HEWLETT-PACKARD

HP-28C
Reference Manual

HP-28C

Reference Manual

HEWLETT
PACKARD

Edition 2 January 1987
Reorder Number 00028-90021

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Printing History

Edition 1	September 1986	Mfg. No. 00028-90022
Edition 2	January 1987	Mfg. No. 00028-90051

Welcome to the HP-28C

Congratulations! With the HP-28C you can easily solve complicated problems, including problems you couldn't solve on a calculator before. The HP-28C combines powerful numerical computation with a new dimension—*symbolic computation*. You can formulate a problem symbolically, find a symbolic solution that shows the global behavior of the problem, and obtain numerical results from the symbolic solution.

The HP-28C offers the following features:

- Algebraic manipulation. You can expand, collect, or rearrange terms in an expression, and you can symbolically solve an equation for a variable.
- Calculus. You can calculate derivatives, indefinite integrals, and definite integrals.
- Numerical solutions. Using the HP-28C Solver, you can solve an expression or equation for any variable. You can also solve a system of linear equations. With multiple data types, you can use complex numbers, vectors, and matrices as easily as real numbers.
- Plotting. You can plot expressions, equations, and statistical data.
- Unit conversion. You can convert between any equivalent combinations of the 120 built-in units. You can also define your own units.
- Statistics. You can calculate single-sample statistics, paired-sample statistics, and probabilities.
- Binary number bases. You can calculate with binary, octal, and hexadecimal numbers and perform bit manipulations.
- Direct entry for algebraic formulas, plus RPN logic for interactive calculations.

The *HP-28C Getting Started Manual* introduces your calculator and leads you through a sampling of examples.

The *HP-28C Reference Manual* (this manual) gives specific information about commands and how the calculator works. The first two chapters explain the fundamentals and basic operations. The third chapter is a dictionary of menus, describing the concepts and commands for each menu.

We recommend that you first work through the examples in Getting Started to get comfortable with the calculator. When you want to know more about a particular command, you can look up the command in the Reference Manual. When you're familiar with the commands and want a broader understanding of the calculator's operation, you can read the theoretical discussions in the Reference Manual.

These manuals show you how to use the HP-28C to do math, but they don't teach math. We assume that you're already familiar with the relevant mathematical principles. For example, to use the calculus features of the HP-28C effectively, you should know elementary calculus.

On the other hand, you don't need to understand all the math topics in the HP-28C to use those parts of interest to you. For example, you don't need to understand calculus to use the statistical capabilities.

Contents

3

Appendixes, Glossary, Indexes

How To Use This Manual

This manual contains general information about how the HP-28C works and specific information about how each operation works. For an overview of the manual, look through the Table of Contents. You can quickly find other types of information as follows.

To Learn About:	Refer to:
A particular operation, command, or function.	The Operation Index (page 381). All operations, commands, and functions are listed alphabetically. Each entry includes a brief description, a reference to a menu or topic in the Dictionary, and a page reference to the Dictionary. For background information, refer to the menu or topic in the Dictionary (listed alphabetically). For specific information, refer to the page number.
A particular menu.	Chapter 3, "Dictionary" (page 57). All menus are listed alphabetically.
Concepts and principles of HP-28C operation.	Chapter 1, "Fundamentals" (page 17).
How to perform general HP-28C operations.	Chapter 2, "Basic Operations" (page 31).
What a displayed message means.	Appendix A, "Messages" (page 349).
What an unfamiliar term means.	The Glossary (page 367).

How This Manual is Organized

Chapters 1 and 2 contain general information. "Fundamentals" is an overview for the experienced user, describing how the HP-28C works. The next chapter, "Basic Operations", describes how to enter objects in the command line, create variables, and perform system operations.

Chapter 3, "Dictionary", is the largest portion of the manual. Organized by menus, it details each individual operation, command, and function. The action of each command and function is defined in a stack diagram. (Refer to "How To Read Stack Diagrams" later in this section.)

Chapter 3 also includes major topics not related to a particular menu. There are entries for Arithmetic, Calculus, CATALOG, Programs, and UNITS.

Appendix A, "Messages," describes status and error messages you might encounter.

Appendix B, "Notes for HP RPN Calculator Users," and appendix C, "Notes for Algebraic Calculator Users," compare the HP-28C with other types of calculators you might be familiar with.

The Glossary defines terms used in this manual.

The Operation Index is an alphabetical listing of all operations, commands, and functions in the HP-28C. Each entry includes a brief description, a reference to the chapter or menu heading in the manual where you can find background information, and a page reference where you can find specific information.

How to Read Stack Diagrams

The action of a command is specified by the values and order of its arguments and results. An *argument* is an object that is taken from the stack, on which the command acts. The command then returns a *result* to the stack. (A few commands affect modes, variables, flags, or the display, rather than returning objects.)

The description of each command includes a *stack diagram*, which provides a tabular listing of the arguments and results of the command. A typical stack diagram looks like this:

XMPL ***Example*** **Function**

Level 2	Level 1		Level 1
obj_1	obj_2	➡	obj_3

This diagram shows:

- The text name (which can appear in the command line) is "XMPL".
- The descriptive name is "Example".
- XMPL is a function (allowed in algebraic expressions).
- XMPL requires two arguments, obj_1 and obj_2, taken from stack levels 2 and 1, respectively.
- XMPL returns one result, obj_3, to level 1.

The arrow ➡ in the diagram separates the arguments (on the left) from the results (on the right). It is a shorthand notation for "with the preceding arguments on the stack, executing XMPL returns the following results to the stack."

The arguments and results are listed in various forms that indicate as much specific information about the objects as possible. Objects of specific types are shown within their characteristic delimiter symbols. Words or formulas included with the delimiters provide additional descriptions of the objects. Stack diagrams generally use the following terms.

Terms Used in Stack Diagrams

Term	Description
obj	Any object.
x or *y*	Real number.
hms	Real number in hours-minutes-seconds format.
n	Positive integer real number.
flag	Real number, zero (false) or non-zero (true).
z	Real or complex number.
(*x*, *y*)	Complex number in rectangular form.
(*r*, *θ*)	Complex number in polar form.
# *n*	Binary integer.
"*string*"	Character string.
[*array*]	Real or complex vector or matrix.
[*vector*]	Real or complex vector.
[*matrix*]	Real or complex matrix.
[*R-array*]	Real vector or matrix.
[*C-array*]	Complex vector or matrix.
{ *list* }	List of objects.
{ *index* }	List of one or two real numbers specifying an array element.
{ *dim* }	List of one or two real numbers specifying the dimension(s) of an array.
'*name*'	Name or local name.
«*program*»	Program.
'*symb*'	Expression, equation, or a name treated as an algebraic.

The stack diagram for a command may contain more than one "argument ➡ result" line, reflecting the various possible combinations of arguments and results. Where appropriate, results are written in a form that shows the mathematical combination of the arguments. For example, the stack diagram for + includes the following entries (among others).

+ **_Add_** **Analytic**

Level 2	Level 1		Level 1
z_1	z_2	➡	z_1+z_2
[$array_1$]	[$array_2$]	➡	[$array_1+array_2$]
z	' $symb$ '	➡	' z+($symb$) '

This diagram shows that:

- Adding two real or complex numbers z_1 and z_2 returns a third real or complex number with the value z_1+z_2.
- Adding two arrays [$array_1$] and [$array_2$] returns a third array [$array_1+array_2$].
- Adding a real or complex number z and a symbolic object ' $symb$ ' returns a symbolic object ' z+($symb$) '.

1

Fundamentals

The HP-28C is based on a few fundamental principles. These principles are somewhat abstract, but their generality is the key to the power and flexibility of the calculator. You don't need a deep understanding of the calculator to use it, but a little understanding of its principles will make its full power available to you.

If you haven't used the HP-28C, we recommend that you begin with *Getting Started With the HP-28C*. That manual gives you step-by-step instructions for solving typical problems, and it demonstrates the fundamentals of the calculator. When you have some experience, you can return here to learn how the calculator works in a more general context.

This chapter begins with a general statement of how the calculator works, followed by sections that elaborate on the general statement. Later sections describe the stack, modes, and errors. Information about object entry, variables, and other basic topics appears in chapter 2, "Basic Operations." Information about individual operations and commands, organized by menus, appears in chapter 3, "Dictionary."

Principle of Operation

Calculator operation centers around the evaluation of objects on the stack. An object can be data, a name, or a procedure. To evaluate an object means to perform the action associated with that object. *Data objects do nothing special (they are just data), name objects refer to other objects, and procedure objects process the objects and commands in their definitions.*

One benefit of this principle is *uniformity*. For operations such as entering, editing, copying, storing, and recalling, you treat all objects alike. This uniformity means fewer rules for you to remember.

Another benefit is *flexibility*. You can use objects in any number of combinations to create the tools you need to solve a particular problem. Because you can choose when, if ever, to evaluate a symbolic object, you can work on a problem both symbolically and numerically.

Data Objects

These objects represent data treated as logical units: numerical data, character strings, and lists of objects.

Data Objects

Type	Object	Description
Real number	Real number	Real-valued decimal floating-point number.
Complex number	Complex number	Complex-valued decimal floating-point number.
Binary integer	Binary integer	64-bit binary integer number.
String	String	Character string.
Real array	Real vector Real matrix	*n*-element real vector. $n \times m$-element real matrix.
Complex array	Complex vector Complex matrix	*n*-element complex vector. $n \times m$-element complex matrix.
List	List	List of objects.

Evaluating a data object has no effect. If you put a data object on the stack and press EVAL, the object simply remains on the stack. Note that the objects contained in a list aren't evaluated when the list is evaluated.

Name Objects

These objects name other objects stored in user memory. *Local names* can be created by procedures and are automatically deleted when the procedure has completed evaluation.

Name Objects

Type	Object	Description
Name	Name	Refers to an object stored in user memory.
	Local name	Refers to an object temporarily held in local memory.

Variables

A variable is a combination of an arbitrary object and a name that are stored together. The name becomes the *name* of the variable; the other object is the *value* or *contents* of the variable. They are stored together in *user memory,* which is separate from the stack. HP-28C variables replace the numbered data registers and program memory found on most calculators.

There are two aspects to the evaluation of variable names: what causes names to be evaluated and the result of evaluating a name.

When is a name evaluated?

- The name on a USER menu label is evaluated when you press the menu key in immediate entry mode.
- An unquoted name in the command line is evaluated when the command line is evaluated.
- An unquoted name in a procedure is evaluated when the procedure is evaluated.
- A name contained in a variable is evaluated when the variable's name is evaluated.
- A name in level 1 is evaluated when the EVAL command is executed.

What happens when a name is evaluated?

- Evaluating a name that corresponds to a variable puts the stored object on the stack and, if the object is a name or program, evaluates it.
- Evaluating a name that doesn't correspond to a variable puts the name back on the stack.

Considering when and how a name is evaluated, note that:

- You can recall a data object contained in a variable simply by evaluating the variable's name.
- An unquoted name that refers to a program acts like a command to evaluate the program.
- If a name refers to another name that refers to yet another name, and so on, evaluating the first name causes all of the names to be evaluated.

Note Do not create a variable whose value includes its own name, such as would happen if you execute `'X' 'X' STO` or `'X+Y' 'X' STO`. Evaluating such a variable causes an endless loop. To halt an endless loop, you must perform a system halt ([ON] [▲], described in "Basic Operations"), which also clears the stack.

Similarly, do not create variables that reference one another in a circular definition. Evaluating a variable included in a circular definition also causes an endless loop.

Local Variables

Local variables are used only within the program structure that creates the variable. For example, user-defined functions and FOR...NEXT program structures use local variables. Names that identify local variables are called *local names* and are described in "Programs." Evaluating a local variable's name simply puts the local variable's contents on the stack.

Formal Variables

In symbolic calculations you can use name objects as variables (in the mathematical sense) before assigning values to the variables. If your goal is a symbolic result—say, the derivative of an expression—you might never assign values to the variables.

Names used as mathematical variables, but not associated with stored objects, are called *formal variables*. We use this term only in a few cases when the distinction is important. Evaluating a formal variable leaves its name on the stack.

Procedure Objects

These objects contain *procedures*—sequences of objects and commands that are processed when the procedure object is evaluated. A *program object* can contain any sequence of objects and commands, including those affecting the stack, user memory, or calculator modes. An *algebraic object* contains a limited number of object types and commands, and its syntax is similar to mathematical expressions and equations.

Procedure Objects

Type	Object	Description
Program	Program	Contains any sequence of objects.
Algebraic	Expression	Contains a mathematical expression.
	Equation	Contains a mathematical equation relating two expressions.

Programs

A program is essentially the object form of a command line. The objects and commands you enter in the command line constitute a procedure. When you surround that procedure by the program delimiters, you indicate that you want to treat the procedure as an object that will be evaluated later.

When is a program evaluated?

- You can evaluate a program in level 1 by executing the EVAL command.
- A program stored in a variable is evaluated when the variable's name is evaluated.
- Commands such as DRAW, ∫, and ROOT repeatedly evaluate a program that is their argument.
- A program that is the procedure part of a local variable structure is evaluated when the structure is evaluated.

What happens when a program is evaluated?

- Evaluating a program puts each object on the stack and, if the object is a command or unquoted name, evaluates it.

Considering when and how a program is evaluated, note that:

- Suppose that a program contains an unquoted name that refers to another program, and that program contains an unquoted name referring to yet another program, and so on. Evaluating the first program causes all of the programs to be evaluated. (The last one referenced is the first one completely evaluated.)
- If you execute a command that evaluates a program, the commands in the program may overwrite the original command arguments stored in LAST.

Expressions

An expression is a procedure representing a mathematical expression that is entered and displayed in a syntax corresponding to ordinary mathematical forms.

When is an expression evaluated?

- You can evaluate an expression by executing the EVAL command. (Evaluating expressions is the most common use of EVAL.)

- Commands such as DRAW, ∫, ROOT, TAYLR, and QUAD repeatedly evaluate an expression that is their argument.
- An expression that defines a user-defined function is evaluated when the function is evaluated.

What happens when an expression is evaluated?

- Evaluating an expression puts each object on the stack and evaluates it. These objects are evaluated in RPN order (the order of the equivalent program), not in the order they appear in the expression.

Considering when and how an expression is evaluated, note that:

- Although evaluating a name that refers to a program evaluates the program, evaluating a name that refers to an expression puts the expression on the stack.
- If a name in an expression refers to a second expression, evaluating the first expression *doesn't* evaluate the second expression. Rather, the second expression is substituted for every occurrence of the name in the first expression.

Equations

Equations are two expressions related by an equals "=" sign. Evaluating an equation produces a new equation. The new left-hand expression is the result of evaluating the original left-hand expression. The new right-hand expression is the result of evaluating the original right-hand expression.

Commands

Commands are built-in procedures that you can include in programs. You can consider a command name as it appears in the command line (for example, DROP or SIN) to be the unquoted name of a procedure object stored in the calculator. This is similar to the names and contents of your own variables. In practice, there's no useful distinction between the unquoted name and the built-in procedure.

We classify built-in procedures according to their uses:

- An *operation* is any procedure built into the calculator, such as ENTER, CATALOG, or TRACE.
- A *command* is a programmable operation, such as SWAP or STO.
- A *function* is a command allowed in algebraics, such as IP or MIN.
- An *analytic function* is a function for which the HP-28C provides a derivative and inverse, such as SIN or +.

Built-in procedures are usually characterized by their highest capability. For example, SWAP is both a command and an operation, and IP is a function, a command, and an operation, but we characterize SWAP as a command and IP as a function.

The Stack

The stack is a sequence of numbered *levels*, each holding one object. Objects enter the stack in level 1, lifting objects already in the stack to higher levels. Objects also leave the stack from level 1, dropping the objects remaining on the stack to lower levels. All objects are treated identically—simply as objects—on the stack.

The HP-28C provides commands to duplicate, delete, and reorder objects on the stack. Several of these commands are found on the keyboard (DROP, ■ SWAP, ■ ROLL, and ■ CLEAR); others are in the STACK menu.

Most commands take input objects (called *arguments*) from the stack and return output objects (called *results*) to the stack. The arguments must be present on the stack before the command is executed. The command removes its arguments and replaces them with its results. For example, the function SIN takes a value (a real or complex number, or an algebraic) from level 1, computes its sine, and returns the result to level 1. The function + takes two values from the stack and returns their sum to the stack.

This type of logic, where the command comes after the arguments, is called *postfix logic* or *RPN*, for *Reverse Polish Notation*, named after the Polish logician Jan Lukasiewicz (1878-1956).

(Note that these postfix-logic commands include operations that may use a prefix syntax on other RPN calculators. For example, to select FIX display mode with two digits after the radix on the HP-28C, you must execute the sequence `2 FIX`. Similarly, to store the number 12 in a variable named FIRST, you must execute the sequence `12 'FIRST' STO`.)

Modes

For many operations you can control the results by selecting a mode. For example, you can control whether the trigonometric functions interpret numbers as degrees or radians by selecting degrees angle mode or radians angle mode.

The following table shows the modes in the HP-28C, grouped by similar topics. The default choice for each mode, selected when you perform a memory reset, is marked by an asterisk.

Most modes are indicated by a flag, an annunciator, or a menu label. The third case occurs if a mode is selected by menu keys: the menu label for the current selection appears in black characters, not the inverse video characters typical of menu labels. Pressing a menu key that changes a mode also changes the menu label from inverse to black characters.

HP-28C Modes

Mode	Choices	Indicator
Angle	Degrees*/radians	MODE labels, flag 60 clear*/set, (2π) annunciator
Beeper	Enabled*/disabled	Flag 51 clear*/set
Principal value	Off*/on	Flag 34 clear*/set

* Default choice.

HP-28C Modes *(Continued)*

Mode	Choices	Indicator
General Entry and Display		
Entry mode	Immediate*/Algebraic/ Alpha	Cursor, α annunciator
Case	Upper*/lower	None
Level 1 display	Multi-line*/compact	MODE labels, flag 45 set*/clear
Real Number Entry and Display		
Radix	Period*/comma	MODE labels, flag 48 clear*/set
Real number format	Standard*/Fixed/ Scientific/Engineering	MODE labels, flags 49–50
Number of decimal digits	0* through 11	Flags 53–56
Binary Integer Entry and Display		
Binary integer base	Decimal*/Hexadecimal/ Octal/Binary	BINARY labels, flags 43–44,
Binary integer wordsize	1 through 64*	Flags 37–42
Recovery		
COMMAND	Enabled*/disabled	MODE labels
UNDO	Enabled*/disabled	MODE labels
LAST	Enabled*/disabled	MODE labels, flag 31 set*/clear
Evaluation		
Evaluation of symbolic constants	Symbolic*/numeric	Flag 35 set*/clear
Evaluation of functions	Symbolic*/numeric	Flag 36 set*/clear
Printer		
Printer trace	Disabled*/enabled	PRINT labels, flag 32 clear*/set
Auto CR	Enabled*/disabled	Flag 33 clear*/set
Faster print	Disabled*/enabled	Flag 52 clear*/set

* Default choice.

Annunciators

The annunciators at the top of the display indicate the angle mode, the entry mode, and other status information.

Annunciators

Annunciator	Indication
○	A program is suspended.
⤴	The shift key has been pressed.
α	Alpha entry mode is active.
((•))	The HP-28C is busy—that is, not ready for keyboard input.
▭	Low battery.
(2π)	The current angle mode is radians.
🖶	The HP-28C is sending output to the printer.

Flags

A *flag* is a quantity that represents a truth value, either *true* or *false*. Flags occur as *numeric flags* and as *user flags*.

Numeric Flags. On the stack, a non-zero real number represents *true* and the real number 0 represents *false*. Numeric flags are used with program branch structures, such as IF...THEN...ELSE, and with logical tests, such as XOR.

User Flags. A separate part of calculator memory contains user flags, each having two possible states: *set* (true) or *clear* (false). You store the value *true* in the flag by setting the flag, and you store the value *false* by clearing the flag. You can test the value of a flag, which returns the corresponding numeric flag, 0 (false) or 1 (true), to the stack.

There are 64 user flags, numbered 1 through 64. Flags 1 through 30 are available for general use. Flags 31–64 have special meanings, as listed below—when you set or clear them you alter the modes associated with the flags.

Reserved User Flags

Number	Description	Default
31	LAST enable	Set
32	Printer trace	Clear
33	Auto CR	Clear
34	Principal value	Clear
35	Symbolic evaluation of constants	Set
36	Symbolic evaluation of functions	Set
37-42	Binary integer wordsize	Set
43-44	Binary integer base	Clear
45	Level 1 display	Set
46	Reserved	Clear
47	Reserved	Clear
48	Radix	Clear
49-50	Real number format	Clear
51	Beeper	Clear
52	Faster print	Clear
53-56	Number of decimal digits	Clear
57	Underflow action	Clear
58	Overflow action	Clear
59	Infinite Result action	Set
60	Angle	Clear
61	Underflow- exception	Clear
62	Underflow+ exception	Clear
63	Overflow exception	Clear
64	Infinite Result exception	Clear

Errors and Exceptions

When an error occurs, the calculator beeps and displays an error message in the top line of the display. Error messages are described in appendix A, "Messages."

If the error occurs during execution of a command that takes arguments from the stack, the arguments are restored to the stack if LAST is enabled. If LAST is disabled, the arguments are lost.

Errors in the Command Line

An error can occur during ENTER, while the calculator is processing the text in the command line. If so, the calculator beeps, displays `Syntax Error`, restores the command line, and attempts to indicate the problem. If the error resulted from illegal syntax, the incorrect text is displayed in inverse characters, followed by the cursor. If the error resulted from incomplete entry, the cursor is positioned at the end of the line.

Errors in Programs

If an error occurs in a program, the remainder of the program (not yet processed) is aborted. If the program's evaluation was started by another program, the remainder of that program is also aborted.

Mathematical Exceptions

Certain errors that can arise during ordinary real number calculations are classified as *mathematical exceptions*. An exception can act as an ordinary error and halt the calculation, or it can supply a default result, allowing the calculation to proceed. You can choose how exceptions act by setting or clearing flags 57, 58, and 59. The following table describes the mathematical exceptions and the related flags.

Mathematical Exceptions

Exception	Description
Infinite Result	This exception occurs when a calculation returns an infinite result. Examples include LN(0), TAN(90°), and dividing by zero. If flag 59 is set,* Infinite Result exceptions are errors. If flag 59 is clear, an Infinite Result exception returns the default result ±MAXR and sets flag 64, the Infinite Result indicator.
Overflow	This exception occurs when a calculation would return a finite result whose absolute value is greater than the largest machine-representable number MAXR. Examples include 9E499 + 9E499, EXP(5000), FACT(2000). If flag 58 is set, Overflow exceptions are errors. If flag 58 is clear,* an Overflow exception returns the default result ±MAXR and sets flag 63, the Overflow indicator.
Underflow	This exception occurs when a calculation returns a finite result whose absolute value is smaller than the smallest machine-representable number MINR. Examples include 1E-499/2 and EXP(-5000). If flag 57 is set, an Underflow exception acts like an error. It returns the error message Negative Underflow or Positive Underflow, depending on the sign of the actual result. If flag 57 is clear,* an Underflow exception returns the default result 0 and sets flag 62, the Underflow+ indicator, or flag 61, the Underflow− indicator, depending on the sign of the actual result.

* Default choice.

2

Basic Operations

This chapter describes how to enter objects in the command line, how to create, recall, and purge variables, how to recover previous command lines, stacks, and arguments, how to deal with low memory conditions, and how to perform system operations.

Object Entry

When you press a key to begin entering new objects, the character on the key is entered into a *command line*. The command line can contain any number of objects, represented in text form. It appears at the bottom of the display (immediately above the menu labels, if present). The command line also appears when you use ■ EDIT or ■ VISIT to view or alter the contents of an existing object.

The contents of the command line are processed when you press ENTER (or any command or function key that automatically performs ENTER). The contents of the command line are evaluated as a program, and the command line disappears from the display.

You can enter any number of characters into the command line. You can break the line into several rows by pressing ■ NEWLINE , which inserts a "newline" character (line-feed) into the command line string at the current cursor position. Newline characters act as object separators, but are otherwise ignored when the command line is evaluated.

If you enter more than 23 characters into the command line, characters scroll off the display to the left. An ellipsis (...) appears in the leftmost character position to indicate the undisplayed characters. If you try to move the cursor past the left end of the display, the leftmost characters scroll back into the display, and characters scroll off the display to the right. An ellipsis then appears at the right end of the display. When the command line contains multiple rows of text, all rows scroll left and right together.

Entering Numbers

Real numbers are entered by pressing the digit keys, CHS and EEX to produce the desired number. Digit keys always just add a single digit to the command line.

Change Sign: CHS. Pressing CHS changes the sign of a number in the command line. (If no command line is present, pressing CHS executes the command NEG, which negates the object in level 1. NEG is described in "Arithmetic.")

"Number" can be either the mantissa or the exponent of a number— the position of the cursor determines which is changed. If no sign is present, a minus sign (−) is inserted at the beginning of the number. If a plus sign (+) or minus sign is present, it is changed to the opposite sign.

If the cursor is not positioned at a valid number, pressing CHS adds a minus sign to the command line.

To key in a negative number as the first object in the command line, you must key in at least one digit, to create the command line, before pressing CHS. In all other cases you can press CHS before, during, or after keying in the number.

Enter Exponent: EEX. You can enter numbers that are quite large or small by using scientific notation. A number can be represented by a *mantissa* and an *exponent*, where the value of the number is the product of the mantissa and 10 raised to the power of the exponent.

To key in a number in scientific notation, key in the mantissa, press [EEX], and then key in the exponent. Pressing [EEX] adds a character E to the command line, separating the mantissa from the exponent.

If the cursor is not positioned at a valid number (or no command line is present), pressing [EEX] adds the characters 1E to the command line. If the cursor is positioned at a number that already has an exponent, pressing [EEX] moves the cursor to the first digit in the exponent.

Backspace: [◀]

Pressing [◀] deletes the character to the left of the cursor, moving the cursor (and any characters to the right) one space to the left. If you press and hold [◀], the action is repeated until you release the key. Pressing [◀] has no effect when the cursor is at the left end of a line.

Lower-Case Letters: [LC]

Pressing [LC] causes letter keys [A] through [Z] to append the corresponding lower case letters a through z to the command line. Lower-case entry continues until you press [LC] a second time, execute ENTER, or press [ON] to clear the command line.

Object Delimiters and Separators

The various object types are entered in the same form in which they are displayed. Successive objects or commands entered into the same command line must be separated from each other by one of the following:

- An object delimiter (,), [,], {, }, #, ", ', «, ».
- A space or newline.
- A period or comma, whichever is not currently the radix mark. (If flag 48 is clear, periods are radix marks and commas are separators; if flag 48 is set, commas are radix marks and periods are separators.)

In algebraic objects, spaces are ignored (except with the operators AND, OR, XOR, and NOT) and arguments contained within parenthesis (such as MOD(A,B)) must be separated by the current separator, either comma or period.

When you key in an object, you must follow the correct syntax for that object. Most object types begin and end with *delimiters*, special punctuation marks that identify the object. For example, strings are surrounded by double quotation marks, as in "Hi There" or "m^2", and vectors are surrounded by square brackets, as in [1 2 3] or [-5 6 7 -10.2]. The calculator follows the same format rules when it displays the objects.

The following table is an expanded version of the table printed above the left keyboard on the calculator. Both show the appropriate delimiters for various objects.

Object Formats

Object	Format	Example
Real number	*real*	-1.234E24
Complex number	(*real*, *real*)	(1.23,4.56)
Binary integer	# *digits*	# 123AF
String	"*text*"	"HELLO"
Real vector	[*real* *real* ...]	[1 2 3 4]
Real matrix	[[*real* *real* ...] [*real* *real* ...] ⋮ [*real* *real* ...]]	[[1 2 3 4] [5 6 7 8] [9 0 1 2] [3 4 5 6]]
Complex vector	[(*real*, *real*) ...]	[(1,2) (3,4)]
Complex matrix	[[(*real*, *real*) ...] [(*real*, *real*) ...] ⋮ [(*real*, *real*) ...]]	[[(1,2) (3,4)] [(5,6) (7,8)]]
List	{ *object* *object* ... }	{1 "HI" (1,2)}

Object Formats *(Continued)*

Object	Format	Example
Name	' *name* '	'FRED'
Local name	' *name* '	'FRED'
Program	« *object object* ... »	« DUP 4 ROLL »
Expression	' *expression* '	'A+B'
Equation	' *expression*=*expression* '	'A+B=SIN(X)'

Any missing delimiters at the end of the command line are automatically added when you press [ENTER].

How the Cursor Indicates Modes

The shape of the cursor indicates the current entry mode and the current choice of insert/replace mode. (Entry modes are described next, followed by the cursor menu, which includes insert/replace modes.) The following table shows the six possible combinations of entry mode and insert/replace mode.

	Insert mode	**Replace mode**
Immediate entry mode	◁	□
Algebraic entry mode	⊲	⊟
Alpha entry mode	◀	■

Entry Modes

There are three modes for entering different types of objects. In general, *immediate entry mode* is used to key in data objects, *algebraic entry mode* to key in name objects and algebraic objects, and *alpha entry mode* to key in programs and strings. You can always activate or deactivate alpha entry mode by pressing [α]; in some cases the entry mode changes automatically when you begin to key in a new object.

The current entry mode primarily affects how keys associated with commands operate—whether pressing a key causes the command to execute, or whether the name of the command is added to the command line. In this discussion "key" includes shifted keys such as ■ π and assigned menu keys such as SIN.

The following keys are unaffected by the current entry mode:

- Keys associated with non-programmable operations, such as ENTER, ■ CATALOG, or +ML. Pressing an operation key always executes the operation.
- All single-character keys on the left-hand keyboard. Pressing a character key always adds the character to the command line. (Although keys such as ■ > correspond to the names of functions, they act as character keys.)
- Character keys 0 through 9, ., ,, and ■ π on the right-hand keyboard. Pressing a character key always adds the character to the command line.
- CLUSR in the USER menu, HALT in the PROGRAM CONTROL (CTRL) menu, and any key in the PROGRAM BRANCH menu. Pressing one of these keys always adds the name of the command to the command line.

Along with command keys, the menu keys assigned to variable names in the USER menu are affected by the current entry mode. The following describes each entry mode and how it affects each type of key. The affected keys are all on the right-hand keyboard and are primarily menu keys.

Immediate Entry Mode. This is the default entry mode—a new command line normally begins in this mode. The cursor appears as □ or ◊. In immediate entry mode:

- Pressing a command key (such as STO) executes the command.
- Pressing a function key (such as +) executes the function.
- Pressing a variable key in the USER menu evaluates the variable name.

To save keystrokes, most command keys execute ENTER before executing the command. Exceptions to this rule are STD , DEG , and RAD in the MODE menu, and DEC , HEX , OCT , and BIN in the BINARY menu, which execute their command without executing ENTER—that is, without disturbing the command line.

Algebraic Entry Mode. Pressing ['] to begin a name or algebraic, while in immediate entry mode, activates algebraic entry mode. The cursor appears as ⊟ or ✧. Pressing ['] a second time to complete the object reactivates immediate entry mode. In algebraic entry mode:

- Pressing a command key executes the command, just as in immediate entry mode.
- Pressing a function key adds the function name to the command line. If the function takes its arguments in parentheses, such as `SIN(X)`, the opening parenthesis is also added.
- Pressing a variable key in the USER menu adds the variable name, unquoted, to the command line.

Alpha Entry Mode. Pressing [«] or ■["] to begin a program or string activates alpha entry mode, indicated by the α annnunciator. The cursor appears as ▦ or ✦. While in immediate or algebraic entry mode, pressing [α] activates alpha entry mode, and pressing [α] a second time reactivates the previous entry mode. At any time you can press ■[α LOCK] to "lock" alpha entry mode indefinitely. To "unlock" the entry mode, press [α].

In alpha entry mode:

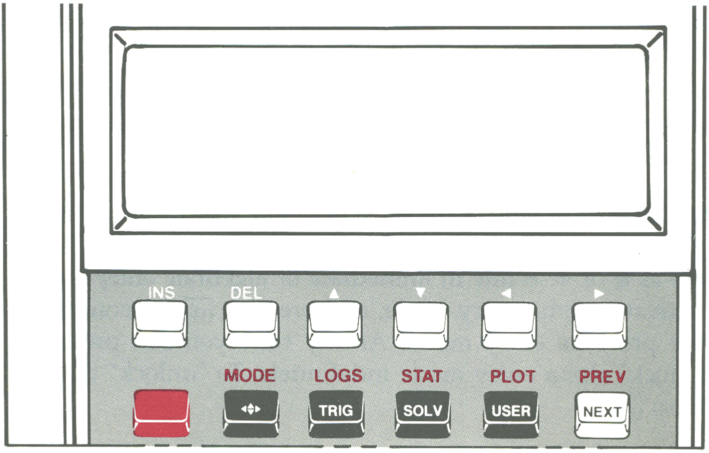
- Pressing a command key adds the command name to the command line.
- Pressing a function key adds the function name to the command line.
- Pressing a variable key in the USER menu adds the variable name, unquoted, to the command line.

If the cursor is in insert mode or positioned at the end of the command line when you press any of the above keys, one space is added to the beginning and end of the appended text to separate commands.

The Cursor Menu: [◂◆▸]

Pressing [◂◆▸] assigns the cursor menu to the menu keys. No menu labels appear; instead, the action of the menu keys is indicated by the white labels above the menu keys. The cursor menu contains editing operations more elaborate than backspacing ([◀]). Pressing [◂◆▸] a second time reassigns the previous menu, whose menu labels reappear in the display.

The cursor menu contains both shifted and unshifted keys. The unshifted keys are labeled in white above the corresponding menu keys, as illustrated.



The following table describes the operations associated with the unshifted cursor menu keys. If you press and hold any of these keys, except [INS], the operation is repeated until you release the key.

Unshifted Cursor Menu Keys

Key	Operation
INS	Switch between replace mode and insert mode. In replace mode, new characters replace existing characters; the cursor appears as □, ⊟, or ■. In insert mode, new characters are inserted between existing characters; the cursor appears as ◇, ⬖, or ◆.
DEL	Delete the character at the cursor position.
▲	Move the cursor up one line.
▼	Move the cursor down one line.
◀	Move the cursor left one space.
▶	Move the cursor right one space.

The following table describes the operations associated with the shifted cursor menu keys. Except for 🟥 INS, these operations are equivalent to repetitions of the unshifted operations.

Shifted Cursor Menu Keys

Key	Operation
🟥 INS	Delete all characters to the left of the cursor.
🟥 DEL	Delete the character at the cursor position and all characters to the right.
🟥 ▲	Move the cursor to the top row of the command line.
🟥 ▼	Move the cursor to the bottom row of the command line.
🟥 ◀	Move the cursor to the left end of the command line.
🟥 ▶	Move the cursor to the right end of the command line.

Enter Command Line: ENTER

Pressing ENTER evaluates the command line. (If no command line is present, pressing ENTER executes the command DUP, which duplicates the contents of level 1. DUP is described in "STACK.")

To evaluate the command line, ENTER must *parse* the text in the command line to make objects, combine the objects into a program, and then evaluate the program. More precisely, here is what happens when you press ENTER to evaluate a command line:

1. The busy annunciator ((•)) is turned on.
2. If UNDO is enabled, a copy of the current stack is saved.
3. The text string in the command line is searched for object delimiters and separators, and then broken into the corresponding substrings.
4. Each substring of text is tested against syntax rules to identify its object type, and the corresponding object is put on the stack.
5. If COMMAND is enabled, a copy of the command line is saved in the command stack.
6. The objects put on the stack are combined into a single program object, which is then evaluated.
7. The busy annunciator ((•)) is turned off.

If a substring fails the syntax tests in step 4, Syntax Error is displayed. The objects that ENTER has put on the stack are dropped, and the command line is restored. The incorrect text is highlighted in inverse characters, followed by the cursor. If the error resulted from incomplete syntax, the cursor is positioned at the end of the line.

Viewing Objects: ■ VIEW↑, ■ VIEW↓

You can view the hidden lines of an object that occupies more than the available display lines with ■ VIEW↑ and ■ VIEW↓. ■ VIEW↑ moves the display window up one line, and ■ VIEW↓ moves the display window down one line. You can also view hidden stack levels in this manner. Both ■ VIEW↑ and ■ VIEW↓ are repeating keys and can be used even during object entry or editing.

Editing Existing Objects

An existing object can be returned to the command line. You can then view its entire definition or change its definition, using the normal editing operations in the command line. EDIT returns an object in level 1 to the command line. VISIT returns an object in higher stack levels, or in user memory, to the command line.

Editing Level 1: 🟥 EDIT

EDIT acts as an inverse of ENTER, taking an object from level 1 and returning it to the command line. In more detail, here is what EDIT does:

1. A copy of the object in level 1 is converted to text form, and entered into the command line.
2. Alpha entry mode is activated.
3. The cursor menu is activated for editing.

The original object in level 1 is highlighted to remind you that you are editing that object and that the original copy is still preserved.

While the text form of the object is in the command line, you can alter it as you please. When you have finished editing you can:

- Press ON to cancel the edit, clear the command line, and leave the original object in level 1 unchanged.
- Press ENTER (or a key that performs ENTER) to replace the original object in level 1. (More precisely, the original object in level 1 is dropped and the command line is evaluated.)

If the cursor menu is still active when you complete the editing, the previous menu is restored.

Editing a Variable or Stack Level: ■ [VISIT]

VISIT is an extended version of EDIT. It enables you to view or edit an object stored in a variable, or in a stack level higher than level 1, without first recalling the object to level 1.

Editing a variable. To edit the object stored in a variable, put the variable name in level 1 and press ■ [VISIT]. The stored object is copied to the command line, alpha entry mode is activated, and the cursor menu is activated.

Editing a stack level. To edit the object in stack level *n*, put *n* on the stack and press ■ [VISIT]. The object is copied to the command line, alpha entry mode is activated, and the cursor menu is activated. The original object is highlighted to remind you that you are editing that object and that the original copy is still preserved.

You terminate VISIT in the same way as EDIT:

- Press [ON] to cancel the edit, clear the command line, and leave the original object unchanged.
- Press [ENTER] (or a key that performs ENTER) to replace the original object. (More precisely, the command line is evaluated, and the resulting object in level 1 replaces the original object.)

If the cursor menu is still active when you complete the editing, the previous menu is restored.

Evaluating Objects

EVAL ***Evaluate Object*** **Command**

Level 1	
obj ➡	

EVAL evaluates the object in level 1. The result of evaluation, including any results returned to the stack, depends on the evaluated object. Evaluation is described in detail in "Fundamentals". The evaluation of functions is affected by flag 36, which selects symbolic or numerical evaluation mode. See the section on "ALGEBRA".

→NUM — *Evaluate to Number* — Command

Level 1		Level 1
obj	➧	*z*

→NUM is identical to EVAL, except it temporarily sets numerical evaluation mode (described in "ALGEBRA") to insure that functions return numerical results. The current function evaluation mode is restored when →NUM is completed.

SYSEVAL — *Evaluate System Object* — Command

Level 1		
# *n*	➧	

SYSEVAL is intended solely for use by Hewlett-Packard in application programming. General use of SYSEVAL can corrupt memory or cause memory loss. *Use SYSEVAL only as specified by Hewlett-Packard applications.*

SYSEVAL evaluates the system object at the absolute address # *n*. You can display the version number of your HP-28C by executing `# 10 SYSEVAL` (assuming DEC base, which is the default base).

Names

Names can be up to 127 characters in length, although practical considerations suggest names no longer than five or six characters. The first character must be a letter. Lower-case letters are distinguished internally from upper-case letters but appear as upper-case in the USER menu. You can not use HP-28C command names for variable names.

The legal characters available on the keyboard are letters, digits, and the characters ?, Σ, π, →, μ, and ▫. The following characters cannot be included in variable names:

- Object delimiters (#, [,], ", ', {, }, (,), «, »).
- Algebraic operator symbols (+, −, *, /, ^, √, =, <, >, ≤, ≥, ≠, ∂, ∫).
- Current separator (. or ,).

Reserved Names

The following names are reserved for specific uses:

- EQ refers to the current equation used by the Solver and PLOT commands.
- ΣPAR refers to a list of parameters used by statistics commands.
- PPAR refers to a list of parameters used by plot commands.
- ΣDAT refers to the current statistical array.
- s1, s2, and so on, are created by ISOL and QUAD to represent arbitrary signs obtained in symbolic solutions.
- n1, n2, and so on, are created by ISOL and QUAD to represent arbitrary integers obtained in symbolic solutions.

You can use any of these names for your own purposes, but remember that certain commands use these names as implicit arguments.

Quoted and Unquoted Names

You can enter a name in the command line with or without quotes, depending on whether you want the name to be evaluated.

Quoted Names. Entering a name in single quotes means, "Put this name on the stack." That is, a quoted name is put on the stack, but not evaluated, when the command line is evaluated.

Unquoted Names. Entering a name without quotes means, "Evaluate the object named by this name." That is, an unquoted name is put on the stack, and then evaluated, when the command line is evaluated.

Duplicate Names

Normally you can't create two user variables that have the same name. However, certain commands (DRAW, ∫, QUAD, TAYLR) create a temporary user variable whose name duplicates the name argument you specify for the command. When the command completes execution, it purges the temporary variable. However, if the command is aborted, by a system halt ([ON] [▲]) or by an `Out of Memory` error, the temporary variable remains in user memory.

If the temporary variable remains, and if you had previously created a variable with that name, there will be two variables with the same name in the USER menu. Use PURGE to clear the duplicate name from the USER menu. (PURGE clears the most recently created variable, which is the temporary variable.)

Creating, Recalling, and Purging Variables

A variable is the combination of a name object and any other object, stored together in user memory. The name object represents the name of the variable; the other object is the value or contents of the variable.

This section tells you how to create an object, how to recall the contents of a variable to the stack without evaluation, and how to purge a variable.

STO Store Command

Level 2	Level 1		
obj	*' name '*	➡	

This command creates a variable whose name is *name* and whose value is *obj*. Subsequent evaluation of *name* puts *obj* on the stack and, if *obj* is a name or program, evaluates *obj*.

RCL Recall Command

Level 1		Level 1
' name '	➡	*obj*

This command searches user memory for the variable *name* and returns its contents *obj*. The object returned is not evaluated.

There is an important distinction between RCL (*recall*) and EVAL (*evaluate*). RCL requires a variable name as an argument and returns the contents of the variable. EVAL accepts any object as an argument and evaluates the object according to the rules for that object. RCL and EVAL have the same effect only when the argument is a name that refers to a data object, an algebraic, or a local variable. In these cases, both return the stored object to the stack.

The following table summarizes the results of executing EVAL and RCL (with the name `'ABC'` in level 1), for different values of the associated variable ABC.

Comparison Between EVAL and RCL

If 'ABC' contains:	'ABC' EVAL:	'ABC' RCL:
(undefined)	Returns 'ABC'.	Causes Undefined Name error
A name 'DEF'.	Evaluates 'DEF'.	Returns 'DEF'.
A program.	Evaluates the program.	Returns the program.
Any other object.	Returns the object.	Returns the object.

PURGE *Purge* Command

Level 1	
'name' ➡	
{ *name*$_1$ *name*$_2$ ... } ➡	

PURGE deletes one or more variables from user memory. If the argument is a name, PURGE deletes the corresponding variable. If the argument is a list of names, PURGE deletes each of the named variables.

Recovery

The HP-28C automatically saves copies of command lines, the stack, and arguments. These copies enable you to recover from a mistake—to go back to where you were before the mistake. You can then redo a calculation correctly without having to start over from the beginning. The copies of command lines and arguments are also handy for repeating calculations.

These copies can consume a significant amount of memory. For each of these recovery features—command lines, the stack, and arguments—you can choose whether to enable or disable the feature. (The operations to enable or disable the recovery features are in the MODE menu.) This section assumes all recovery features are enabled, as they are at memory reset.

The recovery operations are 🟥 COMMAND, which recovers copies of the command line, 🟥 UNDO, which recovers a copy of the stack, and 🟥 LAST, which recovers the last arguments used.

Command Line Recovery: 🟥 COMMAND

Each execution of ENTER saves a copy of the command line. Up to four saved command lines are stored in the *command stack*. Pressing 🟥 COMMAND once retrieves the first (most recently saved) command line, replacing the present contents of the command line. Pressing 🟥 COMMAND a second time retrieves the second command line, and so on. If you press 🟥 COMMAND more than four times, the sequence starts over with the first command line.

Stack Recovery: 🟥 UNDO

Each execution of ENTER saves a copy of the stack before evaluating the command line. Pressing 🟥 UNDO clears the current stack and replaces it with the saved stack. UNDO restores the stack to the same condition as before you pressed ENTER (or the key that executed ENTER), but it doesn't affect any changes that occured in the user flags or in user memory.

While a program is suspended, the UNDO feature, (including +UND, -UND, and UNDO itself), is associated with the suspended program environment. That is, UNDO will restore the stack that was present prior to the last ENTER, but after the (most recent) program was suspended. After a program is continued and completes execution, UNDO will then reference the stack saved before the program was executed.

Last Arguments Recovery

LAST *Last Arguments* Command

	Level 3	Level 2	Level 1
➡			obj_1
➡		obj_1	obj_2
➡	obj_1	obj_2	obj_3

Commands that take arguments from the stack save copies of those objects. Executing ■ LAST returns the objects most recently saved by a command. The objects return to the same stack levels that they originally occupied. Commands that take no arguments leave the current saved arguments unchanged.

Note that when LAST follows a command that evaluates procedures (such as ∫, ∂, ISOL, EVAL, ROOT, and so on), the last arguments saved are from the procedure, not from the original command.

Low Memory

The HP-28C contains 2048 bytes of user memory, of which about 400 are reserved for system use, leaving about 1650 bytes for general use. Virtually every HP-28C operation requires some memory use—even interpreting the command line. The amount of memory used by some algebra commands (COLCT, EXPAN, TAYLR) increases rapidly as their arguments become more complicated.

To use the HP-28C effectively, keep in mind that it is a *calculator* for interactive problem solving. Its power is in its built-in operations, not in its capacity to store large databases or program libraries. Try to leave at least a few hundred bytes of memory free for dynamic system use.

Because the HP-28C operating system shares memory with user objects, you can fill memory so full of user objects that normal calculator operation becomes difficult or impossible. The HP-28C provides a series of low memory warnings and responses. In order of increasing severity—that is, decreasing free memory—these warnings are:

1. Insufficient Memory
2. No Room for UNDO
3. No Room to ENTER
4. Low Memory!
5. No Room to Show Stack
6. Out of Memory

Insufficient Memory

If there isn't enough memory available for a command to execute, the command halts and displays Insufficient Memory. If LAST is enabled, the original arguments are restored to the stack. If LAST is disabled, the arguments are lost.

No Room for UNDO

Suppose that UNDO is enabled and you have a 11 × 11 matrix on the stack. You can't perform even a simple operation such as NEG, because there isn't enough room in memory for both the original matrix and the resulting matrix. In such cases a No Room for UNDO error occurs, which automatically disables UNDO. You can then retry the operation that failed and later reenable UNDO.

No Room to ENTER

If there isn't enough memory available to process the command line, the calculator clears the command line and displays `No Room to ENTER`. A copy of the unsuccessful command line is saved in the command stack if the command stack is enabled.

If you're attempting to edit an existing object, using EDIT or VISIT, and a copy of the unsuccessful command line is saved in the command stack, purge the original copy of the object, press 🟥 COMMAND to recover the command line containing the edited object, and press ENTER to enter the edited version.

Low Memory!

If fewer than 128 bytes of free memory remain, `Low Memory!` flashes once in the top line of the display. This message will flash at every keystroke until additional memory is available. Clear unneeded objects from memory before continuing your calculations.

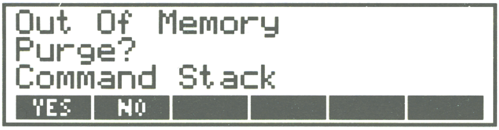
No Room To Show Stack

It is sometimes possible for the HP-28C to complete all pending operations, and not have enough free memory left for the normal stack display. In this case, the calculator displays `No Room to Show Stack` in the top line of the display. Those lines of the display that would normally display stack objects, now show those objects only by type, for example, `Real Number`, `Algebraic`, and so on.

The amount of memory required to display a stack object varies with the object type—algebraics usually require the most memory. Clear one or more objects from memory, or store a stack object as a variable so that it does not have to be displayed.

Out of Memory

The extreme case of low memory is when there is insufficient memory for the calculator to do anything—display the stack, show menu labels, build a command line, and so on. In this situation, you *must* clear some memory before continuing. A special `Out of Memory` procedure is activated, which will create a display:



```
Out Of Memory
Purge?
Command Stack
```

The calculator will sequentially prompt you to clear:

1. The COMMAND stack (if enabled).
2. The UNDO stack (if enabled).
3. LAST Arguments (if enabled).
4. The stack.
5. Each user variable, by name.

For each item that you want to purge, press the YES menu key; for those that you want to keep, press NO. After pressing YES at least once, you can try to terminate the `Out of Memory` procedure by pressing [ATTN]. If sufficient memory is available, the calculator returns to the normal display; otherwise, the calculator beeps and continues through the purge sequence. After cycling once through the choices, the `Out of Memory` procedure attempts to return to normal operation. If there still is not enough free memory, the procedure starts over with the sequence of choices to purge.

System Operations

There are special key combinations that interrupt normal HP-28C operation to perform system operations. These system operations include adjusting display contrast, halting endless program loops that do not respond to the [ON] key, resetting memory, or performing an electronic system test.

Attention: [ON]

Pressing [ON] clears the command line and displays the stack. If a procedure is executing, pressing [ON] halts the procedure. The result is similar to executing ABORT (described in "PROGRAM CONTROL").

Pressing 🟥 [OFF] turns the HP-28C off. When you next press [ON], the HP-28C will resume operation in the same state as when you turned it off. The HP-28C will turn itself off if it is left idle for 10 minutes.

Contrast Control: [ON] [+], [ON] [−]

You can change the HP-28C display contrast as follows:

1. Press and hold the [ON] key.
2. Press [+] to increase the contrast or press [−] to decrease the contrast. As long as you hold the [ON] key down, you can press [+] or [−] repeatedly or continuously, until you find the best contrast.
3. Release the [ON] key.

System Halt: [ON][▲]

To perform a *system halt* press the [ON] key and the [▲] key simultaneously, then release both. A system halt does the following:

- Stops all command or procedure execution.
- Clears any local variables.
- Clears the stack.
- Activates the cursor menu.
- Restarts normal keyboard operation.

The most common use of a system halt is to stop an "endless" name evaluation loop. Remember that evaluation of a name that refers to a second name causes evaluation of the second name. If the second name refers back to the first name, evaluating either name causes an endless loop. For example:

```
'Y' 'X' STO  'X' 'Y' STO X
```

results in an endless loop. Because name evaluation is critical for symbolic algebra, it is optimized for speed, and cannot be halted by the [ON] key. You must use a system halt to interrupt the loop.

Memory Reset: [ON][INS][▶]

To clear and reset the entire HP-28C memory:

1. Press and hold [ON].
2. Press and hold [INS] and [▶].
3. Release [INS] and [▶].
4. Release [ON].

A memory reset does the following:

- Stops all command or procedure execution.
- Clears any local variables.
- Purges all user variables.
- Clears the stack.
- Resets all user flags to their default values.
- Activates the cursor menu.
- Beeps and displays `Memory Lost` in display line 1.
- Restarts normal keyboard operation.

Cancel Reset: [ON][DEL]

If you initiate a system halt ([ON][▲]) or a memory reset ([ON][INS][▶]), the halt or reset does not take place until you release the [ON] key. You can cancel the pending action at any time before you release the [ON] key by doing as follows:

1. Release all keys except [ON].
2. Press and release [DEL].
3. Release the [ON] key.

System Test: [ON][▼], [ON][◀]

The HP-28C includes tests of its system electronics for manufacturing and service operations. You can begin the tests by pressing [ON] and [▼] simultaneously and then releasing them. Each time the calculator completes a test, as shown by a new display pattern, you can advance by pressing any key.

When the display shows KEYBOARD TEST, press [A] through [F], then [G] through [L], then [M] through [R], and so on. When you've completed the left-hand keyboard, test the right-hand keyboard beginning with [INS].

If the calculator successfully completes all tests, it displays OK-28C.

You can begin repeated tests by pressing [ON] and [◀] simultaneously. The calculator repeats the series of tests, skipping the keyboard test, until you press a key. The calculator then displays a FAIL message, indicating that its test was interrupted.

The system tests also perform a system halt ([ON][▲]).

Dictionary

All HP-28C operations (except for the special command SYSEVAL) are available on a key. Frequently used operations such as [ENTER], [+], and 🟥[√] are always available on the key showing their name. All other operations are available on the *menu keys*—the six keys at the top of the right-hand keyboard. (The variables you've created are also available on the menu keys, as described in "SOLVE" and "USER".)

Menus

The current assignment of each menu key appears on its *menu label*, the name displayed in inverse video directly above the key. If no labels are displayed, the cursor menu is active. The cursor operations are used in editing; they are described in the chapter on "Basic Operations."

Operations are grouped in *menus* according to a common application, such as trigonometry, or a common argument type, such as arrays. You select a menu by pressing a *menu selection key* such as [TRIG] or 🟥[ARRAY]. Each menu consists of *menu lines*—a set of six commands assigned to the menu keys at one time. Pressing a menu selection key assigns the first menu line to the menu keys. Pressing [NEXT] assigns the next menu line, eventually returning to the first menu line. Pressing 🟥[PREV] steps through the menu lines in the reverse order from [NEXT].

Pressing a menu key evaluates the assigned operation or adds its name to the command line. Pressing an unassigned menu key (indicated by a blank menu label) causes the calculator to beep. For a complete description, refer to the section on "Object Entry" in "Basic Operations."

The following table lists the menu selection key and description for each menu in the HP-28C.

Menu Selection Key	Description
◀▲▼▶	Cursor movement, editing operations.
🟥 ALGEBRA	Algebra commands.
🟥 ARRAY	Vector and matrix commands.
🟥 BINARY	Integer arithmetic, base conversions, bit manipulations.
🟥 BRANCH	Program branch structures.
🟥 CMPLX	Complex number commands.
🟥 CTRL	Program control, halt, and single-step operations.
🟥 LIST	List commands.
🟥 LOGS	Logarithmic and exponential functions, hyperbolic functions.
🟥 MODE	Display, angle, recovery, and radix mode selection.
🟥 PLOT	Plotting commands.
🟥 PRINT	Printing commands.
🟥 REAL	Real number commands.
SOLV	Numeric and symbolic solution commands, the Solver.
🟥 STACK	Stack manipulation commands.
🟥 STAT	Statistics and probability commands.
🟥 STORE	Storage arithmetic commands, in-place matrix commands.
🟥 STRING	Character string commands.
🟥 TEST	Flag commands and logical test functions.
TRIG	Trigonometric functions, rectangular/polar conversion, degrees/radians conversion, and Hour/Minute/Second arithmetic commands.
USER	Variables, user-memory commands.

ALGEBRA

COLCT	EXPAN	SIZE	FORM	OBSUB	EXSUB
TAYLR	ISOL	QUAD	SHOW	OBGET	EXGET

Algebraic Objects

An algebraic object is a procedure that is entered and displayed in mathematical form. It can contain numbers, variable names, functions, and operators, defined as follows:

Number: A real number or a complex number.

Variable name: Any name, whether or not there is currently a variable associated with the name. We will use the term *formal variable* to refer to a name that is not currently associated with a user variable. When such a name is evaluated, it returns itself.

Function: An HP-28C command that is allowed in an algebraic procedure. Functions must return exactly one result. If one or more of a function's arguments are algebraic objects, the result is algebraic. Most functions appear as a function name followed by one or more arguments contained within parentheses; for example, `'SIN(X)'`.

Operator: A function that generally doesn't require parentheses around its arguments. The operators NOT, √, and NEG (which appears in algebraics as the unary − sign) are *prefix* operators: their names appear before their arguments. The operators +, −, *, /, ^, =, ==, ≠, <, >, ≤, ≥, AND, OR, and XOR are *infix* operators: their names appear between their two arguments.

...ALGEBRA

Precedence

The *precedence* of operators determines the order of evaluation when expressions are entered without parentheses. The operations with higher precedence are performed first. Expressions are evaluated from left to right for operators with the same precedence. The following lists HP-28C algebraic functions in order of precedence, from highest to lowest:

1. Expressions within parentheses. Expressions within nested parentheses are evaluated from the inside out.
2. Functions such as SIN, LOG, and FACT, which require arguments in parentheses.
3. Power (^) and square root (√).
4. Negation (−), multiplication (*), and division (/).
5. Addition (+) and subtraction (−).
6. Relational operators (==, ≠, <, >, ≤, ≥).
7. AND and NOT.
8. OR and XOR.
9. =

Algebraic objects and programs have identical internal structures. Both types of procedures are sequences of objects that are processed sequentially when the procedures are evaluated. The algebraic `'X+Y'` and the program `« X Y + »` are both stored as the same sequence (the RPN form). Algebraics are "marked" as algebraics so that they will be displayed as mathematical expressions and to indicate that they satisfy algebraic syntax rules.

Algebraic Syntax and Subexpressions

A procedure obeys *algebraic syntax* if, when evaluated, it takes no arguments from the stack and returns exactly one argument to the stack, and if it can be subdivided completely into a hierarchy of *subexpressions*. A subexpression can be a number, a name, or a function and its arguments. By *hierarchy*, we mean that each subexpression can itself be an argument of a function. For example, consider the expression:

```
'1-SIN(X+Y)'
```

The expression contains one number, `1`, and two names, `X` and `Y`, each of which can be considered as a simple subexpression. The expression also contains three functions, `+`, `-`, and `SIN`, each of which defines a subexpression along with its arguments. The arguments of `+` are `X` and `Y`; `X+Y` is the argument of `SIN`, and `1` and `SIN(X+Y)` are the arguments of `-`. The hierarchy becomes more obvious if the expression with its operators is rewritten as ordinary functions (*Polish notation*):

$$-(1, \text{SIN}\ (+(X,\ Y)))$$

An object or subexpression within an expression is characterized by its *position* and *level*.

The *position* of an object is determined by counting from left to right in the expression. For example, in the expression `'1-SIN(X+Y)'`, `1` has position 1, `-` has position 2, `SIN` has position 3, and so on.

The position of a subexpression is the position of the object that defines the subexpression. In the same example, `'SIN(X+Y)'` has position 3, since it is defined by `SIN` in position 3.

The *level* of an object within an algebraic expression is the number of pairs of parentheses surrounding the object when the expression is written in purely functional form. For example, in the expression '1-SIN(X+Y)', - has level 0, 1 and SIN have level 1, + has level 2, and X and Y have level 3. Every algebraic expression has exactly one level 0 object.

(User-defined functions are an apparent exception to the rule for determining the levels of a subexpression. In the expression 'F(A,B)', for example, where F is a user-defined function, F, A, and B are all at level 1; there is no explicit level 0 function. This is because F and its arguments A and B are all arguments for a special "invisible" function that provides display and evaluation logic for user-defined functions.)

If we take the above expression and rewrite it again, by removing the parentheses, and placing the functions after their arguments, we obtain the RPN form of the expression:

```
1 X Y + SIN -
```

This defines a *program* that has algebraic syntax, and is effectively equivalent to the corresponding algebraic object. Programs, however, are more flexible than algebraic objects; for example, we could insert a DUP anywhere in the above program and still have a valid program, but it would no longer obey algebraic syntax. Since DUP takes one argument and returns two, it cannot define or be part of an algebraic subexpression.

Equations

An algebraic *equation* is an algebraic object containing two expressions combined with an equals sign (=). Mathematically, the equals sign implies the equality of the two subexpressions on either side of the sign. In the HP-28C, = is a function of two arguments. It is displayed as an infix operator, separating the two subexpressions that are its arguments. Internally, an equation is an expression with = as its level 0 object.

When an equation is numerically evaluated, = is equivalent to −. This feature allows expressions and equations to be used interchangeably as arguments for symbolic and numerical rootfinders. An equation is equivalent to an expression with = replaced by −, and an expression is equivalent to the left side of an equation in which the right side is zero.

When an equation is an argument of a function, the result is also an equation, where the function has been applied to both sides. Thus

`'X=Y' SIN` returns `'SIN(X)=SIN(Y)'`.

Conventional mathematical usage of the equals sign = is ambiguous. The equals sign is used to equate two expressions, as in $x + \sin y = 2z + t$. This type of equation is suitable for solving, that is, adjusting one or more variables to achieve the equality of the two sides.

The equals sign is also used to assign a value to a variable, as in $x = 2y + z$. This equation means that the symbol x is a substitution for the longer expression $2y + z$; it is meaningless to "solve" this equation.

The ambiguity of the equals sign is compounded by certain computer languages such as BASIC, where "=" means "replace by," as in X = Y + Z. Such notation doesn't imply a mathematical equation at all.

In the HP-28C, the equals sign always means equating two expressions, such that solving the equation is equivalent to making the difference between the two expressions zero. (Assignment is performed by STO, which is strictly a postfix command that takes two arguments.)

= *Equal* Analytic

Level 2	Level 1		Level 1
z_1	z_2	➡	' $z_1=z_2$ '
z	' *symb* '	➡	' *z=symb* '
' *symb* '	*z*	➡	' *symb=z* '
' $symb_1$ '	' $symb_2$ '	➡	' $symb_1=symb_2$ '

This function combines two arguments, which must be names, expressions, real numbers or complex numbers.

If the HP-28C is in symbolic evaluation mode (flag 36 set), the result is an algebraic equation, with the level 2 argument on the left side of the equation, and the level 1 argument on the right.

If the HP-28C is in numerical evaluation mode (flag 36 clear), the result is the numerical difference of the two arguments. In effect, = acts as the − operator in numerical evaluation mode.

Functions of Symbolic Arguments

Function Evaluation Mode

Symbolic Evaluation Mode (flag 36 set). In symbolic evaluation mode, functions return symbolic results if their arguments are symbolic. This is the default evaluation mode. For example:

```
        'X' SIN returns 'SIN(X)'
    'X^2+5' LN returns 'LN(X^2+5)'.
        3 'X' + returns '3+X'
    2 'X' + SIN returns 'SIN(2+X)'.
  'X' 1 2 IFTE returns 'IFTE(X,1,2)'.
```

Numeric Evaluation Mode (flag 36 clear). In numeric evaluation mode, each function attempts to convert symbolic arguments to data objects. Once the arguments are converted to numbers, the function is applied to those arguments, returning a numeric result. The arguments are repeatedly evaluated until they become data objects or formal variables. If the final arguments are formal variables, an `Undefined Name` error occurs.

Automatic Simplification

Certain functions, when evaluated, replace certain arguments or combinations of arguments with simpler forms. For example, when `'1*X'` is evaluated, the `*` function detects that one of its arguments is a `1`, so the expression is replaced by `'X'`. Automatic simplification occurs in the following cases:

Original Expression	Simplified Expression
Negation, Inverse, Square	
`-(-X)`	`X`
`INV(INV(X))`	`X`
`SQ(√X)`	`X`
`SQ(X^Y)`	`X^(Y*2)`
`SQ(i)`	`-1`
Addition and Subtraction	
`0+X` or `X+0`	`X`
`X-0`	`X`
`0-X`	`-X`
`X-X`	`0`
Multiplication	
`X*0` or `0*X`	`0`
`X*1` or `1*X`	`X`
`X*(-1)` or `-1*X`	`-X`
`-X*(-1)` or `-1*(-X)`	`X`
`i*i`	`-1`
`-X*INV(Y)`	`-(X/Y)`
`-X*Y`	`-(X*Y)`
`X*INV(Y)`	`X/Y`

Original Expression	Simplified Expression
Division	
X/1	X
0/X	0
1/INV(X)	X
1/X	INV(X)
Power	
1^X	1
X^0	1
X^1	X
(√X)^2	X
INV(X)^(-1)	X
X^(-1)	INV(X)
i^2	-1 or (-1,0)*
i^(2,0)	(-1,0)
SIN, COS, TAN	
SIN(ASIN(X))	X
SIN(-X)	-SIN(X)
SIN(π)	0†
SIN(π/2)	1†
COS(ACOS(X))	X
COS(-X)	COS(X)
COS(π)	-1†
COS(π/2)	0†
TAN(ATAN(X))	X
TAN(-X)	-TAN(X)
TAN(π)	0†
ABS, MAX, MIN, MOD, SIGN	
ABS(ABS(X))	ABS(X)
ABS(-X)	ABS(X)
MAX(X,X)	X
MIN(X,X)	X
MOD(X,0)	X
MOD(0,X)	0
MOD(X,X)	0
MOD(MOD(X,Y),Y)	MOD(X,Y)
SIGN(SIGN(X))	SIGN(X)

* Depends on symbolic evaluation mode (flag 36 set) or numerical evaluation mode (flag 36 clear).

† Applies only when the angle mode is radians.

Original Expression	Simplified Expression
ALOG, EXP, EXPM, SINH, COSH, TANH	
`ALOG(LOG(X))`	`X`
`EXP(LN(X))`	`X`
`EXPM(LNP1(X))`	`X`
`SINH(ASINH(X))`	`X`
`COSH(ACOSH(X))`	`X`
`TANH(ATANH(X))`	`X`
IM, RE, CONJ	
`IM(IM(X))`	`0`
`IM(RE(X))`	`0`
`IM(CONJ(X))`	`-IM(X)`
`IM(i)`	`1`
`RE(RE(X))`	`RE(X)`
`RE(IM(X))`	`IM(X)`
`RE(CONJ(X))`	`RE(X)`
`RE(i)`	`0`
`CONJ(CONJ(X))`	`X`
`CONJ(RE(X))`	`RE(X)`
`CONJ(IM(X))`	`IM(X)`
`CONJ(i)`	`-i`

Functions of Equations

Functions applied to equations in symbolic evaluation mode return equations as results.

If a function of one argument is applied to an equation, the result is an equation obtained by applying the function separately to the left and right sides of the argument equation. For example:

`'X+2=Y' SIN` returns `'SIN(X+2)=SIN(Y)'`.

If both arguments of a two argument function are equations, the result is an equation derived by equating the expressions obtained by applying the function separately with the two left sides of the equation as arguments, and with the two right sides. For example:

`'X+Y=Z+T' 'SIN(Q)=5' +` returns `'X+Y+SIN(Q)=Z+T+5'`.

If one argument of a two argument function is a numeric object or an algebraic expression, and the other is an equation, the former is converted to an identity equation with the original object on both sides. Then the function acts as in the case where both arguments are equations. For example:

`'X=Y' 3 -` returns `'X-3=Y-3'`.

These properties define the behavior of algebraic objects when they are evaluated (see the next section) as well as allow you to perform algebraic calculations in an interactive RPN style, much as you carry out ordinary numerical calculations.

Evaluation of Algebraic Objects

Evaluation of algebraic objects is a powerful feature of the HP-28C that allows you to consolidate expressions by carrying out explicit numerical calculations, and substitute numbers or expressions for variables. In order to understand what to expect when you evaluate an algebraic object remember that an algebraic object is equivalent to a program, and that evaluating a program means to put each object in the program on the stack and, if the object is a command or name, evaluate the object.

To demonstrate what this means, let us suppose that we have defined variable X to have the value 3 (that is, `3 'X' STO`), Y to have the value 4, and Z to have the value `'X+T'`. We will also assume that symbolic evaluation mode (flag 36) is set, so that functions will accept symbolic arguments.

First consider the expression 'X+Y'. When we evaluate this expression ('X+Y' EVAL), we obtain the result 7. Here's why: Internally, 'X+Y' is represented as X Y +. So when 'X+Y' is evaluated, X, Y, and + are evaluated in sequence:

1. Since X is a name, evaluating it is equivalent to evaluating the object stored in the variable X, the number 3. Evaluating X puts 3 in level 1.
2. Similarly, evaluating Y puts 4 in level 1, pushing the 3 into level 2.
3. Now + is evaluated, with the numeric arguments 3 and 4 on the stack. This drops the 3 and the 4, and returns the numeric result 7.

Now try evaluating 'X+T':

1. Evaluating X puts 3 in level 1.
2. T is a name not associated with a variable, so it just returns itself to level 1, pushing the 3 into level 2.
3. This time + has 3 and T as arguments; since T is symbolic, + returns an algebraic result, '3+T'.

Finally, consider evaluating 'X+Y+Z'. Internally, this expression is represented as X Y + Z +. Following the same logic as in the above examples, evaluation gives the result '7+X+T'. We can evaluate this result again and obtain the new result '10+T'. Further evaluation makes no additional changes, since T has no value.

Notice that evaluating 'X+T+Y' (keeping the same values as above) returns '3+T+4', not '7+T' or 'T+7'. The values 3 and 4 obtained by evaluating X and Y are not arguments to the same + operator in the expression, and hence are not combined. If you want to combine the 3 and the 4, you can use either the COLCT command for automatic collection of terms, or the FORM command for more general rearrangement of the expression.

Symbolic Constants: e, π, i, MAXR, and MINR

There are five built-in algebraic objects that return a numerical representation of certain constants. These objects have the special property that their evaluation is controlled by symbolic constants evaluation mode (flag 35) as well as by the functions evaluations mode (flag 36). When flag 36 is clear (numeric evaluation mode), these objects evaluate to their numerical values (regardless of constants evaluation mode (flag 35)). When flag 36 is set (symbolic evaluation mode):

- If flag 35 is clear, these objects will evaluate to their numeric values. For example:

  `'2*i' EVAL` returns `(0,2)`.

- If flag 35 is set, these objects will retain their symbolic form when evaluated. For example:

  `'2*i' EVAL` returns `'2*i'`.

The following table lists the five objects and their numerical values.

HP-28C Symbolic Constants

Object Name	Numerical Value
e	2.71828182846
π	3.14159265359
i	(0.00000000000,1.00000000000)
MAXR	9.99999999999E499
MINR	1.00000000000E-499

The numerical values of `e` and `π` are the closest approximations of the constants ϵ and π that can be expressed with 12-digit accuracy. The numerical value of `i` is the exact representation of the constant i. `MAXR` and `MINR` are the largest and smallest non-zero numerical values that can be represented by the HP-28C.

For greater numerical accuracy, use the expression `'EXP(X)'` rather than the expression `'e^X'`. The function EXP uses a special algorithm to compute the exponential to greater accuracy.

When the angle mode is radians and flags 35 and 36 are set, trigonometric functions of `π` and `π/2` are automatically simplified. For example, evaluating `'SIN(π)'` gives a result of 0.

COLCT EXPAN SIZE FORM OBSUB EXSUB

These commands alter the form of algebraic expressions, much as you might if you were dealing with the expressions "on paper". COLCT, EXPAN, and FORM are identity operations, that is, they change the form of an expression without changing its value. OBSUB and EXSUB allow you to alter the value of an expression by substituting new objects or subexpressions into the expression.

COLCT *Collect Terms* Command

Level 1		Level 1
'$symb_1$'	➡	'$symb_2$'

COLCT rewrites an algebraic object so that it is simplified by "collecting" like terms. Specifically, COLCT:

- Evaluates numerical subexpressions. For example: `'1+2+LOG(10)'` is replaced by 4.
- Collects numerical terms. For example: `'1+X+2'`is replaced by `'3+X'`.
- Orders factors (arguments of *), and combines like factors. For example: `'X^Z*Y*X^T*Y'` is replaced by `'X^(T+Z)*Y^2'`.
- Orders summands (arguments of +), and combines like terms differing only in a numeric coefficient. For example: `'X+X+Y+3*X'` is replaced by `'5*X+Y'`.

COLCT operates separately on the two sides of an equation, so that like terms on opposite sides of the equation are not combined.

The ordering (that is, whether X precedes Y) algorithm used by COLCT was chosen for speed of execution rather than conforming to any obvious or standard forms. If the precise ordering of terms in a resulting expression is not what you desire, you can use FORM to rearrange the order.

EXPAN *Expand Products* Command

Level 1		Level 1
'*symb*$_1$'	➡	'*symb*$_2$'

EXPAN rewrites an algebraic object by expanding products and powers. More specifically, EXPAN:

- Distributes multiplication and division over addition. For example: `'A*(B+C)'` expands to `'A*B+A*C'`; `'(B+C)/A'` expands to `'B/A+C/A'`.
- Expands powers over sums. For example: `'A^(B+C)'` expands to `'A^B*A^C'`.
- Expands positive integer powers. For example: `'X^5'` expands to `'X*X^4'`. The square of a sum `'(X+Y)^2'` or `'SQ(X+Y)'` is expanded to `'X^2+2*X*Y+Y^2'`.

EXPAN does not attempt to carry out all possible expansions of an expression in a single execution. Instead, EXPAN works down through the subexpression hierachy, stopping in each branch of the hierarchy when it finds a subexpression that can be expanded. It first examines the level 0 subexpression; if that is suitable for expansion, it is expanded and EXPAN stops. If not, EXPAN examines each of the level 1 subexpressions. Any of those that are suitable are expanded; in the remainder, the level 2 subexpressions are examined. This process continues down through the hierarchy until an expansion halts further searching down each branch. For example:

Expand the expression `'A^(B*(C^2+D))'`.

1. The level 0 operator is the left ^. Since it cannot be expanded, the level 1 operator * is examined. One of its arguments is a sum, so the product is distributed yielding:

   ```
   'A^(B*C^2+B*D)'
   ```

2. The level 0 operator is still the left ^, but now its power is a sum, so the power is expanded over the sum when EXPAN is executed again:

   ```
   'A^(B*C^2)*A^(B*D)'
   ```

3. One more expansion is possible. The level 0 operator is now the middle ✱. Since it cannot be expanded, the level 1 operators, the outside ^'s, are examined. They cannot be expanded, so the level 2 operators, the outside ✱'s, are examined. Since they cannot be expanded, the level 3 operator, the middle ^, is examined. Its power is a positive integer, so the power is expanded:

```
'A^(B*(C*C))*A^(B*D)'
```

SIZE *Size* Command

Level 1		Level 1
"*string*"	➡	*n*
{ *list* }	➡	*n*
[*array*]	➡	{ *list* }
'*symb*'	➡	*n*

SIZE returns the number of objects that comprise an algebraic object.

Refer to "ARRAY," "LIST," and "STRING" for the use of SIZE with other object types.

FORM *Form Algebraic Expression* Command

Level 1		Level 3	Level 2	Level 1
'$symb_1$'	➡			'$symb_2$'
'$symb_1$'	➡	'$symb_2$'	*n*	'$symb_3$'

FORM is an interactive expression editor that enables you to rearrange an algebraic expression or equation according to standard rules of mathematics. Its operation is described in the next section, "ALGEBRA (FORM)."

OBSUB *Object Substitute* Command

Level 3	Level 2	Level 1		Level 1
'*symb*$_1$'	*n*	{ *obj* }	➧	'*symb*$_2$'

OBSUB substitutes an object in the specified position of an algebraic object. The object is the contents of a list in level 1, the position *n* is in level 2, and the algebraic object is in level 3. For example:

`'A*B' 3 { C } OBSUB` returns `'A*C'`.

You can substitute functions as well as user variables. For example:

`'A*B' 2 { + } OBSUB` returns `'A+B'`.

EXSUB *Expression Substitute* Command

Level 3	Level 2	Level 1		Level 1
'*symb*$_1$'	*n*	'*symb*$_2$'	➧	'*symb*$_3$'

EXSUB substitutes the algebraic (or name) '*symb*$_2$' for the subexpression in the *n*th position of the algebraic '*symb*$_1$' and returns the result expression '*symb*$_3$'. The *n*th subexpression consists of the *n*th object in an algebraic object definition plus the arguments, if any, of the object. For example:

`'(A+B)*C' 2 'E^F' EXSUB` returns `'E^F*C'`.

...ALGEBRA

TAYLR ISOL QUAD SHOW OBGET EXGET

TAYLR is described in "Calculus," along with ∂ and ∫. ISOL, QUAD, and SHOW are described in "SOLV."

OBGET *Object Get* Command

Level 2	Level 1		Level 1
'symb'	*n*	➡	*{ obj }*

OBGET returns the object in the *n*th position of the algebraic object *symb* in level 2. The object is returned as the only object in a list. For example:

`'(A+B)*C' 2 OBGET` returns `{ + }`.

If *n* exceeds the number of objects, OBGET returns the level 0 object.

EXGET *Expression Get* Command

Level 2	Level 1		Level 1
'$symb_1$'	*n*	➡	'$symb_2$'

EXGET returns the subexpression in the *n*th position of the algebraic $symb_1$ in level 2. The *n*th subexpression consists of the *n*th object in an algebraic object definition plus the arguments, if any, of the object. For example:

`'(A+B)*C' 2 EXGET` returns `'A+B'`.

If *n* exceeds the number of objects, EXGET returns the level 0 subexpression.

FORM — *Form Algebraic Expression* — Command

	Level 1		Level 3	Level 2	Level 1
	'$symb_1$'	➡			'$symb_2$'
	'$symb_1$'	➡	'$symb_2$'	n	'$symb_3$'

FORM is an interactive expression editor that enables you to rearrange an algebraic expression or equation according to standard rules of mathematics. All of FORM's mathematical operations are identities; that is, the result expression $symb_2$ will have the same value as the original argument expression $symb_1$, even though the two may have different forms. For example, with FORM you can rearrange `'A+B'` to `'B+A'`, which changes the form but not the value of the expression.

A variation of the command EXGET is available while FORM is active. It allows you to duplicate a subexpression $symb_3$ contained in $symb_1$, and return $symb_3$ and its position n to the stack.

When FORM is executed, the normal stack display is replaced by a special display of the algebraic object, along with a menu of FORM operations at the bottom of the display. The special display initially starts in line two of the display (second from top), and wraps into line three if the object is too long to display in a single line. If the object requires more than two display lines, you will have to move the FORM cursor through the object to view the remainder.

...ALGEBRA (FORM)

To exit FORM and continue with other calculator operations, press [ON]. Alternatively, you can press the EXGET menu key, which also returns the selected subexpression *symb*$_3$ and its position *n* to the stack.

The FORM cursor highlights an individual object in the expression display. (It is not a character cursor like that of the command line.) The highlighted object appears as white characters against a black background. The cursor identifies both the *selected object*, which is highlighted, and the *selected subexpression*, which is the subexpression consisting of the selected object and its arguments, if any.

You can move the cursor to the left or right in the expression by pressing the [←] or [→] keys in the menu; when the cursor moves, it moves directly from object to object, skipping any intervening parentheses. The cursor is always in line two of the display. If you attempt to move the cursor past the right end of line two, the expression scrolls up one line in the display, and the cursor moves back to the left end of line two. Similarly, if you try to move the cursor past the left end of line two, the expression scrolls down one line, and the cursor moves to the right end of line two.

The expression display differs from the normal stack algebraic object display by inserting additional parentheses in order to make all operator precedence explicit. This feature helps you identify the selected subexpression associated with the selected object as shown by the cursor. This is important, since all FORM menu operations operate on the selected subexpression.

While FORM is active, a special set of operations is available as menu keys. The initial menu contains six operations common to all subexpressions. Additional menus of operations are available via the [NEXT] and ■ [PREV] keys; the contents of the additional menus vary according to the selected object. Only those operations that apply to the selected object are shown.

You can reactivate the first six menu keys at any time by pressing [ENTER].

FORM Operations

In the following subsections, all of the operations that can appear in the FORM menus will be described. The descriptions consist primarily of examples of the "before" and "after" structures of the selected subexpressions relevant to each operation. Each possible operation is represented by an example like this:

←D Distribute to the left.

Before	After
((A+B)*C)	((A*C)+(B*C))

For simplicity variable names such as A, B, and C will be used, but each of these can represent a general object or subexpression. The example shows that applying ←D (distribute to the left) to '(A+B)*C' returns 'A*C+B*C'.

Individual FORM operations appear in the FORM menu when they are relevant for the selected object. For example, ←D appears in the menu when + is the selected object, but not when SIN is selected. Furthermore, if an operation does appear, you will be able to execute it only if it applies to the selected subexpression. For example, D→ appears when * is the selected object, since distribution is a property of multiplication. However, the menu key is inactive (it will just beep if pressed) unless the subexpression is of the form '(A+B)*C' or '(A-B)*C', which can be distributed.

...ALGEBRA (FORM)

The initial FORM menu contains the following operations:

Operations Common to All Subexpressions

Operation	Description
COLCT	Collects like terms in the selected subexpression. This operation works the same as the command COLCT except that its action is restricted to the selected subexpression. The FORM cursor is repositioned to the beginning of the expression display.
EXPAN	Expands products and powers in the selected subexpression. This operation works the same as the command EXPAN except that its action is restricted to the current subexpression. The FORM cursor is repositioned to the beginning of the expression display.
LEVEL	Displays the level of the selected object or its associated selected subexpression. The level is displayed as long as you hold down the LEVEL key.
EXGET	Exits FORM, leaving the current version of the edited expression in level 3, a copy of the selected subexpression in level 1, and its position in level 2.
[←]	Moves the FORM cursor to the previous object (to the left) in the expression.
[→]	Moves the FORM cursor to the next object (to the right) in the expression.

Commutation, Association, and Distribution

←→ Commute the arguments of an operator.

Before	After
`(A+B)`	`(B+A)`
`(-(A)+B)`	`(B-A)`
`(A-B)`	`(-(B)+A)`
`(A*B)`	`(B*A)`
`(INV(A)*B)`	`(B/A)`
`(A/B)`	`(INV(B)*A)`

←A Associate to the left. The arrow indicates the direction in which the parentheses will "move."

Before	After
`(A+(B+C))`	`((A+B)+C)`
`(A+(B-C))`	`((A+B)-C)`
`(A-(B+C))`	`((A-B)-C)`
`(A-(B-C))`	`((A-B)+C)`
`(A*(B*C))`	`((A*B)*C)`
`(A*(B/C))`	`((A*B)/C)`
`(A/(B*C))`	`((A/B)/C)`
`(A/(B/C))`	`((A/B)*C)`
`(A^(B*C))`	`((A^B)^C)`

...ALGEBRA (FORM)

A→ **Associate to the right.** The arrow indicates the direction in which the parentheses will "move."

Before	After
((A+B)+C)	(A+(B+C))
((A-B)+C)	(A-(B-C))
((A+B)-C)	(A+(B-C))
((A-B)-C)	(A-(B+C))
((A*B)*C)	(A*(B*C))
((A/B)*C)	(A/(B/C))
((A*B)/C)	(A*(B/C))
((A/B)/C)	(A/(B*C))
((A^B)^C)	(A^(B*C))

→() **Distribute prefix operator.**

Before	After
-(A+B)	(-(A)-B)
-(A-B)	(-(A)+B)
-(A*B)	(-(A)*B)
-(A/B)	(-(A)/B)
-(LOG(A))	LOG(INV(A))
-(LN(A))	LN(INV(A))
INV(A*B)	(INV(A)/B)
INV(A/B)	(INV(A)*B)
INV(A^B)	(A^-(B))
INV(ALOG(A))	ALOG(-(A))
INV(EXP(A))	EXP(-(A))

...ALGEBRA (FORM)

Note that any time an expression is rewritten, the sequence * INV is collapsed to /. Similarly, +− is replaced by −.

←D **Distribute to the left.** The arrow points to the subexpression that is distributed.

Before	After
((A+B)*C)	((A*C)+(B*C))
((A-B)*C)	((A*C)-(B*C))
((A+B)/C)	((A/C)+(B/C))
((A-B)/C)	((A/C)-(B/C))
((A*B)^C)	((A^C)*(B^C))
((A/B)^C)	((A^C)/(B^C))

D→ **Distribute to the right.** The arrow points to the subexpression that is distributed.

Before	After
(A*(B+C))	((A*B)+(A*C))
(A*(B-C))	((A*B)-(A*C))
(A/(B+C))	INV((INV(A)*B)+(INV(A)*C))
(A/(B-C))	INV((INV(A)*B)-(INV(A)*C))
(A^(B+C))	((A^B)*(A^C))
(A^(B-C))	((A^B)/(A^C))
LOG(A*B)	(LOG(A)+LOG(B))
LOG(A/B)	(LOG(A)-LOG(B))
ALOG(A+B)	(ALOG(A)*ALOG(B))
ALOG(A-B)	(ALOG(A)/ALOG(B))
LN(A*B)	(LN(A)+LN(B))

...ALGEBRA (FORM)

(Continued)

Before	After
LN(A/B)	(LN(A)-LN(B))
EXP(A+B)	(EXP(A)*EXP(B))
EXP(A-B)	(EXP(A)/EXP(B))

←M **Merge left factors.** This operation merges arguments of +, −, *, and /, where the arguments have a common factor or a common single-argument function EXP, ALOG, LN, or LOG. In the case of common factors, the arrow indicates that the left-hand factors are common.

Before	After
((A*B)+(A*C))	(A*(B+C))
((A*B)-(A*C))	(A*(B-C))
((A^B)*(A^C))	(A^(B+C))
((A^B)/(A^C))	(A^(B-C))
(LN(A)+LN(B))	LN(A*B)
(LN(A)-LN(B))	LN(A/B)
(LOG(A)+LOG(B))	LOG(A*B)
(LOG(A)-LOG(B))	LOG(A/B)
(EXP(A)*EXP(B))	EXP(A+B)
(EXP(A)/EXP(B))	EXP(A-B)
(ALOG(A)*ALOG(B))	ALOG(A+B)
(ALOG(A)/ALOG(B))	ALOG(A-B)

M→ **Merge right factors.** This operation merges arguments of +, −, ∗, and /, where the arguments have a common factor. The arrow indicates that the right-hand factors are common.

Before	After
((A*C)+(B*C))	((A+B)*C)
((A/C)+(B/C))	((A+B)/C)
((A*C)-(B*C))	((A-B)*C)
((A/C)-(B/C))	((A-B)/C)
((A^C)*(B^C))	((A*B)^C)
((A^C)/(B^C))	((A/B)^C)

Double-Negation and Double-Inversion

DNEG **Double-negate.** Negate a subexpression twice.

Before	After
A	-(-(A))

...ALGEBRA (FORM)

-() **Double-negate and distribute.** This operation is equivalent to a double negate DNEG followed by distribution -() of the resulting inner negation.

Before	After
(A+B)	-(-(A)-B)
(A-B)	-(-(A)+B)
(-(A)-B)	-(A+B)
(A*B)	-(-(A)*B)
(-(A)*B)	-(A*B)
(-(A)/B)	-(A/B)
(A/B)	-(-(A)/B)
LOG(A)	-(LOG(INV(A)))
LOG(INV(A))	-(LOG(A))
LN(A)	-(LN(INV(A)))
LN(INV(A))	-(LN(A))

DINV **Double-invert.** Invert a subexpression twice.

Before	After
A	INV(INV(A))

1/() **Double-invert and distribute.** This operation is equivalent to double inversion DINV followed by distribution →() of the resulting inner INV:

Before	After
(A*B)	INV(INV(A)/B)
(A/B)	INV(INV(A)*B)
(A^B)	INV(A^-(B))
(A^-(B))	INV(A^B)
ALOG(A)	INV(ALOG(-(A)))
ALOG(-(A))	INV(ALOG(A))
EXP(A)	INV(EXP(-(A)))
EXP(-(A))	INV(EXP(A))

Identities

*1 **Multiply by 1.**

Before	After
A	A*1

/1 **Divide by 1.**

Before	After
A	A / 1

...ALGEBRA (FORM)

^1 **Raise to the power 1.**

Before	After
A	A ^ 1

+1-1 **Add 1 and subtract 1.**

Before	After
A	(A + 1) - 1

Rearrangement of Exponentials

L* **Replace log-of-power with product-of-log.**

Before	After

LOG(A^B)
LN(A^B)
	(LOG(A)*B)
	(LN(A)*B)

L() **Replace product-of-log with log-of-power.**

Before	After

(LOG(A)*B)
(LN(A)*B)
	LOG(A^B)
	LN(A^B)

...ALGEBRA (FORM)

E^ **Replace power-product with power-of-power.**

Before	After
ALOG(A*B)	(ALOG(A)^B)
ALOG(A/B)	(ALOG(A)^INV(B))
EXP(A*B)	(EXP(A)^B)
EXP(A/B)	(EXP(A)^INV(B))

E() **Replace power-of-power with power-product.**

Before	After
(ALOG(A)^B)	ALOG(A*B)
(ALOG(A)^INV(B))	ALOG(A/B)
(EXP(A)^B)	EXP(A*B)
(EXP(A)^INV(B))	EXP(A/B)

Adding Fractions

AF **Combine over a common denominator.**

Before	After
(A+(B/C))	(((A*C)+B)/C)
((A/B)+C)	((A+(B*C))/B)
((A/B)+(C/D))	(((A*D)+(B*C))/(B*D))
(A-(B/C))	(((A*C)-B)/C)
((A/B)-C)	((A-(B*C))/B)
((A/B)-(C/D))	(((A*D)-(B*C))/(B*D))

If the denominator is already common between two fractions, use M→ .

...ALGEBRA (FORM)

FORM Operations Listed by Function

The following tables show which operations will appear in the FORM menu when a given function is the selected object. The form of the original subexpression and the result is shown for each operation.

The operations COLCT, EXPAN, LEVEL, DNEG, DINV, *1, /1, and +1-1 are available for all functions and variables. These common operations don't appear in the tables. If only the common operations are available for a function, no table appears for that function. (Only the common operations are available for √ and SQ; to use other operations, substitute ^.5 and ^2.)

Addition (+)

Operation	Before	After
←→	(A+B) (-(A)+B)	(B+A) (B-A)
←A	(A+(B+C)) (A+(B-C))	((A+B)+C) ((A+B)-C)
A→	((A+B)+C) ((A-B)+C)	(A+(B+C)) (A-(B-C))

←M	((A*B)+(A*C))
	(LN(A)+LN(B))
	(LOG(A)+LOG(B))
		(A*(B+C))
		LN(A*B)
		LOG(A*B)

M→	((A*C)+(B*C)) ((A/C)+(B/C))	((A+B)*C) ((A+B)/C)
-()	(A+B) -(A)+B	-(-(A)-B) -(A-B)
AF	(A+(B/C)) ((A/B)+(C/D)) ((A/B)+C)	(((A*C)+B)/C) (((A*D)+(B*C))/(B*D)) ((A+(B*C))/B)

Subtraction (−)

Operation	Before	After
←→	(A-B)	(-(B)+A)
←A	(A-(B+C)) (A-(B-C))	((A-B)-C) ((A-B)+C)
A→	((A+B)-C) ((A-B)-C)	(A+(B-C)) (A-(B+C))

←M	((A*B)-(A*C))
	(LN(A)-LN(B))
	(LOG(A)-LOG(B))
		(A*(B-C))
		LN(A/B)
		LOG(A/B)

M→	((A*C)-(B*C)) ((A/C)-(B/C))	((A-B)*C) ((A-B)/C)
-()	(A-B) (-(A)-B)	-(-(A)+B) -(A+B)
AF	(A-(B/C)) ((A/B)-C) ((A/B)-(C/D))	(((A*C)-B)/C) ((A-(B*C))/B) (((A*D)-(B*C))/(B*D))

Multiplication (*)

Operation	Before	After
←→	(A*B) (INV(A)*B)	(B*A) (B/A)
←A	(A*(B*C)) (A*(B/C))	((A*B)*C) ((A*B)/C)
A→	((A*B)*C) ((A/B)*C)	(A*(B*C)) (A/(B/C))
←D	((A+B)*C) ((A-B)*C)	((A*C)+(B*C)) ((A*C)-(B*C))
D→	(A*(B+C)) (A*(B-C))	((A*B)+(A*C)) ((A*B)-(A*C))

...ALGEBRA (FORM)

(Continued)

Operation	Before	After

←M	((A^B)*(A^C))
	(ALOG(A)*ALOG(B))
	(EXP(A)*EXP(B))
		(A^(B+C))
		ALOG(A+B)
		EXP(A+B)

M→	((A^C)*(B^C))	((A*B)^C)
-()	(A*B) (-(A)*B)	-(-(A)*B) -(A*B)

1/()	(A*B)
	(INV(A)*B)
		INV(INV(A)/B)
		INV(A/B)

L()	(LOG(A)*B)
	(LN(A)*B)
		LOG(A^B)
		LN(A^B)

Division (/)

Operation	Before	After
←→	(A/B)	(INV(B)*A)
←A	(A/(B*C)) (A/(B/C))	((A/B)/C) ((A/B)*C)
A→	((A*B)/C) ((A/B)/C)	(A*(B/C)) (A/(B*C))
←D	((A+B)/C) ((A-B)/C)	((A/C)+(B/C)) ((A/C)-(B/C))

D→	(A/(B+C))
	(A/(B-C))
		INV((INV(A)*B) +(INV(A)*C))
		INV((INV(A)*B) -(INV(A)*C))

←M	((A^B)/(A^C))
	(ALOG(A)/ALOG(B))
	(EXP(A)/EXP(B))
		(A^(B-C))
		ALOG(A-B)
		EXP(A-B)

(*Continued*)

Operation	Before	After
M→	((A^C)/(B^C))	((A/B)^C)
-()	(A/B) (-(A)/B)	-(-(A)/B) -(A/B)

L()	(LN(A)/B)
	(LOG(A)/B)
		LN(A^INV(B))
		LOG(A^INV(B))

1/()	(A/B)	INV(INV(A)*B)

Power (^)

Operation	Before	After
←A	(A^(B*C))	((A^B)^C)
A→	((A^B)^C)	(A^(B*C))
←D	((A*B)^C) ((A/B)^C)	((A^C)*(B^C)) ((A^C)/(B^C))
D→	(A^(B+C)) (A^(B-C))	((A^B)*(A^C)) ((A^B)/(A^C))

1/()	(A^B)
	(A^-(B))
		INV(A^-(B))
		INV(A^B)

E()	(ALOG(A)^B)
	(ALOG(A)^INV(B))
	(EXP(A)^B)
	(EXP(A)^INV(B))
		ALOG(A*B)
		ALOG(A/B)
		EXP(A*B)
		EXP(A/B)

...ALGEBRA (FORM)

Negation (−)

Operation	Before	After
→()	-(A+B)	(-(A)-B)
	-(A-B)	(-(A)+B)
	-(LOG(A))	(-(A)*B)
	-(A/B)	(-(A)/B)
	-(A*B)	LOG(INV(A))
	-(LN(A))	LN(INV(A))

Inverse (INV)

Operation	Before	After
→()	INV(A*B)	(INV(A)/B)
	INV(A/B)	(INV(A)*B)
	INV(A^B)	(A^-(B))
	INV(ALOG(A))	ALOG(-(A))
	INV(EXP(A))	EXP(-(A))

Logarithm (LOG)

Operation	Before	After
D→	LOG(A*B)	(LOG(A)+LOG(B))
	LOG(A/B)	(LOG(A)-LOG(B))
-()	LOG(A)	-(LOG(INV(A)))
	LOG(INV(A))	-(LOG(A))
L*	LOG(A^B)	(LOG(A)*B)
	LOG(A^INV(B))	(LOG(A)/B)

Antilogarithm (ALOG)

Operation	Before	After
D→	ALOG(A+B)	(ALOG(A)*ALOG(B))
	ALOG(A-B)	(ALOG(A)/ALOG(B))
1/()	ALOG(A)	INV(ALOG(-(A)))
	ALOG(-(A))	INV(ALOG(A))
E^	ALOG(A*B)	(ALOG(A)^B)
	ALOG(A/B)	(ALOG(A)^INV(B))

Natural Logarithm (LN)

Operation	Before	After
D→	LN(A*B)	(LN(A)+LN(B))
	LN(A/B)	(LN(A)-LN(B))
-()	LN(A)	-(LN(INV(A)))
	LN(INV(A))	-(LN(A))
L*	LN(A^INV(B))	(LN(A)*B)

Exponential (EXP)

Operation	Before	After
D→	EXP(A+B)	(EXP(A)*EXP(B))
	EXP(A-B)	(EXP(A)/EXP(B))
1/()	EXP(A)	INV(EXP(-(A)))
	EXP(-(A))	INV(EXP(A))
E^	EXP(A*B)	(EXP(A)^B)
	EXP(A/B)	(EXP(A)^INV(B))

Arithmetic

This section describes the arithmetic functions +, −, *, /, ^, INV, √, SQ, and NEG. These functions apply to several object types. They're described here for all appropriate object types; they're described in other sections, such as "ARRAY" and "COMPLEX," only as they apply to that particular object type.

+ Add Analytic

Level 2	Level 1		Level 1
z_1	z_2	➡	z_1+z_2
[$array_1$]	[$array_2$]	➡	[$array_1+array_2$]
z	' $symb$ '	➡	' $z+(symb)$ '
' $symb$ '	z	➡	' $symb+z$ '
' $symb_1$ '	' $symb_2$ '	➡	' $symb_1+(symb_2)$ '
{ $list_1$ }	{ $list_2$ }	➡	{ $list_1 list_2$ }
" $string_1$ "	" $string_2$ "	➡	" $string_1 string_2$ "
# n_1	n_2	➡	# n_1+n_2
n_1	# n_2	➡	# n_1+n_2
# n_1	# n_2	➡	# n_1+n_2

+ returns the sum of its arguments, where the nature of the sum is determined by the type of arguments. If the arguments are:

Two real numbers. The sum is the ordinary real sum of the arguments.

A real number *u* and a complex number (*x*, *y*). The result is the complex number $(x + u, y)$ obtained by treating the real number as a complex number with zero imaginary part.

Two complex numbers (x_1, y_1) and (x_2, y_2). The result is the complex sum ($x_1 + x_2$, $y_1 + y_2$).

A number and an algebraic. The result is an algebraic representing the symbolic sum.

Two algebraics. The result is an algebraic representing the symbolic sum.

Two lists. The result is a list obtained by concatenating the objects in the list in level 1 to the end of the list of objects in level 2.

Two strings. The result is a string obtained by concatenating the characters in the string in level 1 to the end of the string in level 2.

Two arrays. The result is the array sum, where each element is the real or complex sum of the corresponding elements of the argument arrays. The two arrays must have the same dimensions.

A binary integer and a real number. The result is a binary integer that is the sum of the two arguments, truncated to the current wordsize. The real number is converted to a binary integer before the addition.

Two binary integers. The result is a binary integer that is sum of the two arguments, truncated to the current wordsize.

— Subtract Analytic

Level 2	Level 1		Level 1
z_1	z_2	➡	z_1-z_2
[$array_1$]	[$array_2$]	➡	[$array_1-array_2$]
z	' $symb$ '	➡	' $z-symb$ '
' $symb$ '	z	➡	' $symb-z$ '
' $symb_1$ '	' $symb_2$ '	➡	' $symb_1-symb_2$ '
# n_1	n_2	➡	# n_1-n_2
n_1	# n_2	➡	# n_1-n_2
# n_1	# n_2	➡	# n_1-n_2

— returns the difference of its arguments, where the nature of the difference is determined by the type of arguments. The object in level 1 is subtracted from the object in level 2. If the arguments are:

Two real numbers. The result is the ordinary real difference of the arguments.

A real number *u* and a complex number (*x*, *y*). The result is the complex number $(x - u, y)$ or $(u - x, -y)$ obtained by treating the real number as a complex number with zero imaginary part.

Two complex numbers (x_1, y_1) and (x_2, y_2). The result is the complex difference $(x_1 - x_2, y_1 - y_2)$.

A number and an algebraic. The result is an algebraic representing the symbolic difference.

Two algebraics. The result is an algebraic representing the symbolic difference.

Two arrays. The result is the array difference, where each element is the real or complex difference of the corresponding elements of the argument arrays. The two arrays must have the same dimensions.

A binary integer and a real number. The result is a binary integer that is the sum of the number in level 2 plus the twos complement of the number in level 1. The real number is converted to a binary integer before the subtraction.

Two binary integers. The result is a binary integer that is the sum of the number in level 2 plus the twos complement of the number in level 1.

* *Multiply* Analytic

Level 2	Level 1		Level 1
z_1	z_2	➡	$z_1 z_2$
[*matrix*]	[*array*]	➡	[*matrix* × *array*]
z	[*array*]	➡	[*z* × *array*]
[*array*]	*z*	➡	[*array* × *z*]
z	' *symb* '	➡	' *z**(*symb*) '
' *symb* '	*z*	➡	' (*symb*)**z* '
' $symb_1$ '	' $symb_2$ '	➡	' $symb_1$*$symb_2$ '
# n_1	n_2	➡	# $n_1 n_2$
n_1	# n_2	➡	# $n_1 n_2$
# n_1	# n_2	➡	# $n_1 n_2$

...Arithmetic

* returns the product of its arguments, where the nature of the product is determined by the type of arguments. If the arguments are:

Two real numbers. The result is the ordinary real product of the arguments.

A real number *u* and a complex number (x, *y*). The result is the complex number (xu, yu) obtained by treating the real number as a complex number with zero imaginary part.

Two complex numbers (x_1, y_1) and (x_2, y_2). The result is the complex product $(x_1x_2 - y_1y_2, x_1y_2 + x_2y_1)$.

A number and an algebraic. The result is an algebraic representing the symbolic product.

Two algebraics. The result is an algebraic representing the symbolic product.

A number and an array. The result is the product obtained by muliplying each element of the array by the number.

A matrix and an array. The result is the matrix product of the arguments. The array in level 1 must have the same number of rows (elements, if a vector) as the number of columns of the matrix in level 2.

A binary integer and a real number. The result is a binary integer that is the product of the two arguments, truncated to the current wordsize. The real number is converted to a binary integer before the multiplication.

Two binary integers. The result is a binary integer that is the product of the two arguments, truncated to the current wordsize.

/　　Divide　　Analytic

Level 2	Level 1		Level 1
z_1	z_2	➡	z_1/z_2
[*array*]	[*matrix*]	➡	[*array* × *matrix*$^{-1}$]
z	' *symb* '	➡	' *z*/(*symb*) '
' *symb* '	*z*	➡	' (*symb*)/*z* '
' $symb_1$ '	' $symb_2$ '	➡	' $symb_1/symb_2$ '
# n_1	n_2	➡	# n_1/n_2
n_1	# n_2	➡	# n_1/n_2
# n_1	# n_2	➡	# n_1/n_2

/ ($\boxed{\div}$) returns the quotient (the object in level 2 divided by the object in level 1) of its arguments, where the nature of the quotient is determined by the type of arguments. If the arguments are:

Two real numbers. The result is the ordinary real quotient of the arguments.

A real number *u* in level 2 and a complex number (*x*, *y*) in level 1. The result is the complex number

$$(ux/(x^2 + y^2), \ -uy/(x^2 + y^2))$$

obtained by treating the real number as a complex number with zero imaginary part.

A complex number (*x*, *y*) in level 2 and a real number *u* in level 1. The result is the complex number $(x/u, \ y/u)$ obtained by treating the real number as a complex number with zero imaginary part.

A complex number (x_1, y_1) in level 2, and a complex number (x_2, y_2) in level 1. The result is the complex quotient

$$((x_1x_2 + y_1y_2)/(x_2^2 + y_2^2),\ (y_1x_2 - x_1y_2)/(x_2^2 + y_2^2)).$$

A number and an algebraic. The result is an algebraic representing the symbolic quotient.

Two algebraics. The result is an algebraic representing the symbolic quotient.

An array and a matrix. The result is the matrix product of the array in level 2 with the inverse of the matrix in level 1. The array in level 2 must have the same number of rows (elements, if a vector) as the number of columns of the matrix in level 1.

A binary integer and a real number. The result is a binary integer that is the integer part of the quotient of the two arguments. The real number is converted to a binary integer before the division. A divisor of 0 returns # 0.

Two binary integers. The result is a binary integer that is the integer part of the quotient of the two arguments. A divisor of zero returns # 0.

^ **Power** **Analytic**

Level 2	Level 1		Level 1
z_1	z_2	➡	$z_1^{z_2}$
z	' *symb* '	➡	' *z*^(*symb*) '
' *symb* '	*z*	➡	' (*symb*)^*z* '
' *symb*$_1$ '	' *symb*$_2$ '	➡	' *symb*$_1$^(*symb*$_2$) '

^ returns the value of the object in level 2 raised to the power given by the object in level 1. Any combination of real number, complex number, and algebraic arguments may be used. If either argument is complex, ^ returns a complex result.

INV *Inverse* Analytic

Level 1		Level 1
z	➡	1/*z*
[*matrix*]	➡	[*matrix*$^{-1}$]
' *symb* '	➡	' INV(*symb*) '

INV (■ 1/x) returns the inverse (reciprocal) of its argument.

For a complex argument (x, y), the inverse is the complex number

$$(x/(x^2 + y^2), \ -y/(x^2 + y^2)).$$

Array arguments must be square matrices.

√ *Square Root* Analytic

Level 1		Level 1
z	➡	$\sqrt{z}$
' *symb* '	➡	' √(*symb*) '

...Arithmetic

√ returns the (positive) square root of its argument. For a complex number (x_1, y_1), the square root is the complex number

$$(x_2, y_2) = (\sqrt{r} \cos \theta/2, \sqrt{r} \sin \theta/2)$$

where

$$r = \text{abs}\ (x_1, y_1), \quad \theta = \text{arg}\ (x_1, y_1).$$

If $(x_1, y_1) = (0, 0)$, then the square root is (0, 0).

Refer to "Principal Branches and General Solutions" in "COMPLEX."

SQ **Square** Analytic

Level 1		Level 1
z	➡	z^2
[*matrix*]	➡	[*matrix* × *matrix*]
' *symb* '	➡	' SQ(*symb*) '

SQ (🟥 x^2) returns the square of its argument.

For a complex argument (x, y), the square is the complex number

$$(x^2 - y^2, 2xy).$$

Array arguments must be square matrices.

NEG **Negate** Analytic

Level 1		Level 1
z	➡	$-z$
[*array*]	➡	[−*array*]
' *symb* '	➡	' −(*symb*) '

NEG returns the negative of its argument.

For an array, the negative is an array composed of the negative of each element in the array. The [CHS] key can be used to execute NEG if no command line is present. If a command line is present, [CHS] acts on the command line as described in "Basic Operations."

Menu keys for NEG are found in the REAL and ARRAY menus.

ARRAY

→ARRY	ARRY→	PUT	GET	PUTI	GETI
SIZE	RDM	TRN	CON	IDN	RSD
CROSS	DOT	DET	ABS	RNRM	CNRM
R→C	C→R	RE	IM	CONJ	NEG

Arrays are ordered collections of real or complex numbers that satisfy various mathematical rules. In the HP-28C, one-dimensional arrays are called *vectors*; two-dimensional arrays are called *matrices*. We will use the term "array" to refer collectively to vectors and matrices.

Although vectors are entered and displayed as a *row* of numbers, the HP-28C treats vectors, for the purposes of matrix multiplication and computations of matrix norms, as $n \times 1$ matrices.

An array can contain either real numbers or complex numbers. We will use the terms *real array* (*real vector* or *real matrix*) and *complex array* when describing properties of arrays that are specific to real numbers or complex numbers.

Arrays are entered and displayed in the following formats:

vector	[*number number* ...]
matrix	[[*number number* ...] [*number number* ...] ⋮ [*number number* ...]]

where *number* represents a real number or a complex number.

When you enter an array you can mix real and complex numbers. If any one number in an array is complex, the resulting array will be complex.

You can include any number of newlines anywhere in the entry, or you can enter the entire array in a single command line.

When entering matrices, you can omit the delimiter] that ends each row. The [that starts each row is required. If additional objects follow the array in the command line, you must end the array with]] before starting the new object.

The term *row order* refers to a sequential ordering of the elements of an array, starting with the first element (first row, first column), then: from left to right along each row; from the top row to the bottom row (for matrices).

The STORE menu contains commands that allow you to perform array operations using the name of a variable that contains an array, rather than requiring the array itself to be on the stack. In these cases, the result of an operation is stored in the variable, replacing its original contents. This method requires less memory than operations on the stack, and hence can allow you to deal with larger arrays.

Array operations that may be time-consuming for large arrays can be interrupted via the ON key. If you press ON during such an operation, the HP-28C will halt execution of the array command and clear the array arguments from the stack. You can recover the original arguments by using UNDO or LAST.

In addition to the functions present in the ARRAY and STACK menus, the keyboard functions described in the next section accept arrays as arguments.

Keyboard Functions

Complete stack diagrams for these functions appear in "Arithmetic."

+ **Add** **Analytic**

Level 2	Level 1		Level 1
[$array_1$]	[$array_2$]	➡	[$array_1+array_2$]

+ returns the array sum of two array arguments. The two arguments must have the same dimensions. The sum of a real array and a complex array is a complex array, where each element x of the real array is treated as a complex element (x, 0).

− **Subtract** **Analytic**

Level 2	Level 1		Level 1
[$array_1$]	[$array_2$]	➡	[$array_1-array_2$]

− returns the array difference of two array arguments. The two arguments must have the same dimensions. The difference between a real array and a complex array is a complex array, where each element x of the real array is treated as a complex element (x, 0).

* Multiply Analytic

Level 2	Level 1		Level 1
z	[*array*]	➧	[*z* × *array*]
[*array*]	*z*	➧	[*z* × *array*]
[*matrix*]	[*array*]	➧	[*matrix* × *array*]

* returns the product of its arguments, where the nature of the product is determined by the type of arguments. If the arguments are:

An array and a number. The product is the matrix product of the number (real or complex number) and the array, obtained by multiplying each element of the array by the scalar.

Two arrays. The product is the matrix product of the two arrays. The array in level 2 must be a matrix (that is, it can not be a vector). Level 1 can contain either a matrix or a vector. The number of rows in the array in level 1 must equal the number of columns in the matrix in level 2.

The product of a real array and a complex array is a complex array. Each element *x* of the real array is treated as a complex element (*x*,0).

/ Divide Analytic

Level 2	Level 1		Level 1
[*matrix* **B**]	[*matrix* **A**]	➧	[*matrix* **X**]
[*vector* **B**]	[*matrix* **A**]	➧	[*vector* **X**]

...ARRAY

/ ($\boxed{\div}$) applied to array arguments solves the system of equations $\mathbf{AX} = \mathbf{B}$ for $\mathbf{X}$. That is, / computes $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$. / uses 16-digit internal computation precision to provide a more accurate result than obtained by applying INV to **A** and multiplying the result by **B**.

A must be a square matrix, and **B** can be either a matrix or a vector. If **B** is a matrix, it must have the same number of rows as **A**. If **B** is a vector, it must have the same number of elements as the number of columns of **A**.

In many cases the HP-28C will arrive at a correct solution even if the coefficient array is singular (**A** has no proper inverse). This feature allows you to solve under-determined and over-determined systems of equations.

For an under-determined system (containing more variables than equations), the coefficient array will have fewer rows than columns. To find a solution:

1. Append enough rows of zeros to the bottom of your coefficient array to make it square.
2. Append corresponding rows of zeros to the constant array.

You can now use these arrays with / to find a solution to the original system.

For an over-determined system (containing more equations than variables), the coefficient array will have fewer columns than rows. To find a solution:

1. Append enough columns of zeros on the right of your coefficient array to make it square.
2. Add enough zeros on the bottom of your constant array to ensure conformability.

You can now use these arrays with / to find a solution to the original system. Only those elements in the result array that correspond to your original variables will be meaningful.

For both under-determined and over-determined systems, the coefficient array is singular, so you should check the results returned by / to see if they satisfy the original equation.

Improving the Accuracy of System Solutions

Because of rounding errors during calculation, a numerically calculated solution $\mathbf{Z}$ is not in general the solution to the original system $\mathbf{AX} = \mathbf{B}$, but rather the solution to the perturbed system $(\mathbf{A} + \Delta\mathbf{A})\mathbf{Z} = \mathbf{B} + \Delta\mathbf{B}$. The perturbations $\Delta\mathbf{A}$ and $\Delta\mathbf{B}$ satisfy $\|\Delta\mathbf{A}\| \leqslant \epsilon\|\mathbf{A}\|$ and $\|\Delta\mathbf{B}\| \leqslant \epsilon\|\mathbf{B}\|$, where ϵ is a small number and $\|\mathbf{A}\|$ is the *norm* of $\mathbf{A}$, a measure of its size analogous to the length of a vector. In many cases $\Delta\mathbf{A}$ and $\Delta\mathbf{B}$ will amount to less than one in the 12th digit of each element of $\mathbf{A}$ and $\mathbf{B}$.

For a calculated solution $\mathbf{Z}$, the *residual* is $\mathbf{R} = \mathbf{B} - \mathbf{AZ}$. Then $\|\mathbf{R}\| \leqslant \epsilon\|\mathbf{A}\|\ \|\mathbf{Z}\|$. So the expected residual for a calculated solution is small. Nevertheless, the *error* $\mathbf{Z} - \mathbf{X}$ may not be small if $\mathbf{A}$ is ill-conditioned, that is, if $\|\mathbf{Z} - \mathbf{X}\| \leqslant \epsilon\|\mathbf{A}\|\ \|\mathbf{A}^{-1}\|\ \|\mathbf{Z}\|$.

A rule-of-thumb for the accuracy of the computed solution is

$$\begin{aligned}&\text{(number of correct digits)}\\&\qquad \geqslant \text{(number of digits carried)} - \log(\|\mathbf{A}\|\ \|\mathbf{A}^{-1}\|) - \log 10n\end{aligned}$$

where n is the dimension of $\mathbf{A}$. For the HP-28C, which carries 12 accurate digits,

$$\text{(number of correct digits)} \geqslant 11 - \log(\|\mathbf{A}\|\ \|\mathbf{A}^{-1}\|) - \log n.$$

In many applications, this accuracy may be adequate. When additional accuracy is desired, the computed solution $\mathbf{Z}$ can usually be improved by *iterative refinement* (also known as *residual* corrections). Iterative refinement involves calculating a solution to a system of equations, then improving its accuracy using the residual associated with the solution to modify that solution.

To use iterative refinement, first calculate a solution $\mathbf{Z}$ to the original system $\mathbf{AX} = \mathbf{B}$. Then $\mathbf{Z}$ is treated as an approximation to $\mathbf{X}$, in error by $\mathbf{E} = \mathbf{X} - \mathbf{Z}$. Then $\mathbf{E}$ satisifies the linear system

$$\mathbf{AE} = \mathbf{AX} - \mathbf{AZ} = \mathbf{R},$$

where $\mathbf{R}$ is the residual for $\mathbf{Z}$. The next step is to calculate the residual and then solve $\mathbf{AE} = \mathbf{R}$ for $\mathbf{E}$. The calculated solution, denoted by $\mathbf{F}$, is treated as an approximation to $\mathbf{E}$ and is added to $\mathbf{Z}$ to obtain a new approximation to $\mathbf{X}$.

For $\mathbf{F} + \mathbf{Z}$ to be a better approximation to $\mathbf{X}$ than is $\mathbf{Z}$, the residual $\mathbf{R} = \mathbf{B} - \mathbf{AZ}$ must be calculated to extended precision. The function RSD does this (see the description of RSD below for details of its use).

The refinement process can be repeated, but most of the improvement occurs in the first refinement. The / function does not attempt to perform a residual refinement because of the memory required to maintain multiple copies of the original arrays. Here is an example of a user program that solves a matrix equation, including one refinement using RSD:

```
« → B A « B A / B A 3 PICK RSD A / + » »
```

The program takes two array arguments $\mathbf{B}$ and $\mathbf{A}$ from the stack, the same as /, and returns the result array $\mathbf{Z}$, which will be an improved approximation to the solution $\mathbf{X}$ over that provided by / itself.

INV *Inverse* **Analytic**

Level 1		Level 1
[*matrix*]	➡	[$matrix^{-1}$]

INV (■ 1/x) returns the matrix inverse of its argument. The argument must be a square matrix, either real or complex.

SQ *Square* **Analytic**

Level 1		Level 1
[$matrix_1$]	➡	[$matrix_2$]

SQ (■ x^2) returns the matrix product of a square matrix with itself.

NEG *Negate* **Analytic**

Level 1		Level 1
[*array*]	➡	[−*array*]

Pressing CHS when no command line is present executes the function NEG. For an array, each element of the result is the negative of the corresponding element of the argument array.

To enter the NEG function in the command line, use NEG (on the fourth row of the ARRAY menu).

...ARRAY

→ARRY ARRY→ PUT GET PUTI GETI

This group of commands allows you to recall or alter individual elements of an array.

→ARRY — *Stack to Array* — Command

Level $nm+1$ ... Level 2	Level 1		Level 1
x_1 ... x_n	{ *n* }	➡	[*vector*]
x_{11} ... x_{nm}	{ *n m* }	➡	[*matrix*]

→ARRY returns an array comprised of real or complex elements taken one-by-one from the stack, in levels 2 and above. →ARRY takes a list representing the size of the result array from level 1:

Vectors. If the list contains a single integer n, n numbers are taken from the stack, and an n element vector is returned.

Matrices. If the list contains two integers n and m, nm numbers are removed from the stack and returned as the elements of an $n \times m$ matrix.

The elements of the result array should be entered into the stack in row order, with x_{11} (or x_1) in level $nm + 1$ (or $n + 1$), and x_{nm} (or x_n) in level 2. If one or more of the elements is a complex number, the result array will be complex.

ARRY→ — *Array to Stack* — Command

Level 1		Level n+1 ... Level 2	Level 1
[*vector*]	➡	x_1 ... x_n	{ *n* }
[*matrix*]	➡	x_{11} ... x_{nm}	{ *n m* }

ARRY→ takes an array from the stack, and returns its elements to the stack as individual real or complex numbers. ARRY→ also returns a list representing the size of the array to level 1. The elements are placed on the stack in row order:

Vectors. If the argument is an n-element vector, the first element is returned to level $n + 1$, and the nth element to level 2. Level 1 will contain the list $\{ n \}$.

Matrices. If the argument is an $n \times m$ matrix, element x_{nm} is returned to level 2, and element x_{11} to level $(nm + 1)$.

PUT

Put Element **Command**

Level 3	Level 2	Level 1		Level 1
[*array*$_1$]	{ *index* }	*x*	➡	[*array*$_2$]
' *name* '	{ *index* }	*x*	➡	
[*C-array*$_1$]	{ *index* }	*z*	➡	[*C-array*$_2$]
' *name* '	{ *index* }	*z*	➡	
{ *list*$_1$ }	*n*	*obj*	➡	{ *list*$_2$ }
' *name* '	*n*	*obj*	➡	

PUT stores an element into an array or a list. This section describes its use with an array; its use with a list is described in "LIST."

PUT takes three arguments from the stack. Level 1 must contain the number you wish to store in the array. If the number is complex, the array must also be complex.

Level 2 contains the index of the array element you wish to replace. If the array is a vector, the index list contains one integer specifying the element number. If the array is a matrix, the index list has the form { *row column* }.

Level 3 may contain either an array or a name. If level 3 contains an array, PUT returns that array to the stack with one element replaced by the number taken from level 1.

If level 3 contains a name, PUT stores the number from level 1 as one element in the array contained in the variable *name*. (*name* can't be a local name.)

GET is the reverse operation of PUT.

GET *Get Element* Command

Level 2	Level 1		Level 1
[*array*]	{ *index* }	➡	*z*
' *name* '	{ *index* }	➡	*z*
{ *list* }	*n*	➡	*obj*
' *name* '	*n*	➡	*obj*

GET is a general mechanism for recalling an element from an array or a list. In this section, we will describe array element recall. List element recall is described in the "LIST" section.

GET takes two arguments from the stack. Level 1 should contain an index list specifying the array element you wish to recall. If the array is a vector, the index list should contain one integer specifying the element number. If the array is a matrix, the index list must have the form { *row column* }.

Level 2 may contain either an array or a name. If level 2 contains an array, GET returns the indexed element of that array to the stack. If level 2 contains a name, GET returns to the stack the indexed element of the array contained in the variable *name*.

PUT is the reverse operation of GET.

PUTI *Put and Increment Index* Command

Level 3	Level 2	Level 1		Level 2	Level 1
[*array*$_1$]	{ *index*$_1$ }	*x*	➡	[*array*$_2$]	{ *index*$_2$ }
' *name* '	{ *index*$_1$ }	*x*	➡	' *name* '	{ *index*$_2$ }
[*C-array*$_1$]	{ *index*$_1$ }	*z*	➡	[*C-array*$_2$]	{ *index*$_2$ }
' *name* '	{ *index*$_1$ }	*z*	➡	' *name* '	{ *index*$_2$ }
{ *list*$_1$ }	n_1	*obj*	➡	{ *list*$_2$ }	n_2
' *name* '	n_1	*obj*	➡	' *name* '	n_2

PUTI is a general mechanism for element storage into an array or object storage into an list. List storage is described in "LIST."

PUTI stores a number into an array in the same manner as PUT, but also returns the array (or variable name) and the element index incremented to the next element. This leaves the stack ready for you to enter a new number, then execute PUTI again to store the number into the next element in the array. When the element index is equal to the maximum index in the array, PUTI returns the index {1} (for vectors) or {1 1} (for matrices), starting over at the beginning of the array.

PUTI takes three arguments from the stack. Level 1 must contain the number you wish to store in the array. If the number is complex, the array must also be complex.

Level 2 contains the index list specifying the number of the array element you wish to replace. If the array is a vector, the index list should contain one integer specifying the element number. If the array is a matrix, the index list must have the form { *row column* }.

Level 3 may contain either an array or a name. If level 3 contains an array, PUTI returns that array to level 2, with the indexed element replaced by the number taken from level 1. The next index is returned to level 1.

If level 3 contains a name, PUTI stores the number from level 1 as the *n*th element in the array contained in the variable *name*. Name is returned to level 2, and the next index to level 1.

GETI is the reverse operation of PUTI.

GETI — *Get and Increment Index* — Command

Level 2	Level 1		Level 3	Level 2	Level 1
[*array*]	{ $index_1$ }	➡	[*array*]	{ $index_2$ }	*z*
' *name* '	{ $index_1$ }	➡	' *name* '	{ $index_2$ }	*z*
{ *list* }	n_1	➡	{ *list* }	n_2	*obj*
' *name* '	n_1	➡	' *name* '	n_2	*obj*

GETI is a general mechanism for recalling an element from an array or a list. List element recall is described in "LIST."

GETI recalls an element from an array in the same manner as GET, with the added feature that GETI leaves the array on the stack plus the element index incremented to the next element, to facilitate successive recalls from the same array. When the element index is equal to the maximum index of the array, GETI returns the index {1} (for vectors) or {1 1} (for matrices), starting over at the beginning of the array.

GETI takes two arguments from the stack. Level 1 should contain an index specifying the array element you wish to recall, in the form of a list of one or two integers. If the array is a vector, the index list should contain one integer specifying the element number. If the array is a matrix, the index list must have the form { *row column* }.

Level 2 may contain either an array or a name. If level 2 contains an array, GETI returns the indexed element of that array to level 1. GETI also returns the array to level 3, and the index of the next element to level 2.

If level 2 contains a name, GETI returns to level 1 the indexed element in the array contained in the variable *name*. GETI also returns name to level 3 and the index of the next element to level 2.

PUTI is the reverse operation of GETI.

SIZE **RDM** **TRN** **CON** **IDN** **RSD**

SIZE *Size* Command

Level 1		Level 1
" *string* "	➡	*n*
{ *list* }	➡	*n*
[*array*]	➡	{ *list* }
' *symb* '	➡	*n*

SIZE returns an object representing the size, or dimensions, of a list, array, string, or algebraic argument. For an array, SIZE returns a list containing one or two integers:

- If the original object is a vector, the list will contain a single integer representing the number of elements in the vector.
- If the object is a matrix, the list will contain two integers representing the dimensions of the matrix. The first integer is the number of rows in the matrix; the second is the number of columns.

Refer to sections "STRING," "LIST," and "ALGEBRA" for the use of SIZE with other object types.

RDM — *Redimension* — Command

Level 2	Level 1		Level 1
[*array*$_1$]	{ *dim* }	➧	[*array*$_2$]
' *name* '	{ *dim* }	➧	

RDM rearranges the elements of the array $array_1$ taken from level 2 (or contained in a variable *name*), and returns $array_2$, which has the dimensions specified in the list of one or two integers taken from level 1. If the array in level 2 is specified by name, $array_2$ replaces $array_1$ as the contents of the variable. If the list contains a single integer n, $array_2$ will be an n-element vector. If the list has the form $\{n\ m\}$, $array_2$ will be an $n \times m$ matrix.

Elements taken from $array_1$ preserve the same row order in $array_2$. If $array_2$ is dimensioned to contain fewer elements than $array_1$, excess elements from $array_1$ at the end of the row order are discarded. If $array_2$ is dimensioned to contain more elements than $array_1$, the additional elements in $array_2$ at the end of the row order are filled with zeros ((0, 0) if $array_1$ is complex).

TRN *Transpose* Command

Level 1		Level 1
[*matrix*$_1$]	➡	[*matrix*$_2$]
' *name* '	➡	

TRN returns the (conjugate) transpose of its argument. That is, an $n \times m$ matrix $\mathbf{A}$ in level 1 (or contained in *name*) is replaced by an $m \times n$ matrix $\mathbf{A}^t$, where

$$\mathbf{A}^t_{ij} = \begin{cases} \mathbf{A}_{ji} \text{ for real matrices,} \\ \text{CONJ } (\mathbf{A}_{ji}) \text{ for complex matrices.} \end{cases}$$

If the matrix is specified by name, $\mathbf{A}^t$ replaces $\mathbf{A}$ in *name*.

CON *Constant Array* Command

Level 2	Level 1		Level 1
{ *dim* }	*z*	➡	[*array*]
[*array*$_1$]	*x*	➡	[*array*$_2$]
[*C-array*$_1$]	*z*	➡	[*C-array*$_2$]
' *name* '	*z*	➡	

CON produces a *constant* array—an array with all elements having the same value. The constant value is the real or complex number taken from level 1. The result array is either a new array, or an existing array with its elements replaced by the constant value, according to the object in level 2.

Creating a new array. If level 2 contains a list of one or two integers, a new array is returned to the stack. If the list contains a single integer n, the result is a constant vector with n elements. If the list has the form $\{n\ m\}$, the result is a constant matrix with n rows and m columns.

Replacing the elements of an existing array. If level 2 contains a name, that name must identify a user variable containing an array. In this case, the elements of the array are replaced by the constant taken from level 1. If the constant is a complex number, the original array must be complex.

If level 2 contains an array, an array of the same dimensions is returned, with each element equal to the constant value. If the constant is a complex number, the original array must be complex.

IDN *Identity Matrix* Command

Level 1		Level 1
n	➡	[*R-identity matrix*]
[*matrix*]	➡	[*identity matrix*]
' *name* '	➡	

IDN produces an *identity* matrix—a square matrix with its diagonal elements equal to 1, and its off-diagonal elements 0. The result matrix is either a new matrix, or an existing square matrix with its elements replaced by those of the identity matrix, according to the argument in level 1.

Creating a new matrix. If the argument is a real number, a new real identity matrix is returned to the stack, with its number of rows and number of columns equal to the argument.

Replacing the elements of an existing matrix. If the argument is a name, that name must identify a user variable containing a square matrix. In this case, the elements of the matrix are replaced by those of the identity matrix (complex if the original matrix is complex).

If the argument is a square matrix, an identity matrix of the same dimensions is returned. If the original matrix is complex, the returned identity matrix will also be complex, with diagonal values (1,0).

RSD *Residual* Command

Level 3	Level 2	Level 1		Level 1
[*array* **B**]	[*matrix* **A**]	[*array* **Z**]	➡	[*array* **B**−**AZ**]

RSD computes the *residual* **B** − **AZ** of three arrays **B**, **A**, and **Z**. RSD is typically used for computing a correction to **Z**, where **Z** has been obtained as an approximation to the solution **X** to the system of equations **AX** = **B**. Refer to "Improving the Accuracy of System Solutions", earlier in this section, for a description of the use of RSD with systems of equations.

A, **B**, and **Z** are restricted as follows:

- **A** must be a matrix.
- The number of columns of **A** must equal the number of elements of **Z** if **Z** is a vector, or the number of rows of **Z** if **Z** is a matrix.
- The number of rows of **A** must equal the number of elements of **B** if **B** is a vector, or the number of rows of **B** if **B** is a matrix.
- **B** and **Z** must both be vectors or both be matrices.
- **B** and **Z** must have the same number of columns, if they are matrices.

CROSS **DOT** **DET** **ABS** **RNRM** **CNRM**

CROSS *Cross Product* Command

Level 2	Level 1		Level 1
[*vector* **A**]	[*vector* **B**]	➧	[*vector* **A** × **B**]

CROSS returns the cross product **A** × **B** of two three-element vectors **A** and **B**, where

$$\begin{aligned}(\mathbf{A}\times\mathbf{B})_1 &= \mathbf{A}_2\mathbf{B}_3 - \mathbf{A}_3\mathbf{B}_2\\(\mathbf{A}\times\mathbf{B})_2 &= \mathbf{A}_3\mathbf{B}_1 - \mathbf{A}_1\mathbf{B}_3\\(\mathbf{A}\times\mathbf{B})_3 &= \mathbf{A}_1\mathbf{B}_2 - \mathbf{A}_2\mathbf{B}_1\end{aligned}$$

DOT *Dot Product* Command

Level 2	Level 1		Level 1
[*array* **A**]	[*array* **B**]	➧	*x*

DOT returns the "dot" product **A·B** of two arrays **A** and **B**, computed as the sum of the products of the corresponding elements of the two arrays. For example: `[1 2 3] [4 5 6] DOT` returns 1 × 4 + 2 × 5 + 3 × 6, or 32.

Some authorities define the dot product of two complex arrays as the sum of the products of the conjugated elements of one array with their corresponding elements from the other array. The HP-28C uses the ordinary products without conjugation. However, if you prefer the alternate definition, you can apply CONJ to one or both arrays before using DOT.

DET — *Determinant* — Command

Level 1		Level 1
[*matrix*]	➡	*determinant*

DET returns the determinant of its argument, which must be a square matrix.

ABS — *Absolute Value* — Function

Level 1		Level 1
z	➡	$\lvert z \rvert$
[*array*]	➡	$\lVert array \rVert$
' *symb* '	➡	'ABS(*symb*)'

ABS returns the absolute value of its argument. In the case of an array, ABS returns the Frobenius (Euclidean) norm of the array, defined as the square root of the sum of the squares of the absolute values of all of the elements.

Refer to "REAL," "COMPLEX," and "ALGEBRA" for the use of ABS with other object types.

RNRM — *Row Norm* — Command

Level 1		Level 1
[*array*]	➡	*row norm*

RNRM returns the row norm (infinity norm) of its argument. The row norm is the maximum value (over all rows) of the sums of the absolute values of all elements in a row. For a vector, the row norm is the largest absolute value of any of the elements.

CNRM — *Column Norm* — Command

Level 1		Level 1
[*array*]	➧	*column norm*

CNRM returns the column norm (one-norm) of its argument. The column norm is the maximum value (over all columns) of the sums of the absolute values of all elements in a column. For a vector, the column norm is the sum of the absolute values of the elements.

R→C C→R RE IM CONJ NEG

R→C — *Real-to-Complex* — Command

Level 2	Level 1		Level 1
x	*y*	➧	(*x* , *y*)
[*R-array*$_1$]	[*R-array*$_2$]	➧	[*C-array*]

R→C combines two real numbers, or two real arrays, into a single complex number, or complex array, respectively. The object in level 2 is taken as the real part of the result; the object in level 1 is taken as the imaginary part.

For array arguments, the elements of the complex result array are complex numbers, the real and imaginary parts of which are the corresponding elements of the argument arrays in level 2 and level 1, respectively. The arrays must have the same dimensions.

C→R **Complex-to-Real** Command

Level 1		Level 2	Level 1
(*x*, *y*)	➡	*x*	*y*
[*C-array*]	➡	[*R-array*$_1$]	[*R-array*$_2$]

C→R returns to level 2 and level 1 the real and imaginary parts, respectively, of a complex number or complex array.

The real or imaginary part of a complex array is a real array, of the same dimensions, the elements of which are the real or imaginary parts of the corresponding elements of the complex array.

RE **Real Part** Function

Level 1		Level 1
x	➡	*x*
(*x*, *y*)	➡	*x*
[*R-array*]	➡	[*R-array*]
[*C-array*]	➡	[*R-array*]
' *symb* '	➡	' RE(*symb*) '

RE returns the real part of its argument. If the argument is an array, RE returns a real array, the elements of which are equal to the real parts of the corresponding elements of the argument array.

IM **Imaginary Part** **Function**

Level 1		Level 1
x	➡	*0*
(*x*, *y*)	➡	*y*
[*R-array*]	➡	[*zero R-array*]
[*C-array*]	➡	[*R-array*]
' *symb* '	➡	' IM(*symb*) '

IM returns the imaginary part of its argument. If the argument is an array, IM returns a real array, the elements of which are equal to the imaginary parts of the corresponding elements of the argument array. If the argument array is real, all of the elements of the result array will be zero.

CONJ **Conjugate** **Analytic**

Level 1		Level 1
x	➡	*x*
(*x*, *y*)	➡	(*x*, −*y*)
[*R-array*]	➡	[*R-array*]
[$C\text{-}array_1$]	➡	[$C\text{-}array_2$]
' *symb* '	➡	' CONJ(*symb*) '

CONJ returns the complex conjugate of a complex number or complex array. The imaginary part of a complex number, or of each element of a complex array, is negated. For real numbers or arrays, the conjugate is identical to the original argument.

NEG *Negate* Analytic

Level 1		Level 1
[*array*]	➡	[−*array*]

For an array, each element of the result array is the negative of the corresponding element of the argument array.

When no command line is present, pressing CHS executes the function NEG. A complete stack diagram for NEG appears in "Arithmetic".

BINARY

DEC	**HEX**	**OCT**	**BIN**	**STWS**	**RCWS**
RL	**RR**	**RLB**	**RRB**	**R→B**	**B→R**
SL	**SR**	**SLB**	**SRB**	**ASR**	
AND	**OR**	**XOR**	**NOT**		

Binary integers are unsigned integer numbers that are represented internally in the HP-28C as binary numbers of length 1 to 64 bits. Such numbers must be entered, and are displayed, as a string of digits preceded by the delimiter #.

The entry and stack display of binary integers is controlled by the current integer *base*, which can be binary (base 2), octal (base 8), decimal (base 10), or hexadecimal (base 16). If you change the current base using one of the menu keys BIN, OCT, DEC, or HEX, the internal representation of a binary integer on the stack is not changed, but the digits shown in the display will change to reflect the number's representation in the new base.

Digits included in a binary integer entry must be legal in the current base. In binary base, only the digits 0 and 1 are allowed; in octal, the digits 0-7; in decimal, the digits 0-9; and in hexadecimal, the digits 0-9 and the letters A-F. The default base is decimal.

All binary integers entered in the same command line must be in the same (current) base. Since the four base selection menu keys do not perform an ENTER, you can change the base even after you have begun keying into the command line. However, the digit syntax of binary integers in the command line is checked (against the current base) before any of the command line is executed, so you cannot enter numbers in different bases by including the base selection commands in the command line.

The stack display of binary integers is also affected by the current *wordsize*, which you can set in the range 1 to 64 bits with the command STWS. When a binary integer is displayed on the stack, the display shows only the least significant bits, up to the wordsize, even if the number has not been truncated. If you reduce the wordsize, the display will alter to show fewer bits, but if you subsequently increase the wordsize, the hidden bits will be displayed.

The primary purpose of the wordsize is to control the results returned by commands. Commands that take binary integer arguments truncate those arguments to the number of (least significant) bits specified by the current wordsize, and they return results with that number of bits. The default wordsize is 64 bits.

The current base and wordsize are encoded in user flags 37 through 44. Flags 37–42 are the binary representation of the current wordsize minus 1 (flag 42 is the most significant bit). Flags 43 and 44 determine the current base:

Flag 43	Flag 44	Base
0	0	Decimal
0	1	Binary
1	0	Octal
1	1	Hexadecimal

In addition to the BINARY menu commands described in the next sections, the arithmetic functions +, −, *, and / can be used with pairs of binary integers, or combinations of real integers and binary integers, as described in "Arithmetic."

DEC HEX OCT BIN STWS RCWS

DEC *Decimal Mode* Command

	➡	

DEC sets decimal mode for binary integer operations. Binary integers may contain the digits 0 through 9, and will be displayed in base 10.

DEC clears user flags 43 and 44.

HEX *Hexadecimal Mode* Command

	➡	

HEX sets hexadecimal mode for binary integer operations. Binary integers may contain the digits 0 through 9, and A (ten) through F (fifteen), and will be displayed in base 16.

HEX sets user flags 43 and 44.

OCT *Octal Mode* Command

	➡	

OCT sets octal mode for binary integer operations. Binary integers may contain the digits 0 through 7, and will be displayed in base 8.

OCT sets user flag 43 and clears flag 44.

BIN *Binary Mode* Command

➡

BIN sets binary mode for binary integer operations. Binary integers may contain the digits 0 and 1, and will be displayed in base 2.

BIN clears user flag 43, and sets flag 44.

STWS *Store Wordsize* Command

Level 1		
n	➡	

STWS sets the argument n as the current binary integer wordsize, where n should be a real integer in the range 1 through 64. If $n > 64$, then a wordsize of 64 is set; if $n < 1$, the wordsize will be 1. User flags 37–42 represent the binary representation of $n - 1$ (flag 42 is the most significant bit).

RCWS *Recall Wordsize* Command

		Level 1
	➡	*n*

RCWS returns a real integer n equal to the current wordsize, in the range 1 through 64. User flags 37–42 represent the binary representation of $n - 1$.

...BINARY

RL **RR** **RLB** **RRB** **R→B** **B→R**

The commands RL and RR rotate binary integers (set to the current wordsize) to the left or right by one bit. The commands RLB and RRB are equivalent to RL or RR repeated eight times. R→B and B→R convert real numbers to or from binary integers.

RL *Rotate Left* Command

Level 1		Level 1
# n_1	➡	# n_2

RL performs a 1 bit left rotate on a binary integer number # n_1. The leftmost bit of # n_1 becomes the rightmost bit of the result # n_2.

RR *Rotate Right* Command

Level 1		Level 1
# n_1	➡	# n_2

RR performs a 1 bit right rotate on a binary integer number # n_1. The rightmost bit of # n_1 becomes the leftmost bit of the result # n_2.

RLB *Rotate Left Byte* Command

Level 1		Level 1
# n_1	➡	# n_2

RLB performs a 1 byte left rotate on a binary integer number # n_1. The leftmost byte of # n_1 becomes the rightmost byte of the result # n_2.

RRB *Rotate Right Byte* Command

Level 1		Level 1
# n_1	➡	# n_2

RRB performs a 1 byte right rotate on a binary integer number # n_1. The rightmost byte of # n_1 becomes the leftmost byte of the result # n_2.

R→B *Real to Binary* Command

Level 1		Level 1
n	➡	# n

R→B converts a real integer n, $0 \leqslant n \leqslant 1.84467440737E19$, to its binary integer equivalent # n. If $n < 0$, the result is # 0. If $n > 1.84467440737E19$, the result is # FFFFFFFFFFFFFFFF (hex).

B→R *Binary to Real* Command

Level 1		Level 1
# n	➡	n

B→R converts a binary integer # n to its real number equivalent n. If # $n >$ # 1000000000000 (decimal), only the 12 most significant decimal digits are preserved in the mantissa of the result.

SL **SR** **SLB** **SRB** **ASR**

The commands SL and SR shift binary integers (set to the current wordsize) to the left or right by one bit. The commands RLB and RRB are equivalent to RL or RR repeated eight times.

SL *Shift Left* Command

Level 1		Level 1
# n_1	➡	# n_2

SL performs a 1 bit left shift on a binary integer. The high bit of n_1 is lost. The low bit of n_2 is set to zero. SL is equivalent to binary multiplication by two (with truncation to the current wordsize).

SR *Shift Right* Command

Level 1		Level 1
# n_1	➡	# n_2

SR performs a 1 bit right shift on a binary integer. The low bit of n_1 is lost. The high bit of n_2 is set to zero. SR is equivalent to binary division by two.

SLB *Shift Left Byte* Command

Level 1		Level 1
# n_1	➡	# n_2

SLB performs a 1 byte left shift on a binary integer. SLB is equivalent to multiplication by # 100 (hexadeciamal) (truncated to the current wordsize).

SRB *Shift Right Byte* Command

Level 1		Level 1
# n_1	➡	# n_2

SRB performs a 1 byte right shift on a binary integer. SRB is equivalent to binary division by # 100 (hexadecimal).

ASR *Arithmetic Shift Right* Command

Level 1		Level 1
# n_1	➡	# n_2

ASR performs a 1 bit arithmetic right shift on a binary integer. In an arithmetic shift, the most significant bit retains its value, and a shift right is performed on the remaining *wordsize* - 1 bits.

AND OR XOR NOT

The commands AND, OR, XOR, and NOT can be applied to *flags* (real numbers or algebraics), and to binary integers. In the former case, the commands act as logical operators that combine true or false flags. For binary integers, the commands perform logical operations on the individual bits of the arguments.

The following descriptions apply to the use of the commands with binary integer arguments. "PROGRAM TEST" describes their application to flags.

AND *And* Function

Level 2	Level 1		Level 1
# n_1	# n_2	➧	# n_3

AND returns the logical AND of two binary integer arguments. Each bit in the result is determined by the corresponding bits in the two arguments, according to the following table:

# n_1	# n_2	AND Result # n_3
0	0	0
0	1	0
1	0	0
1	1	1

OR *Or* Function

Level 2	Level 1		Level 1
# n_1	# n_2	➡	# n_3

OR returns the logical OR of two binary integer arguments. Each bit in the result is determined by the corresponding bits in the two arguments, according to the following table:

# n_1	# n_2	OR Result # n_3
0	0	0
0	1	1
1	0	1
1	1	1

XOR *Exclusive Or* Function

Level 2	Level 1		Level 1
# n_1	# n_2	➡	# n_3

XOR returns the logical XOR (exclusive OR) of two binary integer arguments. Each bit in the result is determined by the corresponding bits in the two arguments, according to the following table:

# n_1	# n_2	XOR Result # n_3
0	0	0
0	1	1
1	0	1
1	1	0

NOT *Not* Function

Level 1		Level 1
# n_1	➡	# n_2

NOT returns the ones complement of its argument. Each bit in the result is the complement of the corresponding bit in # n_1.

# n_1	NOT Result # n_2
0	1
1	0

Calculus

The HP-28C is capable of symbolic differentiation of any algebraic expression (within the constraints of available memory), and of numerical integration of any (algebraic syntax) procedure. In addition, the calculator can perform symbolic integration of polynomial expressions. For more general expressions, the ∫ command can automatically perform a Taylor series approximation to the expression, then symbolically integrate the resulting polynomial.

Differentiation

∂ ***Differentiate*** **Analytic**

Level 2	Level 1		Level 1
'*symb*$_1$'	'*name*'	➡	'*symb*$_2$'

∂ (■ d/dx) computes the derivative of an algebraic expression *symb*$_1$ with respect to a specified variable *name*. (*Name* cannot be a local name.) The form of the result expression *symb*$_2$ depends upon whether ∂ is executed as part of an algebraic expression, or as a "stand-alone" object.

Step-wise Differentiation in Algebraics

The derivative function ∂ is represented in algebraic expressions with a special syntax:

'∂*name*(*symb*)',

where *name* is the variable of differentiation and *symb* is the expression to be differentiated.

For example, `'∂X(SIN(Y))'` represents the derivative of `SIN(Y)` with respect to `X`. When the overall expression is evaluated, the differentiation is carried forward one "step"—the result is the derivative of the argument expression, multiplied by a new subexpression representing the derivative of its argument. An example should make this clear. Consider differentiating `SIN(Y)` with respect to `X` in radians mode, where `Y` has the value `'X^2'`:

`'∂X(SIN(Y))' EVAL` returns `'COS(Y)*∂X(Y)'`.

We see that this is a strict application of the *chain rule of differentiation*. This description of the behavior of ∂, along with the general properties of EVAL, is sufficient for understanding the results of subsequent evaluations of the expression:

`EVAL` returns `'COS(X^2)*(∂X(X)*2*X^(2-1))'`,

`EVAL` returns `'COS(X^2)*(2*X)'`.

Fully Evaluated Differentiation

When ∂ is executed as an individual object—that is, in a sequence

`'symb' 'name' ∂`,

rather than as part of an algebraic expression, the expression is automatically evaluated repeatedly until it contains no derivatives. As part of this process, if the variable of differentiation *name* has a value, the final form of the expression will have that value substituted everywhere for the variable name.

To compare this behavior of ∂ with the step-wise differentiation described in the preceding section, consider again the example expression `'SIN(Y)'`, where `Y` has the value `'X^2'`:

`'SIN(Y)' 'X' ∂` returns `'COS(X^2)*(2*X)'`.

All of the steps of the differentiation have been carried out in a single operation.

The function ∂ determines whether to perform the automatic repeated evaluation according to the form of the level 1 argument that specifies the variable of differentiation. If that argument is a name, the full differentiation is performed. When the level 1 argument is an algebraic expression containing only a name, only one step of the differentiation is carried out. Normally, algebraics containing only a single name are automatically converted to name objects. The special syntax of ∂ allows this exception to be used as a signal for full or stepwise differentiation.

Differentiation of User-Defined Functions

When ∂ is applied to a user-defined function:

1. The expression consisting of the function name and its arguments within parentheses is replaced by the expression that defines the function.
2. The arguments from the original expression are substituted for the local names within the function definition.
3. The new expression is differentiated.

For example: Define $F (a, b) = 2a + b$:

```
« → a b '2*a+b' » 'F' STO.
```

Then differentiate `'F(X,X^2)'` with respect to `X`. The differentiation automatically proceeds as follows:

1. `'F(X,X^2)'` is replaced by `'2*a+b'`.
2. `X` is substituted for `a`, and `'X^2'` for `b`. The expression is now `'2*X+X^2'`.
3. The new expression is differentiated.
   - If we evaluated `'∂X(F(X,X^2))'` the result is `'∂X(2*X)+∂X(X^2)'`.
   - If we executed `'F(X,(X^2))'  'X'  ∂`, the differentiation is carried through to the final result `'2+2*X'`.

User-Defined Derivatives

If ∂ is applied to an HP-28C function for which a built-in derivative is not available, ∂ returns a formal derivative—a new function whose name is "der" followed by the original function name. For example, the HP-28C definition of % does not include a derivative. If you differentiate `'%(X,Y)'` one step with respect to `Z`, you obtain

```
'der%(X,Y,∂Z(X),∂Z(Y))'
```

Each argument to the % function results in two arguments to the der% function. In this example, the `X` argument results in `X` and `∂Z(X)` arguments, and the `Y` argument results in `Y` and `∂Z(Y)` arguments.

You can further differentiate by creating a user-defined function to represent the derivative. Here is a derivative for %:

```
« → x y dx dy '(x*dy+y*dx)/100' » 'der%' STO.
```

With this definition you can obtain a correct derivative for the % function. For example:

`'%(X,2*X)' 'X' ∂ COLCT` returns `'.04*X'`.

Similarly, if ∂ is applied to a formal user function (a name followed by arguments in parentheses, for which no user-defined function exists in user memory), ∂ returns a formal derivative whose name is "der" followed by the original user function name. For example, differentiating a formal user function `'f(x1,x2,x3)'` with respect to x returns

```
'derf(x1,x2,x3,∂x(x1),∂x(x2),∂x(x3))'
```

Integration

∫ ***Integrate*** **Command**

Level 3	Level 2	Level 1		Level 2	Level 1
'symb'	*'name'*	*degree*	➡		*'integral'*
x	*{ name a b }*	*accuracy*	➡	*integral*	*error*
'symb'	*{ name a b }*	*accuracy*	➡	*integral*	*error*
«program»	*{ name a b }*	*accuracy*	➡	*integral*	*error*
«program»	*{ a b }*	*accuracy*	➡	*integral*	*error*

∫ returns either a polynomial expression representing a symbolic indefinite integral, or two real numbers for a definite numerical integral. The nature of the result is determined by the arguments. In general, ∫ requires three arguments. Level 3 contains the object to be integrated; the level 2 object determines the form of the integration; the level 1 object specifies the accuracy of the integration.

...Calculus

Symbolic Integration

∫ includes a limited symbolic integration capability. It can return an exact (indefinite) integral of an expression that is a polynomial in the variable of integration. It can also return an approximate integral by using a Taylor series approximation to convert the integrand to a polynomial, then integrating the polynomial.

To obtain a symbolic integral, the stack arguments must be:

3: *Integrand* (name or algebraic)
2: *Variable of integration* (name)
1: *Degree of polynomial* (real integer)

The *degree of polynomial* specifies the order of the Taylor series approximation (or the order of the *integrand* if it is already a polynomial).

Numerical Integration

To obtain a numerical integral, you must specify:

- The integrand.
- The variable of integration.
- The numerical limits of integration.
- The accuracy of the integrand, or effectively, the acceptable error in the result of the integration.

Using an Explicit Variable of Integration. A numerical integration, in which the variable of integration is named with a name object that (usually) appears in the definition of the object used as the integrand, is called *explicit variable integration*. In the next section, *implicit variable integration* will described, in which the variable of integration does not have to be named.

For explicit variable integration, you must enter the relevant objects as follows:

```
3: Integrand
2: Variable of integration and limits
1: Accuracy
```

The integrand is an object representing the mathematical expression to be integrated. It can be:

- A real number, representing a constant integrand. In this case, the value of the integral will just be:

  number (upper limit − lower limit).

- An algebraic expression.
- A program. The program must satisfy algebraic syntax—that is, take no arguments from the stack, and return a real number.

The variable of integration and the limits of integration must be included in a list in level 2 of the form:

{ *name lower-limit upper-limit* },

where *name* is a name object, and the limits are real numbers.

The *accuracy* is a real number that specifies the error tolerance of the integration, which is taken to be the relative error in the evaluation of the integrand (the accuracy determines the spacing of the points, in the domain of the integration variable, at which the integrand is sampled for the approximation of the integral).

The accuracy is specified as a fractional error, that is,

$$\mathit{accuracy} \geqslant \left| \frac{\mathit{true\ value} - \mathit{computed\ value}}{\mathit{computed\ value}} \right|$$

where *value* is the value of the integrand at any point in the integration interval. Even if your integrand is accurate to or near 12 significant digits, you may wish to use a larger accuracy value to reduce integration time, since the smaller the accuracy value, the more points that must be sampled.

The accuracy of the integrand depends primarily on three considerations:

- The accuracy of empirical constants in the expression.
- The degree to which the expression may accurately describe a physical situation.
- The extent of round-off error in the internal evaluation of the expression.

Expressions like $\cos(x - \sin x)$ are purely mathematical expressions, containing no empirical constants. The only constraint on the accuracy then, is the round-off errors which may accumulate due to the finite (12-digit) accuracy of the numerical evaluation of the expression. You can, of course, specify an accuracy for integration of such expressions larger than the simple round-off error, in order to reduce computation time.

When the integrand relates to an actual physical situation, there are additional considerations. In these cases, you must ask yourself whether the accuracy you would like in the computed integral is justified by the accuracy of the integrand. For example, if the integrand contains empirical constants that are accurate to only 3 digits, it may not make sense to specify an accuracy smaller than 1E-3.

Furthermore, nearly every function relating to a physical situation is inherently inaccurate because it is only a mathematical model of an actual process or event. The model is typically an approximation that ignores the effects of factors judged to be insignificant in comparison with the factors in the model.

To illustrate numerical integration, we will compute

$$\int_1^2 exp\ x\ dx$$

to an accuracy of .00001. The stack should be configured as follows for $\int$:

```
3: 'EXP(X)'
2: {  X  1  2  }
1: .00001
```

Numerical integration returns two numbers to the stack. The value of the integral is returned to level 2. The error returned to level 1 is an upper limit to the fractional error of the computation, where normally

$$error = accuracy \int |integrand|$$

If the error is a negative number, it indicates that a convergence of the approximation was not achieved, and the level 2 result is the last computed approximation.

For the integral of `'EXP(X)'` in the example, $\int$ returns a value 4.67077 to level 2, and the error 4.7E-5 to level 1.

Using an Implicit Variable of Integration. The use of an explicit variable of integration allows you to enter the integrand as an ordinary algebraic expression. However, it is also possible to enter the integrand in RPN form, which can appreciably reduce the time required to compute the integral by eliminating repeated evaluation of the variable name. In this method, an *implicit* variable of integration is being used. The stack should be configured like this:

3: *Integrand* (program)

2: *Limits of integration* (list)

1: *Accuracy* (real number)

The *integrand* must be a program that takes one real number from the stack, and returns one real number. ∫ evaluates the program at each of the sample points between the limits of integration. For each evaluation ∫ places the sample value on the stack. The program takes that value, and returns the value of the integrand at that point.

The *limits of integration* must be entered as a list of two real numbers, in the format {*lower-limit upper-limit*}. The *accuracy* specifies the fractional error in the computation, as described in the preceding section.

For example to evaluate the integral:

$$\int_1^2 exp\ (x)\ dx$$

to an accuracy of .00001, you should execute ∫ with the stack as follows:

```
3: « EXP »
2: {  1  2  }
1: .00001
```

This returns the same value 4.67077 and accuracy 4.7E-5 as the example in the preceding section, where we used an explicit variable of integration.

Taylor Series

TAYLR — *Taylor Series* — Command

Level 3	Level 2	Level 1		Level 1
'*symb*$_1$'	'*name*'	*n*	➡	'*symb*$_2$'

TAYLR (in the ALGEBRA menu) computes a Taylor series approximation of the algebraic *symb*$_1$, to the *n*th order in the variable *name*. The approximation is evaluated at the point *name* = 0 (sometimes called a MacLaurin series). The Taylor approximation of f(x) at $x = 0$ is defined as:

$$\sum_{i=1}^{n} \frac{x^i}{i!} \left(\frac{\partial^i}{\partial x^i} f(x) \right) \Bigg|_{x=0}$$

...Calculus

Translating the Point of Evaluation

If you're using TAYLR simply to put a polynomial in power form, the point of evaluation makes no difference because the result is exact. However, if you're using TAYLR to approximate a mathematical function, you may need to translate the point of evaluation away from zero.

For example, if you're interested in the behavior of a function in a particular region, its TAYLR approximation will be more useful if you translate the point of evaluation to that region. Also, if the function has no derivative at zero, its TAYLR approximation will be meaningless unless you translate the point of evaluation away from zero.

Note

Executing TAYLR can return a meaningless result if the expression is not differentiable at zero. For example, if you clear flag 59 (to prevent `Infinite Result` errors) and execute:

```
'√X' 'X' 2 TAYLR
```

you will obtain the result `'5.E499*X-1.25E499*X^2'`. The coefficient of `X` is `∂X(X^.5)`, which equals `.5 * X^ - .5` and evaluates to 5.E499 for $x = 0$.

Although TAYLR always evaluates the function and its derivatives at zero, you can effectively translate the point of evaluation away from zero by changing variables in the expression. For example, suppose the function is an expression in X, and you want the TAYLR approximation at X = 2. To translate the point of evaluation by changing variables:

1. Store `'Y+2'` in `'X'`.
2. Evaluate the original function to change the variable from X to Y.

3. Find the Taylor approximation at Y = 0.
4. Purge X (if it still exists as a variable).
5. Store `'X-2'` in `'Y'`.
6. Evaluate the new function to change the variable from Y to X.
7. Purge Y.

Approximations of Rational Functions

A *rational function* is the quotient of two polynomials. If the denominator evenly divides the numerator, the rational function is equivalent to a polynomial. For example:

$$\frac{x^3 + 2x^2 - 5x - 6}{x^2 - x - 2} = x + 3$$

If your expression is such a rational function, you can convert it to the equivalent polynomial form by using TAYLR. However, if the denominator doesn't evenly divide the numerator—that is, if there is a remainder—the rational function is *not* a polynomial. For example:

$$\frac{x^3 + 2x^2 - 5x - 2}{x^2 - x - 2} = x + 3 + \frac{4}{x^2 - x - 2}$$

There is no equivalent polynomial form for such a rational function, but you can use TAYLR to calculate a polynomial that is accurate for small x (close to zero). You can translate the region of greatest accuracy away from $x = 0$, and you can choose the accuracy of the approximation. For the example above, the first-degree TAYLR approximation at $x = 0$ is $2x + 1$.

Polynomial Long Division. Another useful approximation to a rational function is the quotient polynomial resulting from long division. Consider the righthand side of the equation above as a polynomial plus a remainder. The polynomial is a good approximation to the rational function when the remainder is small—that is, when x is large. Note the difference between the quotient polynomial $(x + 3)$ and the TAYLR approximation of the same degree $(2x + 1)$.

The steps below show you how to perform polynomial long division on the HP-28C. The general process is the same as doing long division for numbers.

1. Create expressions for the numerator and denominator, with both in power form.
2. Store the denominator in a variable named 'D' (for "divisor").
3. Store an initial value of zero in a variable named 'Q' (for "quotient").

With the numerator on the stack, proceed with the steps below. The numerator is the initial value for the dividend. Each time you repeat steps 4 through 8, you'll add a term to Q and reduce the dividend.

4. Put D on the stack (in level 1).
5. Divide the highest-order term of the dividend (in level 2) by the highest-order term of the divisor (in level 1). You can calculate the result by inspection and key it in, or you can key in an expression

   ' *dividend-term/divisor-term* '

   and then put it in power form.

   For example, if the dividend is $x^3 + 2x^2 - 5x - 2$ and the divisor is $x^2 - x - 2$, the result is x; if the dividend is $3x^3 + x^2 - 7$ and the divisor is $2x^2 + 8x + 9$, the result is $1.5x$.

   The result is one term of the quotient polynomial.

6. Make a copy of the quotient term, and add this copy to Q.
7. Multiply the quotient term and the divisor.
8. Subtract the result from the dividend. The result is the new dividend.

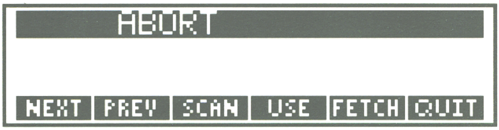
If the new dividend's degree is greater than or equal to the divisor's degree, repeat steps 4 through 8.

When the new dividend's degree is less than the divisor's degree, stop. The polynomial quotient is stored in Q, and the remainder equals the final dividend divided by the divisor.

CATALOG

The HP-28C command catalog, activated by the ■ CATALOG key, is an alphabetical listing of all of the commands recognized by the HP-28C. You can use the catalog to determine the existence and correct spelling of a command, and to learn which object types can be used as arguments for each command. Commands that start with a non-alphabetic character are listed after XPON, the last command beginning with a letter.

When you press ■ CATALOG, the normal HP-28C display is superceded by the catalog display:


ABORT

The top line shows a command name. ABORT is the first command alphabetically in the HP-28C catalog.

The catalog menu is shown in the bottom line of the display. The six menu keys act as follows:

Menu Key	Description
NEXT	Advance the catalog display to the next command in the catalog (repeating key).
PREV	Move the catalog display to the previous command in the catalog (repeating key).
SCAN	Scan automatically forward through the command catalog, showing each command briefly. The SCAN menu label changes to STOP ; pressing STOP halts the scan at the current command. The scan will stop automatically at the last command in the catalog (→STR).
USE	Activate the USAGE display (see below) for the current command, to show the stack arguments used by the command.
FETCH	Exit the catalog, and add the current command abbreviation to the command line at the cursor position (start a new command line if none is present).
QUIT	Exit the catalog, leaving any current command line unchanged.

In addition to the operations available in the command catalog menu, you can also:

- Press a key on the left-hand keyboard to move the catalog display to the first command that starts with the letter or character indicated.
- If there are no commands starting with the letter corresponding to a key you press, the catalog will move to the last command starting with the alphabetically previous letter.

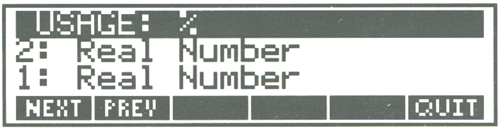
...CATALOG

- If there are no commands starting with the non-alphabetic character corresponding to a key you press, the catalog will move to +, the first command that starts with a non-alphabetic character. (The [αLOCK] key moves the catalog to →STR, the last entry in the catalog.)
- Press [ON] to exit the catalog and clear any current command line.

Any keys that are not active during the catalog display will beep when pressed.

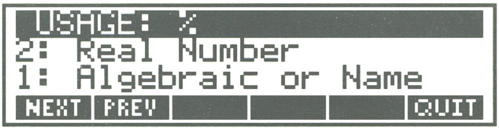
USAGE

Pressing the USE menu key activates a second level of the catalog, called the USAGE display. For the % command, for example, the initial USAGE display looks like this:



```
USAGE: %
2: Real Number
1: Real Number
```

The display indicates that % can take two real numbers as arguments. If the PREV and NEXT menu keys are shown, this indicates that there are additional argument options. For %, if you press NEXT, the display becomes:



```
USAGE: %
2: Real Number
1: Algebraic or Name
```

This shows that % will also accept a real number in level 2 and an algebraic or name in level 1 as arguments. There are two additional combinations for %, which you can view by pressing NEXT twice. You can also back up through the USAGE display by pressing PREV.

To exit from the USAGE display, you can:

- Press QUIT to return to the catalog display for the current command. From here you can move through the catalog to other commands, or exit by pressing QUIT again.
- Press [ON] to exit the catalog. [ON] also clears the current command line.

COMPLEX

R→C	C→R	RE	IM	CONJ	SIGN
R→P	P→R	ABS	NEG	ARG	

The COMPLEX menu (■ CMPLX) contains commands specific to complex numbers.

Complex number objects in the HP-28C are ordered pairs of numbers that are represented as two real numbers enclosed within parentheses and separated by the non-radix character, for example, `(1.234,5.678)`. A complex number object (x, y) can represent:

- A complex number z in rectangular notation, where x is the real part of z, and y is the imaginary part.
- A complex number z in polar notation, where x is the absolute value of z, and y is the polar angle.
- The coordinates of a point in two dimensions, in rectangular coordinates, where x is the abscissa or horizontal coordinate, and y is the ordinate or vertical coordinate.
- The coordinates of a point in two dimensions, in polar coordinates, where x is the radial coordinate, and y is the polar angle.

If you are not familiar with complex number analysis, you may prefer to consider complex number objects as two-dimensional vectors or point coordinates. Most of the complex number commands return results that are meaningful in ordinary two-dimensional geometry as well as for complex numbers.

With the exception of the P→R (polar-to-rectangular) command, all HP-28C commands that deal with values of complex number objects assume that their arguments are expressed in rectangular notation. Similarly, all commands that return complex number results, except R→P (rectangular-to-polar), express their results in rectangular form.

...COMPLEX

In addition to the commands described in the following sections, certain commands in other menus accept complex number arguments:

- Arithmetic functions +, −, *, /, INV, √, SQ, ^.
- Trigonometric functions SIN, ASIN, COS, ACOS, TAN, ATAN.
- Hyperbolic functions SINH, ASINH, COSH, ACOSH, TANH, ATANH.
- Logarithmic functions EXP, LN, LOG, ALOG.

R→C C→R RE IM CONJ SIGN

The commands R→C, C→R, RE, IM, and CONJ also appear in the fourth row of the ARRAY menu. For their use with array arguments, refer to page 000.

R→C *Real to Complex* Command

Level 2	Level 1		Level 1
x	y	➡	(x, y)
[*R-array*$_1$]	[*R-array*$_2$]	➡	[*C-array*]

R→C combines two real numbers x and y into a complex number. x is the real part, and y the imaginary part of the result. x and y may also be considered as the horizontal and vertical coordinates, respectively, of the point (x, y) in a two-dimensional space.

...COMPLEX

C→R *Complex to Real* Command

Level 1		Level 2	Level 1
(*x*, *y*)	➡	*x*	*y*
[*C-array*]	➡	[*R-array*$_1$]	[*R-array*$_2$]

C→R separates a complex number (or coordinate pair) into its components, returning the real part (or horizontal coordinate) to level 2, and the imaginary part (or vertical coordinate) to level 1.

RE *Real Part* Function

Level 1		Level 1
(*x*, *y*)	➡	*x*
'*symb*'	➡	'RE(*symb*)'
[*array*$_1$]	➡	[*array*$_2$]

RE returns the real part *x* of its complex number argument (*x*, *y*). *x* may also be considered as the horizontal or abscissa coordinate of the point (*x*, *y*).

IM *Imaginary Part* Function

Level 1		Level 1
(*x*, *y*)	➡	*y*
'*symb*'	➡	'IM(*symb*)'
[*array*$_1$]	➡	[*array*$_2$]

IM returns the imaginary part *y* of its complex number argument (*x*, *y*). *y* may also be considered as the vertical or ordinate coordinate of the point (*x*, *y*).

CONJ Conjugate Analytic

Level 1		Level 1
x	➡	x
(x, y)	➡	(x, $-y$)
[*R-array*]	➡	[*R-array*]
[$C\text{-}array_1$]	➡	[$C\text{-}array_2$]
' *symb* '	➡	' CONJ(*symb*) '

CONJ returns the complex conjugate of a complex number. The imaginary part of a complex number is negated.

SIGN Sign Function

Level 1		Level 1
z_1	➡	z_2
' *symb* '	➡	' SIGN(*symb*) '

For a complex number argument (x_1, y_1), SIGN returns the unit vector in the direction of (x_1, y_1):

$$(x_2, y_2) = \left(x_1 / \sqrt{x_1^2 + y_1^2},\;\; y_1 / \sqrt{x_1^2 + y_1^2}\right)$$

...COMPLEX

R→P P→R ABS NEG ARG

R→P — *Rectangular to Polar* — Function

Level 1		Level 1
x	➡	`(x,0)`
`(x,y)`	➡	`(r,θ)`
`'symb'`	➡	`'R→P(symb)'`

R→P converts a complex number in rectangular notation (x, y) to polar notation (r, θ), where

$$r = \text{abs}\ (x,\ y), \quad \theta = \text{arg}\ (x,\ y).$$

P→R — *Polar to Rectangular* — Function

Level 1		Level 1
`(r,θ)`	➡	`(x,y)`
`'symb'`	➡	`'P→R(symb)'`

P→R converts a complex number in polar notation (r, θ) to rectangular notation (x, y), where

$$x = r\cos\theta, \quad y = r\sin\theta.$$

ABS **Absolute Value** Function

Level 1		Level 1
z	➡	|z|
[*array*]	➡	‖array‖
' *symb* '	➡	' ABS(*symb*) '

ABS returns the absolute value of its argument. For a complex argument (x, y), the absolute value is $\sqrt{(x^2 + y^2)}$.

NEG **Negate** Analytic

Level 1		Level 1
z	➡	−z
' *symb* '	➡	' −(*symb*) '
[*array*]	➡	[−*array*]

NEG returns the negative of its argument. When no command line is present, pressing CHS executes NEG. A complete stack diagram for NEG appears in "Arithmetic."

ARG **Argument** Function

Level 1		Level 1
z	➡	θ
' *symb* '	➡	' ARG(*symb*) '

ARG returns the polar angle θ of a complex number (x, y) where

$$\theta = \begin{cases} \arctan y/x & \text{for } x \geqslant 0, \\ \arctan y/x + \pi \operatorname{sign} y & \text{for } x < 0, \text{ radians mode,} \\ \arctan y/x + 180 \operatorname{sign} y & \text{for } x < 0, \text{ degrees mode.} \end{cases}$$

The current angle mode determines whether θ is expressed as degrees or radians.

...COMPLEX

Principal Branches and General Solutions

In general the inverse of a function is a *relation*—for any argument the inverse has more than one value. For example, consider $\cos^{-1} z$; for each z there are infinitely many w's such that $\cos w = z$. For relations such as $\cos^{-1}$ the HP-28C defines functions such as ACOS. These functions return a principal value, which lies in the part of the range defined as the *principal branch*.

The principal branches used in the HP-28C are analytic in the regions where their real-valued counterparts are defined—that is, the branch cut occurs where the real-valued inverse is undefined. The principal branches also preserve most of the important symmetries, such as ASIN($-z$) = $-$ASIN(z).

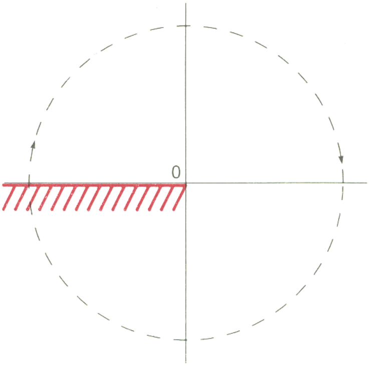
The illustrations below show the principal branches for $\sqrt{\ }$, LN, ASIN, ACOS, ATAN, ACOSH. The graphs of the domains show where the cuts occur: the solid color or black lines are on one side of the cut, and the shaded color or black regions are on the other side. The graphs of the principal branches show where each side of the cut is mapped under the function. Additional dotted lines in the domain graphs and the principal branch graphs help you visualize the function.

Also included are the general solutions returned by ISOL (assuming flag 34, Principal Value, is clear, and radians angle mode is selected). Each general solution is an expression that represent the multiple values of the inverse relations.

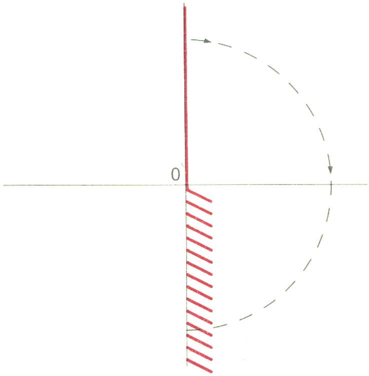
The functions LOG, ^, ASINH, and ATANH are closely related to the illustrated functions. You can determine principal values for LOG, ^, ASINH, and ATANH by extension from the illustrations. Also given are the general solutions for these functions.

Principal Branch for $\sqrt{Z}$

Domain: Z = (*x*, *y*)



Principal Value: W = (*u*, *v*) = √(*x*, *y*)

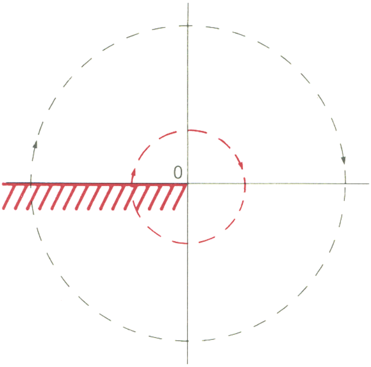


General Solution: 'SQ(W)=Z' 'W' ISOL returns 's1*√Z'.

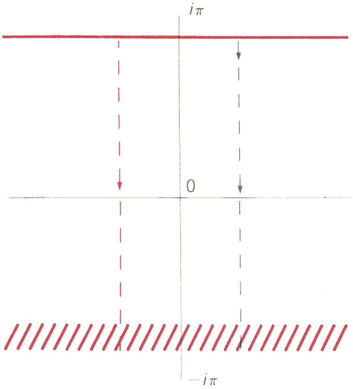
...COMPLEX

Principal Branch for LN(Z)

Domain: Z = (*x*,*y*)



Principal Value: W = (*u*,*v*) = LN(*x*,*y*)



General Solution: 'EXP(W)=Z' 'W' ISOL returns 'LN(Z)+2*π*i*n1'.

Principal Branch for LOG(Z)

You can determine the principal branch for LOG from the illustrations for LN (on the previous page) and the relationship log (z) = ln (z)/ln (10).

General Solution: `'ALOG(W)=Z'  'W'  ISOL` returns `'LOG(Z)+2*π*i*n1/2.30258509299'`

Principal Branch for U^Z

You can determine the principal branch for complex powers from the illustrations for LN (on the previous page) and the relationship $u^z = \exp(\ln(u)\, z)$.

Principal Branch for ASINH(Z)

You can determine the principal branch for ASINH from the illustrations for ASIN (on the following page) and the relationship asinh $z = -i$ asin iz.

General Solution: `'SINH(W)=Z'  'W'  ISOL` returns `'ASINH(Z)+2*π*i*n1'`

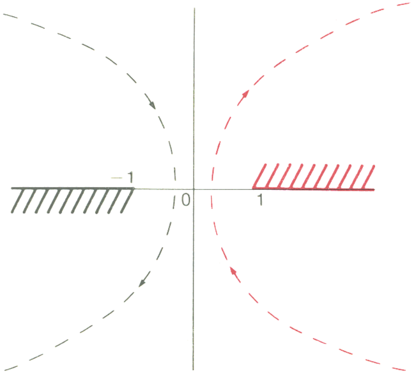
Principal Branch for ATANH(Z)

You can determine the principal branch for ATANH from the illustrations for ATAN (on page 000) and the relationship atanh $z = -i$ atan iz.

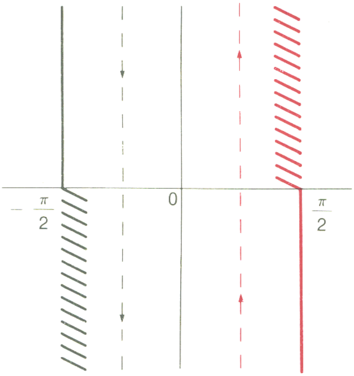
General Solution: `'TANH(W)=Z'  'W'  ISOL` returns `'ATANH(Z)+π*i*n1'`

Principal Branch for ASIN(Z)

Domain: Z = (*x*,*y*)



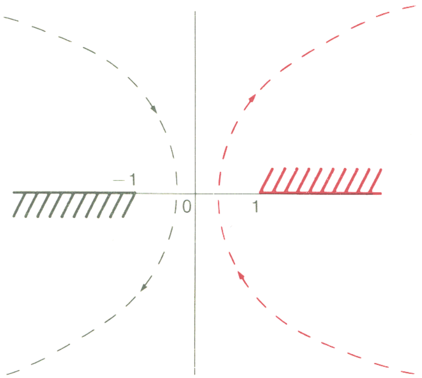
Principal Value: W = (*u*,*v*) = ASIN(*x*,*y*)



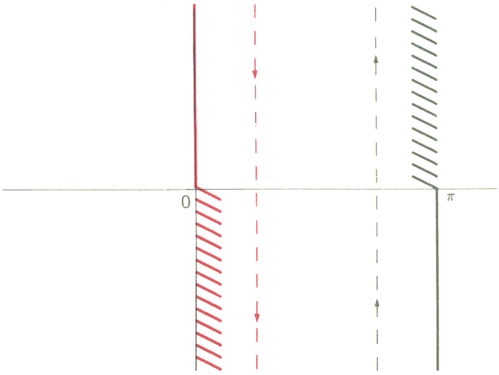
General Solution: 'SIN(W)=Z' 'W' ISOL returns 'ASIN(Z)*(-1)^n1+π*n1'.

Principal Branch for ACOS(Z)

Domain: Z = (*x*, *y*)



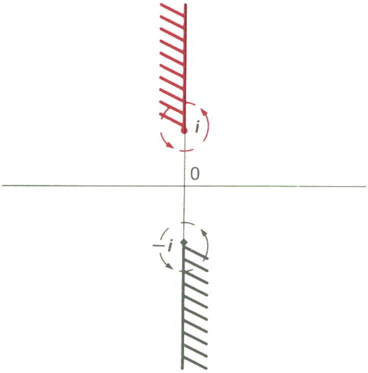
Principal Value: W = (*u*, *v*) = ACOS(*x*, *y*)



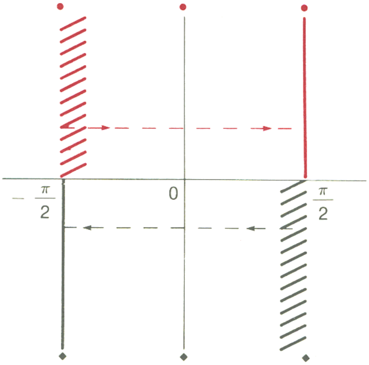
General Solution: 'COS(W)=Z' 'W' ISOL returns 's1*ACOS(Z)+2*π*n1'

Principal Branch for ATAN(Z)

Domain: `Z = (x,y)`



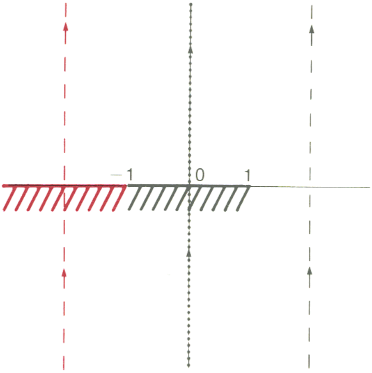
Principal Value: `W = (u,v) = ATAN(x,y)`



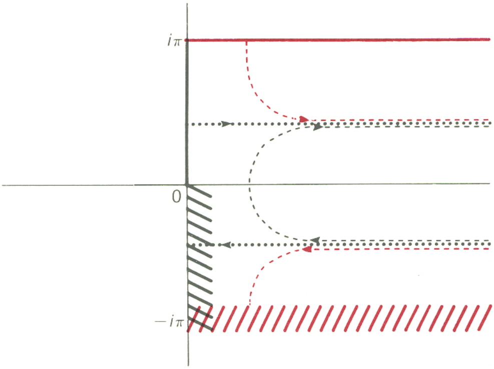
General Solution: `'TAN(W)=Z' 'W' ISOL` returns `'ATAN(Z)+π*n1'`

Principal Branch for ACOSH(Z)

Domain: Z = (*x*,*y*)



Principal Value: W = (*u*,*v*) = ACOSH(*x*,*y*)



General Solution: 'COSH(W)=Z' 'W' ISOL returns 's1*ACOSH(Z)+2*π*i*n1'

LIST

→LIST	**LIST→**	**PUT**	**GET**	**PUTI**	**GETI**
SUB	**SIZE**				

A *list* is an ordered collection of arbitary objects, that is itself an object and hence can be entered into the stack or stored in a variable. The objects in the list are called *elements*, and are numbered from left to right starting with element 1 at the left. The commands in the LIST menu enable you to create and alter lists, and to access the objects contained in lists.

In addition to the LIST menu commands, you can also use the keyboard function + to combine two lists.

+ ***Add*** **Analytic**

Level 2	Level 1		Level 1
{ *list*$_1$ }	{ *list*$_2$ }	➧	{ *list*$_1$ *list*$_2$ }

+ concatenates two lists. That is, it takes two lists from the stack and returns a single list containing the objects from the original two lists.

A complete stack diagram for + is given in the "Arithmetic" section.

→LIST LIST→ PUT GET PUTI GETI

→LIST ***Stack to List*** **Command**

Level *n*+1 … Level 2	Level 1		Level 1
obj$_1$ … *obj*$_n$	*n*	➧	{ *obj*$_1$ … *obj*$_n$ }

→LIST takes an integer number n from level 1, plus n additional objects from levels 2 through $n + 1$, and returns a list containing the n objects.

→LIST is also available in the STACK menu.

LIST→ *List to Stack* **Command**

Level 1		Level n+1 ... Level 2	Level 1
{ obj_1 ... obj_n }	➡	obj_1 ... obj_n	n

LIST→ takes a list of n objects from the stack, and returns the objects comprising the list into separate stack levels 2 through $n + 1$. The number n is returned to level 1.

LIST→ is also available in the STACK menu.

PUT *Put Element* **Command**

Level 3	Level 2	Level 1		Level 1
[$array_1$]	{ *index* }	*x*	➡	[$array_2$]
' *name* '	{ *index* }	*x*	➡	
[$C\text{-}array_1$]	{ *index* }	*z*	➡	[$C\text{-}array_2$]
' *name* '	{ *index* }	*z*	➡	
{ $list_1$ }	*n*	*obj*	➡	{ $list_2$ }
' *name* '	*n*	*obj*	➡	

...LIST

PUT is a general mechanism for element storage into an array or object storage into a list. In this section, we will describe list storage. Array storage is described in the "ARRAY" section.

PUT takes three arguments from the stack. Level 1 must contain the object you wish to store in the list. Level 2 contains a real number specifying the number of the list element you wish to replace. Level 3 may contain either a list or a name:

- If level 3 contains a list, PUT returns that list to the stack with its nth element replaced by the object taken from level 1.
- If level 3 contains a name, then PUT replaces the *n*th element in the list contained in the variable with the object in level 1. The variable can't be a local variable.

GET is the reverse operation to PUT.

GET **_Get Element_** **Command**

Level 2	Level 1		Level 1
[*array*]	{ *index* }	➡	*z*
' *name* '	{ *index* }	➡	*z*
{ *list* }	*n*	➡	*obj*
' *name* '	*n*	➡	*obj*

GET is a general mechanism for recalling an element from an array or a list. In this section, list element recall will be described. Array element recall is described in the "ARRAY" section.

GET takes two arguments from the stack. Level 1 should contain a real number specifying the number of the list element you wish to recall. Level 2 may contain either a list or a name:

- If level 2 contains a list, GET returns the *n*th element of that list to the stack.
- If level 2 contains a name, GET returns to the stack the *n*th element in the list contained in the variable.

PUT is the reverse operation to GET.

PUTI *Put and Increment Index* Command

Level 3	Level 2	Level 1		Level 2	Level 1
[$array_1$]	{ $index_1$ }	x	➡	[$array_2$]	{ $index_2$ }
' *name* '	{ $index_1$ }	x	➡	' *name* '	{ $index_2$ }
[$C\text{-}array_1$]	{ $index_1$ }	z	➡	[$C\text{-}array_2$]	{ $index_2$ }
' *name* '	{ $index_1$ }	z	➡	' *name* '	{ $index_2$ }
{ $list_1$ }	n_1	*obj*	➡	{ $list_2$ }	n_2
' *name* '	n_1	*obj*	➡	' *name* '	n_2

PUTI is a general mechanism for element storage into an array or object storage into a list. In this section, list storage will be described. Array storage is described in the "ARRAY" section.

PUTI stores an object into a list in the same manner as PUT, but also returns the list (or a variable name) and the element number incremented by 1. This leaves the stack ready for you to enter a new object, and then execute PUTI again to store the object into the next position in the list. When the element number is equal to the number of elements in the list, PUTI returns the element number 1, starting over at the start of the list.

PUTI takes three arguments from the stack. Level 1 must contain the object you wish to store in the list. Level 2 contains a real number specifying the number of the list element you wish to replace. Level 3 may contain either a list or a name:

- If level 3 contains a list, PUTI returns that list to level 2, with its *n*th element replaced by the object taken from level 1. The next element number ($n_1 + 1$, or 1) is returned to level 1.
- If level 3 contains a name, PUTI stores the object as the *n*th element in the list contained in the variable. The variable can't be a local variable. The name is returned to level 2, and the next element number ($n_1 + 1$, or 1) is returned to level 1.

GETI is the reverse operation to PUTI.

GETI *Get and Increment Index* Command

Level 2	Level 1		Level 3	Level 2	Level 1
[*array*]	{ $index_1$ }	➡	[*array*]	{ $index_2$ }	*z*
' *name* '	{ $index_1$ }	➡	' *name* '	{ $index_2$ }	*z*
{ *list* }	n_1	➡	{ *list* }	n_2	*obj*
' *name* '	n_1	➡	' *name* '	n_2	*obj*

GETI is a general mechanism for recalling an element from an array or a list. Array element recall is described in the "ARRAY" section.

GETI recalls an element from a list in the same manner as GET, with the added feature that GETI leaves the list on the stack plus the element number incremented by 1, to facilitate successive recalls from the same list. When the element number is equal to the number of elements in the list, GETI returns the element number 1, starting over at the beginning of the list.

GETI takes two arguments from the stack. Level 1 should contain a real number, specifying the number of the list element you wish to recall. Level 2 may contain either a list or a name:

- If level 2 contains a list, GETI returns the *n*th element of that list to level 1. GETI also returns the list to level 3, and the next element number ($n_1 + 1$, or 1) to level 2.
- If level 2 contains a name, GETI returns to level 1 the *n*th element in the list contained in the variable. GETI also returns the name to level 3 and the next element number ($n_1 + 1$ or 1) to level 2.

PUTI is the reverse operation of GETI.

SUB **SIZE**

SUB *Subset* Command

Level 3	Level 2	Level 1		Level 1
"*string*$_1$"	n_1	n_2	➡	"*string*$_2$"
{ *list*$_1$ }	n_1	n_2	➡	{ *list*$_2$ }

SUB takes a list and two integer numbers n_1 and n_2 from the stack, and returns a new list containing the objects that were the elements n_1 through n_2 of the original list. If $n_2 < n_1$, SUB returns an empty list.

SUB works in an analogous manner for character strings. Refer to "STRING".

SIZE *Size* Command

	Level 1		Level 1
	"*string*"	➡	*n*
	{ *list* }	➡	*n*
	[*array*]	➡	{ *list* }
	' *symb* '	➡	*n*

SIZE returns an object representing the size, or dimensions, of a list, array, algebraic, or string argument. For a list, SIZE returns a number n that is the number of elements in the list.

Refer to "ALGEBRA", "ARRAY" and "STRING" for the cases of algebraics, arrays and strings, respectively.

LOGS

LOG	**ALOG**	**LN**	**EXP**	**LNP1**	**EXPM**
SINH	**ASINH**	**COSH**	**ACOSH**	**TANH**	**ATANH**

The LOGS menu contains exponential, logarithmic, and hyperbolic functions. All of these functions accept real and algebraic arguments; all except LNP1 and EXPM accept complex arguments.

LOG ALOG LN EXP LNP1 EXPM

LOG — *Common Logarithm* — Analytic

Level 1		Level 1
z	➡	log *z*
'*symb*'	➡	'LOG(*symb*)'

LOG returns the common logarithm (base 10) of its argument.

An `Infinite Result` exception results if the argument is 0 or (0, 0).

ALOG — *Common Antilogarithm* — Analytic

Level 1		Level 1
z	➡	10^z
'*symb*'	➡	'ALOG(*symb*)'

ALOG returns the common antilogarithm (base 10) of its argument—that is, 10 raised to the power given by the argument.

For complex arguments:

$$\text{alog}\,(x, y) = \exp cx \cos cy + i \exp cx \sin cy,$$

where $c = \ln 10$. (Computation is performed in radians mode).

LN   *Natural Logarithm*   Analytic

Level 1		Level 1
z	➡	ln z
'*symb*'	➡	'LN(*symb*)'

LN returns the natural logarithm (base e) of its argument.

An Infinite Result exception results if the argument is 0 or (0, 0).

EXP   *Exponential*   Analytic

Level 1		Level 1
z	➡	exp z
'*symb*'	➡	'EXP(*symb*)'

EXP returns the exponential, or natural antilogarithm (base e) of its argument—that is, e raised to the power given by the argument. EXP returns a more accurate result than e^, since EXP uses a special algorithm to compute the exponential.

For complex arguments:

$$\exp\,(x, y) = \exp x \cos y + i \exp x \sin y.$$

(Computation is performed in radians mode).

LNP1 *Natural Log of 1+x* Analytic

Level 1		Level 1
x	➡	ln (1+*x*)
'*symb*'	➡	'LNP1(*symb*)'

LNP1 returns ln(1 + *x*), where *x* is the real-valued argument. LNP1 is primarily useful for determining the natural logarithm of numbers close to 1. LNP1 provides a more accurate result for ln(1 + *x*), for *x* close to zero, than can be obtained using LN.

Arguments less than 1 cause an `Undefined Result` error.

EXPM *Exponential Minus 1* Analytic

Level 1		Level 1
x	➡	exp (*x*)−1
'*symb*'	➡	'EXPM(*symb*)'

EXPM returns $e^x - 1$, where *x* is the real-valued argument. EXPM is primarily useful for determining the exponential of numbers close to 0. EXPM provides a more accurate result for $e^x - 1$, for *x* close to 0, than can be obtained using EXP.

...LOGS

SINH ASINH COSH ACOSH TANH ATANH

These are the hyperbolic functions and their inverses.

SINH — *Hyperbolic Sine* — Analytic

Level 1		Level 1
z	➡	sinh z
'*symb*'	➡	'SINH(*symb*)'

SINH returns the hyperbolic sine of its argument.

ASINH — *Inverse Hyperbolic Sine* — Analytic

Level 1		Level 1
z	➡	asinh z
'*symb*'	➡	'ASINH(*symb*)'

ASINH returns the inverse hyperbolic sine of its argument. For real arguments $|x| > 1$, ASINH returns the complex result for the argument $(x, 0)$.

COSH *Hyperbolic cosine* Analytic

Level 1		Level 1
z	➡	cosh *z*
'*symb*'	➡	'COSH(*symb*)'

COSH returns the hyperbolic cosine of its argument.

ACOSH *Inverse Hyperbolic Cosine* Analytic

Level 1		Level 1
z	➡	acosh *z*
'*symb*'	➡	'ACOSH(*symb*)'

ACOSH returns the inverse hyperbolic cosine of its argument. For real arguments $|x| < 1$, ACOSH returns the complex result obtained for the argument $(x, 0)$.

TANH *Hyperbolic Tangent* Analytic

Level 1		Level 1
z	➡	tanh *z*
'*symb*'	➡	'TANH(*symb*)'

TANH returns the hyperbolic tangent of its argument.

...LOGS

ATANH *Inverse Hyperbolic Tangent* Analytic

Level 1		Level 1
z	➡	atanh *z*
'*symb*'	➡	'ATANH(*symb*)'

ATANH returns the inverse hyperbolic tangent of its argument. For real arguments $|x| > 1$, ATANH returns the complex result obtained for the argument $(x, 0)$.

For a real argument $x = \pm 1$, an `Infinite Result` exception occurs. If flag 59 is clear, the sign of the result (MAXR) is that of the argument.

MODE

STD	**FIX**	**SCI**	**ENG**	**DEG**	**RAD**
+CMD	**−CMD**	**+LAST**	**−LAST**	**+UND**	**−UND**
+ML	**−ML**	**RDX.**	**RDX,**	**PRMD**	

The MODE menu contains menu keys that control various calculator *modes*: number display mode, angle mode, recovery modes, radix mode, and multi-line display mode.

The menu key labels in this menu also act as annunciators to show you the current states of the modes. Each mode has two or more keys that set the mode; at any time, one menu key label for each mode is displayed in normal video (black-on-white). The normal video key label indicates the current mode setting. For example, the angle mode is set by the DEG and RAD commands. When the angle mode is *degrees*, the DEG label is normal video and the RAD label is shown in inverse video (white-on-black). If you press RAD, the RAD label changes to normal video (and the DEG label to inverse) to indicate that the angle mode is *radians*.

In immediate entry mode, all MODE commands except FIX, SCI, and ENG (which require arguments) execute without performing ENTER, leaving the command line unchanged.

STD FIX SCI ENG DEG RAD

These functions set the number display mode and the angle mode.

The number display functions STD, FIX, SCI, and ENG control the display format of floating-point numbers, as they appear in stack displays of all types of objects. In the algebraics, non-integer floating-point numbers are displayed in the current format and integers are always displayed in STD format.

...MODE

The current display mode is encoded in flags 49 and 50. Executing any of the display functions alters the states of these flags; conversely, setting and clearing these flags will affect the display mode. The correspondence is as follows:

Mode	Flag 49	Flag 50
Standard	0	0
Fix	1	0
Scientific	0	1
Engineering	1	1

Flags 53–56 encode (in binary) the number of decimal digits, from 0 through 11. Flag 56 is the most significant bit.

STD *Standard* Command

➡

STD sets the number display mode to *standard format*. Standard format (ANSI Minimal BASIC Standard X3J2) produces the following results when displaying or printing a number:

- Numbers that can be represented exactly as integers with 12 or fewer digits are displayed without a radix or exponent. Zero is displayed as `0`.
- Numbers that can be represented exactly with 12 or fewer digits, but not as integers, are displayed with a radix but no exponent. Leading zeroes to the left of the radix and trailing zeroes in the fractional part are omitted.

- All other numbers are displayed in the following format:

  (sign) mantissa E (sign) exponent

  where the value of the mantissa is in the range $1 \leqslant x < 10$, and the exponent is represented by one to three digits. Trailing zeroes in the mantissa and leading zeroes in the exponent are omitted.

The following table provides examples of numbers displayed in standard format:

Number	Displayed As	Representable With 12 Digits?
10^{11}	`100000000000`	Yes (integer)
10^{12}	`1.E12`	No
10^{-12}	`.000000000001`	Yes
1.2×10^{-11}	`.000000000012`	Yes
1.23×10^{-11}	`1.23E-11`	No
12.345	`12.345`	Yes

FIX *Fix* Command

Level 1	
n ➡	

FIX sets the number display mode to *fixed format*, and uses a real number argument to set the number of fraction digits to be displayed in the range 0 through 11. The rounded value of the argument is used. If this value is greater than 11, 11 is used; if less than 0, 0 is used.

In fixed format, displayed or printed numbers appear as

(sign) mantissa

The mantissa appears rounded to *n* places to the right of the decimal, where *n* is the specified number of digits. While fixed format is active, the HP-28C automatically displays a value in scientific format in either of these two cases:

- If the number of digits to be displayed exceeds 12.
- If a non-zero value rounded to *n* places past the decimal point would be displayed as zero in fixed format.

SCI *Scientific* Command

Level 1	
n ➡	

SCI sets the number display mode to *scientific format*, and uses a real number argument to set the number of significant digits to be displayed in the range 0 through 11. The rounded value of the actual argument is used. If this value is greater than 11, 11 is used; if less than 0, 0 is used.

In scientific format, numbers are displayed or printed in scientific notation to $n + 1$ significant digits, where *n* is the specified number of digits (the argument for SCI). A value appears as

```
(sign) mantissa E (sign) exponent
```

where $1 \leqslant \textit{mantissa} < 10$.

ENG *Engineering* Command

Level 1	
n ➡	

ENG sets the number display mode to *engineering format*, and uses a real number argument to set the number of significant digits to be displayed, in the range 0 through 11. The rounded value of the argument is used. If this value is greater than 11, 11 is used; if less than 0, 0 is used.

In engineering format, a displayed or printed number appears as

```
(sign) mantissa E (sign) exponent
```

where $1 \leqslant mantissa < 1000$, and the exponent is a multiple of 3. The number of significant digits displayed is one greater than the argument specified. If a displayed value has an exponent of -499, it is displayed in scientific format.

DEG *Degrees* Command

➡	

DEG (*degrees*) sets the current angle mode to degrees. In degrees mode:

Real-number arguments. Functions that take real-valued angles as arguments interpret those angles as being expressed in degrees. (Complex arguments for SIN, COS and TAN are always assumed to be in radians.)

Real-number results. Functions that give real-valued angles as results return those angles expressed in degrees: ASIN, ACOS, ATAN, ARG, and R→P. (Complex results returned by ASIN or ACOS for arguments outside of the domain $x \leqslant 1$ are always expressed in radians.)

Executing DEG turns off the (2π) annunciator and clears user flag 60.

RAD *Radians* Command

	➡	

RAD (*radians*) sets the current angle mode to radians. In radians mode:

Real-number arguments. Functions that take real-valued angles as arguments interpret those angles as being expressed in radians. (Complex arguments for SIN, COS and TAN are always assumed to be in radians.)

Real-number results. Functions that give real-valued angles as results return those angles expressed in radians: ASIN, ACOS, ATAN, ARG, and R→P. (Complex results returned by ASIN or ACOS for arguments outside of the domain $x \leqslant 1$ are always expressed in radians.)

Executing RAD turns on the (2π) annunciator, and sets user flag 60.

+CMD −CMD +LAST −LAST +UND −UND

These six menu keys control the three recovery mechanisms COMMAND, LAST, and UNDO. In each label, "+" means "ON," and "−" means "OFF."

None of these operations can be included in programs; the menu keys are always immediate execute keys. LAST can be enabled or disabled programmatically by setting or clearing flag 31.

COMMAND Enabled/Disabled

Operation	Description
+CMD	Enables COMMAND. Previous command lines can now be recovered by pressing ■ COMMAND.
-CMD	Disables COMMAND and recovers the memory in which previous command lines are saved. Pressing ■ COMMAND now causes a COMMAND Stack Disabled error.

LAST Enabled/Disabled

Operation	Description
+LAST	Enables LAST. Commands that take one to three arguments now save those arguments for recall by LAST.
-LAST	Disables LAST and recovers the memory used to save arguments. Executing LAST now causes a LAST Disabled error.

UNDO Enabled/Disabled

Operation	Description
+UND	Enables UNDO. The stack is now saved at each execution of ENTER, and pressing ■ UNDO replaces the current stack with the last saved stack.
-UND	Disables UNDO and recovers the memory occupied by the saved stack. Pressing ■ UNDO now causes an `UNDO Disabled` error.
	The effects of +UND and -UND are "local" to the current suspended program. That is, if a program is suspended, executing +UND and -UND changes the stack save feature only while that program is suspended.

+ML   −ML   RDX.   RDX,   PRMD

Multi-line Enabled/Disabled

Objects on the stack are displayed in either of two general display formats: multi-line and compact. Objects in all levels higher than level 1 are always displayed in compact format, in which only one display line is used for the object (if the complete object cannot be shown, an ellipsis "..." replaces the rightmost character).

You have the option, using the +ML and -ML menu keys, of displaying objects in level 1 in compact format (-ML), or in multi-line format (+ML). In multi-line display mode:

- Matrices are shown with each row on a separate display line, with as many elements displayed on each line as possible with the current number display mode.
- Procedures and lists are displayed with newlines inserted into the displayed text so that all of their definitions are visible. Additional procedures, lists, and matrices contained within the definitions are also displayed in multiple lines.
- Numbers, complex numbers, vectors, names, and strings are not broken across display lines, and thus still may be truncated in the display. As in compact display, hidden characters are indicated by an ellipsis "..." in the rightmost character position.

If the full display of the object requires more lines of the display than are available (four with the cursor menu, less one for any other menu, less one or more if there is a command line active), you can view additional lines with ■ VIEW↑ or ■ VIEW↓.

The current multi-line/compact mode choice is also represented by flag 45. Multi-line display mode corresponds to flag 45 set.

Operation	Description
+ML	Selects multi-line display mode for objects in level 1, and sets flag 45. With multi-line mode on, one or more display lines are used to display the object in level 1. Level 1 objects printed on the HP-82240A printer in trace mode will also be printed in a multi-line mode.
-ML	Selects compact display mode for objects in level 1, and clears flag 45. Level 1 objects printed on the HP-82240A printer in trace mode are also printed in compact mode.

...MODE

Period/Comma for Radix Mark

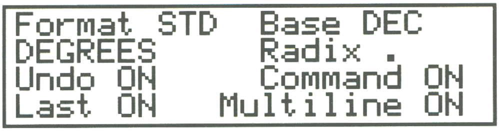
RDX. and RDX, allow you to select the character that is used to represent the *radix,* that is, the symbol that separates the integer and fractional parts of a floating-point number. The choice is either the period "." or the comma ","; the character that is not the current radix character (called the *non-radix*) can be used interchangeably with the space as an object separator when you enter objects into the command line (except within algebraic objects, where the non-radix is used to separate arguments of multiple-argument functions).

Operation	Description
RDX.	Sets the radix mark that separates the integer and fractional parts of a number to be a period (decimal point), and clears user flag 48. In this mode, the comma is called the *non-radix*, and can be used in the command line interchangeably with the space, except within algebraic objects.
RDX,	Sets the radix mark that separates the integer and fractional parts of a number to be a comma, and sets user flag 48. In this mode, the period is called the *non-radix*, and can be used in the command line interchangeably with the space, except within algebraic objects.

PRMD *Print Modes* Command

➡

PRMD displays and prints a listing of current HP-28C modes. The listing shows the states of the number display mode, multiline mode, the angle mode, the binary integer base, and the radix mode, and whether the UNDO, COMMAND, and LAST features are enabled or disabled. A typical listing looks like this:



```
Format STD  Base DEC
DEGREES     Radix .
Undo ON     Command ON
Last ON   Multiline ON
```

PLOT

STEQ	**RCEQ**	**PMIN**	**PMAX**	**INDEP**	**DRAW**
PPAR	**RES**	**AXES**	**CENTR**	***W**	***H**
STOΣ	**RCLΣ**	**COLΣ**	**SCLΣ**	**DRWΣ**	
CLLCD	**DISP**	**PIXEL**	**DRAX**	**CLMF**	**PRLCD**

The commands in the PLOT menu give you the capability of creating special displays that supersede the normal stack and menu display. You can plot mathematical functions, make scatter plots of statistical data, display data while a program is executing, and digitize information from plots.

The Display

The HP-28C liquid-crystal display (LCD) is an array of 32 rows of 137 *pixels* (dots), which is organized as four rows of 23 character spaces. A character space is six pixels wide by eight pixels high, with the exception of the rightmost character space in each row, which is five pixels wide. Normally, display characters are five pixels wide, which leaves a blank column of pixels between characters.

The default display shows menu key labels, the command line, and the stack. You can temporarily replace the default display in the following ways:

- **Clearing the display.** The command CLLCD blanks the entire display (except for the annunciators).
- **Displaying messages.** Using DISP, you can display objects in one or more of the four display lines.
- **Displaying graphical data.** The commands PIXEL, DRAW, and DRWΣ allow you to plot individual data points, mathematical functions, and points from the statistics matrix, respectively. DRAX allows you to draw axes within a plot.

When you execute any of these commands, a system message flag is set automatically, to prevent the normal stack and menu display from overwriting your special display. The message flag is cleared, and the normal display restored, when you press any key after all active procedures have completed execution, that is, when the busy annunciator ((•)) is off. You can use the command CLMF in a program to clear the message flag.

There are also keys that produce special displays and redefine the keyboard temporarily:

- ■[CATALOG] and ■[UNITS] create catalog displays, and provide menu keys and letter keys for you to control the catalogs. ■[CATALOG] and ■[UNITS] are described in "The Command CATALOG" and "Unit Conversion", respectively.
- DRAW and DRWΣ execute DRAW and DRWΣ, respectively, to produce mathematical function graphs and scatter plots of statistical data. In addition, if DRAW or DRWΣ is ececuted by pressing its menu key while the plot displays are visible, you can use the menu keys to move a cursor about the display or to return digitized coordinates to the stack.

Display Positions

For the purpose of displaying character strings, the display is treated as four lines of characters. The topmost line of the display is line 1, the next line is line 2, and so on. Each line must be displayed as a single entity; you can not display single characters at arbitary positions. DISP displays an object, where the object is represented as a character string equivalent to the multi-line stack display of the object.

...PLOT

For graphical data displays, the display is treated as a grid of 32 × 137 dots, or *pixels*. A pixel is specified by its *coordinates*, a complex number representing an ordered pair of coordinates (x, y), where x is the horizontal coordinate and y is the vertical coordinate. (We will use the letters x and y to represent the horizontal and vertical directions during this discussion, but you can use any variable names you choose for plotting on the HP-28C.)

The scaling of coordinates to pixels is established by the coordinates of the corner points P_{max} and P_{min}, which you set with the commands PMAX and PMIN, respectively. P_{max} is the upper-rightmost pixel in the display; its coordinates are (x_{max}, y_{max}). P_{min} (x_{min}, y_{min}) is the lower-leftmost pixel. The default coordinates of these points are $P_{max} = (6.8, 1.6)$ and $P_{min} = (-6.8, -1.5)$. The coordinates of the center of a particular pixel are

$$x = n_x\, w_x + x_{min}$$

$$y = n_y\, w_y + y_{min}$$

where n_x is the horizontal pixel number and n_y is the vertical pixel number (P_{min} has $n_x = 0$ and $n_y = 0$; P_{max} has $n_x = 136$, $n_y = 31$). w_x and w_y are the horizontal and vertical pixel widths:

$$w_x = (x_{max} - x_{min})/136.$$

$$w_y = (y_{max} - y_{min})/31.$$

The pixel with $n_x = 68$ and $n_y = 15$ is defined as the *center* pixel. With the default values for P_{max} and P_{min}, the center pixel has coordinates $(0, 0)$.

Mathematical Function Plots

A mathematical *function plot* is a plot of the values of a procedure stored in the variable EQ (the same used by the Solver), as a function of a specified *independent variable*. The procedure is fully evaluated for each of 137/*r* values of the independent variable from x_{min} to $x_{max}$where *r* is the *resolution* of the plot. A dot (pixel) is added to the graph for each coordinate pair (*independent-variable-value, procedure-value),* as long as the procedure value is within the plot range between y_{min} and y_{max}. The plot also includes axes with tick marks every 10 pixels.

The actual plot is produced by the command DRAW. If you execute DRAW directly by pressing the menu key DRAW, you will be able to use the cursor keys to digitize data from the plot.

A function plot will produce one or two plotted curves, according to the definition of the EQ procedure:

- If EQ contains an algebraic expression without an equals sign, DRAW will plot a single curve corresponding to the value of the expression for each value of the independent variable within the plot range.
- If EQ contains an algebraic equation, DRAW will plot two curves, one for each side of the equation. Note that the intersections of the two curves occur at the values of the independent variable that are the roots of the equation, that can be found by the Solver.
- If EQ contains a program, it will be treated as an algebraic expression and plotted as a single curve. This presumes that the program obeys the syntax of an algebraic expression: it must take no arguments from the stack, and return exactly one object to the stack.

...PLOT

The general procedure for obtaining a function plot is summarized below. For details, refer to the descriptions of the individual commands.

1. Store the procedure to be plotted in EQ, using STEQ.
2. Select the independent variable with INDEP.
3. Select the plot ranges, using PMIN, PMAX, CENTR, *H, and *W.
4. Specify the intersection of the axes, using AXES.
5. Select the plot resolution with RES.
6. Execute DRAW.

Any of steps 1–5 can be omitted, in which case the current values are used.

Statistical Scatter Plots

A statistical *scatter plot* is a plot of individual points taken from the current statistics array stored in variable ΣDAT. You may specify any column of coordinate values from the array to correspond to the horizontal coordinate, and any other column for the vertical coordinate. One point is then plotted for each data point in the matrix.

The general procedure for obtaining a scatter plot is summarized below. For details, refer to the descriptions of the individual commands.

1. Store the statistical data to be plotted in ΣDAT, using STOΣ.
2. Select the horizontal and vertical coordinate columns with COLΣ.
3. Select the plot ranges, using SCLΣ for automatic scaling, or PMIN, PMAX, CENTR, *H, and *W.

4. Specify the intersection of the axes, using AXES.
5. Execute DRWΣ.

Any of the steps 1–4 can be omitted, in which case the current values are used.

Interactive Plots

If you execute DRAW or DRWΣ by pressing the corresponding menu key, the HP-28C enters an interactive plot mode that allows you to digitize information from the plot while viewing it. When you start an interactive plot:

1. The display is cleared.
2. Either DRAW or DRWΣ is executed to produce the appropriate plot. (If you press [ON] before the plotting is finished, plotting of points halts, and the interactive mode begins).
3. A cursor in the form of a small cross (+) appears at the center of the display. (If the axes are drawn through the center, the cursor will not be visible until you move it.)
4. The menu keys are activated as cursor/digitizer keys:
   - The two leftmost menu keys return the coordinates of the cursor to the stack without terminating the plot display. [INS] returns the coordinates as a complex number *(x, y)*. [DEL] returns the coordinates as two real numbers, *x* in level 2 and *y* in level 1.
   - The four rightmost menu keys act like the regular cursor control keys, moving the cursor up, down, left, or right, or all the way in one direction if you first press 🟥.

The interactive plot mode continues until you press the [ON] key.

You can digitize as many points as you wish during the interactive plot, by repeated use of [INS], [DEL], and the cursor keys.

...PLOT

Plot Parameters

The scaling factors necessary for converting a coordinate pair to a display position, and vice-versa, are stored as a list of objects in the variable PPAR. We will refer to them collectively as the *plot parameters*. They are:

Parameter	Description
P_{min}	A complex number representing the coordinates of the lower leftmost pixel. Set by PMIN, CENTR, ∗H, ∗W, and SCLΣ.
P_{max}	A complex number representing the coordinates of the upper rightmost pixel. Set by PMAX, CENTR, ∗H, ∗W, and SCLΣ.
Independent variable	The variable name corresponding to the horizontal axis in a mathematical function plot. Set by INDEP.
Resolution	A real positive integer representing the spacing of plotted points in a function plot. Set by RES.
P_{axes}	A complex number representing the coordinates of the intersection of the plot axes. Set by AXES.

STEQ RCEQ PMIN PMAX INDEP DRAW

This set of commands allows you to select a procedure for a function plot, set the primary plot parameters, and plot the procedure.

STEQ *Store Equation* Command

Level 1	
obj ➡	

STEQ takes an object from the stack, and stores it in the variable EQ ("EQuation"). It is equvalent to `'EQ' STO`.

EQ is used to hold a procedure (the current equation) used as an implicit argument by the Solver and by DRAW, so STEQ's argument should normally be a procedure.

RCEQ *Recall Equation* Command

	Level 1
➡	*obj*

RCEQ returns the contents of the variable EQ. It is equivalent to `'EQ' RCL`.

PMIN *Plot Minima* Command

Level 1	
(x,y) ➡	

PMIN sets the coordinates of the lower leftmost pixel in the display to be the point (*x*, *y*). The complex number (*x*, *y*) is stored as the first item in the list contained in the variable PPAR.

...PLOT

PMAX *Plot Maxima* Command

Level 1	
(x,y) ➡	

PMAX sets the coordinates of the upper-rightmost pixel in the display to be the point (*x*, *y*). The complex number (*x*, *y*) is stored as the second item in the list contained in the variable PPAR.

INDEP *Independent* Command

Level 1	
'*name*' ➡	

INDEP takes a name from the stack, and stores it as the independent variable name, the third item in the list contained in the variable PPAR. For subsequent executions of DRAW, the name will be used as the independent variable corresponding to the horizontal axis (abscissa) of the plot.

DRAW *Draw* Command

➡

DRAW produces mathematical function plots on the HP-28C display. If you execute DRAW by pressing the DRAW menu key, an interactive plot is produced, as described in "Interactive Plots" on page 203.

DRAW automatically executes DRAX to draw axes, then plots one or two curves representing the value(s) of the current equation at each of $137/r$ values of the independent variable. The current equation is the procedure stored in the variable EQ.

If EQ contains an algebraic equation, the two sides of the equation are plotted separately, yielding two curves. If the current equation is an algebraic expression or a program, one curve is plotted.

The resolution r determines the number of plotted points. $r = 1$ means a point is plotted for every column of display pixels; $r = 2$ means every other column; and so on. r is set by the RES command. The default value of r is 1; larger values of r may be used to reduce plotting time.

DRAW checks the current equation to see if it contains at least one reference, direct or indirect, to the independent variable. If the independent variable was never selected, the first variable in the current equation is used (and stored in PPAR). If the independent variable is not referenced in the current equation, the message

name$_1$ `Not In Equation`
`Using` *name*$_2$

is displayed momentarily before the display is cleared and before the actual plot begins. Here *name*$_1$ is the current independent variable defined in PPAR, and *name*$_2$ is the first variable found in the current equation. If the current equation contains no variables, the second line of the warning message is replaced by `Constant Equation`. (The independent variable name in PPAR will then be constant.)

PPAR RES AXES CENTR *W *H

These commands provide alternate ways of setting plot parameters.

PPAR — *Recall Plot Parameters* — Operation

	Level 1
➡	*{ plot parameters }*

The PPAR command is a convenient way for you to examine the current plot parameters.

PPAR is a variable containing a list of the plot parameters, in the form

{ (x_{min}, y_{min}) (x_{max}, y_{max}) *independent resolution* (x_{axis}, y_{axis}) }

Pressing PPAR returns the list to the stack. The contents of the list are described in "Plot Parameters" on page 204.

RES — *Resolution* — Command

Level 1	
n →	

RES sets the *resolution* of mathematical function plots (DRAW) to the value n. n is stored as the fourth item in the list contained in the variable PPAR. n determines the number of plotted points: $n = 1$ means a point is plotted for every column of display pixels; $n = 2$ means every other column; and so on. The default value of n is 1; you may wish to use larger values of n to reduce plotting time.

AXES — *Axes* — Command

Level 1	
(x,y) ➡	

AXES sets the coordinates of the intersection of the plot axes (drawn by DRAX, DRAW, or DRWΣ), to be the point (x, y). The complex number (x, y) is stored as the fifth and last item in the list contained in the variable PPAR. The default coordinates are (0, 0).

CENTR — *Center* — Command

Level 1	
(x,y) ➡	

CENTR adjusts the plot parameters so that the point represented by the argument (x, y) corresponds to the center pixel ($n_x = 68$, $n_y = 15$) of the display. The height and width of the plot are not changed. P_{max} and P_{min} are replaced by P_{max}' and P_{min}', where:

$$x_{max}' = x + \tfrac{1}{2}(x_{max} - x_{min}), \quad y_{max}' = y + \tfrac{16}{31}(y_{max} - y_{min})$$

$$x_{min}' = x - \tfrac{1}{2}(x_{max} - x_{min}), \quad y_{min}' = y - \tfrac{15}{31}(y_{max} - y_{min})$$

*W — *Multiply Width* — Command

Level 1	
x ➡	

*W adjusts the plot parameters so that both x_{min} and x_{max} are multiplied by the number x.

$$x_{min}' = x \times x_{min}$$

$$x_{max}' = x \times x_{max}$$

*H — *Multiply Height* — Command

Level 1	
x ➡	

*H adjusts the plot parameters so that both y_{min} and y_{max} are multiplied by the number x.

$$y_{min}' = x \times y_{min}$$

$$y_{max}' = x \times y_{max}$$

STOΣ RCLΣ COLΣ SCLΣ DRWΣ

This group of commands allows you to create statistics scatter plots. See "STAT" for a description of the general statistical capabilities of the HP-28C.

STOΣ — *Store Sigma* — Command

Level 1	
[*R-array*] ➡	

STOΣ takes a real array from the stack and stores it in the variable ΣDAT. Executing STOΣ is equivalent to executing `'ΣDAT' STO`. The stored array becomes the current statistics matrix.

RCLΣ — *Recall Sigma* — Command

	Level 1
➡	*obj*

RCLΣ returns the current contents of the variable ΣDAT. RCLΣ is equivalent to `'ΣDAT' RCL`.

COLΣ *Sigma Columns* Command

Level 2	Level 1		
n_1	n_2	➡	

COLΣ takes two real integers, n_1 and n_2, and stores them as the first two items in the list contained in variable ΣPAR. The numbers identify column numbers in the current statistics matrix ΣDAT, and are used by statistics commands that work with pairs of columns. Refer to "Stat" for details about ΣPAR.

n_1 designates the column corresponding to the independent variable for LR, or the horizontal coordinate for DRWΣ or SCLΣ. n_2 designates the dependent variable or the vertical coordinate. For CORR and COV, the order of the two column numbers is unimportant.

If a two-column command is executed when ΣPAR does not yet exist, it is automatically created with default values $n_1 = 1$ and $n_2 = 2$.

SCLΣ *Scale Sigma* Command

➡	

SCLΣ causes an automatic scaling of the plot parameters in PPAR so that a subsequent statistics scatter plot exactly fills the display. That is, the horizontal coordinates of P_{max} and P_{min} are set to be the maximum and minimum coordinate values, respectively, in the independent data column of the current statistics matrix. Similarly, the vertical coordinates of P_{max} and P_{min} are set from the dependent data column. The independent and dependent data column numbers are those stored in the variable ΣPAR.

DRWΣ *Draw Sigma* Command

➡

DRWΣ automatically executes DRAX to draw axes, then creates a statistical scatter plot of the points represented by pairs of coordinate values taken from the independent and dependent columns of the current statistics matrix ΣDAT. If you execute DRWΣ by pressing the DRWΣ menu key, an interactive plot is produced, as described in "Interactive Plots" on page 203.

The independent and dependent columns are specified in the variable ΣPAR (default 1 and 2, respectively). DRWΣ plots one point for each data point in the statistics matrix. For each point, the horizontal coordinate is the coordinate value in the independent data column, and the vertical coordinate is the coordinate value in the dependent data column.

CLLCD DISP PIXEL DRAX CLMF PRLCD

These commands allow you to create special displays, and to print an image of the display on the HP-82440A printer.

CLLCD *Clear LCD* Command

➡

CLLCD clears (blanks) the HP-28C display (except the annunciators) and sets the system message flag.

DISP *Display* Command

Level 2	Level 1	
obj	*n*	➡

DISP displays *obj* in the *n*th line of the display, where *n* is a real integer. *n* = 1 indicates the top line of the display; *n* = 4 is the bottom line. DISP sets the system message flag to suppress the normal stack display.

An object is displayed by DISP in the same form as would be used if the object were in level 1 in the multi-line display format, except for strings, which are displayed without the surrounding " delimiters, to facilitate the display of messages. If the object display requires more than one display line, the display starts in line *n*, and continues down the display either to the end of the object or the bottom of the display.

PIXEL *Pixel* Command

Level 1	
(*x*, *y*)	➡

PIXEL turns on one pixel at the coordinates represented by the complex number (*x*, *y*) and sets the system message flag.

DRAX *Draw Axes* Command

➡

...PLOT

DRAX draws a pair of axes on the display, and sets the system message flag. The axes intersect at the point P_{axes}, specified in the variable PPAR. Tick marks are placed on the axes at every 10th pixel.

CLMF — *Clear Message Flag* — Command

➡

CLMF clears the internal message flag set by CLLCD, DISP, PIXEL, DRAX, DRAW, and DRWΣ. Including CLMF in a program, after the last occurrence of any of these words, causes the normal stack display to be restored when the program completes execution.

PRLCD — *Print LCD* — Command

➡

PRLCD provides a means by which you can print copies of mathematical function plots and statistical scatter plots. Since PRLCD will print only a copy of the current display, you must include PRLCD and DRAW (or DRWΣ) in the same command line. For example:

```
CLLCD DRAW PRLCD [ENTER]
```

will clear the LCD, plot the current equation, then print a replica of the display.

PRINT

PR1	**PRST**	**PRVAR**	**PRLCD**	**TRACE**	**NORM**
PRSTC	**PRUSR**	**PRMD**	**CR**		

The HP-28C transmits text and graphics data to the HP 82240A Printer via an infrared light link. The infrared light-emitting diode is situated on the top edge of the right-hand HP-28C case. Before printing, check that the printer can receive the infrared beam from the HP-28C. Refer to the printer manual for more information about printer operation.

You can use the print commands to print objects, variables, stack levels, plots, and so on. In addition, you can select TRACE mode to automatically print a continuous record of your calculations.

The 🖨 annunciator appears whenever the HP-28C transmits data from the infrared diode. The calculator can't determine whether printing is actually occurring because the transmission is one-way only. Make sure that TRACE mode is not active unless a printer is present—otherwise, the frequent infrared transmissions slow down keyboard operations.

Print Formats

Multi-line objects can be printed in compact format or multi-line format. Compact print format is identical to compact display format. Multi-line printer format is similar to multi-line display format, except that the following objects are fully printed:

- Strings and names that are more than 23 characters long are continued on the next printer line.

- The real and imaginary parts of complex numbers are printed on separate lines if they don't fit on the same line.
- Arrays are printed with an index before each element. For example, the index `1,1:` precedes the first element.

In TRACE mode, the print format depends on whether multi-line display format is enabled or disabled (flag 45 is set or clear). The print command PRSTC (*print stack compact*) prints in compact format. All other print commands print in multi-line format.

Faster Printing

When the printer is battery powered, its speed declines as its batteries discharge. The HP-28C normally paces data transmission to match the printer's speed when its batteries are nearly exhausted.

When your printer is powered by an AC adapter, it can sustain a higher speed. You can increase the calculator's data transmission rate to match the higher speed of the printer by setting flag 52. For subsequent battery-powered printing, clear flag 52 to return to slower data transmission.

Don't set flag 52 when the printer is battery powered. Although a printer with fresh batteries can print at the higher rate, it will eventually slow down enough to lose data sent by the HP-28C. This loss of data corrupts printed output and can cause the printer to change its configuration.

Configuring the Printer

You can set various printer modes by sending *escape sequences* to the printer. An escape sequence consists of the escape character (character 27) followed by an additional character. When the printer receives an escape sequence, it switches into the selected mode. The escape sequence itself isn't printed. The HP 82240A printer recognizes the following escape sequences.

Printer Mode	Escape Sequence
Print Column Graphics	27 001 . . .166
No Underline*	27 250
Underline	27 251
Single Wide Print*	27 252
Double Wide Print	27 253
Self Test	27 254
Reset	27 255

* Default mode.

You can use CHR and + to create escape sequences and use PR1 to send them to the printer. For example, you can print Underline as follows:

```
27 CHR 251 CHR + "Under" + 27 CHR + 250 CHR +
 "line" + PR1
```

...PRINT

PR1 **PRST** **PRVAR** **PRLCD** **TRACE** **NORM**

PR1 *Print Level 1* Command

Level 1		Level 1
obj	➡	*obj*

PR1 prints the contents of level 1 in multi-line printer format. All objects except strings are printed with their identifying delimiters. Strings are printed without the leading and trailing " delimiters. If level 1 is empty, the message [Empty Stack] is printed.

Printing a Text String

You can print any sequence of characters by creating a string object that contains the characters, placing the string object in level 1, and executing PR1. The printer prints the characters and leaves the print head at the right end of the print line. Subsequent printing begins on the following line.

Printing a Graphics String

You can print graphics by printing a string object that begins with the escape character (character 27) and a character whose number n is from 1 through 166. Together, these characters instruct the printer to interpret the next n characters ($n \leqslant 166$) as *graphics codes*, with each character specifying one column of graphics. Refer to the printer manual for details about graphics codes.

The printer prints the graphics and leaves the print head at the right end of the print line. Subsequent printing begins on the following line. When you turn on the printer, you must print text or execute CR before printing graphics.

Accumulating Data in the Printer Buffer

You can print any combination of text, graphics, and objects on a single print line by accumulating data in the printer. The printer stores the data in a part of its memory called a *buffer*.

Normally, each print command completes data transmission by sending CR (*carriage right*) to the printer. When the printer receives CR, it prints the data in its buffer and leaves the print head at the right end of the print line.

You can prevent the automatic transmission of CR by setting flag 33. Subsequent print commands send your data to the printer but don't send CR. The data accumulates in the printer buffer and is printed only at your command. When flag 33 is set, observe the following rules:

- Send CR (character 4) or newline (character 10), or execute the command CR, when you want the printer to print the data that it has received.
- Don't send more than 200 characters without causing the printer to print. Otherwise, the printer buffer fills up and subsequent characters are lost.
- Allow time for the printer to print a line before sending more data. The printer requires about 1.8 seconds per line.
- Clear flag 33 when you're done to restore the normal operation of the print commands.

PRST *Print Stack* **Command**

. . . **Level 1**		. . . **Level 1**
. . . *obj*	➧	. . . *obj*

PRST prints all objects in the stack, starting with the object in the highest level. Objects are printed in multi-line printer format.

PRVAR *Print Variable* Command

Level 1	
' *name* '	➡

PRVAR prints the object stored in the variable *name*. The object is printed in multi-line printer format.

PRLCD *Print LCD* Command

➡

PRLCD prints a pixel-by-pixel image of the current HP-28C display (excluding the annunciators).

The width of the printed image of an object is narrower using PRLCD than using a print command such as PR1. The difference results from the spacing between characters. On the display there is a single blank column between characters, and PRLCD prints this spacing. Print commands such as PR1 print two blank columns between adjacent characters.

TRACE Mode: TRACE, NORM

You can print an on-going record of your calculations by selecting TRACE mode. Each time you execute ENTER, either by pressing ENTER or by pressing an immediate-execute key, the calculator prints the contents of the command line, the immediate-execute command, and the resulting contents of level 1.

To enable TRACE mode, press TRACE. The TRACE menu label then appears as black letters in a white label, indicating that TRACE mode is enabled. You can enable TRACE mode within a program by setting flag 32.

To disable TRACE mode, press NORM. The NORM menu label then appears as black letters in a white label, indicating that TRACE mode is disabled. You can disable TRACE mode within a program by clearing flag 32.

The print format for the object in level 1 depends on whether multi-line display format is enabled or disabled (flag 45 is set or clear). If multi-line display mode is enabled (flag 45 is set), the object is printed in multi-line printer format. If compact display mode is active (flag 45 is clear), the object is printed in compact format.

PRSTC PRUSR PRMD CR

PRSTC *Print Stack (Compact)* Command

. . . Level 1	. . . Level 1
. . . *obj* ➧	. . . *obj*

PRSTC prints all objects in the stack, starting with the object in the highest level. Objects are printed in compact format.

...PRINT

PRUSR *Print User Variables* Command

	➡	

PRUSR prints the names of the current user variables. The names are printed in the order they appear in the USER menu. If there are no user variables, PRUSR prints `No User Variables`.

PRMD *Print Modes* Command

	➡	

PRMD displays and prints the current selections for number display mode, binary integer base, angle mode, radix mode, and whether UNDO, COMMAND, LAST, and multi-line display are enabled or disabled.

CR *Carriage Right* Command

	➡	

CR prints the contents, if any, of the printer buffer.

Programs

A *program* is a procedure object delimited by « » characters containing a series of commands, objects, and *program structures*, that are executed in sequence when the program is evaluated. Certain program structures, such as those described in "PROGRAM BRANCH" or those specifying local names, must satisfy specific syntax rules, but otherwise the contents of a program are much more flexible than that of algebraic objects, the other type of procedure.

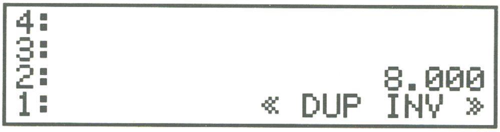
A program, in simplest terms, is a command line for which evaluation is deferred. Any command line can be made into a program by inserting a « at the beginning of the line; then when [ENTER] is pressed, the entire command line is put on the stack as a program. The individual objects in the program are not executed until the program is evaluated.

By making a command line into a program, you can not only defer evaluation, you can also repeat execution as many times as desired. Any number of copies of the program can be made on the stack, using ordinary stack manipulation commands; or you can store a program in a variable and then execute it by name—or by pressing the corresponding menu key in the USER menu. Once a program is stored in a named variable, it becomes essentially indistinguishable from a command. (Actually, the commands themselves are just programs that are entered in ROM instead of RAM.) As you program the HP-28C, you are extending its programming language.

...Programs

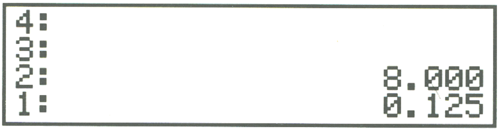
Evaluating Program Objects

Evaluating a program puts each object in the program on the stack and, if the object is a command or unquoted name, evaluates the object. For example, with the stack:



```
4:
3:
2:                8.000
1:         « DUP INV »
```

pressing EVAL yields:



```
4:
3:
2:                8.000
1:                0.125
```

DUP was evaluated, copying 8.000 into level 2, then INV was evaluated, replacing the 8.000 in level 1 with its inverse.

Simple and Complex Programs.

The simplest kind of program is just a single sequence of objects, which are sequentially executed without halting or looping. For example, the program `« 5 * 2 + »` multiplies a number in level 1 by 5 and adds 2.

If this were an operation you performed frequently, you could store the program in a variable, then execute the program as many times as you want by pressing the USER menu key assigned to the variable.

You can add complexity to a program in one or more of the following ways:

Conditionals. By using the IF...THEN...END or IF...THEN...ELSE...END branch structures (or the equivalent commands IFT and IFTE), programs can make decisions based upon computed results, then select execution options accordingly.

Loops. You can cause repeated execution of a program or portion of a program, a definite or indefinite number of times, by using the program loops FOR...NEXT, START...NEXT, DO...UNTIL...END, and WHILE...REPEAT...END.

Error Traps. By using the IFERR...THEN...END or IFERR...THEN...ELSE...END conditional, you can make a program deal with expected or unexpected errors.

Halts. The HALT command allows you to suspend program execution at predetermined points for user input or other purposes, then resume with ■ CONT or SST .

Programs Within Programs. Just as you can postpone evaluation of a command line by enclosing it with « », you can create program objects within other programs by enclosing a program sequence within « ». When the "inner" program is encountered during execution of the "outer" program, it is placed on the stack rather than evaluated. It can be subsequently evaluated with EVAL or any other command that takes a program as an argument.

...Programs

As you add length and complexity to a program, it can grow beyond a size that is conveniently readable on the HP-28C display or too big to enter. For this reason, and to promote orderly programming practices, it is recommended that you break up long programs into multiple short programs. For example, the program `« A B C D »` can be rewritten as `« AB CD »`, where AB is the program `« A B »`, and CD is the program `« C D »`.

The process of writing a large program as a series of small programs makes it straightforward to "debug" the large program. Each secondary program can be tested independently of the others, to insure that it takes the correct number and type of arguments from the stack, and returns the correct results to the stack. Then it is simple to link the secondary programs together by creating a main program consisting of the unquoted names of the secondary programs.

Local Variables and Names

A *local variable* is the combination of an object and a *local name,* which are stored together in a portion of memory temporarily reserved for use only during execution of a procedure. When a procedure completes execution, any local variables associated with that procedure are purged automatically.

Local names are objects used to name local variables. They are subject to the same naming restrictions as ordinary names. You can use local variables, within their defining procedures, almost interchangeably with ordinary names. However, there are several important differences:

- When local names are evaluated, they return the object stored in the associated local variables, unevaluated. They do not automatically evaluate names or programs stored in their local variables, as ordinary names do.
- You cannot use a quoted local name as an argument for 🟥 VISIT or for any of the following commands: CON, IDN, PRVAR, PURGE, PUT, PUTI, RDM, SCONJ, SINV, SNEG, STO+, STO-, STO*, STO/, TAYLR, or TRN.
- Local variables will not appear in the Solver variables menu.

If you have an ordinary variable with the same name as a local variable, any use of the common name within the local variable procedure will refer only to the local variable, and leave the ordinary variable unchanged. Similarly, if a local variable structure is nested within another, the local names of the first (outer) structure can be used within the second (inner).

It is possible for local names to remain on the stack or within procedures and lists even after their associated local variables has been purged. For example, `1 → x « 'x' »` ENTER leaves the local name `'x'` on the stack. If you attempt to evaluate the local name, or use it as an argument for STO, RCL, or PURGE, the error `Undefined Local Name` will be reported.

To minimize any confusion that might arise between names and local names, it is recommended that you adopt a special naming convention for local names. One such convention used in this manual is to use lower-case letters to name local variables (which can never appear in menu key labels), and upper-case for ordinary variables.

...Programs

Creating Local Variables

Local variables are created by using program structures. This section describes two *local variable structures*, which are the primary means of creating local variables. There are also two program branch structures, FOR...NEXT and FOR...STEP, which define definite loops in which the loop index is a local variable. These program branch structures are described in "PROGRAM BRANCH."

The local variable structures have the form:

→ *name*$_1$ *name*$_2$...« *program* »

→ *name*$_1$ *name*$_2$...' *algebraic* '

The → command begins a local variable structure. (The → character is ■ U on the left-hand keyboard. Here → is a command in itself, so it is followed by a space.) The names specify the local names for which local variables are created. The program or algebraic is called the *defining procedure* of the local variable structure. Its initial delimiter, « or ', terminates the sequence of local names.

When → is evaluated, it takes one object from the stack for each of the local names, and stores each object in a local variable named by the corresponding name. The objects and local names are matched up so the order of the names is the same as the order in which the objects were entered into the stack. For example:

```
1 2 3 4 5 → a b c d e
```

assigns the number 1 to the local variable a, 2 to b, 3 to c, 4 to d, and 5 to e. (Since these are local variables, there is no conflict with the symbolic constant e.)

Once the local variables are created and their values assigned, the procedure that follows the name list is evaluated. Within that procedure, you can use the local variable names just like ordinary names (except for the restrictions listed above). When the procedure has finished execution, the local variables are purged automatically.

As an example, suppose you wish to take 3 numbers from the stack, and multiply the first (level 3) by 4, the second (level 2) by 3, and the third (level 1) by 2, and add the results. A simple program for this purpose would be:

```
« 2 * SWAP 3 * + SWAP 4 * + ».
```

Using local variables, the program would become:

```
« → a b c « a 4 * b 3 * + c 2 * + » ».
```

The use of local variables has eliminated the SWAP operations. In this simple case, the use of local variables is of marginal value, but as the complexity of a program grows, local variables can help you write the program in a simpler, less error-prone manner than if you try to manage everything on the stack.

Our example problem also lends itself to an algebraic form. We can write our program this way:

```
« → a b c '4*a+3*b+2*c' »
```

and obtain the same result.

User-Defined Functions

The → command in a special syntax can be used to create new algebraic functions. An algebraic function is a command that can be used within algebraic object definitions. Within those definitions, the functions takes its arguments from a sequence contained within parentheses following the function name. The command SIN, for example, is a typical algebraic function taking one argument. Within an algebraic definition, it is used in the form `'SIN(X)'` where the `X` represents its argument.

A *user-defined function* of *n* arguments is defined by a program with the following syntax:

« → $name_1$ $name_2$ ... $name_n$ '*expression*' »

where $name_1$ $name_2$ ... $name_n$ is a series of *n* local variable names. *expression* is an algebraic expression, containing the local variable names, that represents the mathematical definition of the function. No objects can precede the → in the program, and none can follow '*expression*'.

As an example, consider the algebraic form of the program defined in the preceding section:

```
« → a b c '4*a+3*b+2*c' »
```

It takes three arguments, multiplies them by 4, 3, and 2, respectively, and sums the products. Because nothing precedes the → nor follows the algebraic, this program is a user-defined function. Suppose that we name the user-defined function XYZ by storing the program in variable XYZ:

```
« → a b c '4*a+3*b+2*c' » 'XYZ' STO.
```

In RPN syntax, we can execute `1 2 3 XYZ` to obtain the result 16 (4 × 1 + 3 × 2 + 2 × 3). But we can also use algebraic syntax: `'XYZ(1,2,3)' EVAL` also returns the result 16. You are not restricted to numerical arguments; any of XYZ's three arguments can be an algebraic. XYZ itself can appear in any other algebraic expression.

PROGRAM BRANCH

IF	**IFERR**	**THEN**	**ELSE**	**END**	
START	**FOR**	**NEXT**	**STEP**	**IFT**	**IFTE**
DO	**UNTIL**	**END**	**WHILE**	**REPEAT**	**END**

The PROGRAM BRANCH menu (■ BRANCH) contains commands for making decisions and loops within a program. These commands can appear only in certain combinations called *program structures*. Program branch structures can be grouped into four types: decision, error trap, definite loops, and indefinite loops.

In the following, a *clause* is any program sequence.

1. Decision structures.

- IF *test-clause* THEN *true-clause* END. If *test-clause* is true, then execute *true-clause*. (IFT is a single-command form of this structure.)
- IF *test-clause* THEN *true-clause* ELSE *else-clause* END. If *test-clause* is true, execute *true-clause*; otherwise, execute *else-clause*. (IFTE is a single-command form of this structure.)

2. Error trapping structures.

- IFERR *trap-clause* THEN *error-clause* END. If an error occurs during execution of *trap-clause*, then execute *error-clause*.
- IFERR *trap-clause* THEN *error-clause* ELSE *normal-clause* END. If an error occurs during execution of *trap-clause*, then execute *error-clause*; otherwise, execute *normal-clause*.

3. Definite loop structures.
   - *start finish* START *loop-clause* NEXT. Execute *loop-clause* once for each value of a loop counter incremented by one from *start* through *finish*.
   - *start finish* START *loop-clause step* STEP. Execute *loop-clause* once for each value of a loop counter incremented by *step* from *start* through *finish*.
   - *start finish* FOR *name loop-clause* NEXT. Execute *loop-clause* once for each value of a local variable *name*, used as a loop counter, incremented by ones from *start* through *finish*.
   - *start finish* FOR *name loop-clause step* STEP. Execute *loop-clause* once for each value of a local variable *name*, used as a loop counter, incremented by *step* from *start* through *finish*.
4. Indefinite loop structures.
   - DO *loop-clause* UNTIL *test-clause* END. Execute *loop-clause* repeatedly until *test-clause* is true.
   - WHILE *test-clause* REPEAT *loop-clause* END. While *test-clause* is true, execute *loop-clause* repeatedly.

These structures are described later in this section, following two introductory topics.

...PROGRAM BRANCH

Tests and Flags

All program structures (except definite loops) make a branching decision based upon the evaluation of a *test clause*. A test clause is any program sequence that returns a *flag* when evaluated. A flag is an ordinary real number that nominally has the value 0 or 1. If the flag has value 0, we say that it is "false" or "clear"; for any other value, we say that the flag is "true" or "set".

All program branch decisions are made by testing a flag taken from the stack. For example, in an IF *test-clause* THEN *true-clause* END structure, if evaluation of *test-clause* leaves a non-zero (real) result, *true-clause* will be evaluated. If *test-clause* leaves 0 in level 1, execution will skip past END.

A *test* command is one that explicitly returns a flag with a value 0 or 1. For example, the command < tests two real numbers (or binary integers, or strings) to see if the number in level 2 is less than the number in level 1. If so, < returns the flag 1; otherwise, it returns 0. The other test commands are >, ≤, ≥, ==, ≠, FS?, FC?, FS?C, and FC?C, all of which are described in "PROGRAM TEST."

Replacing GOTO

Programmers accustomed to other calculator programming languages, such as the RPN language of other HP calculators, or BASIC, may note the absence of a simple GOTO instruction in the HP-28C language. GOTO's are commonly used to branch depending on a test and to minimize program size by reusing program steps. We'll look at how GOTO's are used in HP-41 RPN and BASIC, and show how to obtain equivalent results with the HP-28C.

...PROGRAM BRANCH

- Using GOTO instructions to branch depending on a test. For example, the programs below execute the sequence ABC DEF if the number in the X register or variable is positive, or execute the sequence GHI JKL otherwise.

HP-41 RPN	BASIC
01 X>0?	10 IF X>0 THEN GOTO 50
02 GTO 01	20 GHI
03 GHI	30 JKL
04 JKL	40 GOTO 70
05 GTO 02	50 ABC
06 LBL 01	60 DEF
07 ABC	⋮
08 DEF	
09 LBL 02	
⋮	

Here is an HP-28C equivalent:

```
IF 0 > THEN ABC DEF ELSE GHI JKL END
```

- Using a GOTO instruction to minimize program size by reusing program steps. Both programs below contain a sequence MNO PQR STU that is common to two branches of the program.

HP-41 RPN	BASIC
01 ABC	10 ABC
02 DEF	20 DEF
03 GTO 01	30 GOTO 200
⋮	⋮
10 GHI	100 GHI
11 JKL	110 JKL
12 GTO 01	120 GOTO 200
⋮	⋮
20 LBL 01	200 MNO
21 MNO	210 PQR
22 PQR	220 STU
23 STU	⋮
⋮	

...PROGRAM BRANCH

In the HP-28C, the common sequence MNO PQR STU...would be stored as a separate program:

```
« MNO PQR STU ... » 'COMMON' STO
```

Then each branch of the program would execute COMMON:

```
... ABC DEF COMMON ... GHI JKL COMMON ...
```

The advantage of HP-28C programming is that any program has only one entrance and one exit. This makes it simple to write programs and test them independently. When you combine the programs into a main program, you need to test only that the programs work together as you intended.

IF IFERR THEN ELSE END

These commands can be combined in a variety of decision structures and error trapping structures.

IF *test-clause* THEN *true-clause* END. The command THEN takes a flag from the stack. If the flag is true (non-zero), the *true-clause* is evaluated, after which execution continues after END. If the number is false (0), execution skips past END and continues. (Note that only THEN actually uses the flag—the position of the IF is arbitrary as long as it precedes THEN. *test-clause* IF THEN will work the same as IF *test-clause* THEN). For example:

```
IF X 0 > THEN "Positive" END
```

returns the string `"Positive"` if X contains a positive real number.

IF *test-clause* THEN *true-clause* ELSE *false-clause* END. The command THEN takes a flag from the stack. If the flag is true (nonzero), the *true-clause* is evaluated, after which execution continues after END. If the flag is false (0), the *false-clause* is evaluated, after which execution continues after END. (Note that only THEN actually uses the flag—the position of the IF is arbitrary as long as it precedes THEN. *test-clause* IF THEN will work the same as IF *test-clause* THEN). For example:

```
IF X 0 ≥ THEN "Positive" ELSE "Negative" END
```

returns the string `"Positive"` if X contains a non-negative real number, or `"Negative"` if X contains a negative real number.

IFERR *trap-clause* THEN *error-clause* END. This structure evaluates *error-clause* if an error occurs during execution of *trap-clause*.

When *trap-clause* is evaluated, successive elements of the clause are executed normally unless an error occurs. In that case, execution jumps to *error-clause*. The remainder of *trap-clause* is discarded. For example:

```
IFERR WHILE 1 REPEAT + END THEN "OK" 1 DISP END
```

sums all numbers on the stack. The + function is executed repeatedly until an error occurs, indicating that the stack is empty (or a mismatched object type has been encountered). The *error-clause* then displays `OK`.

When you write error clauses, keep in mind that the state of the stack after an error may depend on whether LAST is enabled. If LAST is enabled, commands that error will return their arguments to the stack; otherwise the arguments are dropped.

...PROGRAM BRANCH

IFERR *trap-clause* THEN *error-clause* ELSE *normal-clause* END. This structure enables you to specify an *error-clause* to be evaluated if an error occurs during execution of a *trap-clause,* and also a *normal-clause* for execution if no error occurs.

When *trap-clause* is evaluated, successive elements of the clause are executed normally unless an error occurs.

- If an error occurs, the remainder of the *trap-clause* is discarded and the *error-clause* is evaluated.
- If no error occurs, evaluation of the *trap-clause* is followed by evaluation of the *normal-clause.*

In either case execution continues past END.

START FOR NEXT STEP IFT IFTE

***start finish* START *loop-clause* NEXT.** The START command takes two real numbers, *start* and *finish,* from the stack and stores them as the starting and ending values for a loop counter. Then a sequence of objects *loop-clause* is evaluated. The NEXT command increments the loop counter by 1; if the loop counter is less than or equal to *finish,* *loop-clause* is evaluated again. This continues until the loop counter exceeds *finish,* whereupon execution continues following NEXT. For example:

```
1 10 START XYZ NEXT
```

evaluates XYZ 10 times.

***start finish* START *loop-clause increment* STEP.** This structure is similar to START...NEXT, except that STEP increments the loop counter by a variable amount, whereas NEXT always increments by 1.

START takes two real numbers, *start* and *finish,* from the stack and stores them as the starting and ending values for a loop counter. Then a sequence of objects *loop-clause* is evaluated. STEP increments the loop counter by the real number *increment* taken from level 1.

If *step* is positive and the loop counter is less than or equal to *finish, loop-clause* is evaluated again. This continues until the loop counter exceeds *finish,* whereupon execution continues following STEP.

If *step* is negative and the loop counter is greater than or equal to *finish, loop-clause* is evaluated again. This continues until the loop counter is less than *finish,* whereupon execution continues following STEP. For example:

```
10 1 START XYZ -2 STEP
```

evaluates XYZ five times.

***start finish* FOR *name loop-clause* NEXT.** This structure is a definite loop in which the loop counter *name* is a local variable that can be evaluated within the loop. (The name following FOR should be entered without quotes.) In sequence:

1. FOR takes two real numbers *start* and *finish* from the stack. It creates a local variable *name,* and stores *start* as the initial value of *name.*
2. The sequence of objects *loop-clause* is evaluated. If *name* is evaluated within the sequence, it returns the current value of the loop counter.
3. NEXT increments the loop counter by 1. If its value then exceeds *finish,* execution continues with the object following NEXT, and the local variable *name* is purged. Otherwise, steps 2 and 3 are repeated.

For example:

```
1 5 FOR x x SQ NEXT
```

places the squares of the integers 1 through 5 on the stack.

***start finish* FOR *name loop-clause increment* STEP.** This structure is a definite loop in which the loop counter *name* is a local variable that can be evaluated within the loop. (The name following FOR should be entered without quotes.) It is similar to FOR...NEXT, except that the loop counter is incremented by a variable amount. In sequence:

1. FOR takes two real numbers *start* and *finish* from the stack. It creates a local variable *name*, and stores *start* as the initial value of *name*.
2. The sequence of objects *loop-clause* is evaluated. If *name* is evaluated within the sequence, it returns the current value of the loop counter.
3. STEP takes the real number *increment* from the stack and increments the loop counter by *increment*. If the loop counter then is greater than *finish* (for *increment* > 0) or less than *finish* (for *increment* < 0), execution continues with the object following STEP, and the local variable *name* is purged. Otherwise, steps 2 and 3 are repeated.

For example:

```
1 11 FOR x x SQ 2 STEP
```

places the squares of the integers 1, 3, 5, 7, 9, and 11 on the stack.

IFT — *If-Then* — Command

Level 2	Level 1		
flag	*obj*	➧	

IFT is a single-command form of IF...THEN...END. IFT takes a flag from level 2, and an arbitrary object from level 1. If the flag is true (non-zero), the object is evaluated; if the flag is false (0), the object is discarded. For example:

```
X 0 > "Positive" IFT
```

leaves `"Positive"` in level 1 if X contains a positive real number.

IFTE — *If-Then-Else* — Function

Level 3	Level 2	Level 1		
flag	*true-obj*	*false-obj*	➧	

IFTE is a single-command form of IF...THEN...ELSE...END. IFTE takes a flag from level 3, and two arbitrary objects from levels 1 and 2. If the flag is true (non-zero), *false-object* is discarded, and *true-object* is evaluated. If the flag is false (0), *true-object* is discarded and *false-object* is evaluated. For example:

```
X 0 ≥ "Positive" "Negative" IFTE
```

leaves `"Positive"` on the stack if X contains a non-negative real number, or `"Negative"` if X contains a negative real number.

IFTE is also acceptable in algebraic expressions, with the following syntax:

'IFTE(*test-expression*, *true-expression*, *false-expression*)'

...PROGRAM BRANCH

When an algebraic containing IFTE is evaluated, its first argument *test-expression* is evaluated as a flag. If it returns a non-zero real number, *true-expression* is evaluated. If it returns zero, *false-expression* is evaluated. For example:

```
'IFTE(X≠0,SIN(X)/X,1)'
```

is an expression that returns the value of $\sin(x)/x$, even for $x = 0$, which would normally cause an `Infinite Result` error.

DO UNTIL END WHILE REPEAT END

DO *loop-clause* UNTIL *test-clause* END. This structure repeatedly evaluates a *loop-clause* and a *test-clause*, until the flag returned by *test-clause* is true (non-zero). For example:

```
DO X INCX X - UNTIL .0001 < END.
```

Here INCX is a sample program that increments the variable X by a small amount. This routine will execute INCX repeatedly, until the resulting change in X is less than .0001.

WHILE *test-clause* REPEAT *loop-clause* END. This structure repeatedly evaluates a *test-clause* and a *loop-clause*, as long as the flag returned by *test-clause* is true (non-zero). When the *test-clause* returns a false flag, the *loop-clause* is skipped, and execution resumes following END. The *test-clause* returns a real number, which REPEAT tests as a flag. For example:

```
WHILE STRING "P" POS REPEAT REMOVEP END.
```

Here REMOVEP is a sample program that removes a character `P` from a string stored in the variable STRING. The sequence repeats until no more `P`'s remain in the string.

PROGRAM CONTROL

SST	**HALT**	**ABORT**	**KILL**	**WAIT**	**KEY**
BEEP	**CLLCD**	**DISP**	**CLMF**	**ERRN**	**ERRM**

The PROGRAM CONTROL menu (🟥 CTRL) contains commands for interrupting program execution and for interactions during program execution.

Suspended Programs

Evaluating a program normally executes the objects contained in the program's definition continuously up to the end of the program. The commands in the PROGRAM CONTROL menu allow programs to pause or halt execution at points other than the end of the program:

Command	Description
HALT	Suspends program execution, for continuation later.
ABORT	Stops program execution, which then cannot be resumed.
KILL	Stops program execution, and also clears all other suspended programs.
WAIT	Pauses program execution, which resumes automatically after a specified time.

A *suspended* program is a program that is halted during execution, in such a way that the program can be *continued* (execution resumed) at the point which it stopped. While a program is suspended, you can perform any HP-28C operation (except system halt, memory reset, and the KILL command)—enter data, view results, execute other programs, and so on—then continue the program.

...PROGRAM CONTROL

The ○ annunciator indicates that one or more programs are suspended.

The command HALT causes a program to suspend at the location of the HALT in the program. To resume program execution you can:

- Press 🟥 CONT (continue) to resume continuous execution at the next object in the program after the HALT. You can use HALT in conjunction with 🟥 CONT in a program when you want to stop the program for user input, then continue.
- Press SST (single-step—in the PROGRAM CONTROL menu) to execute the next object in the program after the HALT. Repeated use of SST continues program execution, one step at a time. This is a powerful program debugging tool, since you can view the stack or any other calculator state after each step in a program.

If you do not choose either of these options, the program will remain suspended indefinitely, unless you execute KILL or a system halt, which clear all suspended programs.

You can "nest" suspended programs—that is, you can execute a program that contains a HALT while another program is already suspended. If you continue (🟥 CONT) the second program, execution will halt again when it has finished. Then you can press 🟥 CONT again to resume execution of the first program.

While a program is suspended, the stack save and recovery associated with UNDO are "local" to the program. Refer to "MODE" for information on the use of UNDO with suspended programs.

...PROGRAM CONTROL

SST HALT ABORT KILL WAIT KEY

Single Step

SST executes the "next step" in a suspended program. "Next step," in this context, means the object or command that follows, in the order of program execution, the most recently evaluated object or command.

When you press SST, the program step about to be executed is displayed briefly, in inverse video, then it is executed. After each step, the stack and menu key labels are displayed in the normal fashion. Between steps, you can perform calculator operations without affecting the suspended program. Of course, if you alter the stack, you should insure that it contains the appropriate objects before resuming program execution.

For any of the program loops defined with FOR...NEXT, START...NEXT, DO...UNTIL...END, or WHILE...REPEAT...END, the initial command (FOR, START, DO, or WHILE) is displayed only as a step the first time through the loop. On successive iterations, each loop will start with the first object or command after the initial loop command.

If an error occurs when you single-step an object, the single-step does not advance. This allows you to correct the source of the error, then repeat the single-step.

Pressing SST when an IFERR is the next step executes the entire IFERR...THEN...END or IFERR...THEN...ELSE...END structure as one step. To step through a clause of the structure, include HALT inside the clause.

Similarly, pressing SST when → is displayed executes the entire → *name*$_1$ *name*$_2$ ... *name*$_n$ structure as one step. If the local names are followed by an algebraic, the algebraic is immediately evaluated in that same step.

HALT *Halt Program* Command

➡

HALT causes a program to suspend execution at the location of the HALT command in the program. HALT:

1. Turns on the ○ annunciator.
2. Assigns memory for a temporary saved stack, if UNDO is enabled.
3. Returns calculator control to the keyboard, for normal operations.

Programs resumed with ■ CONT or SST will continue with the object next in the program after the HALT command.

ABORT *Abort Program* Command

➡

ABORT stops execution of a program, at the location of the ABORT command in the program's definition. Execution of the program cannot be resumed.

KILL *Kill Suspended Programs* Command

	➡	

KILL aborts the current program, and also all other currently suspended programs. None of the programs can be resumed.

WAIT *Wait* Command

Level 1		
x	➡	

WAIT pauses program execution for *x* seconds.

KEY *Key* Command

		Level 2	Level 1
	➡		0
	➡	"*string*"	1

KEY returns a string representing the oldest key currently held in the key buffer, and removes that key from the key buffer. If the key buffer is empty, KEY returns a false flag (0). If the key buffer currently holds one or more keys, KEY removes the oldest key from the buffer, and returns a true flag (1) in level 1 plus a string in level 2. The string "names" the key removed from the buffer.

The HP-28C key buffer can hold up to 15 keys that have been pressed but not yet processed. When KEY removes a key from the buffer it is converted to a readable string. The string contains the character(s) on the key top, except for:

Key	String
SPACE	" "
LC	"1"
INS	"INS"
DEL	"DEL"
▲	"UP"
▼	"DOWN"
◀	"LEFT"
▶	"RIGHT"
◂⬍▸	"CURSOR"
⬅	"BACK"

The ON key retains its role as the ATTN key and interrupts the current program.

The action of KEY can be illustrated by the following program:

```
« DO UNTIL KEY END "Y" SAME ».
```

When this program is executed, pressing Y returns 1 (true) to level 1, and pressing any other key returns 0 (false).

...PROGRAM CONTROL

BEEP CLLCD DISP CLMF ERRN ERRM

BEEP *Beep* Command

Level 2	Level 1	
frequency	*duration*	➡

BEEP causes a tone to sound at the specified *frequency* and *duration*. *Frequency* is expressed in Hertz (rounded to an integer). *Duration* is expressed in seconds.

The frequency of the tone is subject to the resolution of the built-in tone generator. The maximum frequency is approximately 4400 Hz; the maximum duration is 1048.575 seconds (# FFFFF msec). Arguments greater than these maximum values will default to the maxima.

Setting flag 51 disables the beeper, so that executing BEEP will produce no sound.

CLLCD *Clear LCD* Command

➡

CLLCD clears (blanks) the LCD display (except the annunciators), and sets the system message flag to suppress the normal stack and menu display.

...PROGRAM CONTROL

DISP *Display* Command

Level 2	Level 1		
obj	*n*	➡	

DISP displays *obj* in the *n*th line of the display, where *n* is a real integer. $n = 1$ indicates the top line of the display; $n = 4$ is the bottom line. DISP sets the system message flag to suppress the normal stack display.

An object is displayed by DISP in the same form as would be used if the object were in level 1 in the multi-line display format, except for strings, which are displayed without the surrounding " delimiters to facilitate the display of messages. If the object display requires more than one display line, the display starts in line *n*, and continues down the display either to the end of the object or the bottom of the display.

CLMF *Clear Message Flag* Command

➡

CLMF clears the internal message flag set by CLLCD, DISP, PIXEL, DRAX, DRAW, and DRWΣ. Including CLMF in a program, after the last occurrence of any of these words, causes the normal stack display to be restored when the program completes execution.

ERRN *Error Number* Command

	Level 1
➡	# *n*

ERRN returns a binary integer equal to the error number of the most recent calculator error. A table of HP-28C errors, error messages, and error numbers is given in Appendix A.

ERRM *Error Message* Command

	Level 1
➡	"*error-message*"

ERRM returns a string containing the error message of the most recent calculator error. A table of HP-28C errors, error messages, and error numbers is given in Appendix A.

PROGRAM TEST

SF	**CF**	**FS?**	**FC?**	**FS?C**	**FC?C**
AND	**OR**	**XOR**	**NOT**	**SAME**	**==**
STOF	**RCLF**	**TYPE**			

The PROGRAM TEST menu (■ TEST) contains commands for changing and testing flags and for logical calculations.

Test commands return a *flag* as the result of a comparison between two arguments, or of a user-flag test. The comparison operators ≠, <, >, ≤, and ≥ are present on the left-hand keyboard as characters. The remaining test commands FS?, FC?, FS?C, FC?C, SAME, and == are present in the TEST menu. In addition, the TEST menu contains the logical operations AND, OR, XOR, and NOT, that allow you to combine flag values. Note that the = function is not a comparison operator; it defines an equation. Both == and SAME test the equality of objects.

Keyboard Functions

≠ ***Not Equal*** **Function**

Level 2	Level 1		Level 1
obj_1	obj_2	➡	*flag*
z	' *symb* '	➡	' $z \neq symb$ '
' *symb* '	*z*	➡	' $symb \neq z$ '
' $symb_1$ '	' $symb_2$ '	➡	' $symb_1 \neq symb_2$ '

≠ takes two objects from levels 1 and 2, and:

- If either object is not an algebraic or a name, returns a false flag (0) if the two objects are the same type and have the same value, or a true flag (1) otherwise. Lists and programs are considered to have the same values if the objects they contain are identical.
- If one object is an algebraic or a name, and the other is a number, a name, or an algebraic, ≠ returns a symbolic comparison expression of the form '$symb_1 \neq symb_2$', where $symb_1$ represents the object from level 2, and $symb_2$ represents the object from level 1. The result expression can be evaluated with EVAL or →NUM to return a flag.

< **Less Than** **Function**

Level 2	Level 1		Level 1
x	y	➡	*flag*
# n_1	# n_2	➡	*flag*
"$string_1$"	"$string_2$"	➡	*flag*
x	'$symb$'	➡	'$x<symb$'
'$symb$'	x	➡	'$symb<x$'
'$symb_1$'	'$symb_2$'	➡	'$symb_1<symb_2$'

> **Greater Than** **Function**

Level 2	Level 1		Level 1
x	y	➡	*flag*
# n_1	# n_2	➡	*flag*
"$string_1$"	"$string_2$"	➡	*flag*
x	'$symb$'	➡	'$x>symb$'
'$symb$'	x	➡	'$symb>x$'
'$symb_1$'	'$symb_2$'	➡	'$symb_1>symb_2$'

⩽ **Less Than or Equal** **Function**

Level 2	Level 1		Level 1
x	y	➧	*flag*
# n_1	# n_2	➧	*flag*
"$string_1$"	"$string_2$"	➧	*flag*
x	' *symb* '	➧	' $x≤symb$ '
' *symb* '	x	➧	' $symb≤x$ '
' $symb_1$ '	' $symb_2$ '	➧	' $symb_1≤symb_2$ '

⩾ **Greater Than or Equal** **Function**

Level 2	Level 1		Level 1
x	y	➧	*flag*
# n_1	# n_2	➧	*flag*
"$string_1$"	"$string_2$"	➧	*flag*
x	' *symb* '	➧	' $x≥symb$ '
' *symb* '	x	➧	' $symb≥x$ '
' $symb_1$ '	' $symb_2$ '	➧	' $symb_1≥symb_2$ '

The following description refers to the four stack disagrams above.

Each of the four commands <, >, ⩽, and ⩾ takes two objects from the stack, applies the logical comparison corresponding to the command name, and returns a flag according to the results of the comparison. The logical order of the comparisons is *level 2 test level 1*, where *test* represents any of the four comparisons. For example, if level 2 contains a real number *x*, and level 1 contains a real number *y*, then < returns a true flag (1) if *x* is less than *y*, and a false flag (0) otherwise.

<, >, ≤, and ≥, because they imply an ordering, apply to fewer object types than ≠, ==, or SAME:

- For real numbers and binary integers, "less than" means numerically smaller (1 is less than 2). For real numbers, "less than" also means "more negative" (−2 is less than −1).
- For strings, "less than" means alphabetically previous ("ABC" is less than "DEF"; "AAA" is less than "AAB"; "A" is less than "AA"). In general, characters are ordered according to their character codes. Note that this means that "B" is less than "a", since "B" is character code 66, and "a" is character code 97.

SF CF FS? FC? FS?C FC?C

This group of commands sets, clears, and tests the 64 user flags. In this context, "to set" means "to make true" or "to assign value 1", and "to clear" means "to make false" or "to assign value 0".

SF *Set Flag* Command

Level 1	
n ➡	

SF sets the user flag specified by the real integer argument n, where $1 \leq n \leq 64$.

CF *Clear Flag* Command

Level 1	
n ➡	

CF clears the user flag specified by the real integer argument n, where $1 \leq n \leq 64$.

...PROGRAM TEST

FS? *Flag Set?* Command?

Level 1		Level 1
n	➡	*flag*

FS? tests the user flag specified by the real integer argument n, where $1 \leqslant n \leqslant 64$. If the user flag is set, FS? returns a true flag (1); otherwise it returns a false flag (0).

FC? *Flag Clear?* Command

Level 1		Level 1
n	➡	*flag*

FC? tests the user flag specified by the real integer argument n, where $1 \leqslant n \leqslant 64$. If the user flag is clear, FC? returns a true flag (1); otherwise it returns a false flag (0).

FS?C *Flag Set? Clear* Command

Level 1		Level 1
n	➡	*flag*

FS?C tests, and then clears, the user flag specified by the real integer argument n, where $1 \leqslant n \leqslant 64$. If the user flag is set, FS?C returns a true flag (1); otherwise it returns a false flag (0).

FC?C *Flag Clear? Clear* Command

Level 1		Level 1
n	➧	*flag*

FC?C tests, and then clears, the user flag specified by the real integer argument *n*, where 1 ≤ *n* ≤ 64. If the user flag is clear , FC?C returns a true flag (1); otherwise it returns a false flag (0).

AND OR XOR NOT SAME ==

The commands AND, OR, XOR, and NOT can be applied to *flags* (real numbers or algebraics), and to binary integers. In the former case, the commands act as logical operators that combine true or false truth values into result flags. For binary integers, the commands perform logical combinations of the individual bits of arguments.

The following descriptions apply to the use of the commands with real number arguments (flags). The "BINARY" section describes their application to binary integers.

AND, OR, XOR, and NOT are allowed in algebraic objects. AND and NOT have higher precedence than OR or XOR. AND, OR, and XOR are displayed within algebraics as *infix* operators:

```
'X AND Y'  '5+X XOR Z AND Y'
```

NOT appears as a *prefix* operator:

```
'NOT X'  'Z+NOT (A AND B)'
```

If you enter the commands in this form, be sure to separate the commands from other commands or objects with spaces. You can also enter these commands into the command line in prefix form:

```
'AND(X,Y)'  'AND(XOR(X,Z),Y)'
```

AND *And* Function

Level 2	Level 1		Level 1
x	*y*	➡	*flag*
x	'*symb*'	➡	'*x* AND *symb*'
'*symb*'	*x*	➡	'*symb* AND *x*'
'$symb_1$'	'$symb_2$'	➡	'$symb_1$ AND $symb_2$'

AND returns a flag that is the logical AND of two flags:

First Argument x	Second Argument y	AND Result
true	true	true
true	false	false
false	true	false
false	false	false

If either or both of the arguments are algebraics, the result is an algebraic of the form '$symb_1$ AND $symb_2$', where $symb_1$ and $symb_2$ represent the arguments.

OR *Or* Function

Level 2	Level 1		Level 1
x	*y*	➡	*flag*
x	'*symb*'	➡	'*x* OR *symb*'
'*symb*'	*x*	➡	'*symb* OR *x*'
'$symb_1$'	'$symb_2$'	➡	'$symb_1$ OR $symb_2$'

OR returns a flag that is the logical OR of two flags:

First Argument x	Second Argument y	OR Result
true	true	true
true	false	true
false	true	true
false	false	false

If either or both of the arguments are algebraics, the result is an algebraic of the form '$symb_1$ OR $symb_2$', where $symb_1$ and $symb_2$ represent the arguments.

XOR **Exclusive Or** **Function**

Level 2	Level 1		Level 1
x	*y*	➡	*flag*
x	'*symb*'	➡	'*x* XOR *symb*'
'*symb*'	*x*	➡	'*symb* XOR *x*'
'$symb_1$'	'$symb_2$'	➡	'$symb_1$ XOR $symb_2$'

XOR returns a flag that is the logical exclusive OR (XOR) of two flags:

First Argument x	Second Argument y	XOR Result
true	true	false
true	false	true
false	true	true
false	false	false

If either or both of the arguments are algebraics, the result is an algebraic of the form `'`$symb_1$ `XOR` $symb_2$`'`, where $symb_1$ and $symb_2$ represent the arguments.

NOT *Not* Function

Level 1		Level 1
x	➡	*flag*
'*symb*'	➡	'NOT *symb*'

NOT returns a flag that is the logical inverse of a flag:

Argument x	NOT Result
true	false
false	true

If the argument is an algebraic, the result is an algebraic of the form `'NOT symb'`, where *symb* represents the argument.

SAME *Same* Command

Level 2	Level 1		Level 1
obj_1	obj_2	➡	*flag*

SAME takes two objects of the same type from levels 1 and 2, and returns a true flag (1) if the two objects are identical, or a false flag (0) otherwise.

SAME is identical in effect to ==, for all object types except algebraics and names. == returns a symbolic (algebraic) flag for these object types.

SAME returns a (real number) flag for all object types, and is not allowed in algebraic expressions.

==   **Equal**   Function

Level 2	Level 1		Level 1
obj_1	obj_2	➡	*flag*
z	'*symb*'	➡	'*z*==*symb*'
'*symb*'	*z*	➡	'*symb*==*z*'
'$symb_1$'	'$symb_2$'	➡	'$symb_1$==$symb_2$'

== takes two objects from levels 1 and 2, and:

- If either object is not an algebraic (or a name), == returns a true flag (1) if the two objects are the same type and have the same value, or a false flag (0) otherwise. Lists and programs are considered to have the same values if the objects they contain are identical.
- If one object is an algebraic (or a name), and the other is a number or an algebraic, == returns a symbolic comparison expression of the form '$symb_1$==$symb_2$', where $symb_1$ represents the object from level 2, and $symb_2$ represents the object from level 1. The result expression can be evaluated with EVAL or →NUM to return a flag.

The function name == is used for the equality comparison, rather than =, to distinguish between a logical comparison (==) and an equation (=).

STOF RCLF TYPE

STOF — *Store Flags* — Command

Level 1		
# n	➡	

STOF sets the states of the 64 user flags to match the bits in a binary integer # *n*. A bit with value 1 sets the corresponding flag; a bit with value 0 clears the corresponding flag. The first (least significant) bit of # *n* corresponds to flag 1; the 64th (most significant) corresponds to flag 64.

If # *n* contains fewer than 64 bits, the unspecified most significant bits are taken to have value 0.

RCLF — *Recall Flags* — Command

		Level 1
	➡	*# n*

RCLF returns a 64-bit binary integer # *n* representing the states of the 64 user flags. Flag 1 corresponds to the first (least significant) bit of the integer; flag 64 is represented by the 64th (most significant) bit.

You can save the states of all user flags, using RCLF, and later restore those states, using STOF. Remember that the current wordsize must be 64 bits (the default wordsize) to save and restore all flags. If the current wordsize is 32, for example, RCLF returns a 32-bit binary integer; executing STOF with a 32-bit binary integer restores only flags 1 through 32 and clears flags 33 through 64.

Following a memory reset, RCLF will return the value # 4001FFC40000000 (hexadecimal), corresponding to the default settings of the 64 flags.

TYPE *Type* Command

Level 1		Level 1
obj	➡	*n*

The command TYPE returns a real integer representing the type of an object in level 1. The object types and their type numbers are as follows:

Object Types and TYPE Numbers

Object	TYPE Number
Real number	0
Complex number	1
String	2
Real (vector or matrix)	3
Complex (vector or matrix)	4
List	5
Name	6
Local name	7
Program	8
Algebraic	9
Binary integer	10

REAL

NEG	**FACT**	**RAND**	**RDZ**	**MAXR**	**MINR**
ABS	**SIGN**	**MANT**	**XPON**		
IP	**FP**	**FLOOR**	**CEIL**	**RND**	
MAX	**MIN**	**MOD**	**%T**		

An HP-28C *real number* object is a floating-point decimal number consisting of a 12-digit mantissa, and a 3-digit exponent in the range −499 to +499. Real numbers are entered and displayed as a string of numeric characters, with no delimiters and no intervening spaces. Numeric characters include the digits 0 through 9, +, −, a radix (“.” or “,” according to the current radix mode), and the letter E to indicate the start of the exponent field. The general real number format is

(sign) mantissa E *(sign) exponent*

When you enter a real number, the format is as follows:

- The mantissa *sign* can be a +, a −, or omitted (implying +).
- The *mantissa* can be any number of digits, with one radix mark anywhere in the sequence. If you enter more than 12 digits, the mantissa is rounded to 12 digits. (Half-way cases are rounded up in magnitude.) Leading zeros are ignored if they are followed by non-zero mantissa digits.
- An exponent is optional; if you include an exponent, it must be separated from the mantissa by an “E”.
- The exponent *sign* can be a +, a −, or omitted (implying +).
- The *exponent* must contain three or fewer digits, and fall in the range 0 to 499. Leading zeros before the exponent are ignored.

Real numbers are displayed according to the current real number display mode. In general, the display may not show all of the significant digits of a number, but the full 12-digit precision of a number is always preserved in the stored version of the number.

...REAL

The REAL menu contains functions that operate upon real number (and real-valued algebraic) arguments, or enter special real numbers into the stack. In addition to the menu functions, % and %CH are provided on the keyboard.

Keyboard Functions

% **_Percent_** **Function**

Level 2	Level 1		Level 1
x	*y*	➡	*xy*/100
x	'*symb*'	➡	'%(*x*, *symb*)'
'*symb*'	*x*	➡	'%(*symb*, *x*)'
'$symb_1$'	'$symb_2$'	➡	'%($symb_1$, $symb_2$)'

% takes two real-valued arguments *x* and *y*, and returns *x percent of y*—that is, *xy*/100.

%CH **_Percent Change_** **Function**

Level 2	Level 1		Level 1
x	*y*	➡	100(*y*−*x*)/*x*
x	'*symb*'	➡	'%CH(*x*, *symb*)'
'*symb*'	*x*	➡	'%CH(*symb*, *x*)'
'$symb_1$'	'$symb_2$'	➡	'%CH($symb_1$, $symb_2$)'

%CH computes the (percent) increase over the real-valued argument *x* in level 2 that is represented by the argument *y* in level 1. That is, %CH returns 100(*y* − *x*)/*x*.

π π **Function**

		Level 1
	➡	3.14159265359
	➡	'π'

π returns the symbolic constant 'π' or the numerical value 3.14159265359, the closest machine-representable approximation to π. For information on symbolic constants, see page 70.

e e **Function**

		Level 1
	➡	2.71828182846
	➡	'e'

e returns the symbolic constant 'e' or the numerical value 2.71828182846, the closest machine-representable approximation to *e*, the base of natural logarithms. For information on symbolic constants, see page 70.

NEG FACT RAND RDZ MAXR MINR

NEG *Negate* **Analytic**

Level 1		Level 1
z	➡	−*z*
'*symb*'	➡	'−*symb*'

NEG returns the negative of its argument. When no command line is present, pressing $\boxed{\text{CHS}}$ executes NEG. A complete stack diagram for NEG appears in "Arithmetic."

FACT *Factorial (Gamma)* Function

Level 1		Level 1
n	➡	$n!$
x	➡	$\Gamma(x+1)$
'*symb*'	➡	'FACT(*symb*)'

FACT returns the factorial $n!$ of a positive integer argument n. For non-integer arguments x, FACT(x) = $\Gamma(x + 1)$, defined for $x > -1$ as

$$\Gamma(x + 1) = \int_0^{\infty} e^{-t} t^x \, dt$$

and defined for other values of x by analytic continuation. For $x \geqslant 253.1190554375$ or x a negative integer, FACT causes an Overflow exception; for $x \leqslant -254.1082426465$, FACT causes an Underflow exception.

RAND *Random Number* Command

		Level 1
	➡	x

RAND returns the next real number in a pseudo-random number sequence, and updates the random number seed.

The HP-28C uses a linear congruous method and a seed value to generate a random number x, which always lies in the range $0 \leqslant x < 1$. Each succeeding execution of RAND returns a value computed from a seed based upon the previous RAND value. You can change the seed by using RDZ.

RDZ *Randomize* Command

Level 1	
x ➡	

RDZ takes a real number as a seed for the RAND command. If the argument is 0, a random value based upon the system clock will be used as the seed. After memory reset, the seed value is .529199358633.

MAXR *Maximum Real* Function

	Level 1
➡	9.99999999999E499
➡	`'MAXR'`

MAXR returns the symbolic constant `'MAXR'` or the numerical value 9.99999999999E499, the largest machine-representable number. For information on symbolic constants, see page 70.

MINR *Minimum Real* Function

	Level 1
➡	1.00000000000E-499
➡	`'MINR'`

MINR returns the symbolic constant `'MINR'` or the numerical value 1E−499, the smallest positive machine-representable number. For information on symbolic constants, see page 70.

ABS SIGN MANT XPON

ABS — *Absolute Value* — Function

Level 1		Level 1
z	➡	$\|z\|$
[array]	➡	$\|\|array\|\|$
' *symb* '	➡	'ABS(*symb*)'

ABS returns the absolute value of its argument. See "ARRAY" and "COMPLEX" for the use of ABS with other object types. ABS can be differentiated but not inverted (solved) by the HP-28C.

SIGN — *Sign* — Function

Level 1		Level 1
z_1	➡	z_2
' *symb* '	➡	'SIGN(*symb*)'

SIGN returns the sign of its argument, defined as +1 for positive real arguments, −1 for negative real arguments, and 0 for argument 0. See "COMPLEX" for complex arguments.

MANT *Mantissa* Function

Level 1		Level 1
x	➡	*y*
'*symb*'	➡	'MANT(*symb*)'

MANT returns the mantissa of its argument. For example,

1.2E34 MANT returns 1.2.

XPON *Exponent* Function

Level 1		Level 1
x	➡	*n*
'*symb*'	➡	'XPON(*symb*)'

XPON returns the exponent of its argument. For example,

1.2E34 XPON returns 34.

IP FP FLOOR CEIL RND

IP *Integer Part* Function

Level 1		Level 1
x	➡	*n*
'*symb*'	➡	'IP(*symb*)'

IP returns the integer part of its argument. The result has the same sign as the argument.

FP *Fractional Part* Function

Level 1		Level 1
x	➡	*y*
'*symb*'	➡	'FP(*symb*)'

FP returns the fractional part of its argument. The result has the same sign as the argument.

FLOOR *Floor* Function

Level 1		Level 1
x	➡	*n*
'*symb*'	➡	'FLOOR(*symb*)'

FLOOR returns the greatest integer less than or equal to its argument. If the argument is an integer, that value is returned.

CEIL *Ceiling* Function

Level 1		Level 1
x	➡	*n*
'*symb*'	➡	'CEIL(*symb*)'

CEIL returns the smallest integer greater than or equal to its argument. If the argument is an integer, that value is returned.

RND *Round* Function

Level 1		Level 1
x	➡	*y*
'*symb*'	➡	'RND(*symb*)'

RND rounds its argument so that the full-precision internal representation of the number is rounded to match the displayed representation, according to the current display mode:

- In SCI or ENG mode, the internal representation of the result is identical to the displayed number.
- In FIX mode, the internal representation of the result is identical to its displayed value, unless the display has underflowed or overflowed to SCI notation. Thus in *n* FIX mode, the result is 0 if the argument is less than 10^{-n}. No rounding is performed if the argument is greater than or equal to 10^{11}.
- In STD mode, no rounding is performed.

Numbers greater than or equal to 9.5E499 are not rounded.

MAX MIN MOD %T

MAX *Maximum* Function

Level 2	Level 1		Level 1
x	*y*	➡	max(***x***,***y***)
x	'*symb*'	➡	'MAX(*x*, *symb*)'
'*symb*'	*x*	➡	'MAX(*symb*, *x*)'
'$symb_1$'	'$symb_2$'	➡	'MAX($symb_1$, $symb_2$)'

MAX returns the greater (more positive) of its two arguments.

MIN *Minimum* Function

Level 2	Level 1		Level 1
x	*y*	➡	min(*x*,*y*)
x	'*symb*'	➡	'MIN(*x*,*symb*)'
'*symb*'	*x*	➡	'MIN(*symb*,*x*)'
'$symb_1$'	'$symb_2$'	➡	'MIN($symb_1$,$symb_2$)'

MIN returns the lesser (more negative) of its two arguments.

MOD *Modulo* Function

Level 2	Level 1		Level 1
x	*y*	➡	*x* mod *y*
x	'*symb*'	➡	'MOD(*x*,*symb*)'
'*symb*'	*x*	➡	'MOD(*symb*,*x*)'
'$symb_1$'	'$symb_2$'	➡	'MOD($symb_1$,$symb_2$)'

MOD applied to real-valued arguments x and y returns a remainder defined by

$$x \bmod y = x - y \text{ floor } (x/y)$$

Mod (x, y) is periodic in x with period y. Mod (x, y) lies in the interval $[0, y)$ for $y > 0$ and in (y, 0] for $y < 0$.

%T **_Percent of Total_** **Function**

Level 2	Level 1		Level 1
x	*y*	➡	100*y*/*x*
x	'*symb*'	➡	'%T(*x*,*symb*)'
'*symb*'	*x*	➡	'%T(*symb*,*x*)'
'$symb_1$'	'$symb_2$'	➡	'%T($symb_1$,$symb_2$)'

%T computes the (percent) fraction of the real-valued argument *x* in level 2 that is represented by the argument *y* in level 1. That is, %T returns 100*y*/*x*.

SOLVE

STEQ **RCEQ** **SOLVR** **ISOL** **QUAD** **SHOW**
ROOT

The SOLVE menu (SOLV) contains commands that enable you to find the solutions of algebraic expressions and equations. By *solution*, we mean a mathematical *root* of an expression—that is, a value of one variable contained in the expression, for which the expression has the value zero. For an equation, this means that both sides of the equation have the same numerical value.

The command ROOT is a sophisticated numerical root-finder that can determine a numerical root for any mathematically reasonable expression. You can use ROOT as an ordinary command, or you can invoke the root-finder through the SOLVR key. SOLVR activates an interactive version of the root-finder called the *Solver*. The Solver provides a menu for data input and for selecting a "solve" variable, and returns labeled results with messages to help you interpret the results.

It is also possible to solve many expressions symbolically, that is, to return symbolic rather than numerical values for the roots of an expression. The command ISOL (isolate) finds a symbolic solution by isolating the first occurrence of a specified variable within an expression. QUAD returns the symbolic solution of a quadratic equation.

In many cases, a symbolic result is preferable to a numerical result. The functional form of the symbolic result gives much more information about the behavior of the system represented by a mathematical expression than can a single number. Also, a symbolic solution can contain *all* of the multiple roots of an expression. Even if you are only interested in numerical results, solving an expression symbolically before using SOLVR can result in a significant time savings in obtaining the numerical roots.

...SOLVE

Interactive Numerical Solving: The Solver

The Solver is an interactive operation that automates the process of storing values into the variables of an equation, and then solving for any one of the variables. The general procedure for using the Solver is as follows:

1. Use STEQ ("Store Equation") to select a *current equation*.
2. Press SOLVR to activate the Solver *variables menu*.
3. Use the variables menu keys to store values for the equation variables, including a "first guess" for the value of the unknown variable.
4. Solve the equation for an unknown, by pressing the shift key (■) then the menu key corresponding to the unknown variable.

Each of these steps is described in detail in the following sections.

The Current Equation

The current equation is defined as the procedure that is currently stored in the user variable EQ. The term *current equation* (and the name EQ) is chosen to reflect the typical use of the procedure; however, the procedure can be an algebraic equation or expression, or a program. A program used with the Solver must be equivalent to an algebraic; that is, it must not take arguments from the stack, and should return one result to the stack.

You can think of the current equation as an "implicit" argument for SOLVR (it is also the argument for DRAW). An implicit argument saves you from having to place a procedure on the stack every time you use SOLVR or DRAW.

For the purpose of solving (root-finding) equations and expressions, you can consider an expression as the left side of an equation with its right side 0. Alternatively, you can interpret an equation as an expression by treating the = sign as equivalent to − (subtract).

Described next are STEQ and RCEQ, which are commands for storing and recalling the contents of EQ.

STEQ *Store Equation* Command

Level 1		
obj	➡	

STEQ takes an object from the stack, and stores it in the variable EQ ("EQuation"). EQ is used to hold the *current equation* used by the Solver and plot applications, so STEQ's argument should normally be a procedure.

RCEQ *Recall Equation* Command

		Level 1
	➡	*obj*

RCEQ returns the contents of the variable EQ. It is equivalent to `'EQ' RCL`.

...SOLVE

Activating the Variables Menu

Pressing SOLVR activates the Solver *variables menu* derived from the current equation. The variables menu contains:

- A menu key label for each independent variable in the current equation. If there are more than six independent variables, you can use the NEXT and ■ PREV keys to activate each group of (up to) six keys.
- One or two menu keys for evaluating the current equation. If EQ contains an algebraic expression or a program, the key EXPR= is provided for evaluating the expression or program. If EQ contains an algebraic equation, LEFT= and RT= allow you to evaluate separately the left and right sides of the equation.

How The Variables Menu Is Configured. An *independent* variable named in the current equation is either a formal variable, or a variable that contains a data object, usually a real number. A variable containing a procedure will not appear in the variables menu. Rather, the names appearing in that procedure are taken as possible independent variables; those that contain data objects are added to the variables menu. The process continues until all independent variables are identified in the menu. The variables menu is continuously updated, so that if you store a procedure into any of the variables in the menu, that variable will be replaced in the menu by the new independent variables contained in the procedure.

For example, if the current equation is 'A+B=C', the variables menu:

A B C LEFT= RT=

results if A, B, and C do not contain procedures. But if we store 'D+E' in C, the menu will become

A B D E LEFT= RT=

(If a current equation variable itself contains an equation, the latter equation is treated as an expression by replacing the = with a −, for the purpose of defining the variable.)

Storing Values into the Independent Variables

Pressing a Solver variables menu key `name`, where *name* is any of the independent variable names, is similar to executing the sequence `'name' STO`. That is, `name` takes an object from the stack and stores it as the value of the variable *name*.

To confirm input, `name` also displays *name*: *object* in display line 1, where *object* is the object taken from the stack. The message will disappear at the next key press.

At any time, you can review the contents of a variable by pressing ['] `name` and then ■[RCL], ■[VISIT], or [EVAL].

Choosing Initial Guesses

In general, algebraic expressions and procedures can have more than one root. For example, the expression *(x − 3) (x − 2)* has roots at $x = 3$ and $x = 2$. The root that the root-finder returns depends on the starting point for its search, called the *initial guess*.

You should always supply an initial guess for the root-finder. The guess is one of the required arguments for the command ROOT. For the Solver, the current value of the unknown variable is taken as the initial guess. If the unknown variable has no value, the Solver will assign it an initial guess value 0 when you solve for it, but there is no guarantee that this default initial guess will yield the root you desire.

You can speed up the root-finding, or guide the root-finder to a particular root, by making an appropriate initial guess. The guess can be any of following objects:

- A number, or a list containing one number. This number is converted to two initial guesses, as described next, by duplicating it and perturbing one copy slightly.
- A list containing two numbers. The two numbers identify a region in which the search will begin. If the two numbers surround an odd number of roots (signified by their procedure values having opposite signs), then the root-finder can usually find a root between the numbers quite rapidly. If the procedure values at the two numbers do not differ in sign, then the root-finder must search for a region where a root lies. Selecting numbers as near a root as possible will tend to speed up this search.
- A list containing three numbers. In this case the first number should represent your best guess for the root of interest. The other two numbers should surround the best guess, and define a region in which the search should begin. The list of three numbers returned when you interrupt the root-finder with the ON key corresponds to the current guess in this format.

Any of the numbers described above can be complex; in that case only the real parts are used.

The best way to choose an initial guess is to plot the current equation. The plot gives you an idea of the global behavior of the equation and lets you see the roots. For an equation, the roots are the values of the independent (horizontal) variable for which the two curves representing the equation intersect; for an expression (or a program), the roots are the points at which the curve intersects the horizontal axis (vertical coordinate = 0). If you use the interactive plotter (DRAW), you can move the cursor to the desired root, and digitize one or more points. Then you can use the point coordinate(s) as the initial guess(es) for the solver.

Solving for the Unknown Variable.

To *solve* the current equation for an "unknown" variable *name*, press the shift key 🟥 and then the menu key name. This activates the numerical root-finder, to determine a value of the unknown variable that is a root of the current equation (that is, makes the current equation have the value zero). While the root-finder is executing, the message

`Solving for` *name*

is displayed in display line 1. When execution is completed, the result is returned to the stack, and display line 1 shows

name`: result`

(until you press a key). Line 2 gives a message that qualifies the result.

While the Solver root-finder is executing, you can:

- Press ON to stop the root-finder iteration and return to the normal stack display. When the root-finder is halted in this manner, it displays its current best value for the root to the unknown variable, and returns a list containing current best value plus two additional real numbers specifying the search region. If you wish to restart the root-finder, you can just press the unknown variable menu key to store the list into the variable, then the shifted menu key. By using the list as a guess, you can restart the root-finder at the same point where it was interrupted.
- Press any other key to display the intermediate results of the root-finder as it seeks a root. Lines 2 and 3 of the display will show two current guesses used by the root-finder, plus the signs of the value of the current equation evaluated at the guesses. If the current equation is undefined at a guess point, the sign is shown as `?`.

The intermediate results that are displayed are two values of the unknown variable that characterize the region in which the root-finder is searching for a root. Although their precise meaning varies according to the nature of the current equation, you can view them as an indicator of the progress of the root search. Typically, the root-finder will search the domain of the procedure until it finds a sign reversal, indicated by opposite signs for the procedure at two guesses. Then you will see the two guesses "bracket" the root and converge from opposite sides to the root value. On the other hand, if you observe the two guesses diverging from each other, it indicates that the root-finder has not yet located a region with a possible root. If the divergence continues, it most likely indicates that there is no finite root.

Interpreting Results

The HP-28C root-finder seeks a real root of a specified procedure, starting with the first guess that you have supplied. In most cases, the root-finder returns a result. The command ROOT just returns the result to the stack. The Solver returns the result to the stack, displays a labeled result in line 1 of the display, and shows a *qualifying message* in line 2. The qualifying message provides a rough guide to the nature of the root found:

Message	Meaning
`Zero`	An "exact" root of the procedure has been found: evaluating the procedure at the root returns the value zero.
`Sign Reversal`	An approximation to a root, correct to 12 digits, has been found. The root-finder has found neighboring points for which the procedure value changes sign, but no point at which it evaluates to zero. This root may or may not be a normal root, or it may be a discontinuity in the procedure value, across which the value changes sign.
`Extremum`	The root-finder has found an approximation to a local minimum or maximum of the numerical absolute value of the procedure. If the "root" is ±9.99999999999E499, it corresponds to an asymptotic extremum.

After you have obtained a result using the Solver or ROOT, you should evaluate the procedure for which the result was obtained, in order to interpret the results. (If you are using the variables menu, you can use EXPR= for an expression or a program, or LEFT= and RT= for an equation.) There are two possibilities: the value of the procedure at the value of the unknown variable returned by the root-finder is close to zero; or it is not close to zero. It is up to you to decide how close is close enough to consider the value a root.

The best way to understand the nature of a root is to plot the procedure in the neighborhood of the root. The plot will show you whether the root is a proper root, or a discontinuity, much more clearly than any qualifying message that the Solver can return.

During its search for a root, the root-finder may evaluate the procedure at values of the unknown variable that cause mathematical exceptions. No error is generated, but the appropriate mathematical exception user flags will be set.

Errors

In two cases the root-finder will fail, indicating the problem with an error message:

Error Message	Meaning
Bad Guess(es)	One or both initial guesses lie outside of the domain of the procedure. That is, the procedure returns an error when evaluated at the guess points.
Constant?	The procedure returns the same value at every point sampled by the root-finder.

ROOT *Root-Finder* Command

Level 3	Level 2	Level 1		Level 1
«program»	*' name '*	*guess*	➡	*root*
«program»	*' name '*	*{ guesses }*	➡	*root*
' symb '	*' name '*	*guess*	➡	*root*
' symb '	*' name '*	*{ guesses }*	➡	*root*

ROOT takes a procedure, a name, and either a single guess (a real number or a complex number) or a list of one, two, or three guesses, and returns a real number *root*. *Root* is a value of the variable *name* that is returned by the HP-28C numerical root-finder. Where the mathematical behavior of the procedure is appropriate, *root* is a mathematical root—a value of the variable for which the procedure has a numerical value zero. Refer to "Interpreting Results" for more information on interpreting the results of the root-finder.

The single *guess*, or the list of *guesses*, are guesses of the value of the root that you must supply to indicate to the root-finder the region in which the search for a root is to begin. "Choosing Initial Guesses" explains how to choose initial guesses.

If you interrupt ROOT by pressing the ON key, the procedure is returned to level 3, the name to level 2, and a list containing three intermediate values of the unknown variable to level 1. The current best value for the root is stored in the unknown variable. The list is suitable for use as a first guess if you wish to restart the root-finder.

Symbolic Solutions

ISOL *Isolate* Command

	Level 2	Level 1		Level 1
	' $symb_1$ '	' *name* '	➧	' $symb_2$ '

ISOL returns an expression $symb_2$ that represents the rearrangement of its argument algebraic $symb_1$ to "isolate" the first occurrence of variable *name*. If the variable occurs only once in the definition of $symb_1$, then $symb_2$ is a symbolic root (solution) of $symb_1$. If *name* appears more than once, then $symb_2$ is effectively the right side of an equation obtained by rearranging and solving $symb_1$ to isolate the first occurrence of *name* on the left side of the equation. (If $symb_1$ is an expression, consider it as the left side of an equation $symb_1 = 0$.)

If *name* appears in the argument of a function within $symb_1$, that function must be an *analytic function*—the HP-28C must be able to compute the inverse of the function. Thus ISOL cannot solve IP(X) = 0 for X, since IP has no inverse. Commands for which the HP-28C can compute an algebraic inverse are identified as *analytic functions* in this manual.

QUAD **Quadratic Form** **Command**

Level 2	Level 1		Level 1
'symb$_1$'	'name'	➡	'symb$_2$'

QUAD solves an algebraic *symb*$_1$ for the variable *name*, and returns an expression *symb*$_2$ representing the solution. QUAD computes the second-degree Taylor series approximation of *symb*$_1$ to convert it to a quadratic form (this will be exact, if *symb*$_1$ is already a second order polynomial in *name*).

QUAD evaluates *symb*$_2$ before returning it to the stack. If you want a symbolic solution, you should purge any variables that you want to remain in the solution as formal variables.

SHOW **Show Variable** **Command**

Level 2	Level 1		Level 1
'symb$_1$'	'name'	➡	'symb$_2$'

SHOW returns *symb*$_2$, which is equivalent to *symb*$_1$, except that all implicit references to a variable *name* are made explicit. For example, if we define

`'X+1' 'A' STO 'Y+2' 'B' STO,`

then

`'A*B' 'Y' SHOW` returns `'A*(Y+2)'`

and

`'A*B' 'X' SHOW` returns `'(X+1)*B'`.

General Solutions

HP-28C functions are *functions* in the strict mathematical sense, that is, they always return exactly one result when evaluated. This means, for example, that $\sqrt{4}$ always returns +2, not −2 or ±2. For other functions, such as ASIN, a principal value is returned, according to common mathematical conventions.

This implies, however, that pairs of functions such as √ and SQ, or SIN and ASIN, do not necessarily represent the general inverse *relation* implied by their names. Consider the equation $x^2 = 2$. If we take the square root of both sides, we obtain the "solutions"

$$x = +\sqrt{2} \text{ and } x = -\sqrt{2}.$$

The HP-28C equation `'X=√2'` cannot represent correctly both solutions—the √ function always returns the positive square root. Similarly, if we solve $\sin x = .5$ for x, there are an infinite number of solutions $x = 30° + 360n°$, where n is any integer. Applying the ASIN function to .5 will only return the single result 30°.

The principal value flag, user flag 34, determines the nature of solutions returned by ISOL and QUAD. If the flag is set, all arbitrary signs and integers are chosen automatically to represent principal values. If the flag is clear, solutions are returned in their full generality.

...SOLVE

General Solution Mode

When the HP-28C is in general solution mode, signified by flag 34 clear, the commands QUAD and ISOL solve expressions in their full generality by introducing, where appropriate, special user variables representing arbitrary signs and arbitrary integers. You can select values for these variables in the usual way by storing the desired values into the corresponding variables, then evaluating the expression. QUAD and ISOL introduce variables in this manner:

- When a command returns a result containing one or more *arbitrary signs*, the first such sign is represented by a variable s1, the second by s2, and so on. Example:

  'X^2+5*X+4' 'X' QUAD returns '(-5+s1*3)/2'.

  The s1 represents the conventional ± symbol. You can choose either root by storing +1 or −1 into s1, then executing EVAL.

- If ISOL returns a result containing one or more arbitrary integers, the first is represented by a variable n1, the next by n2, and so on. Example:

  'X^4=Y' 'X' ISOL returns 'EXP(2*π*i*n1/4)*Y^.25'.

  The exponential represents the arbitrary complex sign of the result; there are three unique values, corresponding to n1 = 0, 1, and 2. You can choose one of these values by storing the appropriate number into n1, then evaluating the expression.

An alternate keyboard method of substituting for the arbitrary variables in an ISOL or QUAD result expression is to EDIT the expression and make the arbitrary variables into temporary variables for which you supply values. For example, to choose the negative root in the above QUAD example, press ■ EDIT to copy the result expression to the command line, then press

INS -1 → s1 ENTER.

This makes s1 into a local variable, assigns it the value −1, and then evaluates the expression. This method has the advantage that it avoids creating "permanent" variables in user memory corresponding to the arbitrary variables.

Principal Value Mode

If you set flag 34, QUAD and ISOL will return "principal" values for their solutions. That is:

- Arbitrary signs are chosen to be positive. This applies both to the ordinary ±, and to the more general complex "sign" exp $(2\pi ni/x)$ that arises from inverting expressions of the form y^x. In the latter case, the arbitrary integer n is chosen to be 0.
- Arbitrary integers are chosen to be 0. Thus

  `'SIN(X)=Y' 'X' ISOL` returns `'ASIN(Y)'`,

  which always lies in the range 0 through 180 degrees.

You should understand that these choices of "principal" values serve primarily to simplify the result expressions. Mathematically, they are no better or worse than any other roots of an expression. If you desire symbolic results that can subsequently be evaluated for purposes other than simple visual inspection, you should work with flag 34 clear, so that the results are completely general.

STACK

DUP	OVER	DUP2	DROP2	ROT	LIST→
ROLLD	PICK	DUPN	DROPN	DEPTH	→LIST

This menu provides commands to manipulate the contents of the stack. The most frequently used of these commands are provided on the keyboard; the remainder are available as menu keys in the STACK menu.

The keyboard commands are [DROP], ■[SWAP], ■[ROLL], and ■[CLEAR].

Keyboard Commands

DROP *Drop* Command

Level 1		
obj	➡	

DROP removes the first object from the stack. The remaining items on the stack drop one level.

You can recover the dropped object by executing LAST if it is enabled.

SWAP *Swap* Command

Level 2	Level 1		Level 2	Level 1
obj_1	obj_2	➡	obj_2	obj_1

SWAP switches the order of the first two objects on the stack.

ROLL *Roll* Command

Level $n+1$ ... Level 2	Level 1		Level n ... Level 2	Level 1
obj_1 ... obj_n	n	➡	obj_2 ... obj_n	obj_1

ROLL takes an integer n from the stack and "rolls" the first n objects remaining on the stack. For example, `4 ROLL` moves the object in level 4 to level 1.

CLEAR *Clear* Command

Level n ... Level 1		
obj_1 ... obj_n	➡	

CLEAR removes all objects from the stack.

If UNDO is enabled, you can recover the stack that has been lost due to an inadvertent CLEAR by pressing ■ UNDO immediately after the CLEAR.

DUP OVER DUP2 DROP2 ROT LIST→

DUP *Duplicate* Command

Level 1		Level 2	Level 1
obj	➡	obj	obj

DUP returns a copy of the object in level 1. Pressing ENTER when no command line is present executes DUP.

OVER *Over* Command

	Level 2	Level 1		Level 3	Level 2	Level 1
	obj$_1$	*obj*$_2$	➡	*obj*$_1$	*obj*$_2$	*obj*$_1$

OVER returns a copy of the object in level 2.

DUP2 *Duplicate Two Objects* Command

	Level 2	Level 1		Level 4	Level 3	Level 2	Level 1
	obj$_1$	*obj*$_2$	➡	*obj*$_1$	*obj*$_2$	*obj*$_1$	*obj*$_2$

DUP2 returns copies of the first two objects on the stack.

DROP2 *Drop* Command

	Level 2	Level 1		
	obj$_1$	*obj*$_2$	➡	

DROP2 removes the first two objects from the stack. The two objects are saved in LAST arguments. They can be recovered with LAST if it is enabled.

ROT *Rotate* Command

Level 3	Level 2	Level 1		Level 3	Level 2	Level 1
obj$_1$	*obj*$_2$	*obj*$_3$	➡	*obj*$_2$	*obj*$_3$	*obj*$_1$

ROT rotates the first three objects on the stack, bringing the third object to level 1. ROT is equivalent to `3 ROLL`.

LIST→ *List to Stack* Command

Level 1		Level n+1 ... Level 2	Level 1
{ obj_1 ... obj_n }	➧	obj_1 ... obj_n	n

LIST→ takes a list of n objects from the stack, and returns the objects comprising the list into separate stack levels 2 through $n+1$. The number n is returned to level 1.

ROLLD PICK DUPN DROPN DEPTH →LIST

ROLLD *Roll Down* Command

Level n+1 ... Level 2	Level 1		Level n	Level n−1 ... Level 1
obj_1 ... obj_n	n	➧	obj_n	obj_1 ... obj_{n-1}

ROLLD takes an integer n from the stack and "rolls down" the first n objects remaining on the stack. For example, `4 ROLLD` moves the object in level 1 to level 4.

PICK *Pick* Command

Level n+1 ... Level 2	Level 1		Level n+1 ... Level 2	Level 1
obj_1 ... obj_n	n	➧	obj_1 ... obj_n	obj_1

PICK takes an integer n from the stack and returns a copy of obj_1 (the nth remaining object). For example, `4 PICK` returns a copy of the object in level 4.

DUPN *Duplicate n Objects* Command

Level $n+1$ ... Level 2	Level 1		Level $2n$ ... Level $n+1$	Level n ... Level 1
obj_n ... obj_1	n	➡	obj_n ... obj_1	obj_n ... obj_1

DUPN takes an integer number n from the stack, and returns copies of the first remaining *n* objects on the stack (the objects in levels 2 through *n* + 1 while *n* is on the stack).

DROPN *Drop n Objects* Command

Level $n+1$ ... Level 2	Level 1		
obj_1 ... obj_n	n	➡	

DROPN removes the first *n* + 1 objects from the stack (the first *n* excluding the number *n* itself). The number *n* is saved in LAST arguments, for recovery by LAST. You can use ■ UNDO to recover the dropped objects that remain.

DEPTH *Depth* Command

		Level 1
	➡	n

DEPTH returns a real number n representing the number of objects present on the stack (before DEPTH was executed).

→LIST *Stack to List* Command

Level $n+1$ ... Level 2	Level 1		Level 1
obj_1 ... obj_n	n	➡	{ obj_1 ... obj_n }

→LIST takes an integer number n from level 1, plus n additional objects from levels 2 through $n + 1$, and returns a list containing the n objects.

Executing `DEPTH →LIST` combines the entire contents of the stack into a list, which you can, for example, store in a variable for later recovery.

STAT

Σ+	Σ−	**NΣ**	**CLΣ**	**STOΣ**	**RCLΣ**
TOT	**MEAN**	**SDEV**	**VAR**	**MAXΣ**	**MINΣ**
COLΣ	**CORR**	**COV**	**LR**	**PRDEV**	
UTPC	**UTPF**	**UTPN**	**UTPT**		

HP-28C statistics commands deal with statistical data collected in an $n \times m$ matrix called the *current statistics matrix*. The current statistics matrix is defined to be a matrix stored in the variable ΣDAT.

The current statistics matrix ΣDAT is created automatically, if it does not already exist, when you begin entry of statistical *data points* with the command Σ+. A data point is a vector of *m coordinate values* (real numbers), and is stored as one row in the statistics matrix. The first data point entered sets the *m* dimension (number of columns) of the statistics matrix. The *n* dimension (number of rows) is the number of data points that have been entered as illustrated below:.

Data Point	**Coordinate Number**			
	1	**2**	. . .	***m***
1 2 ⋮ *n*	X_{11} X_{21} ⋮ X_{n1}	X_{12} X_{22} ⋮ X_{n2}	. . . .	X_{1m} X_{2m} ⋮ X_{nm}

Certain statistics commands combine data from two specified columns of the statistics matrix. User variable ΣPAR contains a list of four real numbers, the first two of which identify the two columns. You select the columns with the command COLΣ. The last two numbers in the list are the slope and intercept computed from the most recent execution of the linear regression command LR.

Because ΣDAT and ΣPAR are ordinary variables, you can use ordinary commands to recall, view, or alter their contents, in addition to the specific statistics commands that deal with the variables.

The commands SDEV (*standard deviation*), VAR (*variance*), and COV (*covariance*) calculate *sample statistics* using data that represent a sample of the population. These commands are described in detail below. If the data represent the entire population, you can calculate the *population statistics* as follows.

1. Execute MEAN to return a data point representing the mean data.
2. Execute Σ+ to add the mean data point to the data.
3. Execute SDEV, VAR, or COV. The result is the statistics for the population.
4. Execute Σ− DROP to remove the mean data point from the data.

Σ+ Σ− NΣ CLΣ STOΣ RCLΣ

These commands allow you to select a statistics matrix, and to add data to or delete data from the matrix.

Σ+ ***Sigma Plus*** **Command**

Level 1	
x	➡
[x_1 x_2 ... x_m]	➡
[[x_{11} x_{12} ... x_{1m}] ⋮ [x_{n1} x_{n2} ... x_{nm}]]	➡

Σ+ adds one or more data points to the current statistics matrix ΣDAT.

For a statistics matrix with m columns, you can enter the argument for Σ+ in several ways:

Entering one data point with a single coordinate value. The argument for Σ+ is a real number.

Entering one data point with multiple coordinate values. The argument for Σ+ is a vector of m real coordinate values.

Entering several data points. The argument for Σ+ is a matrix of n rows of m real coordinate values.

In each case, the coordinate values are added as new rows to the current statistics matrix stored in ΣDAT. If ΣDAT does not exist, Σ+ creates it as an $n \times m$ matrix stored in the variable ΣDAT. If ΣDAT does exist, an error occurs if it does not contain a real matrix, or if the number of coordinate values in each data point entered with Σ+ doesn't match the number of columns in ΣDAT.

Σ− **Sigma Minus** **Command**

	Level 1
➡	x
➡	$[x_1\ x_2\ \ldots\ x_m]$

Σ− returns a vector of m real numbers, or one number if $m = 1$, corresponding to the coordinate values in the last data point entered by Σ+ into the statistics matrix ΣDAT. The last row of the statistics matrix is deleted.

The vector returned by Σ− can be edited or replaced, then restored to the statistics matrix by Σ+.

NΣ *Sigma N* Command

	Level 1
➡	*n*

NΣ returns the number of data points entered in the statistics matrix stored in ΣDAT. The number of points is equal to the number of rows of the matrix.

CLΣ *Clear Sigma* Command

➡	

CLΣ clears the statistics matrix by purging the ΣDAT variable.

STOΣ *Store Sigma* Command

Level 1	
[*matrix*] ➡	

STOΣ takes a matrix from the stack and stores it in the variable ΣDAT.

RCLΣ *Recall Sigma* Command

	Level 1
➡	*obj*

RCLΣ returns the current contents of the variable ΣDAT. RCLΣ is equivalent to `'ΣDAT' RCL`.

TOT MEAN SDEV VAR MAXΣ MINΣ

These commands compute elementary statistics for the data in each column of the current statistics matrix.

TOT *Total* Command

	Level 1
➡	x
➡	$[x_1\ x_2\ \ldots\ x_m]$

TOT computes the sum of each of the *m* columns of coordinate values in the statistics matrix ΣDAT. The sums are returned as a vector of *m* real numbers, or as a single real number if $m = 1$.

MEAN *Mean* Command

	Level 1
➡	x
➡	$[x_1\ x_2\ \ldots\ x_m]$

MEAN computes the mean of each of the *m* columns of coordinate values in the statistics matrix ΣDAT, and returns the mean as a vector of *m* real numbers, or as a single real number if $m = 1$. The mean is computed from the formula

$$\text{mean} = \sum_{i=1}^{n} x_i / n$$

where x_i is the *i*th coordinate value in a column, and *n* is the number of data points.

SDEV *Standard Deviation* Command

		Level 1
	➡	x
	➡	[x_1 x_2 ... x_m]

SDEV computes the sample standard deviation of each of the m columns of coordinate values in the current statistics matrix. The standard deviations are returned as a vector of m real numbers, or as a single real number if $m = 1$. The standard deviations are computed from the formula

$$\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

where x_i is the ith coordinate value in a column, $\bar{x}$ is the mean of the data in this column, and n is the number of data points.

VAR *Variance* Command

		Level 1
	➡	x
	➡	[x_1 x_2 ... x_m]

VAR computes the sample variance of the coordinate values in each of the m columns of the current statistics matrix. The variance is returned as a vector of m real numbers, or as a single real number if $m = 1$. The variance is computed from the formula

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where x_i is the ith coordinate value in a column, $\bar{x}$ is the mean of the data in this column, and n is the number of data points.

MAXΣ *Maximum Sigma* Command

	Level 1
➡	x
➡	$[x_1\ x_2\ \ldots\ x_m]$

MAXΣ finds the maximum coordinate value in each of the m columns of the current statistics matrix. The maxima are returned as a vector of m real numbers, or as a single real number if $m = 1$.

MINΣ *Minimum Sigma* Command

	Level 1
➡	x
➡	$[x_1\ x_2\ \ldots\ x_m]$

MINΣ finds the minimum coordinate value in each of the m columns of the current statistics matrix. The minima are returned as a vector of m real numbers, or as a single real number if $m = 1$.

COLΣ CORR COV LR PREDV

COLΣ *Sigma Columns* Command

Level 2	Level 1	
n_1	n_2	➡

COLΣ takes two column numbers, n_1 and n_2, from the stack and stores them as the first two objects in the list contained in the variable ΣPAR. The numbers identify column numbers in the current statistics matrix ΣDAT, and are used by statistics commands that work with pairs of columns. n_1 designates the column corresponding to the independent variable for LR, or the horizontal coordinate for DRWΣ or SCLΣ. n_2 designates the dependent variable or the vertical coordinate. For CORR and COV, the order of the two column numbers is unimportant.

If any of the two-column commands is executed when ΣPAR does not yet exist, it is automatically created with default values $n_1 = 1$ and $n_2 = 2$.

CORR *Correlation* Command

	Level 1
	➡ *correlation*

CORR returns the correlation of two columns of coordinate values in the current statistics matrix. The columns are specified by the first two elements of ΣPAR (default 1 and 2). The correlation is computed from the formula

$$\frac{\sum_{i=1}^{n} (x_{in_1} - \bar{x}_{n_1})(x_{in_2} - \bar{x}_{n_2})}{\sqrt{\sum_{i=1}^{n} (x_{in_1} - \bar{x}_{n_1})^2 \sum_{i=1}^{n} (x_{in_2} - \bar{x}_{n_2})^2}}$$

where x_{in_1} is the ith coordinate value in column n_1, x_{in_2} is the ith coordinate value in the column n_2, $\bar{x}_{n_1}$ is the mean of the data in column n_1, $\bar{x}_{n_2}$, is the mean of the data in column n_2, and n is the number of data points.

COV *Covariance* Command

	Level 1
➡	*covariance*

COV returns the sample covariance of the coordinate values in two columns of the current statistics matrix. The columns are specified by the first two elements in ΣPAR (default 1 and 2). The covariance is computed from the formula

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_{in_1}-\bar{x}_{n_1})(x_{in_2}-\bar{x}_{n_2})$$

where x_{in_1} is the ith coordinate value in column n_1, x_{in_2} is ith coordinate value in the column n_2, $\bar{x}_{n_1}$ is the mean of the data in column n_1, $\bar{x}_{n_2}$ is the mean of the data in column n_2, and n is the number of data points.

LR *Linear Regression* Command

	Level 2	Level 1
➡	*intercept*	*slope*

LR computes the linear regression of a dependent data column on an independent data column, where the columns of data exist in the current statistics matrix. The columns of independent and dependent data are specified by the first two elements in ΣPAR (default 1 and 2).

The *intercept* and *slope* of the regression line are returned to levels 2 and 1 of the stack, respectively. LR also stores these regression coefficients as the third (intercept) and fourth (slope) items in the list in the variable ΣPAR.

PREDV — *Predicted Value* — Command

Level 1		Level 1
x	➧	*predicted value*

PREDV computes a predicted value from a real number argument x, using the linear regression coefficients most recently computed with LR and stored in the variable ΣPAR:

$$\textit{predicted value} = (x \times \textit{slope}) + \textit{intercept}.$$

The regression coefficients *intercept* and *slope* are stored by LR as the third and fourth items, respectively, in the variable ΣPAR. If you execute PREDV without having previously executed LR, a default value of zero is used for both coefficients, so that PREDV will always return zero.

UTPC UTPF UTPN UTPT

The HP-28C provides four *upper-tail probability* commands, which you can use to determine the statistical significance of test statistics. The upper-tail probability function of a random variable X is the probability that X is greater than a number x, and is equal to $1 - \mathrm{F}(x)$, where $\mathrm{F}(x)$ is the distribution function of X.

The inverses of distribution functions are useful for constructing confidence intervals. The argument of an inverse upper-tail probability function is a value from 0 through 1; when the argument is expressed as a percent, the inverse function values are called percentiles. For example, the 90th percentile of a distribution is the number x for which the probability that the random variable X is greater than x is $100\% - 90\% = 10\%$.

...STAT

You can use the Solver to obtain the inverses of the upper-tail probability functions. Suppose you wish to determine a percentile of the normal distribution. Let

$$P = \text{percentile}/100$$

$$M = \text{mean of the distribution}$$

$$V = \text{variance}$$

$$X = \text{random variable}$$

UTPN (described below) returns the upper-tail probability for normal distribution. To solve the equation

$$1 - \mathrm{P} = \text{utpn}\ (M,\ V,\ X),$$

for X, create the program

```
« 1 P - M V X UTPN - »,
```

and store it as the current equation by pressing SOLV STEQ. Then press SOLVR to produce the Solver menu:

P M V X EXPR= .

Try a normal distribution with $M = 0$, $V = 1$:

0 M **1** V .

Now compute the 95th percentile:

.95 P 🟥 X

yields the result $X = 1.6449$.

UTPC *Upper Chi-Square Distribution* Command

Level 2	Level 1		Level 1
n	*x*	➡	*utpc(n, x)*

UTPC returns the probability utpc(n, x) that a chi-square random variable is greater than x, where n is the number of degrees of freedom of the distribution. n must be a positive integer.

UTPF *Upper Snedecor's F Distribution* Command

Level 3	Level 2	Level 1		Level 1
n_1	n_2	*x*	➡	*utpf*(n_1, n_2, x)

UTPF returns the probability utpf(n_1, n_2, x) that a Snedecor's F random variable is greater than x, where n_1 and n_2 are the numerator and denominator degrees of freedom of the F distribution. n_1 and n_2 must be positive integers.

UTPN *Upper Normal Distribution* Command

Level 3	Level 2	Level 1		Level 1
m	*v*	*x*	➡	*utpn(m, v, x)*

UTPN returns the probability utpn(m, v, x) that a normal random variable is greater than x, where m and v are the mean and variance, respectively, of the normal distribution. v must be a non-negative number.

UTPT *Upper Student's t Distribution* Command

Level 2	Level 1		Level 1
n	*x*	➡	*utpt*(*n*, *x*)

UTPT returns the probability utpt(*n*, *x*) that a Student's *t* random variable is greater than *x*, where *n* is the number of degrees of freedom of the distribution. *n* must be a positive integer.

STORE

STO+	**STO−**	**STO∗**	**STO/**	**SNEG**	**SINV**
SCONJ					

The STORE menu contains storage arithmetic commands which allow you to perform addition, subtraction, multiplication, division, inversion, negation, and conjugation on real and complex numbers and arrays that are stored in variables, without recalling the variable contents to the stack. Besides minimizing keystrokes in many cases, the STORE commands provide an "in-place" method of altering the contents of an array, which requires less memory than manipulating the array while it is on the stack.

Storage arithmetic is restricted to ordinary variables—you cannot use a local name as an argument for any of the commands in the STORE menu.

STO+ STO− STO∗ STO/ SNEG SINV

STO+ *Store Plus* Command

Level 2	Level 1		
z	*'name'*	➡	
'name'	*z*	➡	
[array]	*'name'*	➡	
'name'	*[array]*	➡	

STO+ adds a number or array to the contents of the variable. The variable name and the number or array can be in either order on the stack.

The object on the stack and the object in the variable must be suitable for addition to each other—you can add any combination of real and complex numbers, or any combination of conformable real and complex arrays.

STO− *Store Minus* Command

Level 2	Level 1		
z	' *name* '	➡	
' *name* '	*z*	➡	
[*array*]	' *name* '	➡	
' *name* '	[*array*]	➡	

STO− computes the difference of two numbers or arrays. One object is taken from the stack, and the other is the contents of a variable specified by a name on the stack. The resulting difference is stored as the new value of the variable.

The result depends on the order of the arguments:

- If *name* is in level 1, the difference

  (value in level 2) − (value in *name*)

  becomes the new value of *name*.
- If *name* is in level 2, the difference

  (value in *name*) − (value in level 1)

  becomes the new value in *name*.

The object on the stack and the object in the variable must be suitable for subtraction with each other—you can subtract any combination of real and complex numbers, or any combination of conformable real and complex arrays.

STO*   *Store Times*   Command

Level 2	Level 1	
z	' *name* '	➡
' *name* '	*z*	➡
[*array*]	' *name* '	➡
' *name* '	[*array*]	➡

STO* multiplies the contents of a variable by a number or array. When multiplying two numbers or a number and an array, the variable name and the other object can be in either order on the stack. When multiplying two arrays, the result depends on the order of the arguments:

- If *name* is in level 1, the product

  (array in level 2) × (array in *name*)

  becomes the new value of *name*.
- If *name* is in level 2, the product

  (array in *name*) × (array in level 1)

  becomes the new value in *name*.

The arrays must be conformable for multiplication.

STO/   *Store Divide*   Command

Level 2	Level 1	
z	' *name* '	➡
' *name* '	*z*	➡
[*array*]	' *name* '	➡
' *name* '	[*array*]	➡

STO/ computes the quotient of two numbers or arrays. One object is taken from the stack, and the other is the contents of a variable specified by a name. The resulting quotient is stored as the new value of the variable.

...STORE

The result depends on the order of the arguments:

- If *name* is in level 1, the quotient

  (value in level 2)/(value in *name*)

  becomes the new value of *name*.
- If *name* is in level 2, the quotient

  (value in *name*)/(value in level 1)

  becomes the new value in *name*.

The object on the stack and the object in the variable must be suitable for division with each other. In particular, if both objects are arrays, the divisor (level 1) must be a square matrix, and the dividend (level 2) must have the same number of columns as the divisor.

SNEG **Store Negate** **Command**

Level 1	
' *name* '	➡

SNEG negates the contents of the variable named on the stack; the result replaces the original contents of the variable. The variable may contain a real number, a complex number, or an array.

SINV **Store Invert** **Command**

Level 1	
' *name* '	➡

SINV computes the inverse of the contents of the variable named on the stack; the result replaces the original contents of the variable. The variable may contain a real number, a complex number, or a square matrix.

SCONJ

SCONJ **Store Conjugate** Command

Level 1	
' name '	➡

SCONJ conjugates the contents of the variable named on the stack; the result replaces the original contents of the variable. The variable may contain a real number, a complex number, or an array.

STRING

→STR	**STR→**	**CHR**	**NUM**	**POS**	**DISP**
SUB	**SIZE**				

A *string* object consists of a sequence of characters delimited by double-quote marks " at either end. Any HP-28C character can be included in a string, including the object delimiters (,), [,], {, }, #, ", ', », and «. Characters not directly available on the keyboard can be entered by means of the CHR command.

Although you can include " characters within a string (using CHR and +), you will not be able to EDIT a string containing a " in the usual way. This is because ENTER attempts to match pairs of "'s in the command line—extra "'s within a string will cause the string to be broken into two or more strings that will contain no "'s.

Strings are used primarily for display purposes—prompting, labeling results, and so on. The commands included in the STRING menu provide simple string and character operations. However, the commands →STR and STR→ add an important application for strings—they can convert any object, or sequence of objects, to and from a character-string form. In many cases, the string form requires less memory than the normal form of an object. You can store objects in variables as strings and convert them to the normal form only when you need them. See the descriptions of →STR and STR→ below for more information.

Keyboard Function

+ ***Add*** **Analytic**

Level 2	Level 1		Level 1
"*string*$_1$"	"*string*$_2$"	➧	"*string*$_1$ *string*$_2$"

+ concatenates the characters in the string in level 1 to the characters in the string in level 2, producing a string result.

→STR STR→ CHR NUM POS DISP

→STR

Object to String **Command**

Level 1		Level 1
obj	➧	"*string*"

→STR converts an arbitrary object to a string form. The string is essentially the same as the display form of the object that you would obtain when the object is in level 1, and multi-line display mode is active:

- The result string includes the entire object, even if the displayed form of the object is too large to fit in the display.
- If the object is displayed in two or more lines, the result string will contain newline characters (character 10) at the end of each line. The newlines are displayed as the default character ▪.

- Numbers are converted to strings according to the current number display mode (STD, FIX, SCI, or ENG) or binary integer base (DEC, BIN, OCT, or HEX) and wordsize. The full-precision internal form of the number is not necessarily represented in the result string. You can insure that →STR preserves the full precision of a number by selecting STD mode or a wordsize of 64 bits, or both, prior to executing →STR.
- If the object is already a string, →STR returns the string.

You can use →STR to create special displays to label program output or provide prompts for input. For example, the sequence

```
"Result = " SWAP →STR + 1 DISP
```

displays `Result =` *object* in line 1 of the display, where *object* is a string form of an object taken from level 1.

STR→ *String to Objects* Command

Level 1	
"*string*" ➡	

STR→ is a command form of ENTER. The characters in the string argument are parsed and evaluated as contents of the command line. The string may define a single object, or it may be a series of objects that will be evaluated just like a program.

STR→ can also be used to restore objects that were converted to strings by →STR back to their original form. The combination →STR STR→ leaves objects unchanged except that →STR converts numbers to strings according to the current number display format and binary integer base and wordsize. STR→ will reproduce a number only to the precision represented in the string form.

CHR *Character* Command

Level 1		Level 1
n	➧	"*string*"

CHR returns a one-character string containing the HP-28C character corresponding to the character code *n* taken from level 1. The default character ▪ is used for all character codes that are not part of the normal HP-28C display character set.

Character code 0 is used for special purposes in the command line. You can include this character in strings by using CHR, but attempting to edit a string containing this character causes the `Can't Edit CHR(0)` error.

NUM *Character Number* Command

Level 1		Level 1
"*string*"	➧	*n*

NUM returns the character code of the first character in a string.

The following table shows the relation between character codes (results of NUM, arguments to CHR) and characters (results of CHR, arguments to NUM). For character codes 0 through 147, the table shows the characters as displayed in a string. For character codes 148 through 255, the table shows the characters as printed by the HP 82240A printer; these characters are displayed on the HP-28C as the default character ▪.

...STRING

Character Codes (0–127)

NUM	CHR	NUM	CHR	NUM	CHR	NUM	CHR
0	▪	32		64	@	96	`
1	▪	33	!	65	A	97	a
2	▪	34	"	66	B	98	b
3	▪	35	#	67	C	99	c
4	▪	36	$	68	D	100	d
5	▪	37	%	69	E	101	e
6	▪	38	&	70	F	102	f
7	▪	39	'	71	G	103	g
8	▪	40	(	72	H	104	h
9	▪	41	)	73	I	105	i
10	▪	42	*	74	J	106	j
11	▪	43	+	75	K	107	k
12	▪	44	,	76	L	108	l
13	▪	45	-	77	M	109	m
14	▪	46	.	78	N	110	n
15	▪	47	/	79	O	111	o
16	▪	48	0	80	P	112	p
17	▪	49	1	81	Q	113	q
18	▪	50	2	82	R	114	r
19	▪	51	3	83	S	115	s
20	▪	52	4	84	T	116	t
21	▪	53	5	85	U	117	u
22	▪	54	6	86	V	118	v
23	▪	55	7	87	W	119	w
24	▪	56	8	88	X	120	x
25	▪	57	9	89	Y	121	y
26	▪	58	:	90	Z	122	z
27	▪	59	;	91	[	123	{
28	▪	60	<	92	\	124	|
29	▪	61	=	93	]	125	}
30	▪	62	>	94	^	126	~
31	▪	63	?	95	_	127	▒

Character Codes (128–255)

NUM	CHR	NUM	CHR	NUM	CHR	NUM	CHR
128		160	∡	192	â	224	Á
129	÷	161	À	193	ê	225	Ã
130	×	162	Â	194	ô	226	ã
131	√	163	È	195	û	227	Đ
132	∫	164	Ê	196	á	228	đ
133	Σ	165	Ë	197	é	229	Í
134	▶	166	Î	198	ó	230	Ì
135	π	167	Ï	199	ú	231	Ó
136	∂	168	´	200	à	232	Ò
137	≤	169	`	201	è	233	Õ
138	≥	170	ˆ	202	ò	234	õ
139	≠	171	¨	203	ù	235	Š
140	α	172	˜	204	ä	236	š
141	→	173	Ù	205	ë	237	Ú
142	←	174	Û	206	ö	238	Ÿ
143	µ	175	₤	207	ü	239	ÿ
144	␊	176	¯	208	Å	240	Þ
145	°	177	Ý	209	î	241	þ
146	«	178	ý	210	ø	242	·
147	»	179	°	211	Æ	243	µ
148	⊢	180	Ç	212	å	244	¶
149	₁	181	ç	213	í	245	¾
150	₂	182	Ñ	214	ø	246	—
151	²	183	ñ	215	æ	247	¼
152	³	184	¡	216	Ä	248	½
153	ᵢ	185	¿	217	ì	249	ª
154	ⱼ	186	¤	218	Ö	250	º
155	‥	187	£	219	Ü	251	«
156	ⁱ	188	¥	220	É	252	■
157	ʲ	189	§	221	ï	253	»
158	ᵏ	190	ƒ	222	ß	254	±
159	ⁿ	191	¢	223	Ô	255	

...STRING

POS — *Position* — Command

Level 2	Level 1		Level 1
"*string*$_1$"	"*string*$_2$"	➧	*n*

POS returns a real number *n* that is the position of *string*$_2$ within *string*$_1$. The position is the character position, counting from the left, of the first character of the substring within *string*$_1$ that matches *string*$_2$. A zero result indicates that *string*$_2$ is not matched within *string*$_1$.

DISP — *Display* — Command

Level 2	Level 1		
obj	*n*	➧	

DISP displays *obj* in the *n*th line of the display, where *n* is a real integer. $n = 1$ indicates the top line of the display; $n = 4$ indicates the bottom line. DISP sets the system message flag to suppress the normal stack display.

Strings are displayed without the surrounding " delimiters. Other objects are displayed in the same form as they are in level 1 in multiline display mode. If the object display requires more than one display line, the display starts in line *n* and continues down the display, either to the end of the object or the bottom of the display.

SUB SIZE

SUB *Subset* Command

Level 3	Level 2	Level 1		Level 1
"*string*$_1$"	n_1	n_2	➡	"*string*$_2$"
{ *list*$_1$ }	n_1	n_2	➡	{ *list*$_2$ }

SUB takes a string and two integer numbers n_1 and n_2 from the stack, and returns a new string containing the characters in positions n_1 through n_2 of the original string. If $n_2 < n_1$, SUB returns an empty string.

Arguments less than 1 are converted to 1; arguments greater than the size of the string are converted to the string size.

Refer to "LIST" for the use of SUB with lists.

SIZE *Size* Command

Level 1		Level 1
"*string*"	➡	*n*
[*array*]	➡	{ *list* }
{ *list* }	➡	*n*
' *symb* '	➡	*n*

SIZE returns a number n that is the number of characters in a string.

Refer to "ALGEBRA", "ARRAY" and "LIST" for the use of SIZE with other object types.

TRIG

SIN	ASIN	COS	ACOS	TAN	ATAN
P→R	R→P	R→C	C→R	ARG	
→HMS	HMS→	HMS+	HMS−	D→R	R→D

The TRIG (trigonometry) menu contains commands related to angular measurement and trigonometry: circular functions, polar/rectangular conversions, degrees/radians conversions, and calculations with values expressed in degrees-minutes-seconds or hours-minutes-seconds form.

SIN ASIN COS ACOS TAN ATAN

These are the circular functions and their inverses. SIN, COS and TAN interpret real arguments according to the current angle mode (DEG or RAD), returning real results. ACOS, ASIN, and ATAN express real results according to the current angle mode.

All six functions accept complex arguments, producing complex results. For ACOS and ASIN, real arguments with absolute value greater than 1 also produce complex results. Complex numbers are interpreted and expressed in radians.

ASIN, ACOS, and ATAN return the principal values of the inverse relations, as described in "COMPLEX."

SIN *Sine* Analytic

Level 1		Level 1
z	➡	sin z
'*symb*'	➡	'SIN(*symb*)'

SIN returns the sine of its argument. For complex arguments,

$$\sin(x + iy) = \sin x \cosh y + i \cos x \sinh y.$$

ASIN *Arc sine* Analytic

Level 1		Level 1
z	➡	arc sin z
'*symb*'	➡	'ASIN(*symb*)'

ASIN returns the principal value of the angle having a sine equal to its argument. For real arguments, the range of the result is from −90 to +90 degrees ($-\pi/2$ to $+\pi/2$ radians). For complex arguments, the complex principal value of the arc sine is returned:

$$\text{arc sin } z = -i \ln (iz + \sqrt{1 - z^2})$$

A real argument x outside of the domain $-1 \leqslant x \leqslant 1$ is converted to a complex argument $z = x + 0i$, and the complex principal value is returned.

COS **Cosine** **Analytic**

Level 1		Level 1
z	➡	cos z
'*symb*'	➡	'COS(*symb*)'

COS returns the cosine of its argument. For complex arguments,

$$\cos (x + iy) = \cos x \cosh y - i \sin x \sinh y$$

ACOS **Arc cosine** **Analytic**

Level 1		Level 1
z	➡	arc cos z
'*symb*'	➡	'ASIN(*symb*)'

ACOS returns the principal value of the angle having a cosine equal to its argument. For real arguments, the range of the result is from 0 to 180 degrees (0 to π radians). For complex arguments, ACOS returns the complex principal value of the arc cosine:

$$\text{arc cos } z = -i \ln (z + \sqrt{z^2 - 1})$$

A real argument x outside of the domain $-1 \leqslant x \leqslant 1$ is converted to a complex argument $z = x + 0i$, and the complex principal value is returned.

TAN *Tangent* Analytic

Level 1		Level 1
z	➡	tan *z*
'*symb*'	➡	'TAN(*symb*)'

TAN returns the tangent of its argument. For complex arguments,

$$\tan(x + iy) = \frac{\sin x \cos x + i \sinh y \cosh y}{(\sinh y)^2 + (\cos x)^2}$$

If a real argument is an odd integer multiple of 90, and if DEG angle mode is set, an `Infinite Result` exception occurs. If flag 59 is clear, the sign of the (MAXR) result is that of the argument.

ATAN *Arc tangent* Analytic

Level 1		Level 1
z	➡	arc tan *z*
'*symb*'	➡	'ATAN(*symb*)'

ATAN returns the principal value of the angle having a tangent equal to its argument. For real arguments, the range of the result is from -90 to $+90$ degrees ($-\pi/2$ to $+\pi/2$ radians). For complex arguments, ATAN returns the complex principal value of the arc tangent:

$$\text{arc tan } z = \frac{i}{z} \ln\left(\frac{i + z}{i - z}\right)$$

...TRIG

P→R R→P R→C C→R ARG

The functions P→R (*polar-to-rectangular*), R→P (*rectangular-to-polar*), and ARG (*argument*) deal with complex numbers that represent the coordinates of points in two dimensions. R→C *(real-to-complex)* and C→R *(complex-to-real)* convert pairs of real numbers to and from complex notation.

The functions P→R and R→P can also act on the first two elements of a real vector.

P→R *Polar to Rectangular* Function

	Level 1		Level 1
	`x`	➡	`(x,0)`
	`(r,θ)`	➡	`(x,y)`
	`[ r θ...]`	➡	`[ x y...]`
	`'symb'`	➡	`'P→R(symb)'`

P→R converts a complex number (r, θ) or two-element vector $[\,r\ \theta\,]$, representing polar coordinates, to a complex number (x, y) or two-element vector $[\,x\ y\,]$, representing rectangular coordinates, where:

$$x = r \cos \theta, \qquad y = r \sin \theta.$$

The current angle mode determines whether θ is interpreted as degrees or radians.

If a vector has more than two elements, P→R converts the first two elements and leaves the remaining elements unchanged. For three-element vectors, P→R converts a vector $[\,\rho\ \theta\ z\,]$ from cylindrical coordinates (where ρ is the distance to the z-axis, and θ is the angle in the xy-plane from the x-axis to the projected vector) to the vector $[\,x\ y\ z\,]$ in rectangular coordinates.

You can represent a vector in spherical coordinates as [r ϕ θ], where r is the length of the vector, ϕ is the angle from the z-axis to the vector, and θ is the angle in the xy-plane from the x-axis to the projected vector. To convert a vector from spherical to rectangular coordinates, execute:

```
P→R ARRY→ DROP ROT { 3 } →ARRY P→R
```

R→P *Rectangular-to-Polar* Function

Level 1	Level 1
z ➡	(*r*, *θ*)
[*x* *y*...] ➡	[*r* *θ*...]
'*symb*' ➡	'R→P(*symb*)'

R→P converts a complex number (x, y) or two-element vector [x y], representing rectangular coordinates, to a complex number (r, θ) or two-element vector [r θ], representing polar coordinates, where:

$$r = \text{abs}\ (x,\ y), \qquad \theta = \text{arg}\ (x,\ y)$$

The current angle mode determines whether θ is expressed as degrees or radians. A real argument x is treated as the complex argument $(x, 0)$.

If a vector has more than two elements, R→P converts the first two elements and leaves the remaining elements unchanged. For three-element vectors, R→P converts a vector [x y z] from rectangular coordinates to a vector [ρ θ z] in cylindrical coordinates, where ρ is the distance to the z-axis, and θ is the angle in the xy-plane from the x-axis to the projected vector.

You can represent a vector in spherical coordinates as [r ϕ θ], where r is the length of the vector, ϕ is the angle from the z-axis to the vector, and θ is the angle in the xy-plane from the x-axis to the projected vector. To convert a vector from rectangular to spherical coordinates, execute:

```
R→P ARRY→ DROP ROT ROT { 3 } →ARRY R→P
```

R→C *Real to Complex* Command

Level 2	Level 1		Level 1
x	y	➡	(x,y)

R→C combines real numbers x and y into a coordinate pair (x, y).

Refer to "ARRAY" for the use of R→C with arrays.

C→R *Complex to Real* Command

Level 1		Level 2	Level 1
(x,y)	➡	x	y

C→R converts a coordinate pair (x, y) into two real numbers x and y.

Refer to "ARRAY" for the use of C→R with arrays.

ARG *Argument* Function

Level 1		Level 1
z	➡	*θ*
'*symb*'	➡	'ARG(*symb*)'

ARG returns the polar angle θ of a coordinate pair (x, y) where

$$\theta = \begin{cases} \text{arc tan } y/x & \text{for } x \geqslant 0, \\ \text{arc tan } y/x + \pi \text{ sign } y & \text{for } x < 0, \text{ radians mode,} \\ \text{arc tan } y/x + 180 \text{ sign } y & \text{for } x < 0, \text{ degrees mode.} \end{cases}$$

The current angle mode determines whether θ is expressed as degrees or radians. A real argument x is treated as the complex argument $(x, 0)$.

→HMS HMS→ HMS+ HMS− D→R R→D

The commands →HMS, HMS→, HMS+, and HMS− deal with time (or angular) quantities expressed by real numbers in HMS (*hours-minutes-seconds*) format.

The HMS format is *h.MMSSs*, where:

- *h* is zero or more digits representing the integer part of the number.
- *MM* are two digits representing the number of minutes.
- *SS* are two digits representing the number of seconds.
- *s* is zero or more digits representing the decimal fractional part of seconds.

Here are examples of time (or angular) quantities expressed in HMS format.

Quantity	HMS Format
12h 32m 46s (12° 32′ 46″)	12.3246
−6h 00m 13.2s (−6° 00′ 13.2″)	−6.00132
36m (36′)	0.36

→HMS *Decimal to H-M-S* Command

Level 1		Level 1
x	➡	*hms*

→HMS converts a real number representing decimal hours (or degrees) to HMS format.

HMS→ *H-M-S to Decimal* Command

Level 1		Level 1
hms	➡	*x*

HMS→ converts a real number in HMS format to its decimal form.

HMS+ *Hours-Minutes-Seconds Plus* Command

Level 2	Level 1		Level 1
hms_1	hms_2	➡	$hms_1 + hms_2$

HMS+ adds two numbers in HMS format, returning the sum in HMS format.

HMS− *Hours-Minutes-Seconds Minus* Command

Level 2	Level 1		Level 1
hms_1	hms_2	➡	$hms_1 - hms_2$

HMS− subtracts two real numbers in HMS format, returning the difference in HMS format.

D→R *Degrees to Radians* Function

Level 1		Level 1
x	➡	$(\pi/180)\ x$
'*symb*'	➡	'D→R(*symb*)'

D→R converts a real number expressed in degrees to radians.

R→D *Radians to Degrees* Function

Level 1		Level 1
x	➡	$(180/\pi)\ x$
'*symb*'	➡	'R→D(*symb*)'

R→D converts a real number expressed in radians to degrees.

UNITS

The value of a physical measurement includes units as well as a numerical value. To convert a physical measurement from one system of units to another, you multiply the numerical value by a conversion factor, which is the ratio of the new units to the old units. The HP-28C automates this process through the command CONVERT. You specify a numerical value, the old units, and the new units, and then CONVERT computes the appropriate conversion factor and multiplies the numerical value by the conversion factor.

The HP-28C's unit conversion system is based upon the International System of Units (SI). There are 120 units included in the HP-28C's permanent memory. CONVERT recognizes any multiplicative combination of these units, as well as additional units that you can define. The UNITS catalog lists the built-in units and their values in terms of standard base quantities.

The International System specifies seven base quantities: length (meter), mass (kilogram), time (second), electric current (ampere), thermodynamic temperature (kelvin), luminous intensity (candela), and amount of substance (mole). In addition, the HP-28C recognizes one undefined base quantity, which you may specify as part of user-defined units.

CONVERT *Convert* Command

Level 3	Level 2	Level 1		Level 2	Level 1
x	*"old"*	*"new"*	➡	*y*	*"new"*
x	*"old"*	*'new'*	➡	*y*	*'new'*
x	*'old'*	*"new"*	➡	*y*	*"new"*
x	*'old'*	*'new'*	➡	*y*	*'new'*

CONVERT multiplies a real number *x* by a conversion factor, which is computed from two arguments representing old and new units. The resulting real number *y* is returned to level 2, and the new unit string is returned to level 1.

Generally the old and new units are represented by string objects, as described below. For convenience in simple conversions, however, you can use a name object to represent a unit. For example, assuming you haven't created variables named 'ft' or 'm', you could convert 320 feet to meters by executing:

```
320 ft m CONVERT.
```

The unit strings are string objects that represent algebraic expressions containing unit abbreviations. A unit string may contain:

- Any built-in or user-defined units. Built-in units are represented by their abbreviations (refer to "The UNITS Catalog"). User units are represented by their variable names (refer to "User-Defined Units").
- A unit followed by the ^ symbol, plus a single digit 1-9. For example: `"m^2"` (meters squared), `"g*s^3"` (gram-seconds cubed).
- A unit preceded by a prefix representing a multiplicative power of 10. For example: `"Mpc"` (Megaparsec), `"nm"` (nanometer). (Refer to "Unit Prefixes").
- Two or more units multiplied together, using the * symbol. For example: `"g*cm"` (gram-centimeters), `"ft*lb"` (foot-pounds), `"m*kg*s"` (meter-kilogram-seconds).
- One / symbol to indicate inverse powers of units. If all units in a unit string have inverse powers, the unit string can start with `"1/"`. For example: `"m/sec"` (meters per second), `"1/m"` (inverse meters), `"g*cm/s^2*°K"` (gram-centimeters per second squared per degree Kelvin).
- The ' symbol, which is ignored. This allows you to create an algebraic expression on the stack and then use →STR to change the expression to a unit string. However, parentheses are not allowed in unit strings.

...UNITS

The two unit strings must represent a dimensionally consistent unit conversion. For example, you can convert "l" (liters) to "cm^3" (cubic centimeters), but not to "acre". CONVERT checks that the powers of each of the eight base quantities (seven SI base quantities plus one user-defined base quantity) are the same in both unit strings. (Dimensionality consistency is checked in modulo 256.)

Here are some examples of using CONVERT (numbers shown in STD format):

Old Value	Old Units	New Units		New Value	New Units
1	"m"	"ft"	➧	3.28083989501	"ft"
1	"b*Mpc"	"cm^3"	➧	3.085678	"cm^3"
12.345	"kg*m/s^2"	"dyn"	➧	18585	"dyn"

Temperature Conversions

Conversions between the four temperature scales (°K, °C, °F, and °R) involve additive constants as well as multiplicative factors. If both unit string arguments contain only a single, unprefixed temperature unit with no exponent, CONVERT performs an absolute temperature scale conversion, including the additive constants. For example, to convert 50 degrees Fahrenheit to degrees Celsius, execute:

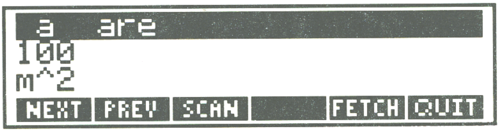
```
50 °F °C CONVERT
```

If either unit string includes a prefix, an exponent, or any unit other than a temperature unit, CONVERT performs a relative temperature unit conversion, which ignores the additive constants.

The UNITS Catalog

Pressing ■ UNITS activates the UNITS catalog, which is analogous to the command catalog obtained with ■ CATALOG. The UNITS catalog lists each unit included in the HP-28C, along with the abbreviation recognized by CONVERT and the value of the unit in terms of the SI base quantities.

When you press ■ UNITS, the normal HP-28C display is superceded by the UNITS display:



```
 a   are
100
m^2
```

The top line shows the unit abbreviation of the selected unit, in this example a, followed by the full name, are. Are is the first unit in the HP-28C alphabetical unit catalog. The second line shows the unit's value in SI base units, which are shown in the third line. Altogether, this display shows that are is abbreviated a and has the value 100 meters squared.

...UNITS

The UNITS menu is shown in the bottom line. The five active menu keys act as follows:

Menu Key	Description
NEXT	Advance the catalog to the next unit in the catalog.
PREV	Move the catalog to the previous unit in the catalog.
SCAN	Scan forward through the unit catalog, showing each unit's name briefly. Pressing SCAN changes the menu label to STOP ; pressing STOP halts the scan at the current unit. The scan stops automatically at the last unit in the catalog (the unit "1").
FETCH	Exit the catalog and add the current unit abbreviation to the command line at the cursor position (start a new command line if none is present).
QUIT	Exit the catalog, leaving any current command line unchanged.

In addition to the operations available in the UNITS menu, you can:

- Press any letter key to move the catalog to the first unit that starts with that letter.
- Press any non-letter key on the left-hand keyboard to move the catalog to "°", the first non-alphabetic unit.
- Press [1] to move the catalog display to "1", the last non-alphabetical unit.
- Press [ON] to exit the catalog and clear the command line.

The following table shows all the units in the UNITS catalog, including descriptions of the units.

HP-28C Units

Unit	Full Name	Description	Value
a	Are	Area	100 m^2
A	Ampere	Electric current	1 A
acre	Acre	Area	4046.87260987 m^2
arcmin	Minute of arc	Plane angle	2.90888208666E-4
arcs	Second of arc	Plane angle	4.8481368111E-6
atm	Atmosphere	Pressure	101325 Kg/m*s^2
au	Astronomical unit	Length	149597900000 m
A°	Angstrom	Length	.0000000001 m
b	Barn	Area	1.E-28 m^2
bar	Bar	Pressure	100000 Kg/m*s^2
bbl	Barrel, oil	Volume	.158987294928 m^3
Bq	Becquerel	Activity	1 1/s
Btu	International Table Btu	Energy	1055.05585262 Kg*m^2/s^2
bu	Bushel	Volume	.03523907 m^3
C	Coulomb	Electric charge	1 A*s
cal	International Table calorie	Energy	4.1868 Kg*m^2/s^2
cd	Candela	Luminous intensity	1 cd
chain	Chain	Length	20.1168402337 m
Ci	Curie	Activity	3.7E10 1/s
ct	Carat	Mass	.0002 Kg
cu	US cup	Volume	2.365882365E-4 m^3
d	Day	Time	86400 s
dyn	Dyne	Force	.00001 Kg*m/s^2
erg	Erg	Energy	.0000001 Kg*m^2/s^2

HP-28C Units (Continued)

Unit	Full Name	Description	Value
eV	Electron volt	Energy	1.60219E-19 Kg*m^2/s^2
F	Farad	Capacitance	1 A^2*s^4/Kg*m^2
fath	Fathom	Length	1.82880365761 m
fbm	Board foot	Volume	.002359737216 m^3
fc	Footcandle	Luminance	10.7639104167 cd/m^2
Fdy	Faraday	Electric charge	96487 A*s
fermi	Fermi	Length	1.E-15 m
flam	Footlambert	Luminance	3.42625909964 cd/m^2
ft	International foot	Length	.3048 m
ftUS	Survey foot	Length	.304800609601 m
g	Gram	Mass	.001 Kg
ga	Standard freefall	Acceleration	9.80665 m/s^2
gal	US gallon	Volume	.003785411784 m^3
galC	Canadian gallon	Volume	.00454609 m^3
galUK	UK gallon	Volume	.004546092 m^3
gf	Gram-force	Force	.00980665 Kg*m/s^2
grad	Grade	Plane angle	1.57079632679E-2
grain	Grain	Mass	.00006479891 Kg
Gy	Gray	Absorbed dose	1 m^2/s^2
h	Hour	Time	3600 s
H	Henry	Inductance	1 Kg*m^2/A^2*s^2
hp	Horsepower	Power	745.699871582 Kg*m^2/s^3
Hz	Hertz	Frequency	1 1/s

HP-28C Units (Continued)

Unit	Full Name	Description	Value
in	Inch	Length	.0254 m
inHg	Inches of mercury	Pressure	3386.38815789 Kg/m*s^2
inH2O	Inches of water	Pressure	248.84 Kg/m*s^2
J	Joule	Energy	1 Kg*m^2/s^2
kip	Kilopound-force	Force	4448.22161526 Kg*m/s^2
knot	Knot	Speed	.514444444444 m/s
kph	Kilometer per hour	Speed	.277777777778 m/s
l	Liter	Volume	.001 m^3
lam	Lambert	Luminance	3183.09886184 cd/m^2
lb	Avoirdupois pound	Mass	.45359237 Kg
lbf	Pound-force	Force	4.44822161526 Kg*m/s^2
lbt	Troy lb	Mass	.3732417 Kg
lm	Lumen	Luminance flux	1 cd
lx	Lux	Illuminance	1 cd/m^2
lyr	Light year	Length	9.46055E15 m
m	Meter	Length	1 m
mho	Mho	Electric conductance	1 A^2*s^3/Kg*m^2
mi	International mile	Length	1609.344 m
mil	Mil	Length	.0000254 m
min	Minute	Time	60 s
miUS	US statute mile	Length	1609.34721869 m
mmHg	Millimeter of mercury	Pressure	133.322368421 Kg/m*s^2
mol	Mole	Amount of substance	1 mol

HP-28C Units (Continued)

Unit	Full Name	Description	Value
mph	Miles per hour	Speed	.44704 m/s
N	Newton	Force	1 Kg*m/s^2
nmi	Nautical mile	Length	1852 m
ohm	Ohm	Electric resistance	1 Kg*m^2/A^2*s^3
oz	Ounce	Mass	.028349523125 Kg
ozfl	US fluid oz	Volume	2.95735295625E−5 m^3
ozt	Troy oz	Mass	.031103475 Kg
ozUK	UK fluid oz	Volume	.000028413075 m^3
P	Poise	Dynamic viscosity	.1 Kg/m*s
Pa	Pascal	Pressure	1 Kg/m*s^2
pc	Parsec	Length	3.08567818585E16 m
pdl	Poundal	Force	.138254954376 Kg*m/s^2
ph	Phot	Luminance	10000 cd/m^2
pk	Peck	Volume	.0088097675 m^3
psi	Pounds per square inch	Pressure	6894.75729317 Kg/m*s^2
pt	Pint	Volume	.000473176473 m^3
qt	Quart	Volume	.000946352946 m^3
r	Radian	Plane angle	1
R	Roentgen	Radiation exposure	.000258 A*s/Kg
rad	Rad	Absorbed dose	.01 m^2/s^2
rd	Rod	Length	5.02921005842 m
rem	Rem	Dose equivalent	.01 m^2/s^2
s	Second	Time	1 s
S	Siemens	Electric conductance	1 A^2*s^3/Kg*m^2
sb	Stilb	Luminance	10000 cd/m^2

HP-28C Units (Continued)

Unit	Full Name	Description	Value
slug	Slug	Mass	14.5939029372 Kg
sr	Steradian	Solid angle	1
st	Stere	Volume	1 m^3
St	Stokes	Kinematic viscosity	.0001 m^2/s
Sv	Sievert	Dose equivalant	1 m^2/s^2
t	Metric ton	Mass	1000 Kg
T	Tesla	Magnetic flux	1 Kg/A∗s^2
tbsp	Tablespoon	Volume	1.47867647813E−5 m^3
therm	EEC therm	Energy	105506000 Kg∗m^2/s^2
ton	Short ton	Mass	907.18474 Kg
tonUK	Long ton	Mass	1016.0469088 Kg
torr	Torr	Pressure	133.322368421 Kg/m∗s^2
tsp	Teaspoon	Volume	4.92892159375E−6 m^3
u	Unified atomic mass	Mass	1.66057E−27 Kg
V	Volt	Electric potential	1 Kg∗m^2/A∗s^3
W	Watt	Power	1 Kg∗m^2/s^3
Wb	Weber	Magnetic flux	1 Kg∗m^2/A∗s^2
yd	International yard	Length	.9144 m
yr	Year	Time	31536000 s
°	Degree	Angle	1.74532925199E−2
°C	Degree Celsius	Temperature	1 °K
°F	Degree Fahrenheit	Temperature	.555555555556 °K
°K	Degree Kelvin	Temperature	1 °K

HP-28C Units (Continued)

Unit	Full Name	Description	Value
°R	Degree Rankine	Temperature	.555555555556 °K
μ	Micron	Length	.000001 m
?	User quantity		1 ?
1	Dimensionless unit		1

Sources: The National Bureau of Standards Special Publication 330, *The International System of Units (SI), Fourth Edition,* Washington D.C., 1981.

The Institute of Electrical and Electronics Engineers, Inc., *American National Standard Metric Practice ANSI/IEEE Std. 268-1982,* New York, 1982.

American Society for Testing and Materials, *ASTM Standard for Metric Practice E380-84,* Philadelphia, 1984.

Aerospace Industries Association of America, Inc., *National Aerospace Standard,* Washington D.C., 1977.

Handbook of Chemistry and Physics, 64th Edition, 1983-1984, CRC Press, Inc., Boca Raton, FL, 1983.

User-Defined Units

You can create a variable containing a list that CONVERT will accept as a *user-defined unit* in a unit string. The list must contain a real number and a unit string (similar to the second and third lines in the UNITS display). For example, suppose you often use weeks as a unit of time. Executing

```
{7 "d"} 'WK' STO
```

allows you to use "WK" in conversions or in creating more complicated user-defined units.

The user defined unit string can contain any element of a conversion unit string, along with two other special units:

- To define a dimensionless unit, specify a unit string "1".
- To define a new unit not expressible in SI units, specify a unit string "?". CONVERT will check dimensionality for this unit along with the SI units. For example, to convert money in three currencies, dollars, pounds, and francs, define:

  ```
   {1 "?"} 'DOLLAR' STO
  {2.25 "?"} 'POUND' STO
   {.4 "?"} 'FRANC' STO
  ```

  and then convert between any two of these currencies (the values chosen are just for illustration).

Unit Prefixes

In a unit string you can precede a built-in unit by a prefix indicating a power of ten. For example, "mm" indicates "millimeter", or meter × 10^{-3}. The table below lists the prefixes recognized by CONVERT.

...UNITS

Unit Prefixes

Prefix	Name	Exponent
E	exa	+18
P	peta	+15
T	tera	+12
G	giga	+9
M	mega	+6
k or K	kilo	+3
h or H	hecto	+2
D	deka	+1
d	deci	−1
c	centi	−2
m	milli	−3
μ	micro	−6
n	nano	−9
p	pico	−12
f	femto	−15
a	atto	−18

Most prefixes used by the HP-28C correspond with standard SI notation. The one exception is "deka", indicating an exponent of +1, which is "D" in HP-28C notation and "da" in SI notation.

Note

You can't use a prefix with a unit if the resulting combination would match a built-in unit. For example, you can't use "min" to indicate milli-inches because "min" is a built-in unit indicating minutes. Other possible combinations that would match built-in units are: "Pa", "cd", "ph", "flam", "nmi", "mph", "kph", "ct", "pt", "ft", "au", and "cu".

Although you can't use a prefix with a user-defined unit, you can create a new user-defined unit whose name includes the prefix character.

USER

ORDER	**CLUSR**	**MEM**

The USER menu contains, in addition to the three "permanent" entries ORDER, CLUSR, and MEM, an entry for each current user variable. User variables appear first in the menu, starting at the left with the most recently created variable. If there are more than six user variables, USER activates the first six; each press of NEXT activates a successive group of six. ORDER, CLUSR, and MEM appear when NEXT is pressed while the last group of user variables is displayed, or when ■ PREV is pressed while the first group is active.

Pressing a USER menu key corresponding to a variable causes the associated variable name to be evaluated in immediate entry mode, or appended to the command line in algebraic or alpha entry mode.

The menu-key labels are derived from the user variable names. A label will show the leading characters from the corresponding variable name, as many as will fit in the label display, up to five characters. A variable name may contain lower-case letters, but these will appear as upper case in the menu label.

ORDER CLUSR MEM

ORDER *Order* Command

Level 1	
{ $name_1$ $name_2$ . . . }	➡

ORDER takes a list of variable names, and rearranges variable memory so that the variables will appear in the user menu in the same order as specified in the list. Those variables named in the list are moved to the start of the user menu. Variables not in the list remain in their current order, following the variables named in the list.

CLUSR *Clear User Memory* Command

➡

CLUSR purges all current variables. To prevent accidental execution, pressing CLUSR always executes ENTER and puts `CLUSR` in the command line. Then you can press [ENTER] to execute CLUSR.

MEM *Memory* Command

	Level 1
➡	x

MEM returns the number of bytes of currently unused memory. The number returned by MEM should be used only as a rough indicator of available memory, since the amount of memory used by the calculator can vary depending on current and pending operations that may not be directly visible. The recovery mechanisms associated with ■ [COMMAND], ■ [UNDO], and ■ [LAST], for example, can consume or release substantial amounts of memory when any operations are executed.

A

Messages

This appendix lists all error and status messages given by the HP-28C. Messages are normally displayed in display line 1 and disappear at the next keystroke. (Solver qualifying messages are shown in line 2.)

Messages noted as *status messages* are for your information, and do not indicate error conditions. Messages noted as *math exceptions* will not appear if the corresponding exception error flag is clear. (Math exceptions are described in "Fundamentals.")

Messages Listed Alphabetically

Message	Error Number		Meaning
	Hex	Decimal	
Bad Argument Type	202	514	A command required a different argument object type.
Bad Argument Value	203	515	An argument value was out of the range of operation of a command.
Bad Guess(es)	A01	2561	The guess or guesses supplied to the Solver or ROOT caused invalid results when the current equation was evaluated.
Can't Edit CHR(0)	102	258	An attempt was made to edit a string containing character 0.

Messages Listed Alphabetically (Continued)

Message	Error Number		Meaning
	Hex	Decimal	
Circular Reference	129	297	An attempt was made to store an object in a variable, using the Solver menu, when the object refers to the variable directly or indirectly.
Command Stack Disabled	125	293	🟥 COMMAND was pressed while COMMAND was disabled.
Constant?	A02	2562	The current equation returned the same value at every point sampled by the root-finder.
Constant Equation	Status		The current equation returned the same value for every point within the specified range sampled by DRAW.
Extremum	Status		The result returned by the Solver is an extremum rather than a root.
HALT not Allowed	121	289	DRAW or the Solver encountered a HALT command in the program EQ.
Improper User Function	103	259	An attempt was made to evaluate an improper user-defined function. Refer to "Programs" for correct syntax.
Inconsistent Units	B02	2818	CONVERT was executed with unit strings of different dimensionality.

Messages Listed Alphabetically (Continued)

Message	Error Number		Meaning
	Hex	Decimal	
Infinite Result	305	773	Math exception. A calculation returned an infinite result, such as 1/0 or LN(0).
Insufficient Memory	001	001	There was not enough free memory to execute an operation.
Insufficient Σ Data	603	1539	A statistics command was executed when ΣDAT did not contain enough data points for the calculation.
Interrupted	Status		The Solver was interrupted by the ON key.
Invalid Dimension	501	1281	An array argument had the wrong dimensionality.
Invalid PPAR	11E	286	DRAW or DRWΣ encountered an invalid entry in PPAR.
Invalid Unit String	B01	2817	CONVERT was executed with an invalid unit string.
Invalid ΣDAT	601	1537	A statistics command was executed with an invalid object stored in ΣDAT.
Invalid ΣPAR	604	1540	ΣPAR is the wrong object type or contains an invalid or missing entry in its list.
LAST Disabled	205	517	LAST was executed with flag 31 clear.
Low Memory!	Status		Indicates fewer than 128 bytes of free memory remain.

Messages Listed Alphabetically (Continued)

Message	Error Number		Meaning
	Hex	Decimal	
Memory Lost	005	005	HP-28C memory has been reset.
Negative Underflow	302	770	Math exception. A calculation returned a negative, non-zero result greater than −MINR.
No Current Equation	104	260	SOLVR or DRAW was executed with a nonexistent variable EQ.
Nonexistent ΣDAT	602	1538	A statistics command was executed with a non-existent variable ΣDAT.
Non-real Result	11F	287	A procedure returned a result other than a real number, which was required for the Solver, ROOT, DRAW, or ∫.
No Room for UNDO	101	257	There was not enough free memory to save a copy of the stack. UNDO is automatically disabled.
No Room to ENTER	105	261	There was not enough memory to process the command line.
No Room to Show Stack	Status		There is not enough memory for the normal stack display.
name$_1$ Not in Equation	Status		DRAW was executed when the independent variable *name*$_1$ in PPAR did not exist in the current equation. This message is followed by either **Constant Equation** or **Using** *name*$_2$.

Messages Listed Alphabetically (Continued)

Message	Error Number		Meaning
	Hex	Decimal	
Out of Memory			No free memory is available for continued calculator operation. You must purge one or more objects to continue.
Overflow	303	771	Math exception. A calculation returned a result greater (in absolute value) than MAXR.
Positive Underflow	301	769	Math exception. A calculation returned a positive, non-zero result less than MINR.
Sign Reversal	Status		The Solver was unable to find a point at which the current equation evaluates to zero, but did find two neighboring points at which the equation changed sign.
Syntax Error	106	262	An object in the command line was entered in an invalid form.
Too Few Arguments	201	513	A command required more arguments than were available on the stack.
Unable to Isolate	120	288	The specifed name could not be isolated in the expression. The name was either absent or contained in the argument of a function with no inverse.
Undefined Local Name	003	003	Attempted to evaluate a local name for which a corresponding local variable did not exist.

Messages Listed Alphabetically (Continued)

Message	Error Number		Meaning
	Hex	Decimal	
Undefined Name	204	516	Attempted to recall the value of an undefined (formal) variable.
Undefined Result	304	772	A function was executed with arguments that lead to a mathematically undefined result, such as 0/0, or LNP1(*x*) for x<−1.
UNDO Disabled	124	292	■ UNDO was pressed while UNDO was disabled.
Using *name*	Status		DRAW has selected the independent variable *name*.
Wrong Argument Count	128	296	A user-defined function was evaluated in an expression, with the wrong number of arguments in parentheses.
Zero	Status		The Solver found a value for the unknown variable at which the current equation evaluated to 0.

Error Messages Listed by Error Number

Hex	Decimal	Message
Errors Resulting From General Operations		
001	001	Insufficient Memory
003	003	Undefined Local Name
005	005	Memory Lost
101	257	No Room for UNDO
102	258	Can't Edit CHR(0)
103	259	Improper User Function
104	260	No Current Equation
105	261	No Room to ENTER
106	262	Syntax Error
11E	286	Invalid PPAR
11F	287	Non-real Result
120	288	Unable to Isolate
121	289	HALT not Allowed
124	292	UNDO Disabled
125	293	Command Stack Disabled
128	296	Wrong Argument Count
129	297	Circular Reference
Errors Resulting From Stack Operations		
201	513	Too Few Arguments
202	514	Bad Argument Type
203	515	Bad Argument Value
204	516	Undefined Name
205	517	LAST Disabled
Errors Resulting From Real Number Operations		
301	769	Positive Underflow
302	770	Negative Underflow
303	771	Overflow
304	772	Undefined Result
305	773	Infinite Result

Error Messages Listed by Error Number (Continued)

Hex	Decimal	Message
Errors Resulting From Array Operations		
501	1281	Invalid Dimension
Errors Resulting From Statistics Operations		
601	1537	Invalid ΣDAT
602	1538	Nonexistent ΣDAT
603	1539	Insufficient Σ Data
604	1540	Invalid ΣPAR
Errors Resulting From the Root-finder		
A01	2561	Bad Guess(es)
A02	2562	Constant?
Errors Resulting From Unit Conversion		
B01	2817	Invalid Unit String
B02	2818	Inconsistent Units

B

Notes for HP RPN Calculator Users

Starting with the HP-35 in 1972, Hewlett-Packard has developed a series of handheld scientific and business calculators based upon the RPN stack interface. Although there are many differences in the capabilities and applications of these various calculators, they all share a common implementation of the basic stack interface, which makes it easy for a user accustomed to one calculator to learn to use any of the others.

The HP-28C also uses a stack and RPN logic as the central themes of its user interface. However, the four-level stack and fixed register structure of the previous calculators is inadequate to support the multiple object types and symbolic mathematical capability of the HP-28C. Thus while the HP-28C is a natural evolution of the "original" RPN interface, there are sufficient differences between the HP-28C and its predecessors to require a little "getting used to" if you are accustomed to other RPN calculators. In this appendix, we will highlight the major differences.

The Dynamic Stack

The most dramatic difference in the basic interface of the HP-28C compared with previous HP RPN calculators is the size of the stack. The other calculators feature a fixed, four-level stack consisting of the X-, Y-, Z- and T-registers, augmented by a single LAST X, or L-register. This stack is always "full"—even when you "clear" the stack, all you are doing is filling the stack with zeros.

The HP-28C has no fixed size to its stack. As you enter new objects onto the stack, new levels are dynamically created as they are needed. When you remove objects from the stack, the stack shrinks, even to the point where the stack is empty. Thus the HP-28C can generate a `Too Few Arguments` error that previous HP RPN calculators could not.

The dynamic versus fixed stack implementation gives rise to the following specific differences between the HP-28C and fixed-stack calculators:

Numbered levels. The indefinite size of the HP-28C stack makes the X Y Z T stack level names inappropriate—instead, the levels are numbered. Thus level 1 is analogous to the X-register, 2 to Y, 3 to Z, and 4 to T. The key labels $1/x$ and x^2 were preserved on the HP-28C for the sake of familiarity—they make the keys more visible than their actual command names INV and SQ, respectively. However, the RPN fixture X<>Y has been renamed SWAP on the HP-28C.

Stack Manipulation. The HP-28C requires a more general set of stack manipulation commands than the fixed-stack calculators. R↑ and R↓, for example, are replaced by ROLL and ROLLD, respectively, each of which require an additional argument to specify how many stack levels to roll. The STACK menu contains several stack manipulation commands that do not exist on the fixed-stack calculators.

No Automatic Replication of the T-register. On fixed-stack calculators, the contents of the T-register are duplicated into the Z-register whenever the stack "drops" (that is, when a number is removed from the stack). This provides a convenient means for constant multiplication—you can fill the stack with copies of a constant, then multiply it by a series of numbers by entering each number, pressing [×], then [CLx] after you have recorded each result. You can't do this on the HP-28C—but it is easy to create a program of the form

```
« 12345 * » 'MULT' STO
```

where 12345 represents a typical constant. Then all you have to do is press [USER], enter a number and press MULT, enter a new number and press MULT again, and so on, to perform constant multiplication. You can leave successive results on the stack.

Stack Memory. A dynamic stack has the advantage that you can use as many levels as you need for any calculation, without worrying about losing objects "off the top" as you enter new ones. This also has the disadvantage that you can tie up a significant amount of memory with old objects, if you leave them on the stack after you are finished with a calculation. With the HP-28C, you should get in the habit of discarding unneeded objects from the stack.

DROP Versus CLX. In fixed-stack calculators, CLX means "replace the contents of the X-register with 0, and disable stack lift" (see below). Its primary purpose is to throw away an old number, prior to replacing it with a new one—but you can also use it as a means to enter 0. On the HP-28C, CLX is replaced by DROP, which does what its name implies—it drops the object in level 1 from the stack, and the rest of the stack drops down to fill in. No extraneous 0 is entered. Similarly, CLEAR drops all objects from the stack, instead of replacing them with zeros as does its fixed-stack counterpart CLST (CLEAR STACK).

Stack-Lift Disable and ENTER

Certain commands on fixed-stack calculators (ENTER↑, CLX, Σ+, Σ−) exhibit a peculiar feature called *stack-lift disable*. That is, after any of these commands is executed, the next number entered onto the stack replaces the current contents of the X-register, rather than pushing it into the Y-register. This feature is entirely absent on the HP-28C. New objects entered onto the stack *always* push the previous stack objects up to higher levels.

The X-register and ENTER on fixed-stack calculators play dual roles that are derived more from the single-line display of the calculators than from the stack structure. The X-register acts as an input register as well as an ordinary stack register—when you key in a number, the digits are created in the X-register, until a non-digit key terminates entry. The [ENTER↑] key is provided for separating two consecutive number entries. But in addition to terminating digit entry, the [ENTER↑] key also copies the contents of the X-register into Y, and disables stack lift.

On the HP-28C each of these dual roles is separated—there is no stack lift disable. A command line completely distinct from level 1 (the "X-register") is used for command entry. ENTER is used *only* to process the contents of the command line—it does not duplicate the contents of level 1. Note, however, that the [ENTER] key will execute DUP (which copies level 1 into level 2) if no command line is present. This feature of [ENTER] is provided partly for the sake of similarity to previous calculators.

Prefix Versus Postfix

HP-28C commands use a strict postfix syntax. That is, all commands using arguments require that those arguments be present on the stack before the command is executed. This departs from the convention used by previous RPN calculators, in which arguments specifying a register number, a flag number, and so on, are not entered on the stack but are entered *after* the command itself—for example, STO 25, TONE 1, CF 03, and so on. This latter method has the advantage of saving a stack level, but the disadvantage of requiring an inflexible format—STO on the HP-41, for example, must always be followed by a two-digit register number.

Similar operations of the HP-28C are closer in style to *indirect* operations on the fixed-stack calculators, where you can use an *i-register* (or any register, in the case of the HP-41) to specify the register, flag number, and so on, addressed by a command. You can view STO, RCL, and so on, on the HP-28C as using level 1 as an *i-register*. RCL, for example, means "recall the contents of the variable ('register') named in level 1"— equivalent to RCL IND X on the HP-41.

You should be aware also that most HP-28C commands remove their arguments from the stack. If you execute, for example, 123 'X' STO, the 123 and the 'X' disappear from the stack. Without this behavior, the stack would be overloaded with "old" arguments. If you want to keep the 123 on the stack, you should execute 123 DUP 'X' STO.

Registers Versus Variables

Fixed-stack calculators can deal efficiently only with real, floating-point numbers for which the fixed, seven-byte register structure of the stack and numbered data register memory is suitable (the HP-41 introduced a primitive alpha data object constrained to the seven-byte format). The HP-28C replaces numbered data registers with named variables. Variables, in addition to having a flexible structure so that they can accomodate different object types, have names that can help you remember their contents more readily than can register numbers.

If you want to duplicate numbered registers on the HP-28C, you can use a vector:

```
{ 50 } 0 CON 'REG' STO
```

creates a vector with 50 elements initialized to 0;

```
« 1 →LIST 'REG' SWAP GET » 'NRCL' STO
```

creates a program NRCL that recalls the *n*th element from the vector, where *n* is a number in level 1;

```
« 1 →LIST 'REG' SWAP ROT PUT » 'NSTO' STO
```

creates the analogous store program NSTO.

LASTX Versus LAST

The LASTX command on fixed-stack calculators returns the contents of the LASTX (or L) register, which contains the last value used from the X-register. This concept is generalized on the HP-28C to the LAST command, which returns the last one, two, or three arguments taken from the stack by a command (no command uses more than three arguments). Thus 1 2 + LASTX returns 3 and 2 to the stack on a fixed-stack calculator, but 1 2 + LAST returns 3, 1, and 2 to the stack on the HP-28C.

Although the HP-28C LAST is more flexible than its LASTX predecessor, you should keep in mind that more HP-28C commands use arguments from the stack than their fixed-stack calculator counterparts. This means that the LAST arguments are updated more frequently, and even such commands as DROP or ROLL will replace the LAST arguments.

Remember also that UNDO can replace the entire stack, which for simple error recovery may be preferable to LAST.

C

Notes for Algebraic Calculator Users

Many calculators, including the great majority of simple, "four-function" calculators, use variations of the *algebraic* calculator interface. The name derives from the feature that the keystroke sequences used for simple calculations closely parallel the way in which the calculation is specified in algebraic expressions "on paper." That is, to evaluate 1 + 2 − 3, you press $\boxed{1}\ \boxed{+}\ \boxed{2}\ \boxed{-}\ \boxed{3}\ \boxed{=}$.

This interface works nicely for expressions containing numbers and *operators*—functions like +, −, ×, and / that are written in infix notation between their arguments. More sophisticated calculators allow you to enter parentheses to specify precedence (the order of operations). However, the introduction of prefix functions, like SIN, LOG, and so on, leads to two different variations:

- Ordinary algebraic calculators use a combination of styles—infix operators remain infix, but prefix functions are entered in a postfix style (like RPN calculators). For example, 1 + SIN(23) is entered as $\boxed{1}\ \boxed{+}\ \boxed{2}\ \boxed{3}\ \boxed{\text{SIN}}\ \boxed{=}$. This approach has the advantages of being able to show intermediate results, and of preserving single-key evaluations of prefix functions (that is, without parentheses), but the disadvantage of losing the correspondence with ordinary mathematical notation that is the primary advantage of the algebraic interface.

- "Direct formula entry" calculators, and BASIC language computers that have an immediate-execute mode, allow you to key in an entire expression in its ordinary algebraic form, then compute the result when you press a termination key (variously labeled [ENTER], [ENDLINE], [RETURN], and so on). This approach has the advantage of preserving the correspondence between written expressions and keystrokes, but usually the disadvantage of providing no intermediate results. (The HP-71B CALC mode is an exception.) You have to know the full form of an expression before you start to enter it—it is difficult to "work your way through a problem," varying the calculation according to intermediate results.

Getting Used to the HP-28C

HP-28C operating logic is based on a mathematical logic known as "Polish Notation," developed by the Polish logician Jan Łukasiweicz (*Wookashye'veech*) (1878–1956). Conventional algebraic notation places arithmetic operators *between* the relevant numbers or variables when evaluating algebraic expressions. Łukasiweicz's notation specifies the operators *before* the variables. A variation of this logic specifies the operators *after* the variables—this is termed "Reverse Polish Notation," or "RPN" for short.

The basic idea of RPN is that you enter numbers or other objects into the calculator first, then execute a command that acts on those entries (called "arguments"). The "stack" is just the sequence of objects waiting to be used. Most commands return their results to the stack, where they can then be used as arguments for subsequent operations.

The HP-28C uses an RPN stack interface because it provides the necessary flexibility to support the wide variety of HP-28C mathematical capabilities in a uniform manner. All calculator operations, including those that can not be expressed as algebraic expressions, are performed in the same manner—arguments from the stack, results to the stack.

Nevertheless, using the RPN stack for simple arithmetic is most likely the biggest stumbling block for algebraic calculator users trying to learn to use RPN calculators. RPN is very efficient, but it does require you mentally to rearrange an expression before you can calculate results. But the HP-28C's capability of interpreting algebraic expressions without translation should make the transition from algebraic calculator use more straightforward than has been possible on previous RPN calculators. The four-line display can also help to take away some of the mystery of the stack, by showing you the contents of up to four levels at a time.

For the purpose of evaluating algebraic expressions, the HP-28C is essentially a "direct formula entry" calculator. That is, to evaluate an algebraic expression, all you have to do is precede it with a ', key in the expression in its algebraic form, including infix operators, prefix functions, and parentheses, and then press EVAL to see the result. You can use this method even for simple arithmetic:

' 1 + 2 − 3 EVAL returns 0.

Except for the preceding ', these are the same keystrokes you would use on a simple algebraic calculator, where you substitute EVAL for =.

Note Don't confuse the HP-28C = key with that found on algebraic calculators—on the HP-28C, = is used for the sole purpose of creating algebraic equations (described in "ALGEBRA").

When you use the HP-28C as a "direct formula entry calculator," each result that you compute is retained on the stack, which takes on the role of a "history stack." This allows you to save old results indefinitely for reuse later. It also allows you to break up large calculations into smaller ones, keeping each partial result on the stack and then combining the results when they are all available. (When carried to the extreme, this is the essence of RPN arithmetic). The stack provides a much easier-to-use and more powerful history stack than the single "result" function available on algebraic or BASIC calculators.

A key feature of the HP-28C is that you really don't need to concern yourself over whether RPN logic is better or worse than algebraic logic. You can choose the logic that is best suited for the problem at hand, and intermix algebraic expressions with RPN manipulations.

Glossary

accuracy: For numerical integration, the numerical accuracy of the integrand, which determines the sampling intervals for computation of the integral.

algebraic: Short for *algebraic object.*

algebraic object: A procedure, entered and displayed between ' ' delimiters, containing numbers, variables, operators, and functions combined in algebraic syntax to represent a mathematical expression or equation.

algebraic entry mode: The entry mode in which a key corresponding to a *function* appends its function name and a left parenthesis (if applicable) to the command line. Keys corresponding to other commands execute their commands immediately.

algebraic syntax: The restrictions on a procedure, that (1) when evaluated, it takes no arguments from the stack and returns one result, and (2) it can be subdivided into a hierarchy of subexpressions. These conditions are satisfied by all algebraic objects and some programs.

alpha entry mode: The entry mode in which all keys corresponding to commands add their command names to the command line.

analytic function: A function that can be differentiated or solved for its argument.

annunciators: The icons at the top of the LCD display that indicate the states of certain calculator modes.

arbitrary integer: A variable *n1*, *n2*, and so on, that appears in the solution of an expression with multiple roots. Different roots are obtained by storing real integers into the variables.

arbitrary sign: A variable *s1*, *s2*, and so on, that appears in the solution of an expression with multiple roots. Different roots are obtained by storing +1 or −1 into the variables.

argument: An object taken from the stack by an operation as its input.

array: An object, defined by the [ ] delimiters, that represents a real or complex matrix or vector.

associate: To rearrange the order in which two functions are applied to three arguments, without changing the value of an expression—for example, (a + b) + c is rearranged to a + (b + c). (In RPN form, a b + c + is rearranged to a b c + +.)

base: The number base in which binary integers are displayed. The choices are binary (base 2), octal (base 8), decimal (base 10) and hexadecimal (base 16).

base unit: One of the seven units that are used as the basis for HP-28C unit conversions. The base units are the meter (length), kilogram (mass), second (time), ampere (electric current), kelvin (thermodynamic temperature), candela (luminous intensity) and mole (amount of substance).

binary integer: An object identified by the delimiter #, which represents an integer number with from 1 to 64 binary digits, displayed according to the current *base*.

clause: A program sequence between two program structure commands, such as IF *test-clause* THEN *then-clause* END.

clear: (1) To empty the stack (CLEAR). (2) To blank the display (CLLCD). (3) To assign the value 0 to a user flag (CF).

command: Any HP-28C operation that can be included in the definition of a procedure or included by name in the command line.

command line: The input string that contains non-immediate-execute characters, numbers, objects, commands, and so on, that are entered from the keyboard. ENTER causes the command line string to be converted to a program and evaluated.

command stack: Up to four previously entered command lines that are stored for future retrieval by COMMAND.

commute: To interchange the two arguments of a function.

complex array: An array in which the elements are complex numbers.

complex number: An object delimited by ( ) symbols, consisting of two real numbers representing the real and imaginary parts of a complex number.

conformable: For two arrays, having the correct dimensions for an arithmetic operation.

contents: The object stored in a variable. Also referred to as the variable's *value*.

coordinate pair: A complex number object used to represent the coordinates of a point in two-dimensional space. The real part is the "horizontal" coordinate, and the imaginary part is the "vertical" coordinate.

current equation: The procedure stored in the variable EQ, used as an implicit argument by DRAW and by the Solver.

current statistics matrix: The matrix stored in the variable ΣDAT, containing the statistical data accumulated with Σ+.

cursor: A display character that highlights a position on the display. (1) The command line cursor indicates where the next character will be entered into the command line. It varies its appearance to indicate the current entry mode. (2) The FORM cursor is an inverse-video highlight that identifies the selected subexpression. (3) The DRAW/DRWΣ cursor is a small cross that indicates the position of a point to be digitized.

data object: An object that, when evaluated, returns itself to the stack. Includes real and complex numbers, arrays, strings, binary integers, and lists.

delimiter: A character that defines the beginning or end of the display or command line form of an object: ', ", #, [,], {, }, (,), «, or ».

dependent variable: A variable whose value is computed from the values of other (independent) variables, rather than being set arbitrarily. Refers also to the vertical coordinate in plots.

digit: One of the characters 0–9, and, when referring to hexadecimal binary integers, one of the characters A–F.

direct formula entry calculator: A calculator in which you perform numerical calculations by entering a complete formula in ordinary mathematical form, without obtaining intermediate results.

distribute: To apply a function to the arguments of the + operator, before performing the addition: a × (b + c) distributes to (a × b) + (a × c).

domain: The range of values of an argument over which a function is defined.

entry mode: The calculator mode that determines whether keys cause immediate command execution or just enter their command names into the command line. The entry mode can be immediate mode, algebraic mode, or alpha mode.

equation: An algebraic object consisting of two expressions combined by a single equals sign (=).

error: Any execution failure, caused by a mathematical error, argument mismatch, low memory, and so on, that causes normal execution to halt with an error message display.

evaluation: The fundamental calculator operation. (1) Evaluation of a data object returns the data object. (2) Evaluation of a name object returns the object stored in the associated variable and, if this object is a name or program, evaluates it. (3) Evaluation of a procedure object returns each object comprising the procedure and, if an object is a command or unquoted name, evaluates it.

exception: A special type of mathematical error for which you can choose, by means of a user flag, whether the calculator returns a default result or halts with an error message.

execute: To evaluate a procedure object or some portion of a procedure, including HP-28C operations, which are procedure objects stored in ROM.

exponent: The power of 10 included in the exponential notation representation of a floating-point number; specifically, the one-, two-, or three-digit signed number following the "E" in a number display. The exponent of x is IP (LOG (x)).

exponential notation: A representation of a number as a sign, a mantissa between 1 and 9.99999999999, and an exponent "E" followed by a signed three-digit integer.

expression: An algebraic object that contains no equals sign (=).

factor: Either of the arguments of * (multiply).

false: A flag value represented by the real number 0.

fixed-stack calculator: An RPN calculator with a fixed, (usually) four-level stack.

flag: A real number used as an indicator to determine a true/false decision. The number 0 represents *false*; any other number, usually +1, represents *true*.

formal variable: A variable that is named but does not exist, that is, has no value.

function: An HP-28C operation that can be included in the definition of an algebraic object. Various functions may take up to three arguments, but all return one result.

function plot: A plot produced by DRAW, for which the current equation is evaluated at up to 137 values of a specified (independent) variable.

hierarchy: The structure of a mathematical expression, which can be organized into a series of levels of subexpressions, each of which can be the argument of a function.

HMS format: A real number format in which digits to the left of the radix mark represent integer hours (or degrees), the first two digits to the right of the radix represent minutes (arc or time), the next two digits integer seconds, and any remaining digits fractional seconds.

independent variable: A variable whose value can be set arbitrarily rather than being computed from the values of the other variables. In plotting, the horizontal coordinate. In the Solver, a variable that doesn't contain a procedure with names in its definition.

infinite result: A mathematical exception resulting from an operation that would return an infinite result, such as divide by zero.

initial guess: One or more numbers supplied to the root-finder to specify the region in which a root is to be sought.

intercept: The vertical coordinate value at which the straight line determined by a linear regression intersects the vertical (dependent variable) axis.

inverse: (1) The reciprocal of a number or array. (2) A function, which when applied to a second function, returns the argument of the second function. Thus SIN is the inverse of ASIN.

iterative refinement: A process of successive approximations to the solution of systems of equations.

key buffer: A memory location that can hold up to 15 pending key codes, representing keys that have been pressed but not yet processed.

level: (1) A position in the stack, capable of containing one object. (2) The position of a subexpression in an algebraic expression hierarchy.

list: A data object, consisting of a collection of other objects.

local name: A name object that names a local variable. Local names are a different object type (type 7) from ordinary names (type 6). Evaluation of a local name returns the contents of the associated local variable, unevaluated.

local variable: A variable created by the command → or FOR for temporary use within a program structure. The variable is automatically purged when the procedure has completed execution.

machine singular: Describes a numerical value that is too large to be represented by an HP-28C floating-point number.

mantissa: The portion of a number represented by the decimal part of its logarithm. Specifically, the part of the number to the left of the "E" when it is displayed in exponential notation.

matrix: A two-dimensional array.

memory reset: A system clear in which all calculator modes and memory locations are reset to their default contents, including clearing the stack, COMMAND stack, UNDO stack, LAST arguments, and user variable memory.

menu: A collection of operations with common properties that are assigned, six at a time, to the menu keys.

menu keys: The six unlabeled keys in the top row of the right-hand keyboard, the operation of which is determined by the active menu shown in the bottom display line.

menu selection key: Any key that activates a menu of operations that can be executed by pressing menu keys.

message flag: An internal flag that determines whether the normal stack display is shown when all pending execution is complete. The message flag is set by errors and by commands that produce special displays.

mode: A calculator state that affects the behavior of one or more operations other than through the explicit arguments of the operation.

name: An object that consists of a character sequence representing a variable name. (1) Evaluation of a name object returns the object stored in the associated variable and, if this object is a name or program, evaluates it. (2) Evaluation of a local name returns the object stored in the associated local variable.

non-singular: The opposite of *singular*.

number: A complex number or a real number.

numeric mode: A mode in which the evaluation of functions causes repeated evaluation of their arguments until those arguments return numbers.

numeric object. A real or complex number or array.

object: The basic element of calculator operation. Data objects represent quantities that have a simple "value;" name objects serve to name variables that contain other objects; and procedure objects represent sequences of objects and commands.

operation: Any built-in HP-28C capability available to the user, including non-programmable keystrokes and programmable commands.

operator: A function that is subject to special rules of precedence when included in an algebraic expression.

overflow: A mathematical exception resulting from a calculation that returns a result too large to represent with a floating-point number.

parse: To convert a character string to a program consisting of the series of objects defined by the string. Usually applied to the action of ENTER on the command line.

pixel: A single LCD picture element, or dot.

plot parameters: The contents of the list variable PPAR, which determine the position and scaling of a plot and the name of the independent variable.

Polish Notation: A mathematical notation in which all functions and operators are written in prefix form. In Polish Notation, "1 plus 2" is written as "+(1, 2)".

precedence: Rules that determine the order of operator execution in expressions where the omission of parentheses would otherwise make the order ambiguous.

principal value: A particular choice among the multiple values of a mathematical relation or solution, chosen for its uniqueness or simplicity. For example, ASIN (.5) returns 30°, a principal value of the more general result $(-1)^n\ 30° + 180n°$, where n is any integer.

procedure: An object of the class that includes programs and algebraics, where evaluation of the object means to put each object comprising the procedure on the stack and, if the object is a command or an unquoted name, evaluate the object.

program: A procedure object defined with RPN logic, identified by the delimiters « ».

program structure: A set of commands that must follow a specific sequence within a program. Program clauses, delimited by the commands, that comprise logical units for decision making and branching.

quadratic form: A second-order polynomial in a specified variable.

qualifying message: A message displayed by the Solver to provide information about the result returned by the Solver.

radix mark: The punctuation that separates the integer and decimal fraction parts of a number.

real array: An array object that contains only real number elements.

real integer: A real number used as the argument for a command that deals with integer values.

real number: An object consisting of a single real floating-point number, displayed in base 10.

recall: To return the object stored in a variable.

resolution: In a plot, the spacing of the points on the abscissa for which ordinate values are computed. Resolution 1 is every point, 2 is every other point, and so on.

results: Objects returned to the stack by commands.

Reverse Polish Notation: A modification of Polish Notation in which functions follow their arguments: 1 2 + means 1 plus 2. This mathematical notation corresponds to the calculator interface where functions take their arguments from a stack and return results to a stack.

root: A value of a variable for which an expression has the value 0, or an equation is satisfied—both sides of the equation have the same value.

row order: An ordering of the elements of an array, from left to right across each row, with successive rows following from top to bottom.

RPN: Reverse Polish Notation.

scatter plot: A plot of data points from the statistics matrix, produced by DRWΣ.

selected subexpression: The subexpression that is subject to the active menu of FORM operations, identified by the inverse video cursor that highlights the object defining the subexpression.

set: To assign the value *true*, or non-zero, to a flag.

simplification: To rewrite an algebraic expression in a form that preserves the original value of the expression, but appears simpler. Simplification may involve combining terms, or partially evaluating the expression.

single step: To execute one object or structure in a program's definition.

singular: Refers to a mathematical quantity that evaluates to 0 at some point, or has derivatives that are 0, such that it can't be evaluated or inverted without returning an infinite result. A singular matrix has determinant 0, so it can't be inverted.

slope: The slope of the straight line obtained from a linear regression.

solution: Equivalent to *root*.

solve: To find a root of an expression or equation.

solver: The HP-28C system that builds a variables menu from the definition of the current equation, enabling you to store values for the variables and solve the equation for any of the variables.

stack: The series of objects that are presented in a "last-in, first-out" stack, providing a uniform interface for dealing with the arguments and results of commands.

stack diagram: A tabular summary of the arguments and results of a command, showing the nature and position of the arguments and results in the stack.

status message: A message displayed by the calculator to inform you of some calculator status that is not an error condition.

storage arithmetic: Performing arithmetic operations on the contents of variables, without recalling the contents to the stack.

string: An object containing a sequence of characters (letters, numbers and other symbols), delimited by " marks.

subexpression: A portion of an algebraic expression consisting of a number, name, or function and its arguments. Any subexpression can contain other subexpressions as arguments, and can itself be an argument to another subexpression.

summand: Either of the arguments of + (addition).

suspended program: A program for which execution has been stopped by HALT, and which may be resumed by SST or ■ CONT.

symbolic: Representing a value by name or symbol rather than with an explicit numerical value.

symbolic constant: Any of the five objects e, i, π, MAXR, and MINR, which either evaluates to its numerical value or retains its symbolic form according to the states of flags 35 and 36.

symbolic mode: The calculator mode in which functions of symbolic arguments return symbolic results.

system halt: An initialization in which all pending operations are stopped and the stack is cleared.

test: To make a program branch decision based upon the value of a flag.

true: A flag value represented by a real number of value other than 0. When a command returns a true flag, it is represented by the number 1.

underflow: A mathematical exception resulting from a calculation that returns a non-zero result too small to represent with a floating-point number.

unit conversion: A multiplication of a real number by a conversion factor determined by the values of two unit strings representing "old" and "new" units for the number.

unit string: A string that represents the physical units associated with a real number value. A unit string can contain unit names, powers, products, and one ratio.

unknown: The variable for which the Solver, ROOT, QUAD, or ISOL attempts to find a numerical or symbolic root.

user flag: A one-bit memory location, the value of which can be set to 0 or 1, and which can be tested. The HP-28C contains 64 user flags, numbered from 1 through 64.

user interface: The procedures, keystrokes, displays, and so on, whereby a user interacts with a calculator.

user memory: The region of memory where user variables are stored.

value: The numerical, symbolic, or logical content of an object. When referring to variables, *value* means the object that is stored in the variable.

variable: A combination of a name object (the variable name) and any other object (the variable value) that are stored in memory together.

variables menu: The menu created by the Solver, where each variable referred to by the current equation is represented by a menu key.

vector: A one-dimensional array.

wordsize: The number of bits to which the results of binary integer commands are truncated.

Operation Index

This index contains basic information and references for all operations in the HP-28C. For each operation this index shows the following:

Name. For operations, the key or menu label associated with the operation. For commands, how the command appears in the command line.

Description. What the operation does.

Type. Where you can use the operation and how its corresponding key acts. This information is given in the following codes.

Code	Description
A	Analytic Function. Can be solved or differentiated.
F	Function. Can be included in algebraic objects or programs.
C	Command. Can be included in programs but not algebraics.
O	Operation. Cannot be included in the command line or in a procedure.
*	The corresponding key or menu key does not perform ENTER in immediate entry mode.
†	The corresponding key or menu key always adds the command name to the command line.

In. How many objects are required on the stack. (This entry is left blank for operations that don't use the stack.)

Out. How many objects are returned to the stack. (This entry is left blank for operations that don't use the stack.)

Where. Where the command is described in this manual.

HP-28C Operation Index

Name	Description	Type	In	Out	Where	
ABORT	Aborts program execution.	C	0	0	PROGRAM CONTROL	246
ABS	Absolute value.	F	1	1	ARRAY COMPLEX REAL	125 165 269
ACOS	Arc cosine.	A	1	1	TRIG	324
ACOSH	Arc hyperbolic cosine.	A	1	1	LOGS	185
AF	Adds fractions.	O*			ALGEBRA (FORM)	89
🟥 ALGEBRA	Selects the ALGEBRA menu.	O*			ALGEBRA	59
ALOG	Antilogarithm (10 to a power).	A	1	1	LOGS	181
AND	Logical or binary AND.	F	2	1	BINARY PROGRAM TEST	138 258
ARG	Argument.	F	1	1	COMPLEX TRIG	165 329
🟥 ARRAY	Selects the ARRAY menu.	O*			ARRAY	106
ARRY→	Replaces an array with its elements as separate stack objects.	C	1	$n+1$	ARRAY	114
ASIN	Arc sine.	A	1	1	TRIG	323
ASINH	Arc hyperbolic sine.	A	1	1	LOGS	184

ASR	Arithmetic shift right.	C	1	1	BINARY	137
ATAN	Arc tangent.	A	1	1	TRIG	325
ATANH	Arc hyperbolic tangent.	A	1	1	LOGS	186
ATTN (ON)	Aborts program execution; clears the command line; exits catalogs, FORM, plot displays.	O*			Basic	53
AXES	Sets intersection of axes.	C	1	0	PLOT	208
A→	Associates to the right.	O*			ALGEBRA (FORM)	82
BEEP	Sounds a beep.	C	2	0	PROGRAM CONTROL	249
BIN	Sets binary base.	C*	0	0	BINARY	133
■ BINARY	Selects the BINARY menu.	O*			BINARY	130
■ BRANCH	Selects the PROGRAM BRANCH menu.	O*			PROGRAM BRANCH	232
B→R	Binary-to-real conversion.	C	1	1	BINARY	135
■ CATALOG	Starts the command catalog.	O*			CATALOG	156
CEIL	Next greater integer.	F	1	1	REAL	271
CENTR	Sets center of plot display.	C	1	0	PLOT	209
CF	Clears a user flag.	C	1	0	PROGRAM TEST	255
CHR	Makes a one-character string.	C	1	1	STRING	317
CHS	Changes the sign of a number in the command line or executes NEG.	O*			Basic	32
CLEAR	Clears the stack.	C	*n*	0	STACK	291

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where
CLLCD	Blanks the display.	C	0	0	PLOT 212 PROGRAM CONTROL 249
CLMF	Clears the system message flag.	C	0	0	PLOT 214 PROGRAM CONTROL 250
CLUSR	Clears all user variables.	C†	0	0	USER 347
CLΣ	Purges the statistics matrix.	C	0	0	STAT 299
■ CMPLX	Selects the COMPLEX menu.	O*			COMPLEX 160
CNRM	Computes a column norm.	C	1	1	ARRAY 126
COLCT	Collects like terms.	C	1	1	ALGEBRA 71
COLCT	Collects like terms in a subexpression.	O*			ALGEBRA (FORM) 80
COLΣ	Selects statistics matrix columns.	C	2	0	PLOT 211 STAT 302
■ COMMAND	Moves an entry from the command stack to the command line.	O			Basic 48
CON	Creates a constant matrix.	C	2	0, 1	ARRAY 121
CONJ	Complex conjugate.	F	1	1	ARRAY 128 COMPLEX 163
■ CONT	Continues a halted program.	O			PROGRAM CONTROL 244

CONVERT	Performs a unit conversion.	C	3	2	UNITS	332
CORR	Correlation coefficient.	C	0	1	STAT	303
COS	Cosine.	A	1	1	TRIG	324
COSH	Hyperbolic cosine.	A	1	1	LOGS	185
COV	Covariance.	C	0	1	STAT	304
CR	Prints a carriage-right.	C	0	0	PRINT	222
CROSS	Cross-product of three-element vectors.	C	2	1	ARRAY	124
■ CTRL	Selects the PROGRAM CONTROL menu.	O*			PROGRAM CONTROL	243
C→R	Complex-to-real conversion.	C	1	2	ARRAY COMPLEX TRIG	127 162 328
■ d/dx	Derivative (∂ function).	F	2	1	Calculus	141
DEC	Sets decimal base.	C*	0	0	BINARY	132
DEG	Sets degrees mode.	C*	0	0	MODE	191
DEL	Deletes character at cursor; digitizes point.	O*	0	0, 2	Basic PLOT	39 203
■ DEL	Deletes character at cursor and all characters to the right.	O*			Basic	39
DEPTH	Counts the objects on the stack.	C	0	1	STACK	294
DET	Determinant of a matrix.	C	1	1	ARRAY	125

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where
DINV	Double inverts.	O*			ALGEBRA (FORM) 86
DISP	Displays an object.	C	2	0	PLOT 213 PROGRAM CONTROL 250 STRING 320
DNEG	Double negates.	O*			ALGEBRA (FORM) 85
DO	Part of DO...UNTIL...END.	C†			PROGRAM BRANCH 233
DOT	Dot product of two vectors.	C	2	1	ARRAY 124
DRAW	Creates a mathematical function plot.	C	0	0	PLOT 206
DRAX	Draws axes.	C	0	0	PLOT 213
DROP	Drops one object from the stack.	C	1	0	STACK 290
DROPN	Drops *n*+1 objects from the stack.	C	*n*+1	0	STACK 294
DROP2	Drops two objects from the stack.	C	2	0	STACK 292
DRWΣ	Creates a statistics scatter plot.	C	0	0	PLOT 212
DUP	Duplicates one object on the stack.	C	1	2	STACK 291
DUPN	Duplicates *n* objects on the stack.	C	*n*+1	2*n*	STACK 294
DUP2	Duplicates two objects on the stack.	C	2	4	STACK 292
D→	Distributes to the right.	O*			ALGEBRA (FORM) 83

D→R	Degrees-to-radians conversion.	F	1	1	TRIG	331
e	Symbolic constant *e*.	F†	0	1	ALGEBRA REAL	70 266
■ EDIT	Copies the object in level 1 to the command line for editing.	O	1	1	Basic	41
EEX	Enters exponent in command line.	O*			Basic	32
ELSE	Begins ELSE clause.	C†			PROGRAM BRANCH	232
END	Ends program structures.	C†	1	0	PROGRAM BRANCH	232 233
ENG	Sets engineering display format.	C	1	0	MODE	191
ENTER	Parses and evaluates the command line or executes DUP.	O			Basic	40
ERRM	Returns the last error message.	C	0	1	PROGRAM CONTROL	251
ERRN	Returns the last error number.	C	0	1	PROGRAM CONTROL	251
EVAL	Evaluates an object.	C	1	0	Basic	42
EXGET	Gets a subexpression.	C	2	1	ALGEBRA	76
EXGET	Gets a subexpression.	O*		2	ALGEBRA (FORM)	80
EXP	Natural exponential.	A	1	1	LOGS	182
EXPAN	Expands an algebraic.	C	1	1	ALGEBRA	72
EXPAN	Expands a subexpression.	O*			ALGEBRA (FORM)	80

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where	
EXPM	Natural exponential minus 1.	A	1	1	LOGS	183
EXPR=	Evaluates the current equation.	O	0	1	SOLVE	278
EXSUB	Substitutes a subexpression.	C	3	1	ALGEBRA	75
E^	Replaces power-of-product with power-of-power.	O*			ALGEBRA (FORM)	89
E()	Replaces power-of-power with power-of-product.	O*			ALGEBRA (FORM)	89
FACT	Factorial or gamma function.	F	1	1	REAL	267
FC?	Tests a user flag.	C	1	1	PROGRAM TEST	256
FC?C	Tests and clears a user flag.	C	1	1	PROGRAM TEST	257
FETCH	Exits CATALOG or UNITS, writes the current command or unit in the command line.	O*			CATALOG UNITS	336 157
FIX	Sets FIX display format.	C	1	0	MODE	189
FLOOR	Next smaller integer.	F	1	1	REAL	271
FOR	Begins definite loop.	C†	2	0	PROGRAM BRANCH	233
FORM	Changes the form of an algebraic.	C	1	1, 3	ALGEBRA ALGEBRA (FORM)	74 77
FP	Fractional part.	F	1	1	REAL	271
FS?	Tests a user flag.	C	1	1	PROGRAM TEST	256

FS?C	Tests and clears a user flag.	C	1	1	PROGRAM TEST	256
GET	Gets an element from an object.	C	2	1	ARRAY LIST	116 176
GETI	Gets an element from an object and increments the index.	C	2	3	ARRAY LIST	118 178
HALT	Suspends program execution.	C†			PROGRAM CONTROL	246
HEX	Sets hexadecimal base.	C*	0	0	BINARY	132
HMS+	Adds in HMS format.	C	2	1	TRIG	331
HMS−	Subtracts in HMS format.	C	2	1	TRIG	331
HMS→	Converts from HMS format.	C	1	1	TRIG	330
i	Symbolic constant *i*.	F†	0	1	ALGEBRA	70
IDN	Creates an identity matrix.	C	1	0, 1	ARRAY	122
IF	Begins IF clause.	C†	0	0	PROGRAM BRANCH	232
IFERR	Begins IF ERROR clause.	C†	0	0	PROGRAM BRANCH	232
IFT	If-then command.	C	2	0	PROGRAM BRANCH	241
IFTE	If-then-else function.	F	3	0	PROGRAM BRANCH	241
IM	Returns the imaginary part of a number or array.	F	1	1	ARRAY COMPLEX	128 162
INDEP	Selects the plot independent variable.	C	1	0	PLOT	206

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where
INS	Switches between insert and replace modes; digitizes point.	O*	0	0, 1	Basic 39 PLOT 203
■ INS	Deletes all characters to the left of the cursor.	O*			Basic 39
INV	Inverse (reciprocal).	A	1	1	Arithmetic 103 ARRAY 113
IP	Integer part.	F	1	1	REAL 270
ISOL	Solves an expression or equation.	C	2	1	ALGEBRA 76 SOLVE 285
KEY	Returns a key string.	C	0	1, 2	PROGRAM CONTROL 247
KILL	Aborts all suspended programs.	C	0	0	PROGRAM CONTROL 247
LAST	Returns last arguments.	C	0	1, 2, 3	Basic 49
LC	Switches between upper-case and lower-case modes.	O*			Basic 33
LEFT=	Evaluates the left side of the current equation.	O	0	1	SOLVE 278
LEVEL	Displays the level of the selected subexpression.	O*			ALGEBRA (FORM) 80
■ LIST	Selects the LIST menu.	O*			LIST 174
LIST→	Moves list elements to the stack.	C	1	$n+1$	LIST 175 STACK 293

LN	Natural logarithm.	A	1	1	LOGS	182
LNP1	Natural logarithm of (*argument*+1).	A	1	1	LOGS	183
LOG	Common (base 10) logarithm.	A	1	1	LOGS	181
■ LOGS	Selects the LOGS menu.	O*			LOGS	181
LR	Computes a linear regression.	C	0	2	STAT	304
L( )	Replaces product-of-log with log-of-power.	O*			ALGEBRA (FORM)	88
L*	Replaces log-of-power with product-of-log.	O*			ALGEBRA (FORM)	88
MANT	Returns the mantissa of a number.	F	1	1	REAL	270
MAX	Returns the maximum of two numbers.	F	2	1	REAL	272
MAXR	Symbolic constant maximum real.	F	0	1	ALGEBRA REAL	70 268
MAXΣ	Finds the maximum coordinate values in the statistics matrix.	C	0	1	STAT	302
MEAN	Computes statistical means.	C	0	1	STAT	300
MEM	Returns available memory.	C	0	1	USER	347
MIN	Returns the minimum of two numbers.	F	2	1	REAL	273
MINR	Symbolic constant minimum real.	F	0	1	ALGEBRA REAL	70 268
MINΣ	Finds the minimum coordinate values in the statistics matrix.	C	0	1	STAT	302

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where	
MOD	Modulo.	F	2	1	REAL	273
■ MODE	Selects the MODE menu.	O*			MODE	187
M→	Merges right factor.	O*			ALGEBRA (FORM)	85
NEG	Negates an argument.	A	1	1	Arithmetic ARRAY COMPLEX REAL	104 129 165 266
NEXT	Ends definite loop.	C†	0	0	PROGRAM BRANCH	233
NEXT	Displays the next row of menu labels.	O*				57
NEXT	Advances to next command or unit in a catalog.	O*			CATALOG UNITS	157 336
NEXT	Advances to next argument option in USAGE.	O*			CATALOG	158
NO	Choose not to purge during `Out of Memory`.	O*			Basic	52
NORM	Disables printer trace mode.	O*			PRINT	221
NOT	Logical or binary NOT.	F	1	1	BINARY PROGRAM TEST	140 260
NUM	Returns character code.	C	1	1	STRING	317
NΣ	Returns the number of data points in the statistics matrix.	C	0	1	STAT	299

OBGET	Extracts an object from an algebraic.	C	2	1	ALGEBRA	76
OBSUB	Substitutes an object into an algebraic.	C	3	1	ALGEBRA	75
OCT	Sets octal base.	C*	0	0	BINARY	132
ON (ATTN)	Turns the calculator on; aborts program execution; clears the command line; exits catalogs, FORM, plot displays.	O*			Basic	53
ON DEL	Cancels system halt or memory reset if pressed before ON is released.	O*			Basic	55
ON INS ▶	(Memory Reset) Stops program execution, clears local and user variables, clears the stack, resets user flags.	O*			Basic	54
ON +	Increases the display contrast.	O*			Basic	53
ON −	Decreases the display contrast.	O*			Basic	53
ON ▲	(System Halt) Stops program execution, clears local variables, clears the stack.	O*			Basic	54
ON ▼	Starts system test.	O*			Basic	55
ON ◀	Starts continuous system test.	O*			Basic	55
■ OFF	Turns the calculator off.	O*			Basic	53
OR	Logical or binary OR.	F	2	1	BINARY PROGRAM TEST	139 258
ORDER	Rearranges the user menu.	C	1	0	USER	346
OVER	Duplicates the object in level 2.	C	2	3	STACK	292

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where
PICK	Duplicates the *n*th object.	C	$n+1$	$n+1$	STACK 293
PIXEL	Turns on a display pixel.	C	1	0	PLOT 213
🟥 PLOT	Selects the PLOT menu.	O*			PLOT 198
PMAX	Sets the upper-right plot coordinates.	C	1	0	PLOT 206
PMIN	Sets the lower-left plot coordinates.	C	1	0	PLOT 205
POS	Finds an substring in a string.	C	2	1	STRING 320
PPAR	Recalls the plot parameters list.	C	0	1	PLOT 208
PREDV	Predicted value.	C	1	1	STAT 305
🟥 PREV	Displays the previous row of menu labels.	O*			57
PREV	Displays the previous command or unit in a catalog.	O*			CATALOG 157 UNITS 336
PREV	Displays the previous argument option in USAGE.	O*			CATALOG 158
🟥 PRINT	Selects the PRINT menu.	O*			PRINT 215
PRLCD	Prints an image of the display.	C	0	0	PLOT 214 PRINT 220
PRMD	Prints and displays current modes.	C	0	0	MODE 196 PRINT 222

PROGRAM						
■ BRANCH	Selects the PROGRAM BRANCH menu.	O*			PROGRAM BRANCH	252
■ CTRL	Selects the PROGRAM CONTROL menu.	O*			PROGRAM CONTROL	232
■ TEST	Selects the PROGRAM TEST menu.	O*			PROGRAM TEST	243
PRST	Prints the stack.	C	0	0	PRINT	219
PRSTC	Prints the stack in compact format.	C	0	0	PRINT	221
PRUSR	Prints a list of variables.	C	0	0	PRINT	222
PRVAR	Prints the contents of a variable.	C	1	0	PRINT	220
PR1	Prints the level 1 object.	C	0	0	PRINT	218
PURGE	Purges one or more variables.	C	1	0	Basic	47
PUT	Put an element into an array or list.	C	3	0, 1	ARRAY LIST	115 175
PUTI	Put an element into an array or list, and increment the index.	C	3	2	ARRAY LIST	117 177
P→R	Polar-to-rectangular conversion.	F	1	1	COMPLEX TRIG	164 326
QUAD	Solves a quadratic polynomial.	C	2	1	ALGEBRA SOLVE	76 286
QUIT	Exits CATALOG or UNITS.	O*			CATALOG UNITS	336 157
QUIT	Exits USAGE display.	O*			CATALOG	159

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where	
RAD	Sets radians mode.	C*	0	0	MODE	192
RAND	Returns a random number.	C	0	1	REAL	267
RCEQ	Recalls the current equation.	C	0	1	PLOT SOLVE	205 277
RCL	Recalls the contents of a variable, unevaluated.	C	1	1	Basic	46
RCLF	Returns a binary integer representing the user flags.	C	0	1	PROGRAM TEST	262
RCLΣ	Recalls the current statistics matrix.	C	0	1	PLOT STAT	210 299
RCWS	Recalls the binary integer wordsize.	C	0	1	BINARY	133
RDM	Redimensions an array.	C	2	0, 1	ARRAY	120
RDX.	Sets "." as the radix.	O*			MODE	196
RDX,	Sets "," as the radix.	O*			MODE	196
RDZ	Sets the random number seed.	C	1	0	REAL	268
RE	Returns the real part of a complex number or array.	F	1	1	ARRAY COMPLEX	127 162
■ REAL	Selects the REAL menu.	O*			REAL	264
REPEAT	Part of WHILE...REPEAT...END.	C†	1	0	PROGRAM BRANCH	233

RES	Sets the plot resolution.	C	1	0	PLOT	208
RL	Rotates left by one bit.	C	1	1	BINARY	134
RLB	Rotates left by one byte.	C	1	1	BINARY	134
RND	Rounds according to real number display mode.	F	1	1	REAL	272
RNRM	Computes the row norm of an array.	C	1	1	ARRAY	125
ROLL	Moves the level *n*+1 object to level 1.	C	*n*+1	*n*	STACK	291
ROLLD	Moves the level 2 object to level *n*.	C	*n*+1	*n*	STACK	293
ROOT	Finds a numerical root.	C	3	1, 3	SOLVE	284
ROT	Moves the level 3 object to level 1.	C	3	3	STACK	292
RR	Rotates right by one bit.	C	1	1	BINARY	134
RRB	Rotates right by one byte.	C	1	1	BINARY	135
RSD	Computes a correction to the solution of a system of equations.	C	3	1	ARRAY	123
RT=	Evaluates the right side of the current equation.	O	0	1	SOLVE	278
R→B	Real-to-binary conversion.	C	1	1	BINARY	135
R→C	Real-to-complex conversion.	C	2	1	ARRAY COMPLEX TRIG	126 161 328
R→D	Radians-to-degrees conversion.	F	1	1	TRIG	331

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where	
R→P	Rectangular-to-polar conversion.	F	1	1	COMPLEX TRIG	164 327
SAME	Tests two objects for equality.	C	2	1	PROGRAM TEST	260
SCAN	Advances automatically through CATALOG or UNITS.	O*			CATALOG UNITS	336 157
SCI	Sets scientific display format.	C	1	0	MODE	190
SCLΣ	Auto-scales the plot parameters according to the statistical data.	C	0	0	PLOT	211
SCONJ	Conjugates the contents of a variable.	C	1	0	STORE	313
SDEV	Computes standard deviations.	C	0	1	STAT	301
SF	Sets a user flag.	C	1	0	PROGRAM TEST	255
SHOW	Resolves all references to a name implicit in an algebraic.	C	2	1	ALGEBRA SOLVE	76 286
SIGN	Sign of a number.	F	1	1	COMPLEX REAL	163 269
SIN	Sine.	A	1	1	TRIG	323
SINH	Hyperbolic sine.	A	1	1	LOGS	184
SINV	Inverts the contents of a variable.	C	1	0	STORE	312

SIZE	Finds the dimensions of a list, array, string, or algebraic.	C	1	1	ALGEBRA 74 ARRAY 119 LIST 180 STRING 321
SL	Shifts left by one bit.	C	1	1	BINARY 136
SLB	Shifts left by one byte.	C	1	1	BINARY 137
SNEG	Negates the contents of variable.	C	1	0	STORE 312
SOLV	Selects the SOLVE menu.	O*			SOLVE 275
SOLVR	Selects the Solver variables menu.	O			SOLVE 278
SQ	Squares a number or matrix.	A	1	1	Arithmetic 104 ARRAY 113
SR	Shifts right by one bit.	C	1	1	BINARY 136
SRB	Shifts right by one byte.	C	1	1	BINARY 137
SST	Single-steps a suspended program.	O			PROGRAM CONTROL 245
■ STACK	Selects the STACK menu.	O*			STACK 290
START	Begins definite loop.	C†	2	0	PROGRAM BRANCH 233
■ STAT	Selects the STAT menu.	O*			STAT 296
STD	Sets standard display format.	C*			MODE 188
STEP	Ends definite loop.	C†	1	0	PROGRAM BRANCH 233
STEQ	Stores the current equation.	C	1	0	PLOT 205 SOLVE 277

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where
STO	Stores an object in a variable.	C	2	0	Basic 46
STOF	Sets all user flags according to the value of a binary integer.	C	1	0	PROGRAM TEST 262
STOP	Stops scanning through CATALOG or UNITS.	O*			CATALOG 336 UNITS 157
■ STORE	Selects the STORE menu.	O*			STORE 309
STO∗	Storage arithmetic multiply.	C	2	0	STORE 311
STO+	Storage arithmetic add.	C	2	0	STORE 309
STO−	Storage arithmetic subtract.	C	2	0	STORE 310
STO/	Storage arithmetic divide.	C	2	0	STORE 311
STOΣ	Stores the current statistics matrix.	O*	1	0	PLOT 210 STAT 299
■ STRING	Selects the STRING menu.	O*			STRING 314
STR→	Parses and evaluates the commands defined by a string.	C	1	0	STRING 315
STWS	Sets the binary integer wordsize.	C	1	0	BINARY 133
SUB	Extracts a portion of a list or string.	C	3	1	LIST 180 STRING 321

SWAP	Swaps the objects in levels 1 and 2.	C	2	2	STACK	290
SYSEVAL	Executes a system object.	C	1	0	Basic	43
TAN	Tangent.	A	1	1	TRIG	325
TANH	Hyperbolic tangent.	A	1	1	LOGS	185
TAYLR	Computes a Taylor series approximation.	C	3	1	ALGEBRA Calculus	76 151
🟥 TEST	Selects the PROGRAM TEST menu.	O*			PROGRAM TEST	252
THEN	Begins THEN clause.	C†	1	0	PROGRAM BRANCH	232
TOT	Sums the coordinate values in the statistics matrix.	C	0	1	STAT	300
TRACE	Enables printer trace mode.	O*			PRINT	221
TRIG	Selects the TRIG menu.	O*			TRIG	322
TRN	Transposes a matrix.	C	1	0, 1	ARRAY	121
TYPE	Returns the type of an object.	C	1	1	PROGRAM TEST	263
🟥 UNDO	Replaces the stack contents.	O*			Basic	48
🟥 UNITS	Selects the units catalog.	O*			UNITS	332
UNTIL	Part of BEGIN...UNTIL...END.	C†			PROGRAM BRANCH	233
USE	Displays USAGE for current command in CATALOG.	O*			CATALOG	157
USER	Selects the USER menu.	O*			USER	346

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where
UTPC	Upper-tail Chi-Square distribution .	C	2	1	STAT 307
UTPF	Upper-tail F-distribution.	C	3	1	STAT 307
UTPN	Upper-tail normal distribution.	C	3	1	STAT 307
UTPT	Upper-tail t-distribution.	C	2	1	STAT 308
VAR	Computes statisical variances.	C	0	1	STAT 301
■ VIEW↑	Moves the display window up one line.	O*			Basic 40
■ VIEW↓	Moves the display window down one line.	O*			Basic 40
■ VISIT	Copies an object to the command line for editing.	O	1	0	Basic 42
WAIT	Pauses program execution.	C	1	0	PROGRAM CONTROL 247
WHILE	Begins WHILE...REPEAT...END.	C†	0	0	PROGRAM BRANCH 233
XOR	Logical or binary XOR.	F	2	1	BINARY 139 PROGRAM TEST 259
XPON	Returns the exponent of a number.	F	1	1	REAL 270
■ x^2	Executes function SQ.	A	1	1	Arithmetic 104 ARRAY 113
YES	Chooses to purge during Out of Memory.	O*			Basic 52

🟥 1/x	Executes function INV.	A	1	1	Arithmetic ARRAY	103 113
1/()	Double invert and distribute.	O*			ALGEBRA (FORM)	87
+	Adds two objects.	A	2	1	Arithmetic ARRAY LIST STRING	96 108 174 315
+CMD	Enables COMMAND.	O*			MODE	193
+LAST	Enables LAST.	O*			MODE	193
+ML	Selects multi-line display mode.	O*			MODE	195
+UND	Enables UNDO.	O*			MODE	194
+1-1	Adds and subtracts 1.	O*			ALGEBRA (FORM)	88
−	Subtracts two objects.	A	2	1	Arithmetic ARRAY	98 108
-CMD	Disables COMMAND.	O*			MODE	193
-LAST	Disables LAST.	O*			MODE	193
-ML	Selects single-line display mode.	O*			MODE	195
-UND	Disables UNDO.	O*			MODE	194
-()	Double negates and distributes.	O*			ALGEBRA (FORM)	86
*	Multiplies two objects.	A	2	1	Arithmetic ARRAY	99 109

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where	
*H	Adjusts the height of a plot.	C	1	0	PLOT	210
*W	Adjusts the width of a plot.	C	1	0	PLOT	209
*1	Multiplies by 1.	O*			ALGEBRA (FORM)	87
/	Divides two objects.	A	2	1	Arithmetic ARRAY	101 109
/1	Divides by 1.	O*			ALGEBRA (FORM)	87
%	Percent.	F	2	1	REAL	265
%CH	Percent change.	F	2	1	REAL	265
%T	Percent of total.	F	2	1	REAL	174
^	Raises a number to a power.	A	2	1	Arithmetic	102
^1	Raises to the power 1.	O*			ALGEBRA (FORM)	88
√	Takes the square root.	A	1	1	Arithmetic	103
∫	Definite or indefinite integral.	C	3	1, 2	Calculus	145
∂	Derivative.	F	2	1	Calculus	141
<	Less-than comparison.	F†	2	1	PROGRAM TEST	253
≤	Less-than-or-equal comparison.	F†	2	1	PROGRAM TEST	254
=	Equals operator.	A†	2	1	ALGEBRA	64

==	Equality comparison.	F	2	1	PROGRAM TEST	261
≠	Not-equal comparison.	F†	2	1	PROGRAM TEST	252
≥	Greater-than-or-equal comparison.	F†	2	1	PROGRAM TEST	254
>	Greater-than comparison.	F†	2	1	PROGRAM TEST	253
🟥	Shift key.	O*				
[◂◆▸]	Selects cursor menu or restores last menu.	O*			Basic	38
[▲]	Moves cursor up.	O*			Basic	39
🟥 [▲]	Moves cursor up all the way.	O*			Basic	39
[▼]	Moves cursor down.	O*			Basic	39
🟥 [▼]	Moves cursor down all the way.	O*			Basic	39
[◀]	Moves cursor left.	O*			Basic	39
🟥 [◀]	Moves cursor left all the way.	O*			Basic	39
[▶]	Moves cursor right.	O*			Basic	39
🟥 [▶]	Moves cursor right all the way.	O*			Basic	39
[←]	Moves FORM cursor left.	O*			ALGEBRA (FORM)	80
[→]	Moves FORM cursor right.	O*			ALGEBRA (FORM)	80
[⬅]	Backspace.	O*			Basic	33
[α]	Switches alpha mode on or off.	O*			Basic	37
🟥 [αLOCK]	Locks alpha mode on.	O*			Basic	37

HP-28C Operation Index (Continued)

Name	Description	Type	In	Out	Where	
π	Symbolic constant π.	F†	0	1	ALGEBRA REAL	70 266
Σ+	Adds a data point to the statistics matrix.	C	1	0	STAT	297
Σ−	Deletes the last data point from the statistics matrix.	C	0	1	STAT	298
←A	Associates to the left.	O*			ALGEBRA (FORM)	81
←D	Distributes to the left.	O*			ALGEBRA (FORM)	83
←M	Merges left factors.	O*			ALGEBRA (FORM)	84
←→	Commutes arguments.	O*			ALGEBRA (FORM)	81
→	Creates local variables.	C†	*n*	0	Programs	228
→ARRY	Combines numbers into an array.	C	*n*+1	1	ARRAY	114
→HMS	Converts a number to HMS format.	C	1	1	TRIG	330
→LIST	Combines objects into a list.	C	*n*+1	1	LIST STACK	174 295
→NUM	Evaluates an object in numerical mode.	C	1	0	Basic	43
→STR	Converts an object to a string.	C	1	1	STRING	315
→()	Distributes prefix operator.	O*			ALGEBRA (FORM)	82

Terms Used in Stack Diagrams

Term	Description
obj	Any object.
x or *y*	Real number.
hms	Real number in hours-minutes-seconds format.
n	Positive integer real number.
flag	Real number, zero (false) or non-zero (true).
z	Real or complex number.
(*x*, *y*)	Complex number in rectangular form.
(*r*, *θ*)	Complex number in polar form.
# *n*	Binary integer.
"*string*"	Character string.
[*array*]	Real or complex vector or matrix.
[*vector*]	Real or complex vector.
[*matrix*]	Real or complex matrix.
[*R-array*]	Real vector or matrix.
[*C-array*]	Complex vector or matrix.
{ *list* }	List of objects.
{ *index* }	List of one or two real numbers specifying an array element.
{ *dim* }	List of one or two real numbers specifying the dimension(s) of an array.
'*name*'	Name or local name.
«*program*»	Program.
'*symb*'	Expression, equation, or a name treated as an algebraic.

Contents

HEWLETT
PACKARD

Reorder Number
00028-90021

00028-90051
Printed in U.S.A. 1/87