IBM Instruments Inc.

PRELIMINARY

Computer System
BASIC Reference Manual

GC22-9184

Preliminary Edition Only (December 1982)

Changes are continually made to the information herein; any such changes will be reported in subsequent revisions.

Requests for copies of IBM Instruments, Inc. publications should be made to your IBM Instruments, Inc. representative or via calling, toll-free, 800-243-3122 (in Connecticut, call collect 265-5791).

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Instruments, Inc., Department 79K, P.O. Box 332, Danbury, CT 06810. IBM Instruments, Inc. may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

01234567890

PREFACE

This manual is a reference manual for CS BASIC. It is not intended as a user manual or a tutorial. Readers are assumed to have some grasp of programming concepts and terminology, and to have at least a minimal understanding of BASIC. There are many tutorial-style BASIC books available on the market that teach programming through the BASIC language.

Chapter 1 -- "Introduction to CS BASIC -- briefly describes some main features of the language, including its syntax and notation.

Chapter 2 -- "CS BASIC Commands" -- covers the commands that can be typed to the CS BASIC interpreter itself: to run a program, save or retrieve programs, display a program's statements, or to change an existing program.

Chapter 3 -- "Elements of CS BASIC" -- contains a description of the elements of the language, including the character set used, line numbers, statements, and identifiers.

Chapter 4 -- "Data Representation" -- describes the formats of numeric constants and variables, string constants and variables, and array (matrix) constants and variables.

Chapter 5 -- "Expressions" -- contains a discussion of expressions, in which operators and operands are combined to generate numeric or string values.

Chapter 6 -- "Standard Functions" -- covers the built-in functions available in CS BASIC.

Chapter 7 -- "Assignment Statements" -- describes assignment statements and other simple statements.

Chapter 8 -- "Control Statements" -- contains a description of the statements that direct the flow of execution of a CS BASIC program.

Chapter 9 -- "Input and Output Statements" -- describes the input and output capabilities of CS BASIC.

Chapter 10 -- "Matrix Operations" -- describes the facilities for manipulating matrices.

Chapter 11 -- "External Linkages" -- describes how external routines are called from CS BASIC.

Chapter 12 -- CS BASIC Program Structure -- discusses the general structure of a CS BASIC program, concentrating on the subroutine calls, functions, and error handling.

The appendixes contain a list of error messages, a treatment of floating point numbers, a summary of the language, a list of reserved words, and a chart of the ASCII character set.

Related Publications:

Publications that discuss related aspects of the Computer System are:

Computer System Product Description, GC22-9183

Computer System BASIC Reference Manual, GC22-9184

Computer System Operating System Reference Manual
Part 1: Operating System, GC22-9199
Part 2: Logical I/O and System Services, GC22-9200

Computer System Problem Isolation Manual, GC22-9192

CONTENTS

1.0 INTRODUCTION TO CS BASIC

1.1 GENERAL

CS BASIC is an implementation of the BASIC programming language for the Computer System. CS BASIC is an extended version of the BASIC language.

BASIC stands for **B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode. BASIC was developed at Dartmouth College in the 1950's and early 1960's. The design goals of BASIC were to provide an interactive and user-friendly environment in which people outside of computer science could program a computer easily and effectively.

BASIC has evolved over the years, both in the application areas to which it has been applied, and in the sophistication and features of its language dialects. CS BASIC is among the more extensive dialects of the BASIC language, while retaining the user-friendly environment that characterizes most implementations of BASIC.

CS BASIC runs in an interactive manner. Programs are entered and run from the terminal. There are facilities to store programs in the computer's disk system for later retrieval.

The language supports real (floating point), integer, and string data types. A numeric string data type provides for arithmetic carried to very high precision. The language also contains facilities to define and manipulate matrices.

The interpretive nature of the language allows an "immediate," or command, mode, which lends itself to a "desk calculator" style of use with almost the same power as the underlying BASIC language. In the immediate mode, it is possible to debug CS BASIC programs easily, by examining and changing the state of the program (after it has been suspended) and then restarting the program's execution.

A CS BASIC program is (ultimately) composed of characters. Characters are grouped into lines and statements, the elements which serve to make up programs. A line is either a remark (comment), a declaration, or an executable statement.

Program elements include constants and variables. A constant is a group of digits or other characters defining a value that does not change. Variables occupy storage and have values that can be changed during program execution. Variables and constants can have both a name and a data type. The name serves to

identify that element in a program. The data type of an element defines, among other things, the amount of storage it occupies, its range and precision, and in some cases, the operations that can be performed on it.

A variable can be a single element or it can be an aggregate. There are two forms of aggregate data elements, namely dimensioned (array) variables (also known as matrices) and string variables. An array variable is a collection of data occupying consecutive storage units. Arrays can have one or two dimensions. A string variable represents string data and is a sequence of ASCII characters, which can be accessed individually or collectively by various string functions. It is possible to define arrays of strings. CS BASIC supports what are called "virtual arrays," whereby array data types can be stored on external disk storage, and retrieved on demand.

A complete CS BASIC program can (but does not have to) contain subroutines and functions. Subroutines are activated via the GOSUB statement to perform out-of-line groups of statements. Functions compute and return a value in the context of an expression.

Variables have a lifetime that is dependent on the way that they are defined. Formal parameters to functions have a lifetime that begins when the function is invoked and ends when the function returns to its caller. All other variables have a lifetime that lasts for the duration of the program.

Expressions combine operands and operators to create new values. CS BASIC supports arithmetic, string, and relational expressions. Mixed-mode expressions are permitted, with well-defined rules for conversions between the operands and for generating results. In addition, CS BASIC provides for treating integers as logical operands, to which logical operators can be applied. There are also built-in functions to treat character strings as numbers, providing a higher degree of precision than is possible with the regular integer or floating point data types.

The assignment statement assigns the value of an expression to a variable. There are two variations of assignment, namely arithmetic and string.

Control statements are those that control the flow of execution in a program. Various kinds of IF statements select other statements for execution, depending on the result of evaluating a logical or arithmetic expression. The FOR, WHILE, and UNTIL statements provide for repetition of a block of statements while a control variable is assigned a sequence of values. The GOSUB and RETURN statements provide for subroutine execution. Variations of the GOTO statement provide for transfer of control within a program unit.

CS BASIC provides a powerful input and output capability. Files can be accessed sequentially or randomly. Format conversion is performed via INPUT, READ,

PRINT, or PRINT USING statements. There is a rich set of format specifications to control the form and layout of converted data.

Functions may have arguments that are passed to them for processing. When a function is declared, its formal arguments are declared. When the function is referenced, actual arguments are assigned to the formal arguments. Control is returned from a function by executing the function's associated FNEND statement. The value of the function is that of the last value "stored" in the function name.

CS BASIC supplies a comprehensive set of intrinsic functions that perform data-type conversion and provide an extensive collection of arithmetic and transcendental functions. There is also a rich set of string manipulation functions built into the language.

1.2 CS BASIC PROGRAMS

The CS BASIC interpreter works on one BASIC program at a time. The program is developed in the computer's memory, which may be thought of as a kind of "workspace" in which the user can enter, run, and debug CS BASIC programs.

Every CS BASIC program has a name associated with it. In the absence of an explicit name given by the user, the system assigns the name "NONAME" to a program. The name of the program is used in header messages that various commands (such as RUN) display when they operate.

CS BASIC programs can be stored in and retrieved from files in the computer's file system. By convention, the filename for a program called (say) PAYROLL is given the name PAYROLL.BAS. The .BAS part of the filename is called an extension, and is an abbreviation for **BA**sic **S**ource.

The names given to programs and files are converted to all upper case by the CS BASIC interpreter before they are stored on the disk system. Thus, the names "program," "Program," and "PROGRAM" are identical when typed to CS BASIC.

1.3 NOTATION AND TERMINOLOGY USED IN THIS MANUAL

This section summarizes the conventions used in this manual to describe the syntax of CS BASIC.

Words appearing in upper case, such as LET, are BASIC keywords.

In general, special characters such as the equals sign (=) represent themselves when they appear in statement syntax.

The angle brackets (< and >) enclose elements of the language.

Elements that appear in the braces ({ and }) are optional.

When an element is followed by an ellipsis (...), that element can be repeated.

The vertical bar character (|) stands for "or". It separates choices in a list of elements.

Example of Syntax Notation

MAT PRINT {#<exp>,} <matrix>{(<subscripts>)} { , | ; }

The example above illustrates the syntactic definition of the MAT PRINT statement, in which the phrase

MAT PRINT

is composed of CS BASIC keywords;

#<exp>,

is a file number, enclosed in braces because it is optional;

<matrix>

is the name of the matrix to be printed; and the optional term

{(<subscripts>)}

indicates that there can be optional dimension information following the name of the matrix. Lastly, there is a choice of an optional comma or semicolon following the statement. The braces indicate that the comma or semicolon are optional; the vertical bar indicates The choice of one or the other.

2.0 CS BASIC COMMANDS

This chapter describes the commands that CS BASIC handles. Commands are not part of the language itself but are the means for directing the actions of the BASIC language interpreter.

2.1 USING THE CS BASIC INTERPRETER

The user invokes CS BASIC by typing the command

basic

on the keyboard or, more simply, by pressing function key F5.

The CS BASIC interpreter will display a header message. It will then prompt the user with the following message:

Ready

which means that the system is ready for input from the user's terminal. The system is now said to be in the immediate mode.

2.1.1 IMMEDIATE (COMMAND) OPERATING MODE

Immediate mode in CS BASIC refers to the execution of statements at the terminal immediately after they are typed, without the need to compile and run them first. Immediate mode thus provides a kind of "desk calculator" feature in the language. Immediate mode is also useful for looking at the state of variables in a program after a STOP or a run-time error has occurred.

CS BASIC recognizes immediate mode statements by the absence of a line number before the statement. Statements that do not start with a line number are executed as soon as they are entered at the terminal.

Immediate mode is useful for debugging programs as well as for trying out the effects of statements before they are entered into a program, and for doing simple calculations at the terminal.

In immediate mode, there may be multiple statements per line. Statement modifiers (but not the FOR statement modifier; see Section 8.9.3) can also be used as the next example shows:

```
PRINT X, SIN(X) IF X = 10
```

2.1.2 RESTRICTIONS ON IMMEDIATE MODE

Certain statements are not allowed in immediate mode, and give rise to an error message. These statements are:

DEF* and FNEND	Function definition,
DIM	Dimension declaration,
DATA	Data definition,
FOR and NEXT	FOR loops of any kind (including statement modifiers).

Note that GOSUB statements and function references are allowed in immediate mode, so the user has access to both user-defined functions and to the built-in functions.

2.2 INTERRUPTS

Pressing the Ctrl and Break keys together while a BASIC program is running will normally return the CS BASIC interpreter to immediate mode. If the interpreter is awaiting the completion of input from the operating system, however, the interpreter will not be interrupted until the input has been received. If this state is entered and it is not possible to satisfy the input request, then the Ctrl-Alt-Del warm start keystroke sequence must be used. This action will return control to the operating system, and any unsaved work will be lost.

Any output from CS BASIC may be paused by pressing the Ctrl and NumLock keys. Output may be resumed by pressing any other key.

2.3 CS BASIC COMMANDS -- SUMMARY

The set of commands listed below operate on CS BASIC programs in various ways, saving them, retrieving them, renaming them, and so forth. The subsections following this summary list discuss the individual commands in more detail.

APPEND	Includes the source of another CS BASIC program in the current program.
AUTO	Automatically generates program line numbers.
BYE	Returns control to the operating system.
CONT	Continues the execution of a program after a STOP statement.
EDIT	Enables the user to modify program lines.
DELETE	Deletes one or more lines from a program.
KILL	Deletes a file.
LENGTH	Displays the length of the current program on the console screen.
LIST	Displays the current program on the console screen.
LISTNH	Displays the current program on the console screen, but without a program heading.
LLIST	Prints out the current program on the printer.
NEW	Clears the program work area, sets the program name, and reads in an existing CS BASIC program from the computer's file system.
OLD	Clears the program work area, sets the program name, and reads in an existing CS BASIC program from the computer's file system.
REPLACE	Same as SAVE except that the command presupposes that the program named already exists in the computer's file system.
RENUM	Renumbers all lines in the current program.
RUN	Initiates execution of a program.
RUNNH	Initiates execution of a program but without displaying a program heading.

SAVE	Appends the .BAS suffix to the program name, then saves the program in the computer's file system.
TROFF	Exits trace mode.
TRON	Enters trace mode.

2.4 APPEND

The APPEND command includes the source of a previously saved program in the current program. The format of the command is:

```
APPEND <filename>
```

where <filename> is the name of the disk file that is to be included in the current program. The extension is .BAS.

The APPEND command performs a line-by-line insertion (or substitution or both) of the existing program with the APPEND'ed program, just as if the user had typed each line of the APPEND'ed program from the terminal. One of the examples below clarifies this.

Example of the APPEND Command

To illustrate, assume that the Computer System has a file called MOREBAS.BAS, containing the following program:

```
15 LET W = 7
30 LET Z = 3
40 PRINT X + Y + Z + W
```

and that the existing program in memory looks like this:

```
listnh
10 LET X = 1
20 LET Y = 2
30 PRINT X + Y
99 END
```

then the APPEND command is used, and the result displayed with the LISTNH command:

```
append more.bas
Ready
listnh
10 LET X = 1
15 LET W = 7
20 LET Y = 2
30 LET Z = 3
40 PRINT X + Y + Z + W
99 END
Ready
```

As the example illustrates, new lines in the APPEND'ed file (those whose line numbers do not exist in the in-memory version of the program) are inserted in their correct sequential place and lines in the APPEND'ed file whose line numbers duplicate those in the in-memory version replace those lines in the in-memory version of the program.

2.5 AUTO

The AUTO command helps the user to input new statements by automatically generating line numbers. The format of the AUTO command is:

AUTO {<line number> {,<increment>}}

This command begins numbering lines at <line-number> with each subsequent line number incremented by <increment>. If the increment is not specified, it is assumed to be 10. If neither the line number nor the increment is specified, both are assumed to be 10.

If the line number being generated already exists, the user will be warned with an asterisk that the old line is about to be replaced. If the user enters an empty line (i.e., a single carriage return) the old line will be retained and a new line number will be generated.

The user returns to the READY mode by entering a carriage return to an unasterisked line number, or an escape-carriage return, <Esc><CR>, to any automatically generated line number prompt.

2.6 BYE

The BYE command exits from the CS BASIC interpreter and returns control to the operating system. Any files that are open are closed and saved. The format of the command is:

```
BYE
```

The system asks for confirmation with the prompt:

```
Confirm:
```

The user should then type one of the following:

Y "Yes" -- go ahead and log out.

N "No" -- negates the BYE command.

F "Fast logout" -- is equivalent to the **Y** response.

It is also possible to include the option directly in the BYE command. For example:

```
BYE/Y
```

in which case the BYE command does not prompt for confirmation.

2.7 CONT

The CONT command continues the execution of a program, after it has previously executed a STOP statement. The format of the command is:

```
CONT
```

Programs that have been halted by the CS BASIC interpreter because of some kind of error cannot be CONT'inued. Similarly, programs that have stopped because an END statement was executed also cannot be CONT'inued.

2.8 DELETE

The DELETE command removes lines from a program. The format of the command is:

DELETE <list of line numbers>

where <list of line numbers> can be any one of the following:

- a single line numbers such as 125
- a list of two or more line numbers separated by commas, such as 100, 230, 455, 690
- a line number range, where the start and end of the range are separated by a minus sign, such as 100-200. In this case, the range is line 100 through 200 inclusive
- a combination of line numbers and line number ranges, such as 10, 20, 50-455, 600-700, 1010, 2050.

Line numbers and ranges of line numbers can appear in any order. Incorrect line number ranges (such as 60-30) are ignored.

Example of the DELETE Command

```
DELETE 20-30,150-200,201,203,212,320-340
```

2.9 EDIT

The EDIT command allows a user to perform simple editing on a source line that has previously been entered. The form of the command is:

EDIT <line-number>

The EDIT command displays the contents of the current line with that number, positions the cursor at the first position of that line, and allows modification of that line with the cursor keys for positioning, and the insert and delete keys for changing the text. Pressing the carriage return key causes the modified line to be used as a replacement of the original line, and returns to command mode.

Pressing the <Esc> key causes the edited line to be discarded and the original line to be retained.

Note that when the Computer System is reset or started from a powered-down state, the NumLock state is on. The cursor control keys, therefore, are not active until NumLock is pressed once (each time NumLock is pressed, the keys toggle between numeric and cursor control mode). The state of the cursor-control keys is remembered across executions of the CS BASIC interpreter.

Similarly, the insert key toggles the insert mode each time it is pressed. The initial state does not allow insertion.

2.10 KILL

The KILL command deletes a file. The format of the command is:

KILL <filename>

where <filename> is the name of the file to be removed. If the file is present it is deleted from the disk. If not, an error message is printed.

KILL first attempts to delete a file with the name as given. If that file is not found, the extension ".BAS" is appended, and it attempts to delete that file. If neither file is present, an error message is printed.

2.11 LENGTH

The LENGTH command displays the length of the current program in 1K increments, as well as the maximum amount of available memory. The format of the command is:

```
LENGTH
```

For example:

```
LENGTH
19 (124)K of memory used
```

means that the program is between 18K and 19K, and the maximum memory available is 124K.

2.12 LIST

The LIST command displays the current program on the console screen with a heading that contains the name of the program. The format of the command is:

LIST {<list of line number>}

LIST alone displays the whole program. Selected portions of the program can be displayed by specifying them in <list of line numbers>, which can be any one of the following:

- a single line number, such as 125
- a list of two or more line numbers, separated by commas, such as 100, 230, 455, 690
- a line number range, where the start and end of the range are separated by a hyphen, such as 100-200. In this case, the range is line 100 through 200 inclusive
- a combination of line numbers and line number ranges, such as 10, 20, 50-455, 600-700, 1010, 2050

Line numbers and ranges of line numbers can appear in any order. Incorrect line number ranges (such as 60-30) are ignored.

Example of the LIST Command

```
LIST 10-20,150-170,200,205,210,300-320
```

2.13 LISTNH

The LISTNH command is exactly the same as LIST except that it displays the current program without any heading to identify the program. The format of the command is:

LISTNH {<list of line numbers>}

See the LIST command for details.

2.14 LLIST

The LLIST command is the same as the LIST command except that it prints out the current program on the printer (#PR). The format of the command is:

```
LLIST {<list of line number>}
```

See the LIST command for details.

2.15 NEW

The NEW command clears the program work area in memory, so that the user can start work on a completely new BASIC program. The format of the command is:

```
NEW {<program name>}
```

If the keyword alone is typed, the system clears the program work area and names the program "NONAME".

If the user specifies a program name, the system clears the program work area and assigns the specified name to the program.

Examples of the NEW Command

```
new

new pipefit
```

The first example clears the program work area and sets the program name to "NONAME". The second example clears the program work area and sets the program name to "PIPEFIT".

2.16 OLD

The OLD command clears the program work area in memory, sets the program name, and recalls a previously saved program from disk. The format of the command is:

OLD <program name>

The CS BASIC system appends a suffix of .BAS to the program name, if the user has not already done so.

Example of the OLD Command

```
OLD PLOTTER
```

The above example clears the program work area, sets the name of the current program to "PLOTTER", and loads the CS BASIC program from a file called PLOTTER.BAS.

2.17 RENUM

The RENUM command automatically renumbers the lines of the BASIC program that is currently in memory. The format of this command is:

RENUM {<new> {,<start> {,<increment>}}}

<new>	specifies the first line number of the newly numbered group of lines. The default is 10.
<start>	specifies the first of the "old" line numbers to be changed. The default is the first line of the program.
<increment>	is the increment of the renumbered lines. The default is 10.

NOTES:

1. When no arguments are specified, all lines of the current program are renumbered, and the renumbering is determined by the default parameters (first line 10, increment 10). All line numbers in the program -- including those that occur in statements (e.g., a GOTO statement) -- are changed in conformity with the new numbering.

2. If only <new> is specified, the first line of the program is changed to <new> and subsequent line numbers are incremented by 10.

3. If <start> and <increment> are both specified, the "old" program line number, designated by <start>, is the first line number to be changed; subsequent line numbers are incremented by <increment>.

4. If only ,<start> is specified, the default increment of 10 is used. (Note that in this case an initial comma is required if <new> is not specified.)

5. If only ,,<increment> is specified, numbering begins with the first statement of the program, and subsequent line numbers are incremented by the <increment> specified. (Note that in this case two initial commas are required if both <new> and <start> are not specified.)

Example of the RENUM Command

In this example, the command renumbers all line numbers from 10 to the end of the program, assigns line number 20 to line 10, and increments all subsequent line numbers by 2.

```
10 PRINT
20 PRINT
30 PRINT
40 NOFOLD
50 PRINT
60 PRINT TAB(75);"*";123456789

READY
RENUM 20,10,2      *RETURN*
```

will generate the following:

```
20 PRINT
22 PRINT
24 PRINT
26 NOFOLD1308
28 PRINT
30 PRINT TAB(75);"*";123456789
```

2.18 REPLACE

The REPLACE command is like SAVE (see below) except that a copy of the program is assumed to exist on disk. The current version of the program in memory then replaces (overwrites) the disk version. The format of the command is:

```
REPLACE <filename>
```

where <filename> is the name of the disk file in which the current program is to be placed. (See the SAVE command for more details.)

2.19 RUN

The RUN command executes a program. The RUN command displays a header containing the name of the program. The format of the RUN command is:

RUN {<program name>}

The keyword alone, without a filename, runs the current program in memory.

If a filename is specified, the system clears the program work area, sets the current program name to that of the filename (minus any .BAS extension), fetches that file from the computer's file system, compiles the program, and then runs the program.

2.20 RUNNH

The RUNNH command is exactly the same as RUN except that it does not display a program heading. The format of the command is:

RUNNH {<program name>}

(See RUN for more details.)

2.21 SAVE

The SAVE command saves the program in memory by storing it on disk. The format of the command is:

```
SAVE {<filename>}
```

where <filename> is the name of the disk file in which the current program is to be saved. The extension .BAS is appended to the name unless the user has already done so. If the file to be SAVE'd already exists in the file system, CS BASIC asks whether the old version should be deleted. A response of "Y" or "y" deletes the old version of the file. Any other response cancels the SAVE command.

If no <filename> is specified, CS BASIC uses the name of the current program with a suffix of .BAS as the name of the file in the computer file system.

Examples of the SAVE and REPLACE Commands

```
save VIEWPACK

save PROGGY
Delete old PROGGY.BAS? y

replace VIEWPACK
```

The middle example shows the dialog that results when the user attempts to SAVE an already existing file.

2.22 TROFF AND TRON

The TROFF and TRON commands turn off and turn on, respectively, the trace mode, which is a BASIC program debugging aid. The command formats are:

```
TROFF

TRON
```

NOTES:

1. Trace mode is a program debugging aid that displays, on the screen, the line numbers of a program while that program is executing. The line numbers displayed are enclosed in brackets, and print commands are actuated.

2. TROFF and TRON can also be used as statements in a BASIC program. In program statements, a line number must precede the keyword.

Example

```
10 PRINT "THIS IS AN EXAMPLE OF TRACE MODE"
20 A=5
30 B=10
40 C=A+B
50 PRINT C

READY
*TRON*

READY
*RUN*
```

will result in the following display:

```
[10]THIS IS AN EXAMPLE OF TRACE MODE
[20][30][40][50]15.0

READY

-
```

The trace mode can now be turned off and the program run again, producing two lines on the display, as follows:

```
*TROFF*

READY
*RUN*
THIS IS AN EXAMPLE OF TRACE MODE
15.0

READY

-
```

3.0 ELEMENTS OF CS BASIC

This chapter describes the lowest level elements of the BASIC language. Topics covered here include the character set used; the definition of line numbers and statements, with multiple-statement lines and continuation lines; and identifiers.

3.1 CHARACTER SET

The CS BASIC character set consists of 26 upper case letters, 26 lower case letters, and 21 other characters.

A <letter> is one of the 52 characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

A <digit> is one of the ten characters:

0 1 2 3 4 5 6 7 8 9

An <alphanumeric character> is a <letter> or a <digit>.

The other printing characters consist of the characters shown in the table below:

Character	Meaning	Character	Meaning
	Blank or space	=	Assignment or Equal
+	Addition	-	Subtraction
*	Multiplication	/	Division
^ or **	Exponentiation	.	Decimal Point
\ :	Multiple Statements	&	Continuation
(	Left Parenthesis	)	Right Parenthesis
,	Compressed	;	Record Suppression
$	String Variable	%	Integer Variable
'	String Delimiter	"	String Delimiter
<	Less than sign	>	Greater than sign
#	Number sign	!	Comment Starter

3.1.1 COLLATING SEQUENCE AND GRAPHICS

CS BASIC uses the ASCII character set. The collating sequence in ASCII is:

- Space (blank) collates lowest, followed by:
- Digits "0" though "9", followed by:
- Upper case letters "A" through "Z", followed by:
- Lower case letters "a" through "z".

The other printing characters appear in between digits and upper case letters and before and after lower case letters. There is an ASCII character set chart in Appendix E.

Within each of the ordered sets, digits, upper case letters, and lower case letters, the characters in those sequences are contiguous -- there are no "holes" in them.

3.2 USE OF SPACES AND TABS

Spaces and tabs can appear only between CS BASIC language elements, where they serve to delimit or separate those elements, and in character strings, where they stand for themselves.

Where one space or tab can appear between language elements, any number of spaces or tabs are equivalent to one.

3.3 LINE NUMBERS

Each program line in a BASIC program must be preceded by a line number. Line numbers have the following functions in the language:

1. Indicate that the statement(s) following the line number are part of the BASIC program, as opposed to a statement that is executed immediately (i.e., a command).

2. Indicate the order in which program statements will normally be executed, in the absence of any changes of control flow.

3. Provide a means whereby other statements can change the order of statement execution by branching (conditionally or unconditionally) to numbered statements.

4. Supply the means to change numbered statements without affecting other statements in the same program.

Line numbers are integers in the range 1 through 32767. Line number 0 is used for special purposes, for example in ON ERROR GOTO statements.

NOTE: If a statement is not preceded by a line number, it is considered an "immediate mode" statement and is executed immediately (see Section 2.1.1).

3.4 STATEMENTS

A statement follows a line number (if it is part of a program), or is executed immediately if there was no preceding line number. In this manual, statements fall into four logical groups:

- Empty statements
- Assignment and other arithmetic statements, including matrix manipulation statements
- Control statements
- Input and output statements.

Assignment and other arithmetic statements are covered in Chapter 7; Control statements are described in Chapter 8; input and output statements are discussed in Chapter 9.

An empty statement is indicated by a line number on its own. Execution of an empty statement has no effect but can be used as the target of a change of control flow.

3.4.1 MULTIPLE STATEMENTS PER LINE

There can be more than one statement per line in a BASIC program. To put multiple statements on a line, each statement except the last one on the line must be terminated either by a colon (:) or by a reverse slash character (\).

Only the first statement on a multiple statement line may have a line number.

Examples of Single and Multiple Statements

Here is a single-statement line:

```
240 LET A = 25
```

and here is a line containing three statements:

```
1125 IF A = 10.5 GOSUB 1230  \ PRINT A  \ LET A = 0
```

The second and subsequent statements on a multiple-statement line should not be considered as separate lines. There are cases in which such statements will not get executed. For instance, if the example above were changed to read:

```
1125  IF A = 10.5 GOTO 1230  \ PRINT A  \ LET A = 0
```

the following PRINT and LET statements would never be executed. If A is equal to 10.5, control passes to statement 1230 as indicated. But if A is not equal to 10.5, control passes to the next line in the program, not to the PRINT statement on the same line.

3.4.2 STATEMENT CONTINUATION

A single statement can be continued over more than one line. Continuation of a statement is signaled by an ampersand character (&) at the end of the line, before the carriage return.

Example of a Continued Statement

```
100 LET TOTAL.FICTION = &
    CRAWLING.ON.THE.GRASS.GOTHICS + &
    SLUSHY.LOVE.STORIES + &
    SOUTHERN.TERMITE.JULEPS + &
    FEDERAL.BUDGET.RETURNS
```

3.5 REMARKS AND COMMENTS

CS BASIC provides for the insertion of commentary material in a program, for the purposes of identification and documentation. There are two ways to insert commentary in a program:

- by using a REM statement
- using a ! sign after a statement.

The REM (for REMark) statement ignores everything that follows it on that line.

Examples of REM Statements

```
200 REM This program computes the whichness of the why.
250
300 REM This program simply asks the question.
```

The other means of introducing commentary is the ! sign after a statement. For example, the following statements have comments affixed:

```
980 LET  AV = TOT.BOOK / BOOK.CATS   !  compute average

1020 VOLUME = 4 / 3 * PI * R 3       !  Spherical volume
```

A single program line with multiple lines can have comments on each of the lines:

```
670 GOLDEN.SECTION = (1 + SQR(5)) / 2 ! compute Golden Ratio &
   \ PRINT  GOLDEN.SECTION             ! and print it
```

3.6 IDENTIFIERS

Identifiers in CS BASIC are formed from the following elements:

- Upper case letters **A** through **Z**
- Lower case letter **a** through **z**,
- Digits **0** through **9**,
- The period character .

The syntactical definition of an identifier is:

```
<letter> {<letter> | <digit> | . ...}
```

An identifier must start with a letter. Including the initial letter, an identifier may contain up to a total of 30 characters, consisting of letters, digits, and the period. Upper-case letters and lower-case letters are considered the same in an identifier -- the BASIC processor "folds" all letters to a single case. Periods are used to break identifiers into words for readability. Spaces and line breaks are not allowed in identifiers.

Examples of Correct Identifiers

```
Clock.Rate          Example.23          RECEIPTS
accounts            x2000               ANSI.and.ISO
```

Examples of Incorrect Identifiers

```
2nd.April          (should not begin with a digit)

.that.moot         (should not begin with a period)

Beginners.All.Purpose.Symbolic.Instruction.Code     (Too long)
```

Identifiers are further qualified by using a trailing **%** sign to indicate an integer variable, or by a trailing **$** sign to indicate a string variable. An unqualified identifier is automatically assumed to denote a floating point variable.

Identifiers that are the names of functions are denoted by the letters **FN** in front of them.

The same identifier can be used to represent a floating point variable, an integer variable, a string variable, a dimensioned variable of any type, and a function name, with no ambiguity. That is, the identifiers A, A$, and A%, are all distinct identifiers that can be used in the same CS BASIC program.

3.6.1 RESERVED IDENTIFIERS

CS BASIC uses many identifiers as reserved words in the language. User programs cannot use these reserved words for user identifiers. There is a list of reserved identifiers in Appendix D. Some of these reserved words are not CS BASIC commands, statements, or functions.

3.7 USE OF UPPER CASE AND LOWER CASE LETTERS

Just as for identifiers, CS BASIC ignores the case of letters except within character strings.

This means that all CS BASIC keywords can be either uppercase, lowercase, or a mixture of both.

Similarly, the letter **E** used for exponents (discussed below) can also be typed either as the upper-case letter **E** or lower-case **e**.

Similarly, the **FN** keyword preceding a function name can be written in lower-case, upper-case, or a mixture of cases.

The only place that upper case and lower case are significant is within character string constants.

4.0 DATA REPRESENTATION IN CS BASIC

4.1 NUMERIC DATA TYPES

Numeric data in CS BASIC is represented internally in floating point format, unless the programmer explicitly states that the data is of integer type.

There is also a means of representing numeric data by character strings.

4.1.1 NUMERIC CONSTANTS

Numeric constants are generally floating point values, unless they are suffixed with a % sign, in which case they are integer values. An <integer> constant is syntactically defined as:

```
<digit> {<digit> ...}
```

A numeric constant consists of the following elements:

```
{ + { - } {<integer>} {.} {<integer>}
                        { E | e { + | - } <integer>}
```

given the rules stated above, the following are examples of correct constants:

```
+1                      -3
 0.0                     2.                  .14142
-10.5                  +128.                -.7071
2.99793E8             -1E-10                1.5E5
```

The following are examples of incorrect constants:

```
.          a decimal point alone is incorrect,
+. or -.   an operator with or without a period is also incorrect
E5         a variable name (not the value 10 5).
```

All the constants above are represented in floating point notation. Floating point numbers are 64-bit quantities. The range is approximately ± 10E ± 308. The precision is approximately 15 decimal places.

The floating point number system used in CS BASIC reserves certain values to indicate that an erroneous operation has taken place. The values of positive infinity, negative infinity, and Not a Number (usually called NaN) are such values. When used in normal arithmetic operations, such values behave as one would expect. For instance,

```
infinity + 1.0
```

is infinity, and

```
NaN + 1.0
```

is NaN, and so on. When printed, the value of positive infinity appears as +.+++++, negative infinity appears as -.-----, and NaN as ?.?????. These strange values are "created" by operations such as dividing a number by zero or taking the logarithm of a negative number. The user cannot represent infinity or NaN as a numeric constant in a CS BASIC program.

Integer constants are indicated by placing a percent sign after the number. Integers are represented by 16-bit numbers internally. The range of integers is therefore -32768 through +32767. However, the most negative integer constant allowed is -32767.

A percent sign placed after a floating point constant containing either a period or an exponent, for instance

```
7.6E6%
```

causes a syntax error. Similarly, a percent sign placed after an integer constant that is too large to be represented as an integer value also causes a syntax error. For example, the numbers:

```
99999% and -50000%
```

generate syntax errors.

If a constant contains a period or an exponent, it will be stored as a floating point data value by the CS BASIC interpreter. Similarly, a constant ending with a % sign will be stored as an integer data value by the CS BASIC interpreter.

If neither a period nor an exponent is present in a numeric constant, and the value is small enough to be stored as an integer, the constant is considered an "ambiguous" constant, and is stored as a floating point value, unless a % sign appears in the expression to the left of the constant, in which case, the CS BASIC interpreter stores the constant as an integer value.

4.1.2 NUMERIC VARIABLES

Numeric variables are designated by identifiers. A plain identifier represents the name of a floating point variable. If the name of the variable is followed by a percent sign, it means that the variable is to contain integer data.

A variable with a % sign following it is completely distinct from a floating variable of the same name, from a string variable of the same name, and from a dimensioned variable of the same name.

4.2 STRING DATA TYPES

CS BASIC implements a string data type. A string is a sequence of characters that can be manipulated as a single entity. A character string has a maximum length of 32767 characters.

Note that strings that are part of virtual arrays have additional restrictions on their length (see the discussion of virtual arrays later in this chapter).

4.2.1 STRING CONSTANTS

String constants in CS BASIC represent a sequence of ASCII characters. A string constant is delimited by either apostrophes (') or by double quote signs (").

If the user wishes to represent the string delimiter as a character in the string, two delimiters must be typed.

Examples of String Constants

```
"Haul on the bowline"

'Splice the mainspring'

'The time is Ten O''Clock'

"Call for the Great O'Reilly"
```

4.2.2 STRING VARIABLES

A string variable in CS BASIC is an identifier followed by a $ sign.

Examples of String Variables

```
wool.or.cotton$      A$
```

A variable with a $ sign following it is completely distinct from a floating, integer, or dimensioned variable of the same name.

4.2.3 NUMERIC STRING DATA

CS BASIC provides a means whereby character strings can be treated as numbers. This mechanism allows for exact, high-precision arithmetic without the need for scaling.

A numeric string is simply a string variable or constant whose characters conform to the rules for numeric constants defined above. A numeric string has a maximum size of 56 characters, including the + or - sign and the decimal point.

The language provides functions to operate on numeric string data, and these are described in Chapter 6.

4.3 ARRAYS, OR DIMENSIONED VARIABLES

CS BASIC provides for dimensioned variables. These are also known as arrays or matrices. Dimensioned variables can be introduced with the DIM (DIMension) statement. Floating, integer, and string variables can be dimensioned.

The same name can be used for a simple variable and for a dimensioned variable with no ambiguity.

There are powerful features in the language for manipulating entire arrays, these facilities are described in Chapter 10.

The format of the DIM statement is:

```
<line number> DIM <identifier>(<subscripts>)
                    {, <identifier>(<subscripts>) ...}
```

where <subscripts> is:

```
<upper bound> {, <upper bound>}
```

and <upper bound> is a numeric constant that determines the upper bound of that dimension of the array. Dimensioned variables can have one or two dimensions.

The DIM statement cannot be used in immediate mode.

Examples of DIM Statements

```
120 DIM hours(5)

130 DIM time.and.motion%(10, 20)

140 DIM days$(7),weeks%(52), seconds(366)

150 DIM names$(100, 100)
```

This example shows that more than one dimensioned variable can be declared per DIM statement, and also that the rules for floating, integer, and string data types apply.

The bounds of each dimension in the declaration must be a positive, nonzero integer. The bounds specify the upper bounds only; there is no facility for specifying the lower bound of a dimension. In all cases, the lower bound of a dimension is zero. In general, the zero'th element of a dimension is not used in array manipulations, but it is used in some cases, such as the CHANGE statement. Of course, the user can access element zero specifically, in the same manner as with any other array element.

Chapter 10 contains a detailed discussion on the way in which matrices are dimensioned and redimensioned.

4.3.1 VIRTUAL ARRAYS

Virtual arrays are a method for associating a dimensioned variable with a file on some external storage device. There are two applications for virtual arrays:

- manipulating arrays that are too big to fit in the available memory,
- performing random access to data stored on external devices.

Virtual arrays are declared in a variant of the DIM statement. The format of a virtual array declaration is:

```
DIM #<integer constant>, <identifier>(<subscripts>)
                {, <identifier>(<subscripts>) ...}
```

The <integer constant> is an integer in the range 1 through 12, which is the internal file designator of a disk file. File descriptors are treated more fully in Chapter 9.

Strings are treated specially in virtual arrays. Whereas string variables can range in length from zero up to 32767 characters, virtual string arrays have specific limits placed on them.

An element of a string virtual array has a maximum length of 512 characters. A string element need not have the maximum length declared. Also, maximum lengths shorter than 512 characters can be declared. The form of a string virtual array variable is:

```
<identifier>$(<subscripts>){=<integer constant>}
```

The =<integer constant> part of the declaration specifies the number of characters allocated for each element. The length specification is optional. The system defaults to 16 characters per element as the maximum length, if the length specification is omitted.

The maximum length specification for a string virtual array element must be one of the following powers of two:

```
2, 4, 8, 16, 32, 64, 128, 256, 512
```

In the event that a maximum length specification other than those stated above is specified, the system rounds up to the next larger power of two.

Examples of Virtual Array Declarations

```
200 DIM #4, ins.and.outs(50, 200)

300 DIM #5, names$(500)=64, addresses$(500)=512
```

The first example declares a 50-by-200 floating array. The second example declares two 500-element string arrays, with their maximum lengths specified.

4.4 INITIAL VALUES OF VARIABLES

When a CS BASIC program first uses a variable, the CS BASIC system gives that variable an initial value. The initial values assigned are:

```
0.0     for a floating variable,
0%      for an integer variable,
""      (the null string) for a string variable.
```

The CS BASIC system initializes arrays (except virtual arrays) by setting all of their elements to the values stated above.

Variables that are formal arguments to functions have their initial values assigned to the value of the actual arguments when the function is referenced. See Chapter 12 -- "CS BASIC Program Structure" -- for a description of how arguments are assigned.

Note that virtual arrays are not automatically initialized by the system. The programmer must explicitly write code to initialize a virtual array, or use the MAT ZER matrix initialization statement.

RUN and RUNNH do not set all variables to their initial values.

4.5 DISTINCTNESS OF VARIABLE NAMES

The same variable name can be used for more than one type of object with no ambiguity. This is because the name can be distinguished from the context in which it is used.

A variable with a % sign following it is completely distinct from a floating variable of the same name.

Similarly, a floating variable and an integer variable are distinct from a string variable of the same name.

Finally, the same names can be used for dimensioned variables and undimensioned variables with no ambiguity.

For instance, each of the references below may appear in the same CS BASIC program and refer to distinct variables:

```
I       I%      I$
I(2)    I%(2)   I$(2)
```

4.6 DEFINING VARIABLE DATA TYPES

Three keywords -- DEFINT, DEFDOUBLE, and DEFSTRING -- may be used to create BASIC statements as follows:

```
DEFINT    <letter range> { , <letter range> ... }
DEFDOUBLE <letter range> { , <letter range> ... }
DEFSTRING <letter range> { , <letter range> ... }
```

where <letter range> is either a single alphabetic character or an alphabetic character followed by a minus sign followed by another alphabetic character, the second of which collates (using the ASCII character set) at least as high as the first. All alphabetic characters are treated as if entered in upper case. These statements are not "executable" but take effect when entered into CS BASIC, and their effect continues until overridden by a subsequent DEFINT, DEFREAL, or DEFSTRING statement is entered or BASIC is completely reinitialized (e.g., via NEW, OLD, or CHAIN).

The effect of a DEFINT statement is to activate the indicated alphabetic character such that variables (including parameters, function names, and array names) subsequently entered into the BASIC system beginning with that character are transformed as follows:

```
<var>  is treated as <var>%
<var>% is treated as <var>%
<var>$ is treated as <var>$
```

Thus, for example, after the statement "DEFINT I-N" has been entered into the BASIC system, a definition or reference to FNIII is treated in exactly the same way as FNIII%. After the DEFINT statement it is no longer possible to enter floating variables (etc.) beginning with the indicated letters.

The effect of the DEFSTRING statement is similar to that of the DEFINT statement and induces the following transformations:

```
<var>  is treated as <var>$
<var>% is treated as <var>%
<Var>$ is treated as <var>$
```

The effect of the DEFDOUBLE statement is to cancel any transformations on the given letters, leaving the default state:

```
<var>  is treated as <var>
<var>% is treated as <var>%
<var>$ is treated as <var>$
```

None of these statements has any effect on program portions already entered into the BASIC system. The transformations are carried out strictly as new BASIC program input is first scanned.

5.0 CS BASIC EXPRESSIONS

Expressions in CS BASIC are mechanisms for computing new values. The values of operands (constants, variables and function values) are combined by operators to generate values.

This chapter describes the operators available for the different types of data and gives the rules for combining those operators and operands.

5.1 MIXED MODE ARITHMETIC

CS BASIC provides for arithmetic operators to operate upon a mixture of floating point and integer operands in the same expression.

If both the left and right operands of an arithmetic operator are integers, the result of the operation is an integer. If both operands are floating point, the result is also floating point. If one operand is a floating point number and the other is an integer, the integer is first converted to a floating point value, the operation is performed using floating point values, and the result is a floating point value.

To ensure that a constant is represented as a floating point constant, it should contain, or be terminated with, a decimal point.

A number should be suffixed with a % sign to ensure that it is represented as an integer.

Constants without a suffix at all are termed ambiguous constants, and their representation as floating point or integer depends on the statement in which they are used. If an integer variable or constant appears anywhere to the left of an ambiguous constant in a statement, that ambiguous constant represents an integer, otherwise it represents a floating point number.

5.2 ARITHMETIC OPERATORS

Arithmetic operators operate upon numeric (integer or floating point) operands and yield numeric results. The arithmetic operators are as shown in the table.

Operator	Meaning	Precedence
^ or **	Exponentiation	Highest
+	Unary Plus	Next Highest
-	Unary Minus	
*	Multiplication	Lower
/	Division	
+	Addition	Lowest
-	Subtraction	

The arithmetic operators have a precedence, ranging from exponentiation (the highest) to addition and subtraction (the lowest). The unary plus and minus bind tighter than the operators below them in the table.

Parentheses may be used to change the precedence of operators in an expression. In the absence of parentheses, operators of equal precedence are applied left to right, including the exponentiation operator.

The result from an attempt to raise a negative number to a fractional power is NaN (Not a Number).

The results of a division by zero depends upon the type of the operands. If the operands are floating point, the result of a division by zero is either positive or negative infinity. For example, the division:

```
1.0 / 0.0
```

generates positive infinity as a result, and the division:

```
-1.0 / 0.0
```

generates negative infinity as a result.

The representations and results of extreme values such as plus and minus infinity and NaN are covered in Appendix B.

If the operands of a division are integers, division by zero, in an expression such as:

```
A% /0%
```

results in a run-time error, that can be trapped by the ON ERROR GOTO facility described in Chapter 12.

The plus (+) and minus (-) signs can also be used as unary operators. The plus sign is simply ignored; the minus sign changes the sign of the expression that follows.

5.3 ARITHMETIC RELATIONAL OPERATORS

Arithmetic relational operators evaluate relationships between numeric operands. The precedence of arithmetic relational operators listed in the table below is the same as that of the arithmetic operators described in the previous section.

Operator	Meaning
=	Equal To
<	Less Than
<=	Less Than or Equal To
>	Greater Than
>=	Greater Than or Equal To
<>	Not Equal To
==	Approximately Equal To

The == sign stands for "approximately equal to" and is used when comparing floating point numbers. Internally, CS BASIC carries floating point numbers to a higher precision than is normally printed by PRINT statements. The use of the == operator is to compare numbers that look equal when printed but are actually unequal in the internal representation the computer. CS BASIC prints numbers to a precision of approximately six decimal places but represents them internally to a precision of approximately 15 decimal places.

5.4 LOGICAL OPERATORS

Logical operators are used to combine relational expressions into compound relational expressions. The logical operators have their usual meanings.

Operator	Meaning	Precedence
NOT	Logical Negation	Highest
AND	Logical Conjunction	Next Highest
OR	Logical Disjunction	
XOR	Logical Exclusive OR	Lower
IMP	Logical Implication	
EQV	Logical Equivalence	Lowest

5.5 STRING OPERATORS

String operators are those that operate upon string operands to produce string expressions. The basic string operator is concatenation, denoted by a plus sign (+).

String relational operators are used for lexicographic comparisons between string values. The string relational operators are as shown in the table below.

Operator	Meaning
=	Equivalent
<	Less Than
<=	Less Than or Equal To
>	Greater Than
>=	Greater Than or Equal To
<>	Not Equal To
==	Identical

The equivalence operator (=) means that its operands are equivalent except for possible trailing spaces.

The identity operator (==) means that its operands are identical, including trailing spaces. That is, they are both of the same length, and contain the same characters, in the same order.

The ASCII character set is used as the collating order for string comparison.

When strings of unequal length are compared, the shorter string (say of length n) is compared with the first n characters of the longer string. If the first n characters are equal, and the rest of the longer string is only spaces, the two strings are equivalent. If the first n characters are equal and the rest of the longer string is not spaces, the longer string is considered greater than the shorter string.

5.6 INTEGERS AS LOGICAL VARIABLES

CS BASIC allows integers to be used as logical variables. Logical operators can be applied to integer operands to generate bitwise logical expressions.

In addition, whenever a logical expression is expected, the values of the integers can represent the values TRUE and FALSE. An integer value of 0% represents the logical value FALSE. Any nonzero value represents the logical TRUE value. CS BASIC uses the integer value -1% (all ones) to represent the logical value TRUE when generated as a truth value in an expression.

When the logical operations in the truth tables below are applied to integer values, the values are considered as bit strings, not as signed integers.

In all cases, A and B are integer values.

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

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

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

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

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

A	NOT A
0	1
1	0

6.0 CS BASIC STANDARD FUNCTIONS

This chapter describes the three major types of functions built into the CS BASIC interpreter:

1. mathematical and arithmetic functions, which operate on numeric arguments and return numeric results;

2. string functions, which operate on character strings and return string results;

3. functions that treat character strings as numeric data. These functions also operate on strings and return string results, but they are considered a distinct type since they perform arithmetic operations on character strings.

Two other kinds of functions are discussed elsewhere in this manual:

4. User-defined functions are discussed in Section 12.3 of the chapter on program structure.

5. Functions that operate on matrixes are described in Section 10.6 of the chapter on matrix operations.

6.1 MATHEMATICAL FUNCTIONS

6.1.1 ABS -- COMPUTE ABSOLUTE VALUE

The ABS function returns the absolute value of its argument. The format of the ABS function is

```
ABS(X)
```

where X is a numeric value.

6.1.2 SGN -- FIND SIGN OF NUMBER

The SGN function determines the sign of its numeric argument. The format of the SGN function is:

```
SGN(X)
```

where X is a numeric argument. SGN returns:

```
-1 for a negative argument
 0 for a zero argument
+1 for a positive argument
```

6.1.3 INT -- ROUND DOWN TO NEAREST INTEGER

The INT function returns the largest integer that is less than or equal to its argument. The format of the INT function is:

```
INT(X)
```

where X is a numeric argument. The value of

```
INT(1.5)
```

is 1. The value of

```
INT(-0.5)
```

is -1.

6.1.4 FIX -- TRUNCATE TO INTEGER

The FIX function truncates its argument to the nearest integer. The form of the FIX function is:

```
FIX(X)
```

where X is a numeric argument. The value of

```
FIX(0.5)
```

is 0.

6.1.5 COS -- TRIGONOMETRIC COSINE

The COS function returns the cosine of its argument. The format of the COS function is:

```
COS(X)
```

where X is a numeric argument. The argument to COS is in radians.

6.1.6 SIN -- TRIGONOMETRIC SINE

The SIN function returns the sine of its argument. The format of the SIN function is:

```
SIN(X)
```

where X is a numeric argument. The argument to SIN is in radians.

6.1.7 TAN -- TRIGONOMETRIC TANGENT

The TAN function returns the tangent of its argument. The format of the TAN function is:

```
TAN(X)
```

where X is a numeric argument. The argument to TAN is in radians.

6.1.8 ATN -- TRIGONOMETRIC ARC TANGENT

The ATN function returns the arctangent of its argument. The format of the ATN function is:

```
ATN(X)
```

where X is a numeric argument. The value of the ATN function is in radians.

6.1.9 SQR -- COMPUTE SQUARE ROOT

The SQR function returns the square root of its argument. The format of the SQR function is:

```
SQR(X)
```

where X is a numeric argument.

If the argument to SQR is negative, the value of the function is NaN (Not a Number). See Appendix B.

6.1.10 EXP -- EXPONENTIAL FUNCTION

The EXP function returns the exponential of its argument, e , where e is 2.71828...The format of the EXP function is:

```
EXP(X)
```

where X is a numeric argument.

6.1.11 LOG -- NATURAL LOGARITHM

The LOG function returns the natural logarithm (log x) of its argument. The format of the LOG function is:

```
LOG(X)
```

where X is a numeric argument.

If the argument X is a negative number, the value of the LOG function is Not a Number (NaN).

6.1.12 LOG10 -- LOGARITHM TO BASE 10

The LOG10 function returns the logarithm to the base 10 (log x) of its argument. The format of the LOG10 function is:

```
LOG10(X)
```

where X is a numeric argument.

If the argument X is a negative number, the value of the LOG10 function is Not a Number (NaN).

6.1.13 PI -- CONSTANT VALUE OF PI

The PI function returns a constant value of pi, the ratio of a circle's circumference to its diameter. The value is approximately 3.14159.

6.1.14 RND -- RANDOM NUMBER GENERATOR

The RND function returns uniformly distributed pseudo-random numbers in the range 0.0 to 1.0. The format of the RND function is

```
RND(X)
```

The argument X is ignored and can be omitted.

The RND function generates the same sequence of numbers each time the program is run.

6.1.15 SWAP% -- SWAP BYTES IN AN INTEGER

The SWAP% function swaps the bytes in an integer value. The format of the SWAP% function is:

```
SWAP%
```

where N% is an integer expression.

6.1.16 CCPOS OR POS -- CURRENT POSITION OF PRINT HEAD

The CCPOS or POS function returns the current position of the "print head" for a specific input-output channel. The format of the function

```
CCPOS(N%)
```

or

```
POS(N%)
```

where N% is an integer expression.

6.1.17 TAB -- SET PRINT POSITION

The TAB function can be used only in a PRINT statement. The TAB function moves the printing position in the current print record to a specified position. The format of the TAB function is:

```
TAB(N%)
```

where N% is an integer expression. The TAB function returns a string of spaces to move the print position to column N on the output line.

If a PRINT statement that refers to file #0 (the user's terminal) contains a TAB(N%) function, the TAB(N%) function returns the number of spaces necessary to move the print position to column N.

6.2 STRING FUNCTIONS

This section describes the functions available in BASIC for manipulating character strings.

6.2.1 LEFT -- TAKE LEFT SUBSTRING OF STRING

The LEFT function extracts a substring from a string, starting at the first character in the string, and extending for the specified number of characters. The format of the LEFT function is:

```
LEFT(A$, N)
```

where A$ is a string variable, and N is the number of characters to extract from it.

If the number of characters to extract is less than 1, the result of the LEFT$ function is a null string. If the number of characters to extract is greater than the length of the string variable A$, the result of the LEFT function is all of A$.

Example of the LEFT Function

```
LET Sport$ = "Hunting The Snark"
PRINT LEFT(Sport$, 7%)

Hunting
```

6.2.2 RIGHT -- TAKE RIGHT SUBSTRING OF STRING

The RIGHT function extracts a substring from a string starting at a specified character in the string, and extending to the last character in the string. The format of the RIGHT function is:

```
RIGHT(A$, N)
```

where A$ is a string variable and N is the character position in the string at which the extraction starts.

If the starting position N is less than 1, the result of the RIGHT function is all of the string A4. If the starting position is greater than the length of the string variable A4, the result of the RIGHT function is a null string.

Example of the RIGHT Function

```
LET Sport$ = "Hunting The Snark"
PRINT RIGHT(Sport$, 9)

The Snark
```

6.2.3 MID -- TAKE SUBSTRING OF STRING

The MID function extracts a substring from a string, starting at a specified character in the string and extending for the specified number of characters. The format of the MID function is:

```
MID(A$, S, L)
```

where A$ is a string variable, S is the position in the string at which extraction starts, and L is the number of characters to be extracted.

If the starting position S is less than 1 or greater than the length of the string variable A$, the result of the MID function is a null string.

If L, the number of characters to extract, is less than 1, the result of the MID function is a null string. If the number of characters to extract is greater than the length of the string variable A$, the result of the MID function extends from the starting position to the end of A$.

If the sum of the starting position S, plus the number of characters to extract, L, is greater than the number of characters remaining in the string variable A$, the result of the MID function extends to the end of A$.

Example of the MID Function

```
LET Sport$ = "Hunting The Snark"
PRINT MID(Sport$, 9, 3)

The
```

6.2.4 LEN -- COMPUTE LENGTH OF STRING

The LEN function returns the length of a string, including trailing spaces in the string. The format of the LEN function is:

```
LEN(A$)
```

where A$ is a string variable.

Example of the LEN Function

```
100  LET S$ = "ABCDEFGHIJKLM"
110  L% = LEN(S)
120  PRINT L%
130  END
runnh
13
Ready
```

6.2.5 STRING CONCATENATION WITH THE + OPERATOR

The plus sign (+), when applied to string data elements, signifies a concatenation of its operands.

Example of the String Concatenation Function

```
100 LET Left.Hand$ = "ABC"
110 LET Right.Hand$ = "XYZ"
120 LET Whole$ = Left.Hand$ + Right.Hand$
130 PRINT Left.Hand$, Right.Hand$, Whole$, LEN(Whole$)
140 END
runnh
ABC     XYZ     ABCXYZ     6
Ready
```

6.2.6 CHR$ -- CHARACTER VALUE OF INTEGER

The CHR$ function generates a one-character string whose value is the ASCII character corresponding to its numeric argument. The format of the CHR$ function is:

```
CHR$(N)
```

where N is a number in the range 0 to 127.

For instance, the function call has as its value the space character,

```
CHR$(32)
```

and

```
CHR$(65)
```

has the value of the letter "A".

If the numeric argument to CHR$ is negative or greater than 127, the lower eight bits are used.

Example of the CHR$ Function

```
PRINT CHR$(66)

B
```

6.2.7 ASCII -- INTEGER EQUIVALENT OF CHARACTER

The function

```
ASCII(A$)
```

generates an integer that is the numeric value of the first character of the string A$.

For example,

```
ASCII("F")
```

is the value 70. If the string variable V$ contains the string "Wizard," then the function

```
ASCII(V$)
```

has the value of the first character, namely the value 87.

6.2.8 INSTR -- SEARCH FOR SUBSTRING IN STRING

The INSTR function performs a search for a specified substring within a string. The format of the INSTR function is

```
INSTR(P, A$, B$)
```

where A$ is a string, B$ is the substring to be searched for in A$, and P is the position in the string A$ at which the search is to start.

IF string B$ is not found in string A$, INSTR returns a value of 0.

If string B$ is found in string A$, INSTR returns the position in string A$ at which B$ was found. Character positions are numbered from 1, starting at the leftmost character of the string.

If the starting search position S is less than 1, or greater than LEN(A$), the INSTR function returns the value zero.

If the length of substring B$ is greater than the length of string A$, the INSTR function returns the value zero.

Example of the INSTR Function

```
LET Sport$ = "Hunting The Snark"
PRINT INSTR(1, Sport$, "The S")
9
PRINT INSTR(1, Sport$, "the S")
0
```

6.2.9 SPACE$ -- GENERATE STRING OF SPACES

The SPACE$ function generates a string of spaces. It is useful for filling a string variable to all spaces or for inserting a number of spaces into a string. The format of the SPACE$ function is:

```
SPACE$(N%)
```

where N% is an integer value specifying the number of spaces to generate.

Example of the SPACE$ Function

```
B$ = SPACE$(16%)
```

This example assigns a string of 16 spaces to the variable B$.

6.2.10 NUM$ -- STRING REPRESENTATION OF NUMBER

The NUM$ function returns a string of characters that represents the value of a numeric argument in the way that a PRINT statement would print it. The format of the NUM$ function is:

```
NUM$(N)
```

where N is a numeric data value.

If the numeric argument is positive, NUM$ generates the string representation of the number, with a space character on either side of the string. If the numeric argument is negative, NUM$ generates the string representation of the number, preceded by a minus sign, and followed by a space.

If the numeric argument is outside the range -999999 through 999999, the string representation is generated in floating point format.

Example of the NUM$ Function

```
A$ = NUM$(567)
B$ = NUM$(-234)
PRINT A$; B$
567 -234
Ready
```

6.2.11 NUM1$ -- STRING REPRESENTATION OF NUMBER

The NUM1$ function returns a string of characters that is the string representation of its numeric argument. The format of the NUM1$ function is:

```
NUM1$(N)
```

where N is a numeric data value. The generated string does not contain any spaces, nor is it converted to floating point format.

The NUM1$ function is useful for converting numbers to string-numeric data types.

Example of the NUM1$ Function

```
A$ = NUM1$(2 24)
PRINT A$
16777216
Ready
```

6.2.12 VAL -- CONVERT STRING TO NUMBER

The VAL function converts its string argument to a floating point number. The format of the VAL function is:

```
VAL(A$)
```

where A$ is a string data value containing the representation of a number. The string A$ can contain a plus sign, minus sign, or decimal point.

If the string A$ contains characters that are not digits, + or - or . characters, the VAL function generates a run-time error, that can be trapped by the ON ERROR GOTO facility if required.

Example of the VAL Function

```
A$ = '-12869.345'
F = VAL(A$)
PRINT F
-12869.3
```

6.2.13 STRING$ -- CREATE REPEATED CHARACTER STRING

The STRING$ function generates a string consisting of a specified number of characters. The format of the STRING$ function is:

```
STRING$(N, V)
```

where N is the number of characters to generate and V is the value of the character.

If V is outside the range of 0 to 255, the lower eight bits of the value are used.

Example of the STRING$ Function

```
A$ = STRING$(10, 35) + STRING$(15, 61) + STRING$(10, 37)
PRINT A$
##########aaaaaaaaaaaaaaa%%%%%%%%%
Ready
```

6.2.14 CVT CONVERSION FUNCTIONS

There are five variations of the CVT function, which maps between string and integer forms. These functions map directly between numerical and string data, and are used primarily to store floating point and integer data in block input-output files.

The variations are described below, and are summarized here:

`S$ = CVT%$(I%)` Maps the integer expression I% into a two-character string S$.

`I% = CVT$%(S$)` Maps the first two characters of the string S$ into an integer I%.

`S$ = CVTF$(X)` Maps the floating point expression X into an eight-character string S$.

`X = CVT$F(S$)` Maps the first eight characters of the string S$ into the floating point value X.

`T$ = CVT$$(S$,M%)` Performs editing on the string argument S$.

6.2.14.1 CVT%$ -- Map Integer to String

The CVT%$ Maps its integer argument into a two-character string. The form of the CVT%$ function is:

```
CVT%$(I%)
```

where I% is an integer expression. The result of the CVT%$ function is a two-character string.

Example of the CVT%$ Function

```
PRINT CVT%$(16730)

AZ
```

6.2.14.2 CVT$% -- Map Characters to Integer

The CVT$% function maps the first two characters of its string argument into an integer value. The format of the CVT$% function is:

```
CVT$%(S$)
```

where S$ is a character string expression. The result of the CVT$% function is an integer.

Example of the CVT$% Function

```
PRINT CVT$%("MN")

19790
```

6.2.14.3 CVTF$ -- Map Floating Point to String

The CVTF$ function maps its floating point argument into an eight-character string. The format of the CVTF$ function is:

```
CVTF$(X)
```

where X is a floating point expression. The result of the CVTF$ function is an eight-character string.

6.2.14.4 CVT$F -- Map Characters to Floating Point

The CVT$F function maps the first eight characters of its string argument into a floating point value. The format of the CVT$F function is:

```
CVT$F(S$)
```

where S$ is a character string expression. The result of the CVT$F function is a floating point value.

6.2.14.5 CVT$$ -- String Editing

The CVT$$ function provides for editing of character string values under control of its arguments. The format of the CVT$$ function is:

```
CVT$$(S$, M)
```

where S$ is a character string data value, and M is a control parameter.

The characters in S$ are edited according to the value of M. M is a "bit-mask" whose individual values have the following meanings:

1 Trim the parity bit from each character.

2 Discard all spaces and tabs from the string.

4 Discard carriage returns, line feeds, form feeds, escape, rubout, and NULL characters.

8 Discard leading spaces and tabs only.

16 Reduce runs of spaces and tabs to a single space.

32 Convert lower case letters to upper case.

64 Convert [and] characters to (and).

128 Discard trailing spaces and tabs only.

256 Do not alter characters inside single or double quotes, except for trimming the parity bit.

The bits in the M parameter can be combined additively. For example, setting M to 96 (64+32) means that lower case letters are converted to upper case, and brackets are converted to parentheses.

Example of the CVT$ Function

```
LET Sport$ = "Hunting The Snark"
PRINT CVT$$(Sport$, 2%)

HuntingTheSnark

PRINT CVT$$(Sport$, 32%)

HUNTING THE SNARK

PRINT CVT$$(Sport$, 34%)

HUNTINGTHESNARK
```

The result of the first CVT$$ function removes all spaces (and tabs) from the string Sport$. The second example uses the mask that raises the entire string to upper case. The third example combines the two previous masks, discarding all spaces and tabs while raising the string to upper case.

6.2.15 XLATE -- CHARACTER TRANSLATION

The XLATE function provides a "table lookup" capability for translating characters in a string. The format of the XLATE function is:

```
XLATE(S$, T$)
```

where S$ is a source string that is to be translated, and T$ is a lookup table string that is used to do the translation.

The action of the XLATE function is that each character from S$ is used as an index into the lookup table T$. Indexing starts from zero, with the first character in T$ indexed by 0, up to the last possible position of 255. The character at that position is used to form the value of the XLATE function.

The translation process terminates under one of two conditions:

1. The lookup table string T$ has a length shorter than the index from the source string S$,

2. A zero value is found at the indexed position in the lookup table T$.

Example of the XLATE Function

```
T$ = STRING(32, ASCII(' ')) + STRING(16, ASCII('!')) &
   + STRING(10, ASCII('0')) + STRING(26, ASCII('A')) &
   + '!!!!!!!!' + STRING(26, ASCII('A')) &
   + '!!!!!!' + STRING(128, ASCII(' '))
A$ = 'Maryann Clark'
B$ = XLATE(A$, T$)
PRINT B$
```

The example shown above has a translation table such that all alphabetic characters translate to 'A', all digits translate to '0', all delimiters translate to '!', and everything else translates to space.

6.2.16 RAD$ -- CONVERT FROM RADIX 50

The RAD$ function converts an integer value into three ASCII characters. The format of the RAD$ function is:

```
RAD$(N%)
```

where N% is an integer expression. The result is a three-character string.

6.3 NUMERIC STRING FUNCTIONS

The BASIC language provides for operating on numbers that are represented by strings of characters. This allows for arithmetic upon numbers up to 56 characters in length. The limit of 56 characters includes any plus sign, minus sign, or decimal point.

The functions described below are for performing arithmetic operations on these data types.

The PROD$ (product), QUO$ (quotient), and PLACE$ (roundoff) functions each have a parameter, **P** (for **P**laces), that is the number of decimal places to which the result of the function is to be rounded or truncated.

The **P** parameter is an integer expression. Values of **P** less than 5000 indicate that the result of the function should be rounded. Values of **P** greater than 5000 indicate that the result of the function should be truncated. In this second case, the actual number of places to which the result is truncated is derived from the expression:

P - 10000

If the number of places to round or truncate is positive, the results are rounded or truncated that number of places to the right of the decimal point.

If the number of places to round or truncate is negative, the results are rounded or truncated to the left of the decimal point.

In all cases, if nonnumeric characters (other than + or - or .) are found in the string, a run-time error is generated.

6.3.1 SUM$ -- ARITHMETIC SUM OF NUMERIC STRINGS

The SUM$ function adds two numeric strings together. The format of the SUM$ function is:

```
SUM$(A$, B$)
```

where A$ and B$ are both numeric strings.

Example of the SUM$ Function

```
PRINT SUM$("12345", "67890")

80235
```

6.3.2 DIF$ -- ARITHMETIC DIFFERENCE OF NUMERIC STRINGS

The DIF$ function subtracts one numeric string from another. The format of the DIF$ function is:

```
DIF$(A$, B$)
```

where A$ and B$ are both numeric strings.

The result of the DIF$ function is A$-B$.

Example of the DIF$ Function

```
PRINT DIF$("33554432", "16777216")

16777216
```

6.3.3 PROD$ -- ARITHMETIC PRODUCT OF NUMERIC STRINGS

The PROD$ function multiplies two numeric strings together. The format the PROD$ function is:

```
PROD$(A$, B$, P)
```

where A$ and B$ are both numeric strings and P is the number of places to which the product is to be rounded or truncated.

The result of the function is the product of A$ and B$, rounded or truncated to P decimal places.

Example of the PROD$ Function

```
PRINT PROD$("1024", "1024" 10%)

1048576
```

6.3.4 QUO$ -- ARITHMETIC QUOTIENT OF NUMERIC STRINGS

The QUO$ function divides one numeric string by another. The format of the QUO$ function is:

```
QUO$(A$, B$, P)
```

where A$ and B$ are both numeric strings and P is the number of places to which the result is to be rounded or truncated.

The result of the QUO$ function is A$ divided by B$, with the quotient rounded or truncated to P decimal places.

Example of the QUO$ Function

```
PRINT QUO$("16777216". "1024", 10%)

16384
```

6.3.5 PLACE$ -- ROUND NUMERIC STRING

The PLACE$ function rounds a numeric string to a specified number of places. The format of the PLACE$ function is:

```
PLACE$(A$, P)
```

where A$ is a numeric string, and P is the number of places to which the string A$ is to be rounded or truncated.

Example of the PLACE$ Function

```
PRINT PLACE$("123.456", 2%)
123.46

PRINT PLACE$("123.456", 10000 + 2%)
123.45

PRINT PLACE$("126.6666", -2%)
13

PRINT PLACE$("126.666", 10000 + -2%)
12
```

6.3.6 COMP% -- NUMERIC STRING COMPARISON

The COMP% function compares two numeric strings and returns an integer truth value based on the results of the comparison. The format of the COMP% function is:

```
COMP%(A$, B$)
```

where A$ and B$ are both numeric strings.

The COMP% function returns a value that represents the result of the comparison, as follows:

```
-1 if A$ < B$
 0 if A$ = B$
+1 if A$ > B$
```

Example of the COMP% Function

```
PRINT COMP%("123", "456")
-1
PRINT COMP%("123", "100")
1
PRINT COMP%("262144", "262144")
0
```

7.0 ASSIGNMENT STATEMENTS

This chapter discusses the assignment statements of CS BASIC: The LET (assignment) statement, the LSET and RSET (in-place string assignment) statements, and the CHANGE statement.

7.1 LET

Assignment statements are the fundamental statements of CS BASIC, where the value of an expression can be assigned to a variable. The form of an assignment statement is:

{LET} <variable> = <expression>

where <variable> is the name of a numeric variable, a string variable, or an element of a dimensioned numeric or string variable.

<expression> is the value of a numeric or string expression, that is to be assigned to the <variable>.

The LET keyword is optional and is retained for compatibility with other implementations of BASIC.

The data type of <variable> must agree with the data type of <expression>. That is, if <expression> is numeric, <variable> must be numeric; if <expression> is a string expression, <variable> must be a string variable.

If <variable> is an integer variable and <expression> is a floating point expression, the result is truncated (if possible) before the assignment. If the result of the expression is too large to fit an integer value, the value of the integer is undefined.

If <variable> is a floating point variable, and the <expression> is an integer expression, it is converted to floating point before the assignment.

Examples of Assignment Statements

```
100 LET Count% = 10%
120 LET Tax.Rate = 23.55
140 Emp%(Emp.Num%) = 795%
```

7.1.1 MULTIPLE ASSIGNMENT

In CS BASIC it is possible to assign the value of an expression to a collection of variables. The form of a multiple assignment is:

{LET} <var> {, <var> ...} = <expression>

Each <var> in the format above is the name of a variable of the same type as the <expression>. All the <var>'s are either numeric (float or integer) or string.

The <expression> is computed once, then the result is assigned to the list of variables.

The order of assignment is undefined.

The same variable can appear in the list of variables more than once.

Note that a matrix can be initialized to certain values (such as all zeros, all ones, or identity) by matrix initialization as described in Chapter 10.

7.1.2 STRING ASSIGNMENT

The form of a string assignment statement is just the same as that for numeric assignment, namely:

{LET} <stringvar> = <expression>

There are some aspects of string assignment that the user must be aware of. String variables are actually implemented as descriptors containing the length of the string and the address of the storage area containing the string. When one string variable is assigned to another, the system simply copies the descriptor. The result is that both descriptors refer to the same area of storage. (This is not true for string virtual arrays -- see below.)

Suppose that the following assignment is made:

```
LET A$ = "Skyhook"
```

The variable A$ now is a descriptor to a storage area with the string "Skyhook" in it. A subsequent assignment:

```
LET B$ = A$
```

makes the descriptor for B$ reference the same storage area as the descriptor for A$.

If either A$ or B$ is subsequently changed, the storage areas are made different. For example, the assignment:

```
LET A$ = "new string"
```

makes a new storage area containing the string "new string". A$ references this new string, while B$ still references the "Skyhook" string.

If, when assigning one string variable to another, a distinct storage area is actually required, it can be done with an assignment statement of the form:

```
LET B$ = A$ + ""
```

In other words, B$ and A$ now reference different areas of storage, even though the concatenation only concatenated a null string. It is not necessary to do this in general, unless LSET and RSET are being used in the same CS BASIC program.

To change strings without moving their actual storage area, the LSET and RSET statements must be used, as described below.

7.1.2.1 Special Notes on Assigning to String Virtual Arrays

For string virtual arrays, the assignment works differently from that of ordinary strings. If the source or target of a string assignment is an element of a string virtual array, the string is copied into the string virtual array. This is in contrast to assignment with ordinary strings, where only the descriptor is changed.

If the target of a string assignment is a string virtual array, and if the source string is shorter in length than maximum length of the target string, the array element is filed with zeros on the right.

7.1.3 LSET AND RSET -- CHANGE STRINGS IN PLACE

The LSET and RSET statements find most use in connection with block input and output, as described in Chapter 7, but they are described here since they are actually variations on the string assignment facilities.

LSET and RSET provide for changing the value of a string without moving the string in storage in any way. The forms of the LSET and RSET statements are:

LSET <stringvar> {, <stringvar> ...} = <string>

RSET <stringvar> {, <stringvar> ...} = <string>

where <stringvar> is the name of an existing string, and <string> is a string expression. Both LSET and RSET change the destination strings in place. The string that was previously stored in <stringvar> is overwritten, but the length of <stringvar> is unchanged.

If the length of <string> is greater than that of <stringvar>, the system truncates <string> to fit.

The LSET statement places the new <string> left justified in <stringvar>. If the length of <string> is less than that of <stringvar>, LSET pads on the right with spaces.

The RSET statement places the new <string> right justified in the <stringvar>. If the length of <string> is less than that of <stringvar>, RSET pads on the left with spaces.

Example of the LSET Statement

```
A$ = 'ABCDE'
B$ = A$
C$ = 'MNOPQ'
D$ = C$
LSET A$ = 'XYZ'
C$ = 'DEFGH'
```

In the example above, the strings A$ and B$ both refer to the same storage area. After the LSET statement, both A$ and B$ will contain the value 'XYZ ', but the strings C$ and D$ will contain different values because the final simple assignment statement creates a new string.

7.2 CHANGE -- CHARACTER AND NUMERIC CONVERSION

The CHANGE statement operates in one of two ways:

- converts every character in a string variable into its numeric equivalent,

  or

- converts an array of numeric values into characters in a string.

The form of the CHANGE statement is:

```
CHANGE <X> TO <Y>
```

If <X> is a string, <Y> must be a numeric array variable. If <X> is a numeric array variable, <Y> must be a string.

When converting from a string variable to a numeric representation, element zero of the destination numeric array will be set to the number of characters in the string. The first converted character from the string is then placed in element 1 of the numeric array, and so on.

If the conversion is from string to numeric representation and the number of characters in the string is greater than the number of elements in the numeric array, a run-time error is produced.

If in changing from numeric to string format, the contents of the zero'th element of the numeric array is zero or negative, a zero length string is generated.

Examples of the CHANGE Statement

```
100 DIM Decimal%(3)
110 LET String$ = "ABC"
120 CHANGE String$ To Decimal%
130 PRINT Decimal%(0); Decimal%(1); &
        Decimal%(2); Decimal%(3)
140 Decimal%(0) = 3 \ Decimal%(1) = 88 &
    Decimal%(2) = 89 \ Decimal%(3) = 90
150 CHANGE Decimal% TO String$
160 PRINT String$
170 END
runnh
3 65 66 67
XYZ
Ready
```

8.0 CONTROL STATEMENTS

This chapter covers CS BASIC control statements, which control the flow of execution of a program. Input and output statements are not discussed in this chapter but are covered in Chapter 9.

In the sections to follow, there are descriptions of statements, such as FOR and WHILE, as well as IF-THEN-ELSE, that have "block structure." Control statements can occur inside the range of other such control statements, but they must "nest" properly. For example, the following structure is erroneous and is not allowed:

```
100 FOR I = 1 TO 10
110  WHILE X% = 0
120    INPUT #1, A(I)
130  NEXT I
140 NEXT
```

The example illustrates incorrect nesting of statements. The NEXT I and the plain NEXT statements should be interchanged. The BASIC interpreter generates an error message in such situations.

8.1 IF THEN AND IF GOTO STATEMENTS

The IF THEN and IF GOTO statements are the most basic control statements in the language. There are three forms of the IF statement:

IF <condition> THEN <statements>

IF <condition> THEN <line number>

IF <condition> GOTO <line number>

The first form executes the <statements> following the THEN keyword if the <condition> is true. If the <condition> is false, the <statements> following the THEN keyword are not executed.

The second and third forms are equivalent: if the <condition> is true, program execution resumes at the line number specified by <line number>. If the

<condition> is false, program execution continues at the statement following the IF statement.

If there are multiple <statements> following the THEN keyword, all of those statements are executed if the <condition> is true, and none of the <statements> is executed if the <condition> is false.

Examples of IF Statements

```
100  IF A < B  THEN  PRINT "A is less"

200  IF A$ <> B$ then 230

300  IF A$ = B$ GOTO 1435

400  IF A = B THEN PRINT A \ PRINT B
```

In the last example (line 400), both of the PRINT statements are executed if A equals B, and neither of the PRINT statements is executed if A does not equal B.

8.2 IF THEN ELSE

The basic forms of IF THEN ELSE are similar to those of the IF THEN and IF GOTO statements, with the addition of an ELSE clause in the construct. The forms are as follows:

```
                  THEN <statement>
                                    { ELSE <statement>}
IF <condition> THEN <line number>
                                    { ELSE <line number>}
                  GOTO <line number>
```

The "piles" in the description above represent alternative ways of typing the statements. Either of the ELSE forms can be optionally combined with any of the forms of the basic IF statement.

If multiple statements follow the ELSE clause, all those statements are executed in the event that the <condition> was false, causing the ELSE branch of the statement to execute. If the <condition> is true, the ELSE branch is not taken, and none of the statements that follow it are executed.

Examples of IF THEN ELSE Statements

```
100 IF X = Y THEN PRINT "Same" ELSE PRINT "Different"

240 IF A$ = "GROOVY" THEN PRINT "All groovy" ELSE 1950

300 IF ASCII(X$) = 9 THEN PRINT 'Starts with TAB' &
                      ELSE PRINT 'Tabless' \ STOP

350 IF A% <> B% THEN A% = 10 ELSE A% = 20 \ B% = 30
```

In the last example above (line 350), both of the statements following the ELSE are executed if A% equals B%, and neither of those statements is executed if A% does not equal B%.

8.3 WHILE NEXT

The WHILE statement executes a block of subordinate statements while some condition remains true. The form of the WHILE NEXT statement is:

```
WHILE <condition>
  <statements>
NEXT
```

The <statements> between the WHILE and the NEXT are executed as long as <condition> remains true. If <condition> is initially false, none of the <statements> is executed.

Example of the WHILE Statement

```
100 WHILE A% <> 0
120   GOSUB 1000
140 NEXT
```

Note that the WHILE NEXT statement is not available in immediate mode.

8.4 UNTIL NEXT

The UNTIL statement executes a block of subordinate statements while some condition remains false. The form of the UNTIL NEXT statement is:

```
UNTIL <condition>
  <statements>
NEXT
```

The <statements> between the UNTIL and the NEXT are executed as long as <condition> remains false. IF <condition> is initially true, none of the <statements> is executed.

Example of the UNTIL Statement

```
100 UNTIL X == 0.0
150   INPUT #1, X
200 NEXT
```

Note that the UNTIL NEXT statement is not available in immediate mode.

8.5 FOR NEXT

The FOR NEXT statement is the basic mechanism for constructing loops in a BASIC program. The form of the FOR NEXT statement is:

```
FOR <var> = <exp1> TO <exp2> {STEP <exp3>}
. . . . . Statements subordinate to the FOR
NEXT <var>
```

The <var> is a numeric variable that is initialized to the value of <exp1>. The second expression, <exp2>, represents a limit to the value of <var>. If the optional third expression -- <exp3> -- following the STEP statement is present, its value increments <var> (or decrements it if <exp3> is negative) every time around the FOR loop, otherwise a value of one (1) is used for the increment.

If the step (<exp2>) is positive, the loop terminates when

```
<var>  >  <exp2>
```

If the step (<exp2>) is negative, the loop terminates when

```
<var>  <  <exp2>
```

If the step (<exp2>) is zero, the effect of the FOR loop is equivalent to:

```
<var>  =  <exp1>
```

```
WHILE <var>  <=  <exp2>
      .
      .
      .
NEXT
```

The NEXT <var> statement selects the next iteration on the loop variable <var>.

The subordinate statements between the FOR statement and the NEXT statement are executed as long as the loop does not terminate.

It is possible for a FOR NEXT loop to execute zero times (in other words, not execute at all) if the value of the control variable is greater than the limit at the start (or less than the limit for a negative increment).

If a NEXT statement is executed without a matching FOR statement having been previously executed (possible due to a transfer into the body of the FOR loop), the results are undefined.

Examples of FOR-NEXT Statements

```
100 FOR I = 1 To 10
110     PRINT 2^I
120 NEXT I
```

The example above is a simple loop that prints powers of 2 between 1 and 10.

Note that the FOR-NEXT statement is not available in immediate mode.

8.6 FOR WHILE AND FOR UNTIL

The FOR WHILE and FOR UNTIL statements are similar to the basic FOR statement, but instead of counting to some specified limit the loop continues iterating WHILE some condition remains true or UNTIL some condition becomes true. The forms of the statements are:

```
FOR <var> = <exp1> {STEP <exp2>} WHILE <condition>
. . . . . Statements subordinate to the FOR
NEXT <var>
```

```
FOR <var> = <exp1> {STEP <exp2>} UNTIL <condition>
. . . . . Statements subordinate to the FOR
NEXT <var>
```

Both of these forms of the FOR statement start off in the same way: a numeric variable <var> is set to an initial value, which is the expression <exp1>.

In the case of the FOR-WHILE form of the statement, the loop terminates when the conditional expression given by <condition> becomes false.

In the case of the FOR-UNTIL form of the statement, the FOR loop terminates as soon as the conditional expression <condition> becomes true.

Each time the NEXT statement is encountered, the variable <var> is incremented either by 1, or the optional expression, <exp2>, specified in the STEP part of the loop.

It is possible that the FOR WHILE and FOR UNTIL execute zero times (in other words they do not execute at all) if the <condition> is initially false (in the case of the WHILE) or true (in the case of the UNTIL). In such a case, the NEXT statement is never executed and <var> retains its initial value.

Example of a FOR-WHILE Statement

```
100 FOR I = 1 WHILE A(I) > 0
110     A(I) = A(I) + 1
120 NEXT I
```

The example above increments each element of an array, while an array element is greater than zero.

Example of a FOR UNTIL Statement

```
100 FOR J = 2 STEP 2 UNTIL J >= 100 OR A(J) <= 0
110     A(J) = A(J) + A(J-1)
120 NEXT J
```

This example demonstrates the use of the UNTIL part of a FOR loop.

Note that neither the FOR WHILE nor the FOR UNTIL statements is available in immediate mode.

8.7 GOTO

The GOTO statement provides for unconditional transfer of control to another part of the program. The form of the GOTO statement is:

GOTO <line number>

The GOTO statement unconditionally transfers control to the statement specified by <line number>.

Example of a GOTO Statement

```
450 GOTO 295
```

8.8 ON GOTO

The ON GOTO statement transfers control according to the value of a numeric expression. It is similar to the computed-GOTO of FORTRAN. The form of the ON GOTO statement is:

ON <exp> GOTO <list of line numbers>

The expression given by <exp> is used to select one line number out of the list of line numbers specified.

Indexing of the list starts at one (1). So in the example below, if I is equal to 1, statement 2000 is selected.

The <exp> part of the ON GOTO statement is converted to integer for the purposes of selection.

If <exp> is less than 0 or greater than the number of elements in the list of line numbers, the ON GOTO statement generates a run-time error. This error can be trapped via an ON ERROR GOTO statement.

Example of an ON GOTO Statement

```
1650  ON I GOTO  2000, 2100, 2200, 2400, 2700
```

8.9 STATEMENT MODIFIERS

The statement modifier facility of the language provides a convenient way to execute a statement conditionally, where the condition is actually a part of the statement.

All the statement modifiers have the same basic format:

<statement> <modifier>

The <modifier> serves as a qualifier for the <statement>.

Note that in a multiple-statement line, a statement modifier affects only the statement that it immediately qualifies. Other statements on the same line are not affected by that specific statement modifier.

8.9.1 IF STATEMENT MODIFIER

The IF statement modifier is a qualifier indicating that the modified statement is executed only IF a specified condition is true. The form of the IF statement modifier is:

<statement> IF <condition>

This means that the <statement> is executed only if the <condition> is true.

Examples of an IF Statement Modifier

```
130 PRINT "Time to go" IF T > 5

140 PRINT 10: PRINT 11 IF I = 99: PRINT 12
```

The first example simply prints the message "Time to go", if T is greater than 5. The second example either prints the values 10, 11 and 12 (if I is equal to 99), or prints the values 10 and 12 (if I is not equal to 99).

8.9.2 UNLESS STATEMENT-MODIFIER

The UNLESS statement modifier is similar to the IF statement modifier, but the statement is executed only if the specified condition is false. The form of the UNLESS statement-modifier is:

<statement> UNLESS <condition>

Example of an UNLESS Statement Modifier

```
495 GOTO 1300 UNLESS A = 10
```

8.9.3 FOR STATEMENT MODIFIER

The FOR statement modifier adds an iterative clause to a statement such that the statement is executed a number of times determined by the FOR clause. The form of the FOR statement modifier is:

<statement> FOR <var> = <exp1> TO <exp2> {STEP <exp3>}

Example of a FOR Statement Modifier

```
1580 A(I) = 0 FOR I = 1 TO 100
```

Note that the FOR statement modifier is not available in immediate mode.

8.9.4 WHILE STATEMENT MODIFIER

The WHILE statement modifier places a conditional qualifier on a statement, which indicates that the statement is executed repeatedly, WHILE a condition is true. The form of the WHILE statement modifier is:

<statement> WHILE <condition>

Example of a WHILE Statement Modifier

```
1210 LET T(I) = T(I) + FNF(I) WHILE I < 100
```

8.9.5 UNTIL STATEMENT MODIFIER

The UNTIL statement modifier is similar to the WHILE statement modifier, but the indicated statement is executed UNTIL a specified condition becomes true. The form of the UNTIL statement modifier is

```
<statement> UNTIL <condition>
```

Example of an UNTIL Statement Modifier

```
1210 LET T(I) = T(I) + FNF(I) UNTIL I >= 100
```

This example is equivalent to the example given for the WHILE statement modifier above.

8.10 MULTIPLE STATEMENT MODIFIERS

It is possible to qualify a given statement with more than one statement modifier. For example:

```
250 A = A + 1 IF A > 0 IF A < 100
```

This example is equivalent to either of the forms below:

```
250 IF A > 0 AND A < 100 THEN LET A = A + 1
```

```
250 IF A > 0 THEN IF A < 100 THEN A = A + 1
```

8.11 END

The END and STOP statements both terminate program execution.

The END statement is the last statement in a CS BASIC program. If an END statement is encountered in the normal flow of program execution (for instance, if the program "falls through" to the END statement), the program is terminated.

8.12 STOP

The STOP statement terminates execution of a program when it is executed. When a STOP statement is executed, a message is displayed to that effect. There can be multiple STOP statements throughout a program, with the (possibly conditional) flow of program execution determining when a STOP statement should be executed.

After a STOP statement has been executed, program execution can be started again with a CONT command, as described in Chapter 3.

8.13 CHAIN

The CHAIN statement finds application when a program is too big to load into memory all at once. If a program is too large, it can be split into independent programs. In any one of the programs, the CHAIN statement starts execution of another program. The form of the CHAIN statement is:

CHAIN <string> {LINE <exp>}

When the CHAIN statement is executed, CS BASIC loads, compiles, and starts executing the program specified by <string>.

If the expression <exp> is present, it specifies a line number at which the chained-to program is to start executing. If the line number is omitted, the chained-to program starts executing at the lowest numbered line, as if a RUN command had been issued to the system. If the specified line number does not exist in the CHAIN'ed-to program, a fatal run-time error is generated.

A CHAIN statement causes a complete replacement of the existing program with the chained-to program. Once a program issues a CHAIN statement, it never regains control, unless of course, another program CHAIN back to it.

The CHAIN statement close all open files in the current program before chaining to the target program. It should be noted, however, that partially filled buffers or modified virtual array elements may be lost. It is better to issue explicit CLOSE requests on open files before any CHAIN statements are issued. If several independent programs use the same files, they must explicitly OPEN the files in each program.

Note that all variables are reinitialized in the environment of a CHAIN'ed-to program. That is, numeric values are set to zero and string values are set to the null string.

Examples of CHAIN Statements

```
CHAIN 'PHASE.TWO'    ! Call up the next pass

CHAIN 'PAYABLES' LINE 1600
```

The first example passes control to the program called 'PHASE.TWO'. Execution of 'PHASE.TWO' starts at the lowest numbered line in that program, just as if a RUN command had been issued.

The second example starts execution of a program called 'PAYABLES', but in this case, execution starts at line 1600 in that program.

8.13.1 MERGE OPTION

If ",MERGE" is appended to a CHAIN statement, then the statements of the original program are not removed (unless the chained-to program uses the same line numbers) and all variables retain their values.

8.14 COMMON

The COMMON statement allows the user to pass variables and their values to another BASIC program when used in conjunction with a CHAIN command. The form of a COMMON statement is:

```
COMMON <var> {, <var>}...
```

where <var> is the name of either a simple variable or an array variable followed by left and right parenthesis '( )'. When a CHAIN statement is executed, the variables that have been listed in a COMMON statement and their current values are passed to the new program. The CHAIN command will have a new optional field ',ALL' which will override the chain command and cause all variables to be passed. Virtual arrays cannot be placed into a COMMON area and are never passed to a new program when the 'ALL' option is specified.

The implementation of the COMMON area, particularly with the ALL option, will require certain resources. If there is not enough memory available to build the necessary data structures, the CHAIN command will abort with an error message.

This is particularly applicable to the 'ALL' option, though if large amounts of data are passed and memory is almost full, it can happen to any CHAIN that involves COMMON.

Note that CHAIN with the MERGE option does a COMMON,ALL equivalent in all cases.

9.0 INPUT AND OUTPUT STATEMENTS

This chapter covers those statements in CS BASIC that perform input and output to and from files and devices in the system.

There are two major divisions of input-output in CS BASIC

- Data can be defined directly in the program itself, with the DATA statement. Data defined via DATA statements can be read with the READ statement, and "rewound" with the RESTORE statement.

- Data can be written to and read from external files. In this method, the external files can be used in one of three different ways. Two of these ways, plain ASCII data transfers and block input-output are described in this chapter. The other way is "virtual arrays", and is described in Chapter 10 -- "Matrix Operations".

9.1 READING DATA FROM WITHIN THE PROGRAM

The simplest form of data transfer capability in CS BASIC is supplied by the DATA, READ and RESTORE statements.

The DATA statement actually defines data in the body of the program itself. The READ statement reads items from the list of elements defined by the DATA statements. The RESTORE statement "rewinds" to the start of the list of data items.

9.1.1 DATA -- DEFINE DATA IN PROGRAM

The DATA statement defines data elements as part of the text of a BASIC Program. The data so defined can be read by a READ statement. The form of a DATA statement is:

```
DATA <value> {, <value> ...}
```

where any <value> can be integer, floating point, or string data values. String data items do not need to be enclosed in quotes, but if they are not, all spaces in the string are removed.

Note that while the objects defined in a DATA statement may have the appearance of integers or floating point numbers, it is a subsequent READ statement that actually determines whether the objects are interpreted as integers, floating point numbers, or strings. For example, the string of characters:

```
12345
```

while having the appearance of an integer, is actually a correct string, and can be read into a string variable, where it is stored as the literal characters "12345".

DATA statements can appear anywhere in a program, although common programming practice seems to lump them all together at the end of the program, where they do not clutter up the flow of the control statements. If DATA statements appear mixed in with executable statements in the program, they are skipped over.

Data defined in DATA statements is defined in the order of the statement numbers associated with the statements.

String data items must be written with quote signs surrounding them if they contain any of the following characters:

- a comma,
- significant spaces or tabs

Integer data values should not have a trailing % sign appended. A subsequent READ statement applied to such a data item will generate a run-time error.

A DATA statement must be the last statement of a multiple-statement line.

It is not possible to place comments after a DATA statement. Consider the DATA statement shown here:

```
5000 DATA 1, 2, 3     ! Define three numbers
```

In the above example, the DATA statement defines two numbers (1 and 2), but the last integer and the apparent comment is taken as the character string "3!Definethreenumbers".

Examples of DATA Statements

```
9000  DATA 1, 1, 3, 5, 8, 13
```

```
9010  DATA 123.45, Mince, 666, "Chicken Soup"
```

The first example defines a list of numeric values (that are also correctly formed character strings).

The second example defines a floating point value, a string data item that has the value "Mince", the numeric value 666, and finally the string "Chicken Soup" (enclosed in quotes to preserve significant spaces).

9.1.2 READ -- READ DATA FROM DATA LIST

The READ statement assigns values to variables. The values to be assigned are obtained from a list defined by DATA statements. The form of the READ statement is:

```
READ <variable> {, <variable> ...}
```

The list of <variables> consists of integer, floating point, or string variables. The variables can be simple variables or subscripted variables.

Data values read via a READ statement must conform in type with the variables to which they are assigned. That means that a numeric variable expects to read a numeric value, while a string variable expects a string value.

A string variable can accept any "value" from a data list, which looks like a correct string; a floating point variable will accept either a floating point value or an integer value; an integer variable must read an integer constant. For example, the string of characters:

```
12345
```

is either the literal character string "12345", the floating point number 12345.0, or the integer value 12345. Similarly, the string 1048576 is either the literal character string "1048576", or the floating point number 1.048576E6 (an attempt to read such a number as an integer would generate a run-time error). Therefore, it is actually the variables in the READ statement that determine the nature of the strings of characters in a DATA statement.

If a READ statement tries to read more data than defined in DATA statements, a run-time error is generated. The error can be trapped by the ON ERROR GOTO statement described in Chapter 12.

Example of READ Statements

```
200  READ F%(I%) FOR I% = 1 TO 6

1025 READ H, E$, K%, M$
```

9.1.3 RESTORE -- REPOSITION TO START OF DATA

The RESTORE statement restores the reading position to the lowest numbered DATA statement in the program. The form of the RESTORE statement is:

```
RESTORE
```

There are no arguments to the RESTORE statement. A RESTORE statement can appear in any position in a multiple-statement line.

The next READ statement to be executed in the program after execution of the RESTORE statement will start reading data from the first (lowest numbered) DATA statement in the program.

9.2 FILE INPUT AND OUTPUT

This section covers data transfers to and from external files. The two major divisions of data transfer described here are ASCII (formatted) input and output that are done with the INPUT and PRINT statements, and block input and output that are done with the GET, PUT, and FIELD statements.

CS BASIC performs data transfers to and from external storage devices via internal file descriptors. A file descriptor is an integer expression in the range 1% through 12%. That is, there may be a maximum of 12 files open at any one time. In addition, file descriptor 0% is the user's terminal.

The OPEN statement (described below) associates a file descriptor with a specific named file.

A file remains open until it is explicitly closed with a CLOSE statement, or until the program terminates, at which time the interpreter closes all open files.

At the time a file is first opened, the CS BASIC system does not know what that file will actually be used for. It might be used for plain ASCII input-output

using INPUT and PRINT statements, for block input-output using GET and PUT statements, or for virtual array storage.

The first time that a file is used for any one of these purpose, the system then designates that file as being used for that specific purpose. If an attempt is subsequently made to use the file for any other purpose, a data run-time error is generated.

9.2.1 OPEN -- OPEN A FILE FOR DATA TRANSFER

The OPEN statement opens a data file for transfer between the computer's memory and external storage. The form of the OPEN statement is:

```
OPEN <string> {FOR INPUT | FOR OUTPUT} AS FILE #<exp>
   {, RECORDSIZE <exp>} {, CLUSTERSIZE <exp>}
   {, FILESIZE <exp> {, MODE <exp>}
```

The basic function of the OPEN statement is to associate the file designated by the <string> immediately following the OPEN statement with the file descriptor designated by the <exp> following the AS FILE clause.

The optional FOR INPUT or FOR OUTPUT clauses do not actually restrict input-output on that file to input only or output only. The rules are as follows:

1. The FOR INPUT clause tries to open an existing file in the file system. If the designated file is not found, a run-time error is generated. This error can be trapped by the ON ERROR GOTO facility.

2. The FOR OUTPUT clause creates a new file if it did not exist prior to this OPEN statement. If the file did exist previously, it is re-created, and its previous contents are lost.

3. If neither the FOR INPUT nor FOR OUTPUT clause appear in the OPEN statement, the system tries to open the file for input as described in item 1 above. If this fails because the file does not exist, the OPEN statement executes as in item 2 above.

Designating a file as open FOR INPUT does not prevent the program from writing on that file. Similarly, a file designated as open FOR OUTPUT can be read from.

Simple Example of the OPEN Statement

```
200 OPEN "NUMBERS" FOR INPUT AS FILE #2
```

The options that follow the basic OPEN statement are to give the user finer control over the physical aspects of the file characteristics.

Options, if specified, must appear in the order shown. That is, RECORDSIZE, CLUSTERSIZE, FILESIZE, and MODE.

The RECORDSIZE option provides the means to specify the physical size of the buffer that the system uses for data transfer to and from the program.

The system normally uses records (blocks) of 512 bytes. The RECORDSIZE option can change this default size. The buffer size can be made larger than 512 bytes, but not smaller.

Example of the RECORDSIZE Option

```
200  OPEN "NUMBERS" FOR INPUT AS FILE #2, RECORDSIZE 1024
```

The CLUSTERSIZE, FILESIZE, and MODE options have no meaning in CS BASIC, but the interpreter accepts those options for compatibility with other implementations of BASIC.

9.2.2 CLOSE -- CLOSE A FILE

The CLOSE statement breaks the connection between a file and its internal file descriptor. The format of the CLOSE statement is

```
CLOSE <exp> {, <exp> ...}
```

Each <exp> is an integer expression referring to the number of an internal file descriptor of the file(s) to be closed. If there are any partially written buffers associated with that file, they are flushed (written to the file) before the file is closed.

If any <exp> is negative, the file designated by the absolute value of the expression is closed immediately. If there are any partially written buffers associated with that file, they are not flushed (written to the file) before the file is closed. This means that a CLOSE with a negative file number might lose some data.

Examples of the CLOSE Statement

```
8090 CLOSE 2, 3

8100 CLOSE 5, -8
```

The examples illustrate closing files 2, 3, 5 and 8. Since file number 8 was specified as a negative number, any data remaining in buffers for that file is lost.

9.3 SCREEN CONTROL

Two CS BASIC statements are provided to facilitate text output to the screen -- CLS and LOCATE.

9.3.1 CLS

CLS clears the display screen of all alphanumeric, graphic, and system information. The format of the command is:

9.3.2 LOCATE

LOCATE moves the display-screen cursor to the specified position. Information from the next PRINT statement will appear at this position. The format of the command is:

```
LOCATE (row, column)
```

The value of "row" ranges from 0 (top of screen) to 24 (bottom of screen), and of "column" from 0 (left of edge) to 79 (right edge).

9.4 PRINTING DATA

The PRINT and PRINT USING statements convert and print data. PRINT simply prints a list of variables on a specified destination, with default formatting rules determined by the system. PRINT USING prints data under control of a format string.

There are three floating point number representations that are out of the normal range. These are plus infinity, minus infinity, and Not a Number (NaN). When printed, plus infinity displays as a row of plus signs, minus infinity displays as a row of minus signs, and NaN displays as a row of question marks (see Appendix B).

9.4.1 PRINT -- PRINT ON FILE

The PRINT command is the simplest mechanism for displaying data at the terminal, or for printing to a file. The format of the PRINT command is:

PRINT {#<exp>,} <exp> {, <exp> ...}

If the optional #<exp> field is present in the PRINT statement, it refers to a file descriptor on which the list of values is to be printed. If the #<exp> is omitted, the display is directed to the user's terminal.

Each <exp> in the list of expressions can be a numeric or a string expression. Elements in the list of expressions are normally separated by commas, but they can alternatively be separated by semicolons. The different effects of the commas and semicolons are described in the discussion on print zones that follows.

The system divides each line of the terminal into a number of print zones, each print zone being 14 characters wide.

The behavior of the PRINT statement is as follows:

1. Each PRINT statement that does not have a comma or a semicolon at the end of the statement, completes printing on a given output line.

2. Leading zeros are suppressed, as are trailing zeros to the right of a decimal point.

3. At most six significant digits are displayed, unless more are requested via the PRINT USING statement described later.

4. Numeric values in the range -999999 through 999999 are printed in decimal format. Numbers outside of this range are printed in exponential format.

5. Numbers are always printed with a trailing space. Positive numbers have a leading space; negative numbers have a leading minus sign.

6. String values are printed verbatim, with no leading or trailing spaces.

7. Extra commas between print elements have the effect of skipping (or tabbing) print zones.

8. After a value is displayed, the system moves the print position to the start of the next available print zone if the value was followed by a comma.

9. A semicolon after an expression inhibits the movement of the print position to the next print zone, causing the displayed values to appear on the line in a packed fashion.

Example of PRINT Statement

```
760 LET Emp.Num% = 795
765 LET Emp.Name$ = 'Harry Bloggs'
770 LET Hour.Rate = 6.45
780 PRINT Emp.Num%, Emp.Name$, Hour.Rate
800 PRINT Emp.Num%; Emp.Name$; Hour.Rate
1000 END
Runnh
 795          Harry Bloggs   6.45
 795 Harry Bloggs 6.45
```

The two PRINT statements above illustrate the differences between printing data in a comma-separated list, and using semicolons to achieve the packed form of the display.

9.4.2 PRINT USING -- FORMATTED PRINTING

The PRINT USING statement provides for "formatted output", something like FORTRANS's formatted WRITE statement. The PRINT USING statement supplies a

"template" that controls the formatting of the list of variables that are to be printed. The format of a PRINT USING statement is as follows:

PRINT {#<exp>,} USING <string>, <exp> {,<exp> ...}

The optional #<exp> field, if specified, designates a file descriptor on which the printing is to take place. If the #<exp> field is omitted, printing is to the user's terminal.

The <string> field after the USING Keyword is a template that controls the layout of elements from the list of expressions following. The characters that can appear in the <string>, and their interpretations, are as follows:

! An exclamation mark in the format field denotes a single character field to appear in the output. For example:

```
PRINT USING '!!!', 'MNO', 'XYZ', 'PQR'
MXP
```

\\ Two backslashes denote a variable length string field of two or more characters. Two contiguous backslashes designates a field of two characters. If there are n spaces between the backslash characters, n+2 characters are printed. For example:

```
PRINT USING '\\ \  \', 'Alfred Bloggs', 'Maryann'
Al Mary
```

The number sign indicates positions at which decimal digits should appear in the output. A decimal point may appear at any position within a string of # signs (or the decimal point can be omitted altogether). When numbers are printed under this format, they are rounded appropriately.

Numbers are right justified in the specified field. If a field is too large to fit inthe space allotted, the system prints a % sign to indicate there is a problem, and the number is then printed without further reference to the format. If the number of characters is smaller than the allotted space, the space is filled with leading spaces, unless the ** format control described below is used. For example:

```
PRINT USING '######', 45.1
   45
PRINT USING '##.##', 45.168
45.17
PRINT USING '###', 5486
% 5486
```

If the value is to be displayed in exponential format, the use of four contiguous signs indicates the placement of the xponent.

```
PRINT USING '##.##     ####.##', 5745.98, 5745.98
57.46E 02  5745.98
```

** Two asterisks at the start of a numeric field designation indicates that unfilled positions in the output field should be filled with asterisks. The two asterisks serve a # signs as well as indicating asterisk fill.

Exponential format cannot be used in a field with leading asterisk fill. In this format, negative numbers must be output with a trailing minus sign.

```
PRINT USING '**##.##', 13.94, 430.70, 3681.00
**13.94
*430.70
3681.00
```

- If a numeric format field is terminated by a minus sign, the sign of the number is printed after the number instead of before it.

```
PRINT USING '###.##-  ###.##', -99.37, -99.37
 99.37-  -99.37
```

$$ If two $ signs are printed in front of a numeric field designation, a $ sign is printed in front of the number on the output. The two dollar signs, in addition to adding a $ sign in the output, serve as a single # sign in the format string.

Exponential format cannot be used with the leading dollar sign. Also, if negative numbers ar to be printed, the minus sign must follow the format.

```
PRINT USING '$$##.##', 37.40, 159.48, 2227.56
 $37.40
$159.48
% 2227.56
```

, A comma appearing in a numeric field format after a # sign and to the left of the decimal point indicates the normal conventions for placing commas in numbers, that is, every three digits. In this case, the comma acts like a # sign in every other way.

A comma appearing in a format to the right of a decimal point terminates the format field, and is printed as a literal character in the output.

```
PRINT USING '#######.###  ###,###.###  ###,###.#,##', &
```

```
                    987654.321, 987654.321, 987654.321
 987654.321  987,654.321  987,654.3,
```

Any character appearing in a PRINT USING string that is not listed above, are passed through to the output directly. For example:

```
PRINT USING 'xyz###<<<', 123
xyz123<<<
```

If a numeric field asks for more significant digits than there are available, trailing zeros are substituted for places after the last significant place. Floating point numbers can have up to 15 significant digits.

In the PRINT USING statement, a comma or a semicolon at the end of the line inhibits the system from printing an end of line at that position. Another PRINT or PRINT USING statement will then print the data on the same line.

If a PRINT USING statement reaches the end of a list of values, and there are no more format fields available in the USING string, the CS BASIC interpreter starts a new line on the output file, and starts using the USING string again from the beginning.

9.4.3 INPUT -- INPUT DATA FROM FILE

The INPUT statement reads data from an external storage device. The format of the INPUT statement is:

```
INPUT {#<exp>,} {<string>;} <var>
                        {{, <string>;} <var>...}
```

The optional #<exp> is the file descriptor of the file to read the data from. If the #<exp> is omitted, or if the #<exp> refers to file #0 (the user's terminal), the system prompts the user's terminal for the data.

The optional <string> parts interspersed in the list of variables represent messages that can be displayed at the user's terminal before prompting for data values. This prompting string is only displayed if the INPUT statement either does not indicate a file, or if the INPUT statement names file #0 (the user's terminal) as the source of input.

The way that the system interacts with the user's terminal is as follows:

- The system prompts with a ? sign if no <string> was used in the INPUT statement,

- The system prompts with a prompt of <string>? if a <string> was typed as part of the INPUT statement,
- A <string> in the INPUT statement that is terminated by a semicolon prints as is, with no ? sign,
- A <string> in the INPUT statement that is terminated by a comma moves the print position to the start of the next print zone,
- It is possible to prompt with several <string> values in a row.

The <var>'s in the INPUT statement definition above are a list of variables, separated by commas, into which the data elements are to be read. The system continues to prompt until sufficient values have been input.

String values can be typed either with or without quotes. Quotes are only necessary if the user wishes to embed commas or spaces in the string.

Example of the INPUT Statement

```
150 INPUT #1, List%(I) FOR I = 1 TO 10

250 INPUT 'Enter Year, Month, Day: ': Year%, Month%, Day%
```

9.4.4 INPUT LINE -- INPUT A STRING FROM A FILE

The INPUT LINE statement reads a line from a specified device into a character string variable. The form of the INPUT LINE statement is:

```
INPUT LINE {#<exp>,} <string variable>
```

A line is read from the file specified by #<exp>, or from the user's terminal if #<exp> is omitted.

Characters in the line are read and stored in <string variable>, without any interpretation, up to and including the first carriage-return.

The optional message display is not available in the INPUT LINE statement.

Example of INPUT LINE Statement

```
200 INPUT LINE #3, Line$
```

9.5 BLOCK INPUT AND OUTPUT STATEMENTS

Block Input and Output is provided for accessing specific blocks in a file in a random manner.

A file must be opened with the OPEN statement (described previously) before block input-output can be performed on that file.

A block input-output file has an input-output buffer associated with it. Data is transferred between block files and the buffer via GET and PUT statements, described below. Data is transferred between the user's program memory and the buffer by defining regions of the buffer with the FIELD statement, and moving data into those fields with the LSET and RSET statements, described in Chapter 7.

Block files are closed with the CLOSE statement described previously.

9.5.1 GET AND PUT -- READ OR WRITE DATA

The GET statement reads blocks from a file. The PUT statement writes blocks to a file. The format of the GET and PUT statements is:

```
GET #<exp1> {, RECORD <exp2> | BLOCK <exp2> }
                    {, COUNT <exp3>} {, USING <exp4>}
PUT #<exp1> {, RECORD <exp2> | BLOCK <exp2> }
                    {, COUNT <exp3>} {, USING <exp4>}
```

The #<exp1> is an expression designating the file descriptor of the file for data transfer.

The optional RECORD and BLOCK clauses are synonymous. They refer to the block number (<exp2>) at which reading or writing is to start. If the RECORD or BLOCK clause is omitted, reading or writing is performed on the next sequential block in the file. When the file is first opened, it is positioned at the first block (block 1).

In order to PUT a specific block, it is not necessary to write all the preceding blocks. In other words, there can be "holes" in the file.

Examples of the GET and PUT Statements

```
100 GET #1%, RECORD 24%

200 PUT #2%, RECORD 12%
```

9.5.1.1 The COUNT Option in GET and PUT

The COUNT clause in a GET statement defines the maximum number of characters to read, regardless of the size of the input-output buffer. For character oriented devices such as terminals, subsequent GET operations read remaining data from the device. For block oriented devices such as disks, data remaining in the block after COUNT characters have been read is discarded. A subsequent GET statement reads the next block from the device.

The COUNT clause in a PUT statement defines the number of characters to write i the current record. The expression (<exp3>) associated with the COUNT Option cannot be larger than the size of the input-output buffer. The COUNT option in PUT can only be used with non-file oriented devices.

Examples of the COUNT Option

```
100 GET #1%, RECORD 24%, COUNT 100%

200 PUT #2%, COUNT 240%
```

9.5.1.2 The USING Option in GET and PUT

The USING clause it he GET or PUT statement defines an offset into the input-output buffer associated with that file.

Examples of the USING Option

```
100 GET #1%, RECORD 24%, USING 128%
```

```
200 PUT #2%, RECORD 16%, USING 64%
```

9.5.2 FIELD -- SET BUFFER STRUCTURE

The FIELD statement associates string names with portions of an input output buffer. The FIELD statement can be thought of as defining a template for the layout of data in the buffer. The form of the FIELD statement is:

```
FIELD #<expr>, <expr> AS <stringvar>
              {, <expr> AS <stringvar> ...}
```

In the above FIELD statement, #<exp> is the internal file descriptor for the file under consideration.

Each <expr> AS <stringvar> clause defines the length (<expr>) of a string variable (<stringvar>) that is associated with a part of the buffer. The <stringvar> names are associated left to right with successive characters in the input output buffer.

FIELD statements do not perform any data movement. Instead, a FIELD statement directly associates a string variable name with an area in an input output buffer.

Example of FIELD Statement

```
FIELD #5%, 31% AS name$, 4% AS Age$, 11% AS Social.Sec
```

The above example associates three fields of the input output buffer of file number 5% with three string variables. The record might be something like an employee record for a company. The first field says there are 31 characters for the name; the second field has four characters for the person's age; the third field has 11 characters for the employee's social security number.

Data is moved into fields in a buffer with the LSET and RSET statements. The plain LET (assignment) statement cannot be used to move data, since LET actually creates new storage for the strings. LSET and RSET, on the other hand, replace data "in place". LSET and RSET are described in Chapter 5.

9.5.3 NOTES ON THE FIELD STATEMENT

The FIELD statement achieves its effect of associating a string name with a portion of the buffer when the FIELD statement is actually executed. In this regard, the FIELD statement should not be considered a static declaration. The association of string names with buffer positions is therefore dynamic.

This means that the association of a string name with a part of the buffer can be changed dynamically, simply by executing a different FIELD statement.

As noted earlier, data is moved into FIELD-defined strings by using the LSET and RSET statements. If a string defined in a FIELD statement is ever used as the target of a plain LET statement, a new string is created, and the association of that string name with a part of the buffer is lost.

9.6 INPUT AND OUTPUT STATUS DATA

The input-output system in CS BASIC maintains some essential information that enables the user to keep track of the status of files. This information is specified in the subsections below.

9.6.1 RECOUNT VARIABLE - NUMBER OF CHARACTERS READ

RECOUNT is a variable that contains the number of characters actually read on an input operation. RECOUNT can be used anywhere an integer value can be used.

9.6.2 BUFSIZ FUNCTION - DETERMINE BUFFER SIZE

BUFSIZ is a function that returns the buffer size of an open input-output channel. The form of the BUFSIZ function is:

```
BUFSIZ(I%)
```

where I% is the file descriptor number of an open file. BUFSIZ returns an integer value that represents the buffer size for that file.

If the file specified by I% is not open, BUFSIZ returns the value zero.

9.7 GRAPHICS CALLS

The Computer System features a graphics display superimposed over the user's display. The points, or "pixels," of this display are numbered from the lower left corner from 0 to 769 horizontally (x) and from 0 to 479 vertically (y). (Note that the origin of the user's console, row 0 column 0 is in the upper left corner).

Each graphics function has an optional "mode" specification as its last parameter. If mode is 0, the affected pixels are turned off (cleared); if mode is 1 (the default) pixels are turned on; and if mode is 2, the state of each pixel is complemented.

PSET (x,y{,mode})

Set the single point at (x,y).

LINE (x1,y1,x2,y2 {,mode})

Draw a line from (x,y) to (x ,y). If x is negative, then draw a line from the last pixel position to (x ,y).

FILL (x1,y1,x2,y2{,mode})

Fill in the rectangular area where (x1,y1) is one corner and (x2,y2) is the corner diagonally opposite. If x is negative, then one corner is the last pixel position affected.

ELIPSE (x1,y1,x2,y2{,mode})

Draw an ellipse with center at (x1,y1) and x-radius of {(x2-x1){ and y-radius of {(y2-y1){. If x is negative, then the center is the last pixel affected by a previous graphics statement. If the y-radius is 1.2 times the x-radius, the ellipse will approximate a circle.

TEXT (x1,y1 string {,size {,orientation {,mode}})

Draw a string of characters beginning at (x1,y1). Character size ranges from 1 to 8, where 1 is the normal character size (default), 2 is double, 3 is 4 times as large, 4 is 8 times as large, and so on. Orientation is 0 to 3, where 0 is left to right (default); 1 is bottom to top; 2 is right to left (upside-down); and 3 is top to bottom. Size, direction, and mode may be omitted and the defaults used; 2 mode alone may be omitted or mode and direction may both be omitted. If y is negative, the string will begin at the pixel last affected by a previous graphics command.

10.0 MATRIX OPERATIONS

CS BASIC has facilities for matrix manipulation. A matrix is declared as a dimensioned variable, as described in Chapter 2. A matrix can have one or two dimensions.

This chapter covers two main areas concerned with matrices. First, there is a discussion on the statements for manipulating matrices. Second, there is a description of virtual arrays, whereby matrices can be stored in the disk storage system.

There are facilities for initializing a matrix to specific values (such as the identity). The MAT statements provide for adding, subtracting, multiplying, and inverting matrices, in addition to reading data into a matrix, and printing a matrix.

The rules for declaring matrices and virtual arrays appear in Chapter 3 -- "Elements of the BASIC Language". The next two sections below discuss the details of how arrays are dimensioned and redimensioned.

10.1 HOW ARRAY VARIABLES ARE DIMENSIONED

This section describes the ways in that the CS BASIC system determines the shape and size of dimensioned variables. It is important that the user understand this process.

An array variable or matrix is given its shape (whether one-dimensional or two-dimensional) either explicitly or implicitly. The shape of the array is determined contextually. Once the shape of an array has been determined, it must be used consistently thereafter. Any attempt to change the shape of an array results in a fatal run-time error.

The size of an array (its dimensions) are either assigned explicitly by declaring the variable in a DIM statement, as described in Chapter 2, or the size is set to the default size by the CS BASIC interpreter. The default sizes for arrays corresponds to an implicit declaration of

```
DIM A(10)
```

for a one-dimensional array, and to an implicit declaration of

```
DIM A(10, 10)
```

for a two-dimensional array. This means that the default sizes are 11 elements (for a one-dimensional array) or 121 elements (for a two-dimensional array). Arrays always have a zero'th element (one-dimensional) or a zero'th row and column (two-dimensional).

Storage for an array is allocated the first time that any portion of the program (or any immediate mode statement) is executed. At such a time, the CS BASIC interpreter performs a pre-pass to allocate storage for all array variables. Once the dimensions for an array are fixed, that is the maximum number of elements tha the array can ever have. But an array can be redimensioned to the same or a smaller size, and this is explained below in the section on array redimensioning.

To explain this idea, the statement:

```
120 DIM A(100)
```

declares **A** as an array with 101 elements. However, if the above DIM statement had never appeared in the program, then the statement:

```
500 MAT A = IDN(50, 50)
```

would have (implicitly) declared **A** as a 10 by 10 array, and the statement above, which would otherwise redimension the array to a size larger than allocated, will generate a run-time error.

Furthermore, if neither of the above statements had ever been executed in the program, the statement:

```
200 MAT A = CON
```

declares A as a single dimensioned array with its upper bound equal to 10, containing 11 elements, and initialized to all ones (1's).

10.2 REDIMENSIONING A MATRIX

As discussed in the previous section, a matrix initially starts with a specific number of elements in it. The number of elements is either explicitly determined by dimension information provided by the user, or default dimensions are provided by the system.

Once the size of a matrix if fixed by the system, that matrix can never grow any bigger (it has a fixed amount of storage allocated for it). The matrix can, however, be made smaller, and can also change its shape, as long as the total number of elements never exceeds the maximum. This process is called redimensioning a matrix.

In no case can redimensioning change the shape of an array. That is, it is not possible to change an array from one-dimensional to two-dimensional or vice versa.

For example, suppose that the array **A** were declared like this:

```
100 DIM A(20, 20)
```

This declares **A** as an array with 441 elements. Now, the statement:

```
MAT A = ZER(12, 16)
```

redimensions **A** as a 12 by 16 array (containing 221 elements), while initializing **A**. It is possible that future statements can change the size yet again. For example, the statement:

```
200 MAT A = IDN(20, 20)
```

sets A to a square identity matrix, having the same number of elements as it did originally.

However, consider this example:

```
100 DIM C(12, 12), D(14, 14)
120 MAT C = D
```

This example is wrong, because the matrix assignment statement at line 120 is attempting to increase the total number of elements in the **C** matrix. This would generate a run-time error. Note that the converse assignment:

```
120 MAT D=C
```

is correct, because a smaller array is being assigned to a larger array.

It is not possible to change the shape of an array from one-dimensional to two-dimensional or vice versa. For example:

```
100 DIM A(5, 5), B(4)
120 MAT A = B + B
```

The above example is incorrect, since the statement at line 120 is attempting to change the array A from two-dimensional to one-dimensional.

10.3 INITIALIZING A MATRIX

CS BASIC provides for initializing a matrix in one of three ways. The form of the matrix initialization statement is:

```
MAT <matrix name> = ZER | CON | IDN {(<subscripts>)}
```

<matrix name> is the name of the matrix to be initialized. The optional <subscripts>, if specified, have the effect of redimensioning the matrix. If the optional <subscripts> are omitted, the dimensions of the matrix are unchanged.

The keywords ZER, CON or IDN have these meanings:

ZER initializes the matrix to zeros. The result is called a zero matrix or a null matrix. Note that when a matrix is first created, it is initialized to all zeros. However, the MAT ZER is useful for initializing virtual arrays that are not initialized when they are created.

CON initializes the matrix to ones.

IDN initializes the matrix as an identity matrix (ones along the principal diagonal, zeros elsewhere).

Examples of Matrix Initialization

```
100 DIM line(5), checker.board(8, 8), diag(5, 5)
110 REM
120 MAT line = ZER
130 REM
140 MAT checker.board = CON
150 REM
160 MAT diag = IDN(4, 4)
```

The statement at line 120 initializes all elements of the array variable 'line' to zero. The statement at line 140 initializes the entire 8-by-8 matrix called 'checker.board' to all ones. The statement at line 160 initializes the 'diag' matrix as an identity matrix, while redimensioning that matrix to be a 4-by-4 matrix.

Note that an identity matrix need not be a square matrix. The action of IDN is to place a 1 in those positions of the matrix where the row and column subscript are equal, and 0 elsewhere inthe matrix.

The MAT statements do not have any effect on row 0 and column 0 of a matrix.

10.4 MATRIX INPUT AND OUTPUT

CS BASIC provides three statements for input and output operations on whole matrices. The MAT READ statement reads data into a matrix from data defined in a DATA statement. The MAT PRINT statement prints the elements of a matrix. The MAT INPUT statement reads elements of a matrix from external storage devices. These statements are described in the following subsections.

Note that the MAT READ and MAT INPUT statements can redimension a matrix.

10.4.1 MAT READ -- READ MATRIX ELEMENTS FROM DATA

The MAT READ statement reads data defined in a DATA statement, and uses that data to initialize the elements of a matrix. The form of the MAT READ statement is:

```
MAT READ <matrix>{(<subscripts>)}
         {, <matrix>{(<subscripts>)} ...}
```

Each element in the list of matrixes is the name of a matrix, optionally followed by <subscripts>.

If a matrix name is specified without any <subscripts>, the entire matrix is read.

If a matrix name is followed by dimension information, the matrix is redimensioned. If there is not enough data to fill the matrix as dimensioned, a run-time error is generated.

The zero'th row and column of a matrix do not participate in a MAT READ statement.

10.4.2 MAT PRINT -- PRINT MATRIX ELEMENTS

The MAT PRINT statement prints elements from a matrix. The form of the MAT PRINT statement is:

```
MAT PRINT {#<exp>.} <matrix>{(<subscripts>)} { , | ; }
```

where <matrix> is the name of the matrix that is to be printed.

If the name of the matrix has no dimension information following, the whole matrix is printed. If dimensions are specified, only those elements are printed. The MAT PRINT operation does not redimension the matrix, even if only a subset of the matrix is actually printed.

The optional semicolon character ; specifies that the elements should be printed in a packed format. The command character , means that the elements are printed across the line, each element in a separate print zone.

If neither a semicolon nor a comma follows the name of the matrix, each element is printed on a separate line.

The zero'th row and column of a matrix do not participate in a MAT PRINT statement.

10.4.3 MAT INPUT -- READ MATRIX ELEMENTS FROM EXTERNAL STORAGE

The MAT INPUT reads matrix elements from the user's terminal or an external file. The form of the MAT INPUT statement is:

```
MAT INPUT {#<exp>,} <matrix>{(<subscripts>)}
                    {, <matrix>{(<subscripts>)}}
```

If the optional #<exp> is present, it refers to a file descriptor from which the data destined for the list of matrixes is to be read. If the #<exp> is omitted, data is read from the user's terminal. The MAT INPUT statement can redimension the target matrix.

Note that only elements from a single line are read in a MAT INPUT statement. Remaining elements in the matrix are left unchanged.

The zero'th row and column of a matrix do not participate in a MAT INPUT statement.

10.4.4 STATUS VARIABLES FOR MAT INPUT

There are two status functions connected with matrix input statements.

The NUM function returns an integer value that is the number of elements input for a one-dimensional matrix, and the number of rows input for a two-dimensional matrix.

The NUM2 function returns an integer value that is the number of elements input in the last row of a two-dimensional matrix.

10.5 MATRIX ARITHMETIC OPERATIONS

There are five arithmetical operations that can be performed on matrices, plus three built-in functions.

The arithmetic operations are:

- matrix assignment,
- addition and subtraction,
- scalar multiplication,
- multiplication of conforming matrices. P.In all matrix arithmetic operations of the form:

  MAT **C** = **A** <op> **B**

where <op> is **+**, **-**, or *****, the matrix on the left of the equals sign is redimensioned to conform to the dimensions of the result of the operation.

Matrix operations can only be done one at a time. That is, it is not possible to do:

```
MAT A = B + C + D
```

The zero'th row and column of a matrix do not participate in matrix operations.

10.5.1 MATRIX ASSIGNMENT

The matrix assignment operation assigns the value of one entire matrix to another. The form of the statement is:

```
MAT A = B
```

where A and B are matrices. The matrix assignment statement redimensions the target matrix if necessary.

10.5.2 ADDITION AND SUBTRACTION OF MATRICES

Matrix addition and subtraction is done with the familiar + and - operators in MAT statements. For example:

```
MAT A = B + C
```

is a matrix addition statement.

Addition or subtraction may only be performed between matrices of equal dimensions.

When matrices are added or subtracted, the result is the element-by-element addition or subtraction of the operands.

10.5.3 SCALAR MULTIPLICATION OF MATRICES

Scalar multiplication of a matrix involves multiplying each element of the matrix by a scalar constant, scalar variable, or scalar expression.

The form of scalar multiplication is:

```
MAT Y = (K) * X
```

where Y and X are matrices (they can be the same matrix), and K is a scalar constant, scalar variable, or scalar expression. The syntax indicates that K must be enclosed in parentheses, even if it is a simple scalar constant or scalar variable.

Example of Scalar Multiplication

```
listnh

100 DIM Triangle(5, 5)   ! declare the matrix
110 MAT Triangle = IDN   ! make it an identity matrix
120 MAT Triangle = (4) * Triangle    ! Scalar multiply
130 MAT PRINT Triangle;     !Print it
140 END

Ready

runnh

4 0 0 0 0
0 4 0 0 0
0 0 4 0 0
0 0 0 4 0
0 0 0 0 4

Ready
```

The example shows the matrix called 'Triangle', first made into an identity matrix. Each element is multiplied by 4, then the matrix is printed out.

10.5.4 MULTIPLICATION OF CONFORMING MATRICES

Two matrices can be multiplied together using the standard * operator. In order for matrices to be multiplied, they must be conforming. That is, in the multiplication

```
MAT C = A * B
```

the number of columns in **A** must be equal to the number of rows in **B**.

If the number of columns in **A** is not equal to the number of rows in **B**, a run-time error is generated, which can be trapped by the ON ERROR GOTO facility.

The definition of multiplication in this case is as follows. Assume that there are three matrices:

A is a m by n matrix whose elements are addressed a

B is a n by p matrix whose elements are addressed b

The result of the multiplication

```
MAT C = A * B
```

is a matrix **C**, which is a m by p matrix, and whose individual elements consist of the sum of the products:

```
c   = SUM(a   * b   ) FOR I = 1 TO n
```

Note that if the **C** matrix is a virtual array, it must be distinct from both the **A** and **B** matrices.

10.6 MATRIX FUNCTIONS

CS BASIC supplies three functions for operating upon matrices. The built-in functions are:

TRN - Take the transpose of a matrix,
INV - Invert a matrix,
DET - Find the determinant of a matrix.

10.6.1 TRN -- TRANSPOSE A MATRIX

The TRN function computes the transpose of a matrix, meaning that the rows and columns of the matrix are interchanged. The target of the transpose function is redimensioned if necessary.

Consider the following example, in which the matrix called 'Portrait' is already initialized as shown in the output from the MAT PRINT statement:

```
listnh

100 DIM Portrait(3, 6), Landscape(6, 3)
105 PRINT "portrait is:"
110 MAT PRINT Portrait;
120 MAT Landscape = TRN(Portrait)
125 PRINT "Landscape is:"
130 MAT PRINT Landscape;
140 END
```

```
Ready

RUNNH
Portrait is:
1 2 3
4 5 6
7 8 9
10 11 12
13 14 15
16 17 18
Landscape is:
1 4 7 10 13 16
2 5 8 11 14 17
3 6 9 12 15 18
Ready
```

Note that in the statement:

```
MAT A = TRN(B)
```

matrix **A** must be distinct from matrix **B** if matrix **A** is a virtual array.

10.6.2 INV -- INVERT A MATRIX

The INV function computes the inverse of a matrix. The matrix to be inverted must be a square matrix.

If the matrix is not invertible, meaning its determinant is zero, a run-time error is generated. The error can be trapped by the ON ERROR GOTO facility if required.

10.6.3 DET -- FIND THE DETERMINANT OF A MATRIX

The DET function returns the determinant of the matrix whose name appeared in the most recently executed INV function.

If the matrix is singular, the most recent INV function will have generated a run-time error. If the error has been trapped via the ON ERROR GOTO facility, the DET function can be used to determine this.

10.7 VIRTUAL ARRAYS

The virtual array facility of CS BASIC provides a means to address data randomly, and to hold very large amounts of data in the form of arrays. Such arrays look like regular arrays to the programmer, but in reality they are stored in the disk system. The data in virtual arrays is stored as unformatted binary data. Thus there is no conversion performed when reading or writing that data.

Two things must be done when using virtual arrays:

- The variables belonging to the virtual arrays must be declared, and also associated with a file descriptor number in a DIM statement.
- The file associated with those arrays must be opened.

The order of these operations is not important.

Unlike matrices stored in memory, virtual arrays cannot be redimensioned.

10.7.1 DECLARING A VIRTUAL ARRAY

Virtual arrays are declared with a variation of the basic DIM statement, as described in Chapter 3 -- "Elements of the BASIC Language".

10.7.2 OPENING AND CLOSING VIRTUAL ARRAY FILES

Virtual array files are opened and closed with the normal OPEN and CLOSE statements described in Chapter 9. The form of the OPEN statement is:

```
OPEN <string> {FOR INPUT | FOR OUTPUT}
                              AS FILE #<expression>
```

where <string> is the external name of the file, and #<expression> is the number of the file descriptor associated with that name.

The other parts of the OPEN statement have their usual meaning. The RECORDSIZE, BLOCKSIZE, CLUSTERSIZE, and MODE options are preset for compatibility with other BASIC versions.

When a virtual array is opened, the system does not initialize the elements of the array. It contains whatever data was previously written on the appropriate parts of the disk file. If the user desires that an array should be initialized, it should be done explicitly. The MAT ZER, MAT IDN, and MAT CON statements are convenient for this purpose.

When assigning values to string virtual array elements, somewhat different rules apply from the normal actions of the assignment statement for string values. This is discussed in Chapter 7 -- "Assignment Statements".

11.0 EXTERNAL LINKAGES

External routines are called as statements in CS BASIC in the following manner:

CALL <externalname> { ({arg { , arg ... }) }

Parameters may be integer, real, or string variables, array elements, arrays, and expressions. Virtual arrays and virtual array elements cannot be passed as parameters. Pointers to the parameters are passed on the stack (call by reference). In the case where any array is being passed, the value passed is the address of the zero'th element of the array -- (A(0) for the one-dimensional case, and A(0,0) for the two-dimensional case). The syntax of an array parameter is the name of the array followed by pair of parentheses. Pointers to the parameters are passed on the stack. Pointers to integer (real) parameters point to 2-byte (8 byte) values. Pointers to string parameters point to a string descriptor. The string descriptor is a 4-byte pointer to the string value followed by a two-byte integer representing the length of the string. The external routine may change a string value in place only if the resulting string is no longer than the string value passed to it (not enforced by CS BASIC, but potentially disasterous if violated).

The name of the external routine is the name of a real variable that has been initialized to the absolute machine address of the subroutine to be called.

An additional, unnamed parameter consisting of a long word containing the number of bytes of parameters actually passed is passed as the last parameter. This count does not include the count itself or the return address.

Missing parameters may be indicated by leaving the corresponding value out of the parameter list, such as CALL PROC(A,,C). The value NIL (zero) will be pushed to indicate missing parameters.

When a CALL statement is made the interpreter will check whether there are at least 2K bytes of stack space remaining. If not, a run-time error will be generated.

CS BASIC implements a call by pushing pointers to the arguments and jump subroutine to the address specified. The called procedure must return to BASIC with the parameters popped off the stack. The interpreter expects a word length value on the stack that represents an error code. If the value is 0, no error is indicated. If the value is nonzero, then that value is the "BASIC error number," and BASIC's usual run-time error path is taken (including ON ERROR GOTO). Thus, the BASIC statement

```
CALL X(A%,B%,C%)
```

might be handled by a Pascal declaration of the form:

```
FUNCTION X(UAR A,B,C:INTEGER;COUNT:LONGINT):INTEGER;
```

or the equivalent machine language code.

PEEK(<floating point expression>) is an integer function that returns the unsigned byte (0..255) at the absolute location that is the truncated value of the <floating point expression>. The floating point expression is converted to a four-byte integer that is used as the address. Floating point values that cannot be represented in a signed four-byte integer representation have undefined conversions to four-byte integers.

POKE (<floating point expression> , <integer expression>) is a BASIC statement that places the least significant byte of the integer expression value into the location specified by the <floating point expression>. The address is computed in the same manner as for PEEK.

12.0 CS BASIC PROGRAM STRUCTURE

This chapter describes the overall structure and layout of a program written in CS BASIC, how subroutines are called, how functions are defined and referenced, and how error conditions can be trapped and handled by the user.

There are four kinds of program constructs that may be executed "out of line" in a program:

1. GOSUB and RETURN provide the familiar subroutine capability. Subroutines in BASIC do not have parameters as in other languages. If the caller of a subroutine wishes to pass parameters to the subroutine, they must be set up in ordinary variables, which are then global to the entire program.

2. Single-line functions, which can have parameters. Single-line functions are defined with the DEF (or DEF*) statement, and their definition consists of a single line expression that is applied to the parameters (if any).

3. Multiline functions are defined via the DEF (or DEF*) statement, and consist of BASIC statements up to and including a FNEND statement that marks the end of the function. A RETURN statement encountered in a multiline function returns control (and the value of the function) to the expression that referenced the function.

4. The ON ERROR GOTO and RESUME constructs, which provides for trapping errors (such as divide by zero) in a program and taking whatever corrective actions the programmer desires.

In principle, a CS BASIC program is simply a collection of statements. For instance, the existence of a subroutine is determined solely by a GOSUB statement making a call to a certain label in the program. The subroutine itself does not impose a structure on the program. A subsequent RETURN statement then returns control to the place in the program from which the subroutine was called.

At another time, however, the collection of statements in the "subroutine" could simply be executed by just GOTO'ing into that body of statements, and then the way out of that "subroutine" might be by another GOTO. In this case, a RETURN statement would cause a return from the currently active subroutine, or a run-time error if there was no currently active subroutine.

Similarly, it is possible to jump into the middle of a function and execute some of the statements in it. In this case, an FNEND statement would cause a return from the currently active function, or a run-time error if there was no currently active function.

As such, statements within subroutines and functions are unrestricted in their flow of control. It is possible to go "outside" the body of a function to somewhere else, assign to the function from outside it, and RETURN from the subroutine or function from elsewhere.

There is of course some structure in a CS BASIC program. Multiline functions must have their DEF and FNEND statements correctly matched. The execution of FOR-NEXT, WHILE-NEXT and UNTIL-NEXT loops must be correctly nested.

12.1 CORRECT NESTING OF SUBROUTINES AND FUNCTIONS

The calling of subroutines and functions can be nested. That is, a subroutine can call other subroutines and can reference functions. Similarly, functions can reference other functions, and call subroutines.

ON ERROR processing can also call subroutines and can reference functions.

In all these constructs, execution must be properly nested.That is, a function that has been referenced from within a subroutine cannot exit the subroutine by executing a RETURN statement. The function must exit first by executing its FNEND statement, which returns control to the subroutine that called it. That subroutine can then exit correctly by executing a RETURN statement.

Similarly, a subroutine that has been called from within a function cannot exit the function by executing an FNEND statement. The subroutine must first exit by executing a RETURN statement, then the calling function can exit correctly by executing its FNEND statement.

Lastly, subroutines or functions called from within an ON ERROR processing routine cannot exit that routine by executing a RESUME statement. All such subroutines and functions must exit correctly and in the right order before any RESUME statement can be executed to exit from the ON ERROR processing routine.

12.2 SUBROUTINES

CS BASIC supplies the facility for subroutines: bodies of code that are executed out of line. The subroutine facility provides the means of constructing a large program from building blocks.

A subroutine looks just like any other sequence of code in a program. The code body is entered when a GOSUB statement references the starting line number of the body of code. The subroutine is exited when a RETURN statement returns control to the part of the program that called the subroutine.

12.2.1 THE GOSUB STATEMENT -- CALLING A SUBROUTINE

The simplest means of calling a subroutine is by means of the GOSUB statement. The format of the GOSUB statement is:

```
GOSUB <line number>
```

Control is transferred to the line specified by <line number>. That line is then the first line of the subroutine. When that subroutine executes a RETURN statement, program execution resumes at the statement after the GOSUB that called the subroutine.

Subroutines can be nested to an arbitrary depth, that is, one subroutine can call another. A subroutine can call itself.

Example of a GOSUB Statement

```
200 GOSUB 1000
210 . . .
          statements
                    . . .
1000 . . .
          statements comprising the
          body of the subroutine
                                  . . .
1155 RETURN
```

In the example above, the GOSUB statement at line 200 calls the subroutine whose first statement starts at line 1000. Execution of the subroutine's code continues until the RETURN statement is executed. At that time, control returns to the statement after the GOSUB statement.

12.2.2 THE ON GOSUB STATEMENT

The ON GOSUB statement supplies a multi-way transfer of control to a subroutine, depending on the value of an expression. The form of the ON GOSUB statement is:

ON <exp> GOSUB <line number> {, <line number> ...}

The expression given by <exp> is used to select one line number out of the list of line numbers specified. Indexing of the list of line numbers starts at one (1). The value of the expression <exp> is automatically converted to integer for the purposes of selecting from the list of line numbers. If the value of <exp> is less than one, or greater than the number of line numbers in the list, a run-time error is generated.

Example of an ON GOSUB Statement

```
ON I GOSUB  2000, 2100, 2200, 2400, 2700
```

12.2.3 RETURN -- RETURNING FROM A SUBROUTINE

The RETURN statement returns from executing a subroutine, and program execution resumes at the statement following the GOSUB that called the subroutine. The format of a RETURN statement is simply:

```
RETURN
```

If a RETURN statement is encountered in a program when no previous GOSUB statement has been executed, a run-time error is generated.

12.3 FUNCTIONS

Functions in CS BASIC are analogous to mathematical functions. A function can be referenced as a part of an expression. A function can take arguments, which it uses to compute a value. The value of the function effectively replaces its reference in an expression.

The DEF and DEF* statements define functions. That is, the actions that are to be performed on the functions arguments (if any) are defined.

Functions cannot be defined in immediate mode. Functions can, however, be referenced in immediate mode.

A function is later referenced when its name and arguments (if any) appear in an expression. Function invocations can be nested, and functions can call themselves.

12.3.1 DEF AND DEF* STATEMENTS -- DEFINING FUNCTIONS

The DEF and DEF* statements are used to define functions. The two different DEF statements are there for historical compatibility. From here on, the DEF* form is used.

There are two forms of function definitions, namely single-line functions and multiple-line functions. The format of a single-line function definition is:

```
DEF* FN<var>{(<arg> {, <arg> ...})} = <exp>
```

The format of a multiple-line function definition is:

```
DEF* FN<var> {(<arg> {, <arg> ...})}  ! defines function
{<statements>}                        ! any number of <statements>
{FN<var> = <exp>}                     ! any number of these
FNEND                                 ! indicates end of function
```

In both forms, the name of the function is a variable identifier (floating, integer, or string), preceded by the letters **FN** to indicate that this is a function name.

The function name is followed by an optional list of arguments. There may be zero to five arguments to a function. In the function definition, the arguments are dummy arguments.

In the single-line version of the definition, the expression immediately following the equals sign designates the computation that is to be done and returned as the value of the function.

In the multiple-line version, the statements in the body of the function, up to the FNEND statement, specify the actions that the function must perform. At some point in the body of a multi-line function, the function is assigned a value via the

```
FN<var> = <exp>
```

line as shown in the definition above. There can be more than one such assignment to the function name. The value of the function is the last value assigned to the function name.

A function is exited when the interpreter executes the FNEND statement.

If a function is exited before any value has been assigned to the function, the value of that function is undefined in the expression that referenced the function, and hence the value of such an expression is unpredictable.

The definition of a function cannot be nested inside the definition of another function. That is, the structure:

```
100 DEF* FNA
      <statements for function A>
200   DEF* FNB
       <statements for function B>
300  FNEND  ! end of B
400 FNEND  ! end of A
```

is an incorrect definition of functions.

Example of a Single Line Function Definitions

```
200  DEF*  FNDiscriminant(A, B, C) = B 2 - 4 * A * C
```

Example of Multi LIne Function Definition

```
430 DEF*  FNLargest.Int%(A%, B%, C%)
440     FNLargest.Int% = A%
450     IF B% > FNLargest.Int% THEN FNLargest.Int% = B%
455     IF C% > FNLargest.Int% THEN FNLargest.Int% = C%
460  FNEND
```

12.3.2 THE FNEND STATEMENT

The FNEND statement indicates the end of a multiple-line function definition. The form fo the FNEND statement is simply:

```
FNEND
```

12.3.3 REFERENCING FUNCTIONS

A function is referenced by stating its name and arguments in an expression. If a function (say, called FNParty) does not have any arguments, it can be referenced either by just stating its name:

```
FNParty
```

or by stating its name followed by empty parentheses:

```
FNParty()
```

Example of a Function Reference

```
100  PRINT FNLargest.Int(-5%, 10%, 100)
```

When the above program is run, it prints the value 100 that the function computed as the largest number among the actual arguments.

12.3.4 PASSING ARGUMENTS TO FUNCTIONS

When a function is referenced, the actual arguments in the reference are assigned to the dummy arguments. The assignment of the actual arguments to the dummy arguments is exactly as if a

```
LET <dummy> = <actual>
```

had been performed.

The function's dummy arguments must agree in number with the actual arguments. If the dummy arguments of a function are string arguments, the actual arguments must also be strings. If the dummy arguments of a function are numeric arguments, the actual arguments must also be numeric. The types of numeric arguments are converted if necessary (and if possible) on entry to the function. For example, consider the following function:

```
1000 DEF* FNM(I, J%, S$)
    <statements>
1050 FNEND
```

when FNM is referenced, its third argument must be a string. However, its first and second arguments may be either floating point or integer:

```
FNM(10.0, 10%, 'ABC')

     or

FNM(10%, 10.0, 'ABC')
```

Either of the above two forms is acceptable. Of course, if the floating point value assigned to the second argument is outside the allowable range for integers, the results would be undefined.

Arguments to functions are passed by value. This means that the function receives a copy of the actual arguments. The function can then alter the dummy arguments, without any effect on the actual values in the program that called the function. The following example illustrates the effects of call by value.

```
100 LET I = 10
110 LET J = 20
130 LET Result = FNF(I,J)
140 PRINT I, J
190 REM
1000 DEF* FNF(I, K)
1010 PRINT I, J
1020 LET I = 13
1030 LET K = 14
1040 LET FNF = 99
1050 FNEND
9000 END
Runnh
 10             20
 10              20
Ready
```

When the function FNF is referenced, the PRINT statement at line 1010 prints the value of I (local to FNF) as 10 (the value it was called with), and the value of J (global to the program as 20 - the value assigned in the program). Then the assignments to the local variables I and K in the function FNF do not change the values of the associated variables I and J in the program so the subsequently executed PRINT statement in the program prints the same values, because the variables I and J did not change. fuctions/arguments/scope of

12.3.5 SCOPE OF FUNCTION ARGUMENTS

Once a function's dummy variables are defined, they remain defined, local to the function, until the function is exited through the FNEND statement. Statements "outside" the body of the function can be executed, where any references to the function's dummy arguments access those arguments, not variables defined elsewhere in the program.

To illustrate this point, consider the function:

```
900 A= 99
910 Y = FNF(3, 4)
1000 DEF* FNF(A. B)
1020 LET A = 13
1030 LET B = 14
1035 GOTO 2100
1040 LET FNF = 99
1050 FNEND
2000 REM
2100 X = SQR(A) >br 2110 GOTO 1040
9000 END
```

In the example above, the first time that statement 2100 is executed, it is in the function (because of the assignment statement at line 910), and the variable A refers to the formal parameters of the function FNF.

The second time that statement 2100 is executed, A refers to the global variable A that is defined at line 900.

Note that after the statement at line 910 is executed, the interpreter skips over the function definition.

12.4 ERROR HANDLING

The CS BASIC language system detects errors generated during execution of a program. Errors fall into two major classes:

1. Computational errors such as dividing by zero, taking the square root of a negative number, and so on,

2. Input-Output errors such as reading from a file that is not opened. In this class, end-of-file also appears as an error, although strictly speaking, end-of-file is a normal occurrence.

Under normal circumstances, when CS BASIC detects a program error, it prints an error message, and terminates the program.

It is possible, however, to set up the program such that a specific body of code is called when errors occur. The programmer can thus write a subroutine to determine the cause of error, and to attempt recovery.

12.4.1 THE ON ERROR GOTO STATEMENT

The ON ERROR GOTO statement caters to a user's code handling errors. The basic form of the ON ERROR GOTO statement is as follows:

```
ON ERROR GOTO {<line number>}
```

The <line number> that is the destination of the GOTO is optional. If the <line number> is present, and non-zero, it specifies the line number of the statement to which control is to be passed when a run-time error occurs.

The ON ERROR GOTO statement must be executed before any statements that could generate errors.

If the <line number> is omitted, or if it is specified as 0, the effect is to disable the error routine.

Examples of ON ERROR GOTO Statements

```
150 ON ERROR GOTO 9500

200 ON ERROR GOTO  ! disables the error routine

210 ON ERROR GOTO 0 ! also disables error routine
```

12.4.2 THE ERR AND ERL VARIABLES

When an error occurs in a CS BASIC program, the interpreter sets up two variables that can assist in handling the error.

The ERR variable contains the error number associated with that specific error. Appendix A contains a list of the user-recoverable error numbers and messages.

The ERL variable contains the line number containing the statement that caused the error.

12.4.3 THE RESUME STATEMENT

The RESUME statement is used to continue the normal flow of program execution after an error has been handled. The RESUME statement is analogous to a RETURN statement in a subroutine. It should appear at the point in an error handling routine where the programmer has complete error handling. The format of the RESUME statement is:

```
RESUME {<line number>}
```

If the <line number> is specified, control is resumed at that specified line. A plain RESUME or RESUME 0 statement resumes execution at the line containing the statement that caused the error. If that statement is on a multiple-statement line, control is passed to the first DIM, DEF*, FNEND, FOR, NEXT or DATA statement that precedes the erroneous statement. If none of these six statements appear on the line in question, control is passed to the first statement on that line.

Note that if a RESUME statement is executed when no previous ON ERROR GOTO statement has been executed, a fatal run-time error is generated.

If an ON ERROR processing routine has called any subroutines or referenced any functions in the course of its processing, those subroutines or functions must be exited correctly, and in the right order before the RESUME statement can be executed.

A.0 APPENDIX A: CS BASIC ERROR MESSAGES

This appendix is a list of the error messages that CS BASIC can issue to the user. The list is divided into two sections. The first contains those errors that are recoverable; the second set of errors are fatal and must be corrected before the program can be rerun.

A.1 RECOVERABLE-ERROR MESSAGES

These error messages can be generated via the ON ERROR GOTO facility of the language. The number shown in the "Error Number" column is the number that appears in the ERR variable.

4 Cannot write values of a virtual array to channel

The actual writing of a file buffer or output characters to the file system failed.

4 Error in writing file

The actual writing of a file buffer or output characters to the file system failed.

4 Error writing virtual array element to channel

The actual writing of a file buffer or output characters to the file system failed.

4 Can't write to file

The actual writing of a file buffer or output characters to the file system failed.

5 Can't open (filename)

The file specified in an OPEN statement cannot be opened. The filename may be misspelled.

5 Can't find (filename)

The system cannot locate the file specified in an OPEN statement. Possibly due to a misspelling.

9 Channel not open

An input-output operation was performed to a channel that has not yet been OPEN'ed.

9 FIELD channel not open

A FIELD statement is referencing an input-output channel that has not yet been OPEN'ed.

9 Attempt to reference an unopened channel

An input-output operation was performed to a channel that has not yet been OPEN'ed.

11 End of file on device

A data input operation attempted to read past the end of a file.

31 Buffer sizes smaller than default not supported

In an OPEN statement, a RECORDSIZE option specified a buffer size smaller than the default value (512 bytes).

31 Can't have USING value larger than recordsize

A using option on a GET or PUT statement has a larger value than the size of the buffer.

35 Stack overflow

Too many nested GOSUBS or function calls. This situation can arise when a subroutine or function calls itself indefinitely with no means of terminating the nesting.

43 Virtual array must be on disk file

An attempt has been made to open a virtual array file on a device that is not a disk (a terminal, for instance).

45 Virtual array not yet open

An attempt was made to reference a virtual array before its associated file was OPEN'ed.

46 Channel number out of range in CLOSE statement

Channel numbers must be between 1% and 12%.

46 Channel number out of range in OPEN statement

Channel numbers must be between 1% and 12%.

46 Channel number out of range

Channel numbers must be between 0% and 12% for statements other than OPEN and CLOSE.

50 Bad input format in READ or INPUT statement

A READ or INPUT statement detected data that is syntactically incorrect.

52 Integer too big

Integer overflow or underflow. Integers must be between -32768 and +32767.

55 Subscript out of range

An attempt was made to reference an array element outside the defined bounds of the array.

55 Array index error

An attempt was made to reference an array element outside the defined bounds of the array.

55 Dimensions or maximum size prevents redimensioning

This message arises from one of the following problems:

- An attempt was made to redimension a one-dimensional array to a two-dimensional array, or vice versa, or,

- An attempt was made to redimension an array to a size with more elements than were originally allocated for that array.

55 Current matrix dimensions smaller than specified

In a MAT PRINT statement of the form

```
MAT PRINT A(x, y)
```

the dimensions x and y exceed the current size of a two-dimensional matrix.

55 Current matrix dimension smaller than specified

In a MAT PRINT statement of the form

```
MAT PRINT A(x)
```

the dimension x exceeds the current size of a one-dimensional matrix.

55 Negative bounds not allowed

Dimensions for a matrix must be positive integers.

55 Matrix dimension error

Operands to a matrix operator do not match. For example, in the matrix addition operation:

```
MAT A = B + C
```

the dimensions of the B and C matrices must be the same.

55 Matrix must be square in order to compute inverse

The INV function can invert only a square matrix.

56 matrix cannot be inverted

An attempt was made to invert a matrix that is singular.

57 Out of data in READ statement

A READ statement tried to read more data than defined in DATA statements.

58 ON GOTO range error

The index value in an ON GOTO statement was outside the specified range of line numbers.

58 ON GOSUB range error

The index value in an ON GOSUB statement was outside the specified range of line numbers.

61 Divide by zero

Attempt to divide by zero (integers only).

63 FIELD overflows buffer

A FIELD statement tried to allocate more space than was available in the input-output buffer.

A.2 NONRECOVERABLE-ERROR MESSAGES

The error messages listed below refer to fatal errors that the programmer cannot recover from via the ON ERROR GOTO facility.

Invalid label number in CHAIN

A line number specified in a CHAIN statement does not exist in the CHAIN'ed to program.

CHAIN file not found

The file specified in a CHAIN statement cannot be found.

Only blanks allowed between \ in USING string

Characters other than spaces appear between consecutive \ characters in a PRINT USING statement.

Missing matching \ in USING string

There is an odd number of \ characters in a PRINT USING statement.

Incorrect USING format to print string

A USING string in a PRINT USING statement has characters that are not correct format characters for the kind of data to be printed.

Must not GET Or PUT virtual array or I/O file

A GET or PUT statement has been applied to a file that was previously used for ASCII input-output or for virtual array storage.

Must not use file as virtual array and for I/O

A file that was previously used for virtual array storage has been referenced for ASCII input-output or for block input-output.

Can't redimension virtual array

An attempt was made to redimension a virtual array.

Virtual array must not be both source and destination

In a matrix operation of the form:

```
MAT A = B * C
```

where the matrix **A** is a virtual array, it must be distinct from the matrices **B** and **C** in the same operation. Similarly, in a matrix operation such as

```
MAT A = TRN(B)
```

where the matrix **A** is a virtual array, it must be distinct from the matrix **B**.

String operand has incorrect format

A statement in the program has attempted to do string arithmetic on a character string that does not have the correct format for a numeric string value. The only characters allowed in a numeric string are digits, plus sign (+), minus sign (-), and decimal point (.).

Result of string arithmetic too long

A string arithmetic operation generated a result that was longer than 56 characters.

Attempt to divide by zero in string arithmetic

It is not possible to divide by zero in string arithmetic.

Result string too long

The string resulting from a NUM1$ function must be less than 256 characters in length.

An ASCII I/O to virtual array or Block I/O file

A file that was previously used for virtual array storage or for block input-output has been referenced in an INPUT or PRINT statement.

Null string can't be used as USING string

A USING string in a PRINT USING statement must have some characters in it.

Incorrect USING format to print numeric

A format character (such as !) was used for a numeric field.

Must not use $$ format with exponential notation

The exponential format cannot be used in conjunction with the $$ format.

Must not use * fill with exponential notation

The exponential format cannot be used in conjunction with the * format.

Can't use * fill with leading minus sign

If the * fill character is used in a PRINT USING statement, only trailing minus signs may be used.

Can't use $$ format with leading minus sign

If the $$ format is used in a PRINT USING statement, only trailing minus signs may be used.

Missing END statement

There is no END statement in the program,

Syntax error

The interpreter detects an incorrect statement syntax.

RETURN without GOSUB

A RETURN statement has been executed, but no previous GOSUB had been executed.

RETURN from DEF FNX

A RETURN statement was encountered in the body of a multiline function. RETURN can be used only to return from a GOSUB subroutine.

GOTO target does not exist

The line number specified in a GOTO statement does not exist in the program.

GOSUB target does not exist

The line number specified in a GOSUB statement does not exist in the program.

Can't LSET or RSET Virtual arrays

An element of a string virtual array may not be the target of an LSET or RSET statement. LSET and RSET can be applied only to strings in memory or to string names associated with an input-output buffer through a FIELD statement.

Can't RESUME

An attempt has been made to RESUME execution of a program, when no ON ERROR routine has been entered.

Can't CONTINUE

A CONT command can be used only after the program has stopped as a result of a previously executed STOP statement.

Call of undefined function

A reference was made to a function that does not exist in the program.

Can't use Virtual arrays in FIELD statement

A string variable names in a FIELD statement is the name of a virtual array element. Only ordinary string names may be used in a FIELD statement.

Negative FIELD width

Size of a field in a FIELD statement must be positive.

B.0 APPENDIX B: IMPLEMENTATION NOTES

This appendix describes the ways the CS BASIC represents data storage and the mechanisms for passing arguments to subroutines and functions.

B.1 STORAGE ALLOCATION

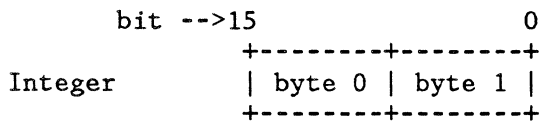
An integer value occupies 16 bits (two bytes).

A floating point value occupies 64 bits (eight bytes). A floating point number has a sign bit, an 11-bit exponent, and a 52-bit mantissa.

B.2 DATA REPRESENTATIONS

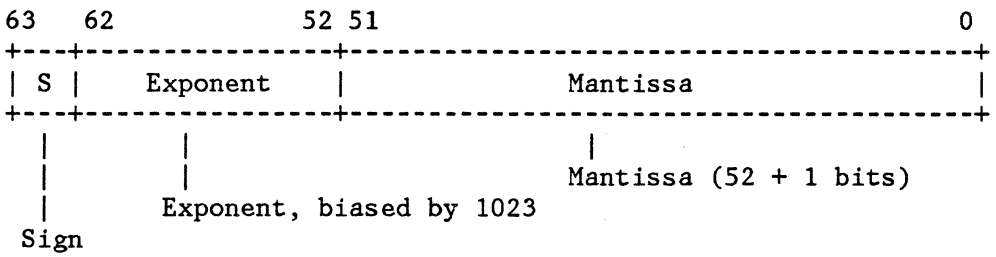
Whatever the size of the data element in question, the most significant bit of that element is always in the lowest numbered byte. See diagrams below.

Representation of Integers



Representation of Floating Point

Floating point data elements are represented according to the proposed IEEE standard described in Computer magazine of March, 1981. The diagram below illustrates the representation.



Floating Point Representation

The parts of floating point numbers are as follows:

- a one-bit sign bit designated by "S" in the diagrams above. The sign bit is a 1 if, and only if, the number is negative.
- a biased exponent. The exponent is eleven bits. The values of all zeros and all ones are reserved values for exponents.
- a normalized mantissa, with the high-order ;.bf 163;1;.pf; bit "hidden". The mantissa is 52 bits for a floating point number. A floating pointnumber is represented by the form:

$$2^{\text{exponent-bias}} * 1.f$$

where 'f' is the bits in the mantissa.

Representation of Extreme Floating Point Numbers

zero (signed) is represented by an exponent of zero, and a mantissa of zero.

denormalized numbers are a product of "gradual underflow". They are non-zero numbers with an exponent of zero. The form of a denormalized number is:

$$2^{\text{exponent-bias+1}} * 0.f$$

where 'f' is the bits in the mantissa.

signed infinity — (that is, affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero mantissa.

Not-a-Number (NaN) — is represented by the largest value that the exponent can assume (all ones), and a non-zero mantissa. The sign is usually ignored.

Normalized floating point numbers are said to contain a "hidden" bit, providing for one more bit of precision than would normally be the case.

Hexadecimal Representation of Selected Numbers

Value	REAL	PRECISION
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7F8xxxxx	7FFxxxxxxxxxxxx

Deviations from the Proposed IEEE Standard

Deviations from the proposed IEEE standard in this implementation are as follows:

- affine mapping for infinities,
- normalizing mode for denormalized numbers,
- rounds approximately to nearest - 7 or more guard bits are computed, but the "sticky" bit is not,

- exception flags are not implemented,
- conversion between binary and decimal is not implemented.

B.3 ARITHMETIC OPERATIONS ON EXTREME VALUES

This subsection describes the results derived from applying the basic arithmetic operations on combinations of extreme values and ordinary values.

No traps or any other exception actions are taken.

All inputs are assumed to be positive. Overflow, underflow, and cancellation are assumed not to happen.

In all the tables below, the abbreviations have the following meanings:

Abbreviation	Meaning
DEN	Denormalized Number
NUM	Normalized Number
Inf	Infinity (positive or negative)
NaN	Not a Number
Uno	Unordered

Addition and Subtraction

Left Operand \ Right Operand	0	Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Note 1	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Note 1: Inf + Inf = Inf; Inf - Inf = NaN

Multiplication						
Left Operand	Right Operand					
	0	Den	Num	Inf	NaN
0	0	0	0	NaN	NaN
Den	0	0	Num	Inf	NaN
Num	0	Num	Num	Inf	NaN
Inf	NaN	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Division						
Left Operand	Right Operand					
	0	Den	Num	Inf	NaN
0	NaN	0	0	0	NaN
Den	Inf	Num	Num	0	NaN
Num	Inf	Num	Num	0	NaN
Inf	Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Comparison						
Left Operand	Right Operand					
	0	Den	Num	Inf	NaN
0	=	<	<	<	Uno
Den	>		<	<	Uno
Num	>	>		<	Uno
Inf	>	>	>		Uno
NaN	Uno	Uno	Uno	Uno	Uno

Notes:

NaN compared with NaN is Unordered, and also results in inequality.

+0 compares equal to -0.

Max						
Left Operand	Right Operand					
	0	Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Min						
Left Operand	Right Operand					
	0	Den	Num	Inf	NaN
0	0	0	0	0	NaN
Den	0	Den	Den	Den	NaN
Num	0	Den	Num	Num	NaN
Inf	0	Den	Num	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

B.4 HOW STRINGS ARE STORED

Strings are stored in a storage area that is allocated for them as required. A string variable is actually a descriptor containing the address of the storeage area, and the length of the string, as shown:

Four-byte Address	Two-byte Length

C.0 APPENDIX C: LANGUAGE SUMMARY

This appendix provides a "quick reference" summary of the syntactic constructions of the CS BASIC language. The organization of this appendix follows that of the entire manual, and the major sections in this appendix correspond one for one with the chapters which have gone before.

C.1 NOTATION USED FOR SYNTACTIC DEFINITIONS

Words appearing in upper case, such as LET, are CS BASIC keywords.

In general, special characters such as = signs represent themselves when they appear in statement syntax.

The "angle brackets" < and > enclose elements of the language.

Elements which appear in the braces { } are optional.

The vertical bar | character stands for "or". It separates choices in the list of elements.

When an element is followed by an ellipsis (...), it indicates that the element may be repeated.

C.2 ELEMENTS OF THE BASIC LANGUAGE

The CS BASIC character set consists of 26 upper case letters, 26 lower case letters, and 21 other characters.

A <letter> is one of the 52 characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

A <digit> is one of the ten characters:

0 1 2 3 4 5 6 7 8 9

An <alphanumeric character> is a <letter> or a <digit>.

The other printing characters consist of the following characters:

Character	Meaning	Character	Meaning
	Blank or space	=	Assignment
+	Addition	-	Subtraction
*	Multiplication	/	Division
^ or **	Exponentiation	.	Decimal Point
:	Multiple Statements	&	Continuation
(	Left Parenthesis	)	Right Parenthesis
,	Compressed	;	Record Suppression
$	String Variable	%	Integer Variable
'	String Delimiter	"	String Delimiter
<	Less than sign	>	Greater than sign
#	Number Sign	!	Comment Starter

CS BASIC uses the ASCII character set.

Line Numbers and Statements

There may be only one line number per statement. Line numbers may be in the range 1 through 32767. Line number 0 is used for special purposes, for example in ON ERROR GOTTO statements.

A line number at the start of a line signals the presence of a (possibly empty) statement which is part of the BASIC program itself.

If a statement is not preceded by a line number, it is considered an "immediate mode" statement, and is executed immediately.

To get multiple statements on a line, each statement on the line, except the last one, is terminated by either a colon : or by a reverse slash character \

Continuation of a statement is signalled by an ampersand character & at the end of the line, before the carriage return.

Comments are indicated in one of two ways:

- by using a REM statement,
- using a ! sign after a statement

Identifiers

An identifier is defined syntactically as:

<letter> { <letter> | <digit> | .... }

Identifiers are further qualified by using a trailing % sign to indicate an integer variable, or by a trailing $ sign to indicate a string variable. An unqualified identifier is automatically assumed to denote a floating point variable.

Identifiers which are the names of functions are denoted by the letters **FN** in front of them.

Constants

The definition of an <integer> is:

<digit> { <digit> ... }

The definition of a numeric constant is:

{ + | - } { <integer> } { } { <integer } { **E** | **e** { + | - } <integer> }

Numeric constants are floating point values, unless they are suffixed with a % sign, in which case they are of type integer. Note that there are some exceptions to the rule, which are defined in chapter 2.

Numeric variables are designated by identifiers. A plain identifier represents the name of a floating point variable. If the name of the variable is followed by a percent sign, it means that the variable is to contain integer data.

String constants represent a sequence of ASCII characters. A string constant is delimited by either apostrophes (') or by double quote signs (").

To represent the string delimiter as a character in a string, two delimiters must be typed.

A string variable is an identifier followed by a **$** sign.

Dimensioned Variables

Dimensioned variables are introduced with the DIM (DIMension) statement. Floating, integer, and string variables can be dimensioned.

The format of the DIM statement is:

```
<line number> DIM <identifier> (<subscripts>)
                    { , <identifier> (< subscripts>) ... }
```

where <subscripts> is:

```
<upper bound> { , <upper bound> }
```

and <upper bound> is a numeric constant which determines the upper bound of that dimension of the array. Dimensioned variables can have one or two dimensions.

The bounds of each dimension in the declaration must be a positive, non-zero integer. The bounds specify the upper bounds only; there is no facility for specifying the lower bound of a dimension. In all cases, the lower bound of a dimension is zero.

Virtual arrays are declared in a variant of the DIM statement, the format of the declaration is:

```
DIM #<integer constant>, <identifier> (<subscripts>)
                { , <identifier>(<subscripts>)... }
```

The <integer constant> is an integer in the range 1 through 12, which is the internal file designator of a desk file.

Initial values assigned to variables are:

```
0.0     for a floating variable,
0%      for an integer variable,
" "     (the null string) for a string variable.
```

The same variable name can be used for more than one type of object with no ambiguity.

C.3 EXPRESSIONS

Arithmetic Operators

Operator	Meaning	Precedence
^ or **	Exponentiation	Highest
+	Unary Plus	Intermediate
-	Unary Minus	
*	Multiplication	Intermediate
/	Division	
+	Addition	Lowest
-	Subtraction	

Arithmetic Relational Operators

Operator	Meaning
=	Equal To
<	Less Than
<=	Less Than or Equal To
>	Greater Than
>=	Greater Than or Equal To
<>	Not Equal To
==	Approximately Equal To

Logical Operators

Operator	Meaning	Precedence
NOT	Logical Negation	Highest
AND	Logical Conjunction	Intermediate
OR	Logical Disjunction	
XOR	Logical Exclusive OR	Intermediate
IMP	Logical Implication	
EQU	Logical Equivalence	Lowest

String Operators

Operator	Meaning
=	Equivalent
<	Less Than
<=	Less Than or Equal To
>	Greater Than
>=	Greater Than or Equal To
<>	Not Equal To
==	Identical

Logical Operators on Integers

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

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

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

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

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

A	NOT A
0	1
1	0

C.4 ASSIGNMENT STATEMENTS

Simple Assignment Statements

{ LET } <variable> = <expression>

Multiple Assignment

{ LET } <var> { , <var> ... } = <expression>

String Assignment

{ LET } <stringvar> = <expression>

LSET and RSET - Change Strings in Place

LSET <stringvar> { , <stringvar> ... } = <string>

RSET <stringvar> { , <stringvar> ... } = <string>

CHANGE - Character and Numeric Conversion

CHANGE <from_var> TO <to_var>

If <from_var> is a string, <to_var> must be an integer array variable. If <from_var> is an integer array variable, <to_var> must be a string.

C.5 CONTROL STATEMENTS

IF THEN and IF GOTO Statements

IF <condition> THEN <statements>

IF <condition> THEN <line number>

IF <condition> GOTO <line number>

IF-THEN-ELSE Statement

```
                THEN <statement>
                                    {ELSE <statement>}
IF <condition> THEN <line number>
                                    {ELSE <statement>}
                GOTO <line number>
```

WHILE NEXT Statement

```
WHILE <condition>
   <statement>
NEXT
```

UNTIL NEXT Statement

```
UNTIL <condition>
   <statements>
NEXT
```

FOR NEXT Statement

```
FOR <var> = <exp1> TO <exp2> { STEP <exp3> }
 . . . . Statements subordinate to the FOR
NEXT <var>
```

FOR WHILE and FOR UNTIL Statements

```
FOR <var> = <exp1> { STEP <exp2> } WHILE <condition>
 . . . . Statements subordinate to the FOR
NEXT <var>

FOR <var> = <exp1> { STEP <exp2> } UNTIL <condition>
 . . . . Statements subordinate to the FOR
NEXT <var>
```

GOTO Statement

```
GOTO <line number>
```

ON GOTO Statement

```
ON <exp> GOTO <list of line numbers>
```

Statement Modifiers

```
<statement> IF <condition>

<statement> UNLESS <condition>

<statement> FOR <var> = <exp1> TO <exp2> { STEP <exp3> } <statement> WHILE
<condition>

<statement> UNTIL <condition>
```

C.6 INPUT AND OUTPUT STATEMENTS

DATA - Define Data in Program

```
DATA <value> { , <value> ... }
```

READ - Read Data From DATA List

```
READ <variable> { , <variable> ... }
```

RESTORE - Reposition to Start of DATA

```
RESTORE
```

OPEN - Open a File for Data Transfer

```
OPEN <string> {FOR INPUT | FOR OUTPUT} AS FILE #<exp>
     {, RECORDSIZE <exp>} {, CLUSTERSIZE <exp>}
     {, FILESIZE <exp> {, MODE <exp>}
```

CLOSE - Close a File

```
CLOSE <exp> {, <exp> ...}
```

PRINT - Print on File

```
PRINT {#<exp>,} <expr> {, <expr> ...}
```

PRINT USING - Formatted Printing

```
PRINT {#<expr>,} USING <string>, <expr> {,<expr> ...}
```

INPUT - Input Data from File

```
INPUT {#<expr>,} {<string>;} <var>
                              {{, <string>;} <var> ...}
```

INPUT LINE - Input a String from a File

```
INPUT LINE {#<expression>,} <string variable>
```

GET and PUT - Read or WRITE Data

```
GET #<expr1> {, RECORD <expr2> | BLOCK <expr2> }
          , COUNT <expr3>, USING <expr4>

PUT #<expr1> {, RECORD <expr2> | BLOCK <expr2> }
          , COUNT <expr3>, USING <expr4>
```

FIELD - Set Buffer Structure

```
FIELD #<expr>, <expr> AS <stringvar>
          {, <expr> AS <stringvar> ...}
```

C.7 MATRIX MANIPULATION

Initializing a Matrix

```
MAT <matrix> = ZER | CON | IDN {(dimensions)}
```

ZER initializes the matrix to zeros.

CON initializes the matrix to ones.

IDN initializes the matrix as an identity matrix (ones along the principal diagonal, zeros elsewhere).

Matrix Input and Output

```
MAT READ <identifier> {(<subscripts>)}
          {, <identifier>{(<subscripts>)} ...}

MAT PRINT  {#exp>,} <matrix name> {, | ; }

MAT INPUT  {#<exp>,} <matrix>{(<dimensions>)}
                     {, <matrix>{(<dimensions>)}}
```

NUM returns the number of elements input for a one-dimensional matrix, and the number of rows input for a two-dimensional matrix.

NUM2 returns the number of elements input in the last row of a two-dimensional matrix.

Matrix Arithmetic Operations

- assignment,
- addition and subtraction,
- scalar multiplication,
- multiplication of conforming matrices.

Matrix Functions

TRN - Take the transpose of a matrix,
INV - Invert a matrix,
DET - Find the determinant of a matrix.

Virtual Arrays

```
DIM # <integer constant>, <matrix> {, <matrix> ...}
```

The form of a string virtual array is:

```
<identifier>}(<dimensions>{=<integer constant>}
```

C.8 PROGRAM STRUCTURE

Subroutines

GOSUB <line number>

ON <exp> GOSUB <line number> {, <line number> ...}

RETURN

Functions

DEF * FN<var>{(<var>{(<arg> {<arg> ...})}

DEF* FN<var> {, <arg> ...})}
{<statements comprising the body of the function } {FN<var> = <exp>}
FNEND

Error Handling

ON ERROR GOTO {<line number>}

The ERR variable contains the error associated with that specific error.

The ERL variable contains the line number containing the statement which caused the error.

RESUME {<line number>}

The END and STOP Statements

The END and STOP statements both terminate program execution.

CHAIN - Execute Another Program

CHAIN <string> {LINE} {<exp>}

D.0 APPENDIX D: RESERVED WORDS IN CS BASIC

Statement names and function names in the BASIC language are reserved words, and cannot be used for any other purpose. This is a list of the reserved words. Note that all these words are actually used in this implementation, but they are retained for compatibility with other implementations of CS BASIC.

ABS	COMMON	END	GE
ABS%	COMP%	EQ	GET
ACCESS	CON	EQV	GO
ALLOW	CONNECT	ERL	GOSUB
ALTERNATE	CONTIGUOUS	ERN$	GOTO
AND	COS	ERR	GT
APPEND	COUNT	ERROR	HT
AS	CR	ERT$	IDN
ASCII	CTRLC	ESCAPE	IF
ATN	CVT$$	EXP	IMP
BACK	CVT$%	EXTEND	INDEXED
BEL	CVT$F	FF	ECHO
BLOCK	CVT%$	FIELD	INSTR
BLOCKSIZE	CVTF$	FILE	INT
BS	DATA	FILESIZE	INV
BUCKETSIZE	DATE$	FILL	KEY
BUFFER	DEF	FILL$	KILL
BUFSIZE	DELETE	FILL%	LEFT
BY	DENSITY	FIND	LEFT$
CALL	DESC	FIX	LEN
CCPOS	DET	FIXED	LET
CHAIN	DIF$	FNEND	LF
CHANGE	DIM	FNEXIT	LINE
CHANGES	DIMENSION	FOR	LINPUT
CHR$	DUPLICATES	FORMAT$	LOC
CLOSE	INPUT	FROM	LOG
CLUSTERSIZE	EDIT$	FSP$	LOG10
COM	ELSE	FSS$	LSET

MAGTAPE	PEEK	SCRATCH	TEMPORARY
MAP	PI	SEG%	THEN
MAT	PLACE$	SEQUENTIAL	TIME
MID	POS	SGN	TIME$
MID$	PRIMARY	SI	TO
MODE	PRINT	SIN	TRM$
MODIFY	PROD	SLEEP	TRN
MOVE	PUT	SO	UNDEFINED
NAME	QUO$	SP	UNLESS
NEXT	RAD$	SPACE$	UNLOCK
NOCHANGES	RANDOM	SPAN	UNTIL
NODUPLICATES	RANDOMIZE	SPEC	UPDATE
NOECHO	RCTRLC	SQR	USEROPEN
NONE	RCTRLO	STATUS	USING
NOREWIND	READ	STEP	VAL
NOSPAN	RECORD	STOP	VAL%
NOT	RECORDSIZE	STR$	VALUE
NUL$	RECOUNT	STREAM	VARIABLE
NUM	REF	STRING$	VIRTUAL
NUM$	RELATIVE	SUB	VT
NUM1$	REM	SUBEND	WAIT
NUM2	RESET	SUBEXIT	WHILE
ON	RESTORE	SUM$	WINDOWSIZE
ONECHR	RESUME	SWAP%	WRITE
ONERROR	RETURN	SYS	WRKMAP
OPEN	RIGHT	TAB	XLATE
OR	RIGHT$	TAN	XOR
ORGANIZATION	RND	TAPE	ZER
OUTPUT	RSET	TASK	

E.0 APPENDIX E: ASCII CHARACTER SET

hex	char	hex	char	hex	char	hex	char
00	NUL	20	SP	40	@	60	'
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

INDEX

C

E

I

K

N

S

W

X

GC22-9184

READER'S COMMENT FORM

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate.

IBM Instruments, Inc. shall have the nonexclusive right, in its discretion, to use and distribute all submitted information, in any form, for any and all purposes, without obligation of any kind to the submitter. Your interest is appreciated.

Note: *Copies of IBM Instruments, Inc. publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM Instruments, Inc. product to your IBM Instruments, Inc. representative or to the IBM Instruments, Inc. office serving your locality.*

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefullness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A.

GC22-9184

Reader's Comment Form

Please do not staple — Fold and Tape

First Class
Permit 40
Armonk
New York

Business Reply Mail

No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

IBM Instruments, Inc.
P.O. Box 332
Danbury, Ct. 06810

Please do not staple — Fold and tape

3M Instruments, Inc.
.O. Box 332
)anbury, Ct. 06810