

The *string* may also contain any of the following metacharacters:

Metacharacter	To be substituted by:
^*letter*	The value of the CTRL-*letter*
#*number*	The character whose ASCII value is specified by the *number.* Note that ASCII values of less than 32 must be preceded by CHR(255) to force output to the printer.
%*letter*	The character whose ASCII value is specified by the *letter* +128

^^	The carat symbol (^)
##	The pound symbol (#)
%%	The percent symbol (%)

For example, an entry in a printer map file might be:

```
HOME=A^A^^B#13##C%1%%D
```

A BASIC program that executes a PRINT #3: CRT(“HOME”) statement would use the above information to send the following escape sequence to the printer opened on channel #3:

```
A<CTRL-A>^B<carriage-return>#C<decimal 129 character>%D
```

CHAPTER 8

Color and Line Drawing Features

Introduction

CET BASIC provides a variety of ways to implement color in your application. You have:

- The CRT$("FC*x*") and CRT("BC*y*") functions that may be used to specify up to 16 foreground and background colors.
- The COLOR statement which provides the eight colors that are available under the THEOS operating system.
- The BW.COLOR function available to programs that use the (optional) Window System package.
- The environment variables B_DFLTBGC and B_DFLTFGC that may be used to set the default background and foreground colors to be used in the (entire) CET application.

The color features that are documented in this chapter use the CRT$ function. This feature is available unless you have set the B_EMULATE environment variable.

The CET BASIC line drawing features that are documented here are intended to be compatible with THEOS BASIC.

Line Drawing Characters

The line-drawing characters are obtained by using either CHR$ with the ASCII value of the character or CRT$ with its mnemonic value. When entering the argument as a character string, case is not important. Both of the following statements will print a horizontal line character:

```
PRINT CHR$ (169);
```

```
PRINT CRT$ (“HORZ”);
```

CET BASIC supports the following line drawing graphic characters:

Mnemonic	ASCII	Description
DDI	183	Double Down Intersection
DFWI	179	Double Four-Way Intersection
DHOR	184	Double Horizontal Line
DI	168	Double Intersection
DLI	180	Double Left Intersection
DLLC	178	Double Lower Left Corner
DLRC	177	Double Lower Right Corner
DRI	181	Double Right Intersection
DUI	182	Double Up Intersection
DULC	175	Double Upper Left Corner
DURC	176	Double Upper Right Corner
DVER	185	Double Vertical
FWI	164	Four-Way Intersection
HOR	169	Horizontal Line
LI	165	Left Intersection
LLC	163	Lower Left Corner
LRC	162	Lower Right Corner
RI	166	Right Intersection
RLLC	174	Round Lower Left Corner
RLRC	173	Round Lower Right Corner
RULC	171	Round Upper Left Corner
RURC	172	Round Upper Right Corner
UI	167	Up Intersection
ULC	160	Upper Left Corner
URC	161	Upper Right Corner
VERT	170	Vertical Line

Color Attributes

When THEOS emulation is not set, the CRT$ function may be used to change the colors on the video screen. The arguments are:

Fc*x* Foreground Color *x*
Bc*x* Background Color x

The *x* in the argument may be any of the following letters or numbers to represent a color:

Letter	Number	Color	Letters	Number	Color
	0	Black	L	8	Gray
B	1	Blue	LB	9	Light Blue
G	2	Green	LG	10	Light Green
C	3	Cyan	LC	11	Light Cyan
R	4	Red	LR	12	Light Red
M	5	Magenta	LM	13	Light Magenta
LY	6	Brown	Y	14	Yellow
W	7	White	LW	15	High White

The colors numbered from 0 to 7 may also be used in the COLOR statement and will be supported under THEOS emulation. For example, the following statements will all produce the same results:

```
COLOR 7,4
PRINT CRT$("FC7");CRT$("BC4");
PRINT CRT$("FCW");CRT$("BCR");
```

Color codes returned from CRT$("FC*x*") and CRT$("BC*y*") may be combined, saved in variables and printed to non-terminal files. They are designed to be machine independent.

The disadvantages of using CRT$ to display color is that, unlike the COLOR statement which explicitly sets reverse video foreground and background colors, a current color pair is simply inverted when RVON is encountered. However, specifying a different set of color pairs for normal and reversed video is easily simulated in CET BASIC. For example:

```
rem 'normal' screen colors are set to white on blue:
norm$=crt$("fc7")&crt$("bc1")
rem reverse video colors are set to black on white:
rv$=crt$("fc0")&crt$("bc7")
```

```
rem clear the screen to display the new colors:
print norm$;crt$("clear")
print "hello"
print rv$;"hello in reverse video";norm$
```

Note that color is also supported under the Window System with special functions to change the color of the window, the text, and/or the window frame. Refer to the *CET BASIC Window System Manual* for details.

CHAPTER 9

The C Preprocessor

Introduction

CET BASIC allows you to use C-style preprocessing from within a BASIC program. All of the directives will be processed before BASIC compilation, hence the name. For example, if the preprocessor directive #include specifies that an external file is to be included in the source program, the directive will be replaced with the entire file before the compilation process begins.

A separate C compiler is not required in order to use these features with the CET DOS or PC UNIX-based products. Check with your CET distributor regarding availability with other CET products.

Preprocessor Directives

All of the valid preprocessor directives are described in alphabetical order in this section. Each directive must be entered in a single line beginning with a # symbol. Note that all commands must be in lower case; #DEFINE would be considered illegal.

#define *name token-string*

This #define directive defines a simple macro *name* and assigns its replacement value. All subsequent occurrences of *name* will be replaced with the *token-string* until a corresponding #undef is encountered for the macro name or the end of the source file is reached.

In the following example, a #define directive is used so that each time "dos" is encountered in the program, it will be replaced with "\ats".

```
#define pos \ats
```

#define *name(arg1, ..., argn) token-string*

This form of the #define directive defines a *function-like* macro with arguments. Use this syntax to replace all subsequent occurrences of the *name*(*arg1, .., argn*) with the *token-string* after having placed the value of the *arg* in the *token-string.* Note that there can be no spaces between *name* and the right parenthesis.

After replacement in either type of mode of #define, the replacement string is rescanned for further references to the macro names that have been defined.

#else

The #else directive is used in conjunction with #if to provide source lines to be performed when the result of the matching #if directive is false. See #if for details.

Directives that perform condition tests may be nested. In that case, the #else clause is optional at each level.

#endif

The #endif directive is required to terminate each #if, #ifdef, and #ifndef directive.

#if *constant-expression*

The #if directive tests the *constant-expression* for a non-zero (true) value. In that case, the source lines following the #if directive, up to the next matching #else or #endif directive are included during the compilation processes.

The *constant-expression* must be made up of actual or #define constants and C operators (with their normal precedences) other than assignment, & and size operators.

The #else in the following example is optional:

```
#if exp
      source line 1
#else
      source line 2
#endif
```

#include "*filename*"

This is similar to the CET BASIC #include statement, with the following exceptions:

- The word include must be in lower case.

- The *filename* must be quoted.

If the specified *filename* is not found in the current directory or in any of the directories entered with the include flag, the standard C include directory will be searched. Under SCO UNIX this directory is usually **/usr/include**.

#ifdef *name*

The #ifdef directive is similar to #if except that only tests for whether a macro *name* has been defined or not. The lines following this directive will appear in the program only if *name* has been previously specified using #define and not subsequently used in an #undef.

#ifndef *name*

The #ifndef directive is similar to #if except that it can only test for whether a macro *name* has not been defined. The lines following this one will appear in the program only if *name* has not been previously specified using #define or has subsequently been used in an #undef command.

#undef *name*

The #undef directive causes the definition of *name* to be removed from this point on.

It is important to note that unlike CET BASIC, the C preprocessor is case-sensitive. In order for *name* to be recognized and replaced, it must be in the same case mode used to specify the *name* with #define. For example:

```
#define Abc 123.4
X = Abc
Y = ABC
```

The code above will produce:

```
X = 123.4
Y = ABC
```

Also note that C-style names must be used. Therefore, a period is not a legal character.

Compiling Programs With the C Preprocessor

The following flags are available to compile BASIC programs that contain C preprocessor commands.

If an error occurs and the -e switch is in use, you will be given the option of editing the program file. However, if there is a preprocessor error because you have used #defune instead of #define, CET BASIC will not stop as the compiler has no way of determining where the error occurred.

Preprocessor Flag (-P)

Invoke the C preprocessor before the BASIC compilation is performed.

Preprocessor Only Flag (-PO)

If the -PO flag is specified, only the preprocessing is done and the result is stored in a file with the same name as the source file and the **.i** extension.

Define Symbol Flag (*-Dsymbol*)

This flag may be used to define and assign a value for the preprocessor symbol #define. For example:

```
ob -P -Dabc -Ddef=xyz newpgm.b
```

This command would be equivalent to entering the following lines at the top of the source file **newpgm.b**:

```
#define abc
#define def xyz
```

Note that the -D parameter will be terminated by the first blank, unless it is enclosed in quotes.

Since CET BASIC does not process the -D*symbol* flag itself, its use implies the -P flag which can be overridden by specifying -PO.

CHAPTER 10

CET BASIC Error Handling System

Introduction

CET BASIC includes facilities for trapping errors and anticipated conditions that may occur during the execution of a program. These errors and conditions include:

- Errors which may arise from improper program execution such as a Truncated Record error.
- Conditions which may arise in normal program execution which require special handling such as a locked record or a file not found condition.
- Abnormal errors which are detected by the operating system in use.

The CET BASIC Error Handling System is a component of the runtime system. It manages:

- The proper execution of the ON ERROR, ON LOCK, ON INTERRUPT, ON MOUSE and RESUME statements.
- The orderly termination of programs which do not have active ON statements.

This chapter covers the details of the error handling system. For other information on syntax and usage, refer to the description of the individual statements.

The ON ERROR Statement

The ON ERROR statement is the mechanism most commonly used to detect and respond to program errors and significant conditions. When an ON ERROR routine is in effect, it must handle a locked file or record condition

unless a special ON LOCK routine is active. Otherwise, the default is to abort with an error message when a locked condition is encountered.

The statement takes one of two forms:

ON ERROR GOTO *line-reference*
ON ERROR GOTO 0

The first form will activate the ON ERROR handler whose first statement is to be found at the *line-reference* which may refer to either a statement number or label name.

The ON ERROR GOTO 0 statement disables the ON ERROR handler. Any subsequent errors will be handled by the BASIC system which displays an appropriate error message and terminates execution.

A user-defined ON ERROR routine may check the value of the following functions to determine what action to take, if any.

ERR

The number associated with the error which invoked the ON ERROR routine. (A list of the error messages may be found in the *Appendix*.)

ERL

The number of the statement which was executing when the error occurred. In programs which contain statements with and without line numbers, the ERL value will refer to the last successfully executed statement that had a number.

ERF

The number of the channel on which a file input/output operation was performed just prior to the error.

An ON ERROR routine should always finish by executing a RESUME statement. A GOTO statement should not be used, otherwise the results will be unpredictable.

RESUME

Re-executes the statement which caused the error to occur, unless the program was interrupted by the operator (ERR=1). In that case, RESUME will return to the statement following the one where the interrupt was detected.

RESUME *ref*

Resumes execution at the label or statement number designated by *ref*.

RESUME 0

Invokes the BASIC system error handler which will display an appropriate error message and terminate the program.

QUIT/END/STOP

Terminates the program.

The ON INTERRUPT Statement

The ON INTERRUPT statement may be used to detect the occurrence of the system Interrupt Key. Under DOS, the Interrupt Key is Ctrl+C. Under UNIX, it is the DELETE key, but the Interrupt Key can be set to another value with the **stty** command.

Consider the following program segment. When an Interrupt Key is detected by the ON INTERRUPT statement, control is transferred to the label intrlab.

```
ON INTERRUPT GOTO intrlab
.
.
intrlab:
PRINT "Do you wish to abort (Y/N)";
LINPUT USING "!", ans$
IF UCASE$(ans$) = "Y" THEN QUIT
RESUME
```

The RESUME statement works differently for ERR=1 conditions than it does for other ERR values. If an ON INTERRUPT condition is generated or an ON ERROR unit is activated with an ERR=1, any subsequent RESUME will return to the statement following the one in which the interrupt occurred. Unlike other conditions in which RESUME is used, the statement will not be re-executed because it has already been performed *successfully*.

When both ON INTERRUPT and ON ERROR routines have been activated, the ON INTERRUPT will take precedence in handling an Interrupt Key.

The ON INTERRUPT statement should be included in all BASIC programs, unless all ON ERROR routines include a check for ERR=1. Failure to do so will mean that a program can not be interrupted. In that case, the program can only be terminated by rebooting the system.

In general, the best way to handle unanticipated errors is to execute a RESUME 0 in the error handler routine.

The ON LOCK Statement

The ON LOCK statement is an event detection statement that works in a manner similar to ON ERROR. When a direct or indexed file is opened with option UPDATE, various I/O statements can cause the program to be suspended if a locked file or record is encountered.

The ON LOCK statement may be used to detect this condition. Consider the following program segment:

```
ON LOCK GOTO locklab
.
.
locklab:
PRINT "Lock on File Number", ERF
SLEEP 2
RESUME
```

In the example above, lock detection is activated by execution of the ON LOCK statement. When a locked record or file is encountered, control is transferred to the label locklab. The ERF function will contain the channel number of the file for which the lock was detected. RESUME works exactly as it does in ON ERROR detection routines, and will reexecute the I/O statement that caused the record lock condition.

Note that if the B_EMULATE variable is set, a locked record is not considered a trappable error. If you wish to use THEOS emulation and special error handling, set the B_THLOCK variable off (to null).

The ON ERROR statement will also detect a record lock or file lock condition. The ERR values would be:

Value	Condition
48	Record Lock
49	File Lock
50	Potential Deadlock

When an ON LOCK is activated, it takes precedence over an active ON ERROR routine. To handle a lock detection, one should code the ON LOCK routine to suspend execution for the duration of the lock condition. This may

be accomplished by coding the ON ERROR module in the following way. This same behavior may be accomplished by setting B_THLOCK.

```
onerror.lab:
IF ERR = 48 OR ERR = 49 OR ERR = 50
      SLEEP 1
IFEND
RESUME
```

This code will not properly handle locks that occur during READNEXT or READPREV statements. Because RESUME reexecutes the entire statement, subsequent attempts to access the record will be executed with different key values.

This condition may be corrected by setting the environment variable B_OPTLOCK so that the file pointer is not moved after the RESUME statement and the (MAT) READNEXT or (MAT) READPREV statements will attempt to read the desired record again.

The ON MOUSE Statement

The ON MOUSE statement works identically to the other statements described in this section. To use this feature, you must load a mouse driver and set the environment variable B_MOUSESUP.

A mouse click terminates input from a WAIT, GET, INPUT, LINPUT or LINPUT USING statement. This permits the operator to use the mouse to enter a response or select an item. For instance, a program might set MOUSE.CLICK% to FALSE%, and then after a LINPUT determine if the statement ended with a string value, an INP value or a mouse click.

Consider the following code segment. When a mouse click is detected by the ON MOUSE statement, control is transferred to the mouselab subroutine. Upon returning, a test could be made for MOUSE.CLICK%=TRUE% to determine what response is associated with the indicated row and column.

```
ON MOUSE GOTO mouselab
.
.
mouselab:
MOUSE.CLICK%=TRUE%
LAST.BUTTON%=ERM_BUTTON
LAST.ROW%=ERM_ROW
LAST.COL%=ERM_COL
```

```
RESUME
```

Note the use of the following variables:

ERM_BUTTON	Returns 0 for a left button click and 1 for a right click
ERM_ROW	Returns the row position of the last button click (the button release).
ERM_COL	Returns the column position of the last button click (the button release).

This feature is only available in the CET DOS and Networking product. The CET W/32 Application Builder for Windows implements mouse support using another method.

Message Files

CET BASIC generates error messages and warnings under various conditions. The text for most of these messages is contained in external text files. The error and compiler diagnostic messages may be modified to reflect local requirements or rewritten completely in accordance with the rules of message definition.

A detailed discussion of the CET BASIC Message System is given in the next chapter. Refer to the *Appendix* for a listing of the errors that may be detected.

CHAPTER 11

CET BASIC Message System

Introduction

The CET BASIC compiler, runtime systems, and development utilities generate error messages and diagnostic information during execution. This data is read from external files stored in an ASCII sequential format. All of the files have the same format so they may be easily read and even edited, if necessary.

The files used to generate the messages are as follows:

CET BASIC Program	DOS File Name	UNIX File Name
Compiler	*path***bcomp.err**	*/lib/***Bcomp_err**
Runtime System	*path***brun.err**	*/lib/***Brun_err**
Development Utilities	*path***butil.err**	*/lib/***Butil_err**

This chapter will provide you with the information you need if you wish to modify any of the **.err** files. Refer to the *Appendix* for a listing of the messages that are generated by default.

CET BASIC Compiler Messages

The CET BASIC compiler prints out an error message when it is invoked with improper command syntax or when an incorrect BASIC statement is detected during compilation.

Compiler messages are stored in the file **\bcomp.err** or **/Bcomp_err.** This message file is composed of individual records that may be edited using any text editor which preserves this format.

Each record in the message file has the following format:

msg-number* TAB *message

msg-number

This is a unique number associated with the message for a particular error or condition.

message

This is the default *message* text that corresponds to the specified *msg number*. Each *message* may contain text, control characters and punctuation. The text is printed exactly as it is found in the file.

Upon printing, each *message* will be output followed by a carriage-return+linefeed, unless the *message* is terminated by two colons ("::").

Control words of the form {*number*} are replaced at execution time with the corresponding argument. For example, the syntax message:

```
Line 190 referenced but not defined
```

is displayed by the CET BASIC compiler when a program references a line number that does not exist. This message is printed by a routine which uses message 604, which is defined in **\bcomp.err** as

```
Line {0} referenced but not defined
```

Similar messages may be modified, but the references in curly braces should be preserved to avoid loss of information regarding the error. For example, the previous message may be edited to read:

```
La linea {0} a que se refiere no está definida
```

Control words with the following format:

{*number* ? *stg1* = *stg2,stg3*}

are printed according to the following rule:

IF error-param *number* = *stg1* THEN PRINT *stg2* ELSE PRINT *stg3*

String designators such as *stg1* may be an asterisk or a sequence of zero (0) or more characters, without surrounding quotation marks. The asterisk designator is substituted for the parameter value itself.

For example, the control string {1?1 = one,*} corresponds to:

```
IF error-parameter 1 = "1"
    THEN PRINT "one"
    ELSE PRINT whatever it is.
```

The control string {1?1=,S} corresponds to:

```
IF error-parameter 1 = "1"
    THEN PRINT nothing
    ELSE
        PRINT "s"
```

This format is often used within **\bcomp.err** to generate English plurals such as those used in message 649.

If the file **\bcomp.err** is not found or unreadable, the CET BASIC compiler will print internally stored error messages.

CET BASIC RunTime Messages

CET BASIC programs execute under the control of the system procedures which detect errors when they occur. When ON ERROR, ON INTERRUPT, and ON LOCK statements are not active, the standard system error handler prints the message associated with the condition.

The runtime messages are stored in the file **\brun.err**. This message file has a format similar to **\bcomp.err,** and may be edited by any text editor which preserves the format.

Each record in the file consists of:

ERR-*number* TAB *message*

ERR-*number*

This is the ERR-*number* which will be stored in ERR if an ON ERROR was used to trap this error.

message

The default *message* text that is printed when the corresponding ERR-*number* occurs. For example, **\brun.err** contains the following message:

```
9    out of dynamic memory space
```

This record may be edited to produce a different error message when the condition associated with ERR= 9 occurs.

If the file **\brun.err** is absent or unreadable, CET BASIC will print internally stored error messages.

CET BASIC Utility Messages

The CET BASIC product includes a number of utility programs such as Bcreate and Blist. If an errors is detected during the execution of any of the utility programs, a message from the file **\butil.err** will be displayed.

This file is formatted just like the other CET BASIC message files so that it may be easily read and modified, if necessary.

The messages from all the CET BASIC error files have been listed in the *Appendix* for your convenience.

APPENDIX A

CET BASIC Error Messages

Introduction

All of the error conditions that can be detected by the CET BASIC compiler, runtime systems, and development utilities have been listed here for your convenience. All of these errors are trappable unless otherwise indicated.

All DOS hard errors are also trapped and reported via errors 301 to 312. This means that your program can detect and account for network errors and conditions such as writing to a *bad* diskette. If a program does not trap for these errors, a message will automatically be displayed in a window on the user's screen before the program aborts.

Note that new CET BASIC features are added to the product on an on-going basis. If your program displays a message that is not listed here, check the CET BBS System for a copy of the current message file.

Compiler Error Messages

The following messages are stored in the **\bcomp.err** file.

2 Unable to link
16 Invalid item in expression
25 File name missing
40 Program too large; out of memory
41 Compiler error {0}; please notify your supplier
42 Bad preprocessor directive
43 #includes nested too deeply
44 #include file {0} not found
45 BASIC compilation terminated by signal {0}
46 Too many numbered program lines
50 Basic usage: '{0} [-switches] filename'

51	No -S file specified
52	-S specified twice
53	Too many -I libraries
54	Bad switch: {0}
55	More than one source file
56	.b extension missing from source file
57	Could not open {0}
58	Could not open work file {0}
59	Fatal error(s); no object produced
60	No -o file specified
84	Bad parameter conversion; put DEF before use
85	BREAK outside loop
98	CET Software Key not present, check parallel port
101	Complete pathnames such as {0} are not supported, use B_LKPATH
120	Line number out of order or invalid
127	Comma required
128	Colon required
129	End of line required
130	Equal sign required
133	Illegal or misspelled keyword
134	Missing parenthesis
135	Numeric expression required
138	String expression required
139	Terminating quote required
140	Too many subscripts
142	Unrecognized statement
144	Variable required
148	Invalid numeric
149	Integer required
316	Nested functions not allowed
604	Line {0} referenced but not defined
605	{0?1=One,*} function{0?1=,s} used but not defined
606	Undefined label: {0}
607	Identifier too long
608	Illegal character
609	Inconsistent usage
610	'(' expected
611	',' or ')' expected
616	Illegal separator
617	BASE must be 0 or 1
618	Too late to specify BASE

619 Bad PRIVILEGE
620 Bad serial
621 Illegal option
622 '#' expected
623 '-' expected
624 RESTORE after last DATA line
625 Bad line reference
656 IF without IFEND::
626 THEN expected
627 GOTO/GOSUB expected
628 TO/SUB expected
629 ON ... GO has too many line references
630 Too few parameters
631 TAB outside PRINT
632 Too many parameters
633 Illegal parameter type
634 Illegal function name
635 Parameter name repeated
636 Incorrect number of parameters
637 FOR index can't be subscripted
638 Too many FOR values
639 TO expected
640 Wrong variable in NEXT
641 Bad mode
642 Bad method
643 Bad subscript
644 Illegal expression
645 SELECT without CEND::
646 CASE expected
647 Too many RESTORE locations
648 Improper use of function name
649 at {1?1=one,*} line{1?1=,s} after line {2}::
650 at {1?1=one,*} line{1?1=,s} after label {2}::
651 at {1?1=one,*} line{1?1=,s} after program start::
652 at line {2}::
653 at label {2}::
654 , ::
655 .
657 WHILE without WEND::
658 Function already defined
659 DEF without FNEND::

660 FOR without NEXT::
661 WEND without WHILE
662 NEXT without FOR
663 ELSE without IF
664 CASE without SELECT
665 OTHERWISE without SELECT
666 IFEND without IF
667 THEN without IF
668 CEND without SELECT
669 FNEND without DEF
670 Unknown error {0}
671 Duplicate label
700 Expecting SORT keyword
701 Expecting SORT range
702 Compiler error : stack calculation
703 LOCK option must be numeric

Runtime Messages

The following messages are stored in the **\brun.err** file.

1 program interrupted
2 divide by 0
3 arithmetic Overflow
4 arithmetic Underflow
5 illegal number
6 square root of negative number
7 log of 0
8 log of negative number
9 out of dynamic memory space
10 RETURN without GOSUB
11 FNEND without function call
12 GOSUB without RETURN
13 GOSUBs nested too deeply
14 Binary floating point error
15 Illegal matrix operation
16 illegal unit number
17 RESUME without error
18 Swap arguments must be of same type
21 tried to reopen file not previously open
22 illegal record number
24 illegal file mode

25 subscript out of range
27 file not open
28 file is still open
29 illegal file name
30 file not found
31 Disk full
33 file protected
36 out of data
37 OPTION BASE after DIM
40 program not found
46 device not attached
47 truncated record on WRITE/PRINT
48 record is locked
49 file is locked. run SHARE on each workstation.
50 potential file deadlock
98 CET Software Key not present, check parallel port
99 invalid item in binary file
231 UNIX/XENIX Signal 1 hangup
232 UNIX/XENIX Signal 2 interrupt
233 UNIX/XENIX Signal 3 quit
234 UNIX/XENIX Signal 4 illegal instruction
235 UNIX/XENIX Signal 5 trace trap
236 UNIX/XENIX Signal 6 IOT instruction
237 UNIX/XENIX Signal 7 EMT instruction
238 UNIX/XENIX Signal 8 floating point exception
240 UNIX/XENIX Signal 10 bus error
241 UNIX/XENIX Signal 11 segmentation violation
242 UNIX/XENIX Signal 12 bad argument to system call
243 UNIX/XENIX Signal 13 write on a pipe with no one to read it
244 UNIX/XENIX Signal 14 alarm clock
245 UNIX/XENIX Signal 15 software termination
246 UNIX/XENIX Signal 16 user defined signal 1
247 UNIX/XENIX Signal 17 user defined signal 2
248 UNIX/XENIX Signal 18 death of a child
249 UNIX/XENIX Signal 19 power-fail restart
300 CSH argument too long
301 Attempted write to protected disk
302 Unknown I/O unit
303 Drive not ready
304 CRC error in data
305 Bad drive request structure

306 Seek error
307 Unknown media type
308 Sector not found
309 Printer out of paper
310 General write fault
311 General read fault
312 General failure
999 Unknown Error

Utility Program Messages

The following messages are stored in **\butil.err:**

1 Insufficient memory.
2 Can't open file "{0}".
3 Can't read file "{0}".
4 Can't create file "{0}".
5 Invalid option: "{0}".
6 Can't find file "{0}".
7 Can't unlink file "{0}".
8 Can't link file "{0}".
9 Invalid mode to open file "{0}".
10 File name missing or invalid.
11 Read failure: {0}.
12 Write failure: {0}.
13 Invalid INDEX file.
14 Invalid/corrupted file format (EOL=LF).
15 Can't start file "{0}".

/***** DOS only *****/
16 Invalid drive specifier "{0}".

/* the following are specific to Bcreate */
51 File "{0}" already exists.
52 KEYLEN option missing.
53 RECLEN option missing.
54 Syntax error.

/* the following are specific to Btermgen */
101 Can't find TERM variable.
102 Can't find termcap entry for {0}.
103 Saving entry in file {0}.

```
104    \7\7Invalid terminal identifier {0}.
105    Exceeded the termcap buffer limit!

/*    the following are specific to Blist      */
151    Comp_expand: line too long.
152    File "{0}" is not CET BASIC format.
153    Comp_expand: read error.
154    Comp_expand: truncated line.
155    Truncated record:
156     invalid type: {0}

/*    the following are specific to Bexpand      */
201    Source isn't compressed format.
202    Source and destination files must be different."*/

/*    the following are specific to Bconvux      */
251    Error 1: unknown non-string type: {0}.
252    Error 2: unknown type: {0}
255    Error 5: cannot convert records larger than {0}
256    Error 6: cannot convert keys larger than {0}
257    Error 7: invalid record type.
258    Error 8: invalid file type.
259    Error 9: record length not properly specified.
260    Error 10: key length not properly specified.

/*    the following are specific to Brenum*/
351    No lines to renumber.
352    Too many lines to renumber.
353    Old and new numbers overlap.
354    Too many lines in file.
355    Bad new start number
356    Bad increment
357    Bad old start number
358    Bad old end number
359    Program "{1}" terminated by signal {2}.

/*    the following are specific to Bpretty */
401    Converting Line
402    Line Number {0},
403    Stmt Number "{0}"
404    Occurred with SP = {0}
405    Expected SELECT.
406    Token[0] = {0}
```

```
407   Token = {0}
408   Last stack entry was {0}.
409   TopCS: no top.
410   Processing Basic Text
411   PushCS: stack overflow.
412   PopCS: stack underflow.
413   Too many token backups.
414   "{0}" unexpected.

/*  the following are specific to Bth8 and Bthrec */
451   Buffer overflowed.
452   Conversion error: {0}.
453   Converting isam format for file "{0}".
454   Converting direct format for file "{0}".
455   Converting seq ASCII format for file "{0}".
456   Converting seq binary format for file "{0}".
457   Can't read directory.
458   Can't read directory.
459   Can't read data.
460   Can't read key.
461   Can't read record pointer.
462   Can't read volume label.
463   Directory too large.
464   Expected: {0}
465   received: {0}
466   Invalid baud rate, default of {0} used.
467   Illegal escape sequence: {0}.
468   Invalid OASIS filename.
469   Record length too long.
470   Unexpected EOF.
471   Unusual length record: {0}.
472   Unrecognized format: "{0}".
473   Waiting for sender, got {0}
474   Too many files to convert.
```

APPENDIX B

Reserved Words

Introduction

In addition to the words listed here, the name of any string function with terminating '$' removed is also reserved. That means that both LEFT and LEFT$ are reserved words.

ABS
ACOS
ACOT
ACSC
ADDROF
AND
ANGLE
ASC
ASEC
ASIN
AT
AT$
ATAN
ATN
BASE
BFLOAT
BIN
BINOF$
BREAK
CALL
CASE
CEIL
CEND
CHAIN
CHR$
CLEAR
CLOSE
CLS$
CMDARG$
COMMA
COMMON
COS
COSH
COT
COTH
CRT$
CSC
CSCH
CSH
DATA
DATE$
DATEFORM
DAY
DEF
DEFAULT
DEG
DEGREE
DEL$
DELETE
DIM
DIRECT
DTE$
ELAPSED
ELSE
END
EOF
EPS
EPS!
EPS!!
EQV
ERF
ERL
ERR
ERROR
EVENT
EXP
EXT$
EXTEND
FIX
FLOAT
FLOOR
FNEND
FOR
LRL
LRR
LSL
LSR
RADIAN
RANDOMIZE
READ
READNEXT

FORMAT
FORMAT$
FP
GCD
GET
GO
GOSUB
GOTO
HEX
HEXOF$
IF
IFEND
IMP
INDEXED
INF
INF!
INF!!
INP
INPUT
IN$$
INT
INTERRUPT
IP
KEYED
LCASES$
LEFT$
LEN
LET
LINE
LINK
LINPUT
LOCK
LOG
LOG10
LOG2
LPAD$
TIME$
TRIM$
TO
UCASE$
UNGET
UNLOCK

LTRIM$
MAT
MATCH
MATIO
MAX
MID$
MIN
MOD
MOUNT
MSEC
NBR
NEWLINE
NEXT
NOT
OCT
OCTOF$
ON
OPEN
OPTION
OR
ORD
OTHERWISE
OUTPUT
OVR$
PAGE
PI
PI!
PI!!
POS
PRINT
PRIV
PROMPT
PUT
QUIT
QUOTE
RAD
UPDATE
USING
UX
VAL
WAIT

READPRIOR
REM
REP$
RESTORE
RESUME
RETURN
RIGHT$
RND
ROUND
RPAD$
RPT$
RTRIM$
RUN
SCH
SEC
SECH
SECOND
SELECT
SEQUENTIAL
SERIAL
SGN
SIN
SINH
SLEEP
SPACE$
SQR
STEP
STOP
STR$
SUB
SYNC
TAB
TAN
TANH
THEN
TIMER
WEND
WHILE
WRITE
XOR
YESNO$

APPENDIX C

Character Codes

ASCII Table

Code	ASCII	CRT$	Code ASCII	Code ASCII	Code ASCII
0	NUL (^@)		32	64 @	96 ‘
1	SOH (^A)	Home	33 !	65 A	97 a
2	STX (^B)	Fon	34 “	66 B	98 b
3	ETX (^C)	Foff	35 #	67 C	99 c
4	EOT (^D)	DC	36 $	68 D	100 d
5	ENQ (^E)	Poff	37 %	69 E	101 e
6	ACK (^F)	Clear	38 &	70 F	102 f
7	BEL (^G)	Bell	39 ‘	71 G	103 g
8	BS (^H)	Left	40 (	72 H	104 h
9	HT (^I)	+++	41)	73 I	105 i
10	LF (^j)	Down	42 *	74 J	106 j
11	VT (^K)	Up	43 +	75 K	107 k
12	FF (^L)	Right	44 ,	76 L	108 l
13	CR (^M)	***	45 –	77 M	109 m
14	SO (^N)	RVon	46 .	78 N	110 n
15	SI (^O)	Rvoff	47 /	79 O	111 o
16	DEL (^P)	---	48 0	80 P	112 p
17	DC1 (^Q)	IL	49 1	81 Q	113 q
18	DC2 (^R)	IC	50 2	82 R	114 r

Code	ASCII	CRT$	Code	ASCII	Code	ASCII	Code	ASCII
19	DC3 (^S)	DL	51	3	83	S	115	s
20	DC4 (^T)	Pon	52	4	84	T	116	t
21	NAK (^U)	Kon	53	5	85	U	117	u
22	SYN (^V)	Uloff	54	6	86	V	118	v
23	ETB (^W)	EOL	55	7	87	W	119	w
24	CAN (^X)	EOS	56	8	88	X	120	x
25	EM (^Y)	Koff	57	9	89	Y	121	y
26	SUB (^Z)	Ulon	58	:	90	Z	122	z
27	ESC (^[ )	Pbon	59	;	91	[	123	{
28	FS (^/)	EU	60	<	92	\	124	|
29	GS (^])	Bon	61	=	93	]	125	}
30	RS (^^)	Boff	62	>	94	^	126	~
31	US (^_)	Pboff	63	?	95	_	127	RUB

+++ Hardware TAB
*** Carriage return (default line separator)
---- Lead-in character to cursor addressing (produced by AT$)

CRT Control Tokens

The left-hand column in the ASCII chart lists the default characters used by the CRT$ and AT$ functions to control the various console characteristics. For example, CRT$("IC") produces the single-character string CHR$(18).

Terminal independence is maintained by using the CRT$ and AT$ values since these *tokens* are translated to the correct escape sequence at the time when they are evaluated for printing to a terminal device. In order to avoid this translation, prefix the character with CHR$(255), the CET BASIC escape character. When B_EMULATE is set to THEOS, the escape character is CHR$(27).

Normally, you can ignore the actual token value for the CRT$ and AT$ functions. However, it is possible to change these values, if necessary. (See the *User-Defined Token Values* section.)

Input Tokens

When a non-ASCII character is input in response to an INPUT, LINPUT or LINPUT USING statement, input is terminated and the value of the function INP is set to a number representing the character.

Control keys are represented by their ASCII value. Whenever possible, special keys such as the cursor movement keys are given the same token value as that used for output by the CRT$ and AT$ functions. For example, the HOME key will set the INP value to 1. Some extra input tokens are assigned to keys which do not correspond to CRT$ functions. The default values are as follows:

Key	Token Value
HOME Key	1
LEFT Key	8
RIGHT Key	12
UP Key	11
DOWN Key	10
EOS Key	24
EOL Key	23
Insert Line Key	17
Delete Line Key	19
Insert Character Key	18
Delete Character Key	4
PageUp	133
PageDown	134
END key	132
Function Key *n*	200 + *n*
Shifted Function Key *n*	216 + *n*

The following is a list of tokens that are returned under THEOS emulation:

Key	Unshifted	Shifted	CTRL-key	ALT-key
Left Arrow	8	8	8	8

Up Arrow	11	11	11	11
Right Arrow	12	12	12	12
Down Arrow	10	10	10	10
END	25	25	25	25
PageDown	16	16	16	16
PageUp	2	2	2	2
HOME	20	20	20	20
Backspace	8	127	127	127
Insert	18	18	18	18
Delete	26	26	26	26
TAB	9	255-15	9	9
F1	23	22	255-224	255-240
F2	24	21	255-225	255-241
F3	1	255-210	255-226	255-242
F4	4	255-211	255-227	255-243
F5	14	255-212	255-228	255-244
F6	28	30	255-229	255-245
F7	3	15	255-230	255-246
F8	7	5	255-231	255-247
F9	17	31	255-232	255-248
F10	6	19	255-233	255-249
F11	255-202	255-218	255-234	255-250
F12	255-203	255-219	255-235	255-251

Some terminals can produce a few of these values as legitimate characters in non-English alphabets. In that case, these characters will not be interpreted as input tokens. They will not trigger the end of INPUT or LINPUT or be returned as the value of the INP function.

It is possible to redefine input token values. A function such as HOME may even be defined with different token values for input and output. This procedure is covered in the next section.

User-Defined Token Values

All CET BASIC input and output functions are associated with a unique *index* value which serves as an index into the Terminal Independence Table which is listed at the end of this section.

When looking at the table, note that index number 4 corresponds to the output function which moves the cursor down while number 28 refers to the input function that detects that a ↓ key has been entered.

Although both of the HOME functions have the default value of 10, it is important to note that CRT$("DOWN") always returns CHR$(table-entry-4) and pressing the ↓ key causes the INP function to be set to the 28th entry in the table.

To change the operation of any input or output function during the execution of a program, you must inform the system by referring to the entry in the table. This is done by calling the Bchgtival function. For example, the following CALL statement may be executed to direct the system to return 196 (instead of the default 10) to the INP value after the ↓ key is entered:

```
CALL Bchgtival(28, 196)
```

The two parameters passed to Bchgtival must be integer expressions; the first is the table entry number (*index*) and the second is the new value for that entry.

The modified token value will be automatically passed from program to program during CHAIN and LINK commands. However, each new program (started directly or RUN from another program) will need to re-execute Bchgtival.

Other examples of changing the token numbers are:

```
CALL Bchgtival(2, 6)  \ REM Change CRT$("RIGHT")
CALL Bchgtival(3, 26) \ REM Change CRT$("UP")
CALL Bchgtival(5, 12) \ REM Change CRT$("CLEAR")
CALL Bchgtival(9, 11) \ REM Change CRT$("ULON")
CALL Bchgtival(13, 4) \ REM Change CRT$("PON")
CALL Bchgtival(22, 18) \ REM Change CRT$("DL")
CALL Bchgtival(23, 19) \ REM Change CRT$("IC")
CALL Bchgtival(24, 20) \ REM Change CRT$("DC")
```

Terminal Independence Table

The following tables lists the table entries corresponding to each terminal output (CRT$ and AT$) and input (INP) function.

Index	Type	Function
0	Output	Move to HOME
1	Output	Move LEFT
2	Output	Move RIGHT
3	Output	Move UP
4	Output	Move DOWN
5	Output	CLEAR Screen
6	Output	Erase to End of Screen
7	Output	Erase to End of Line
8	Output	Erase Unprotected Area
9	Output	Underline On
10	Output	Underline Off
11	Output	Reverse Video On
12	Output	Reverse Video Off
13	Output	Protected On (Half-Intensity)
14	Output	Protected Off
15	Output	Blink On
16	Output	Blink Off
17	Output	Format Mode On
18	Output	Format Mode Off
19	Output	Unlock Keyboard
20	Output	Lock Keyboard
21	Output	Insert Line
22	Output	Delete Line
23	Output	Insert Character or INSERT Mode
24	Output	Delete Character

25	Output	Sound Bell
26		Unused
27	Input	UP Key Received
28	Input	DOWN Key Received
29	Input	RIGHT Key Received
30	Input	LEFT Key Received
31	Input	HOME Key Received
32	Output	Lead-in Character for AT$ Positioning
33	Input	PageUp Key Received
34	Input	PageDown Key Received
35	Input	END Key Received
36	Input	IL Key Received
37	Input	DL Key Received
38	Input	IC or INSERT Key Received
39	Input	DC Key Received
40	Input	EOL Key Received
41	Input	EOS Key Received
42–57	Input	Function Keys 1–16
58–73	Input	Shifted Function Keys 1–16