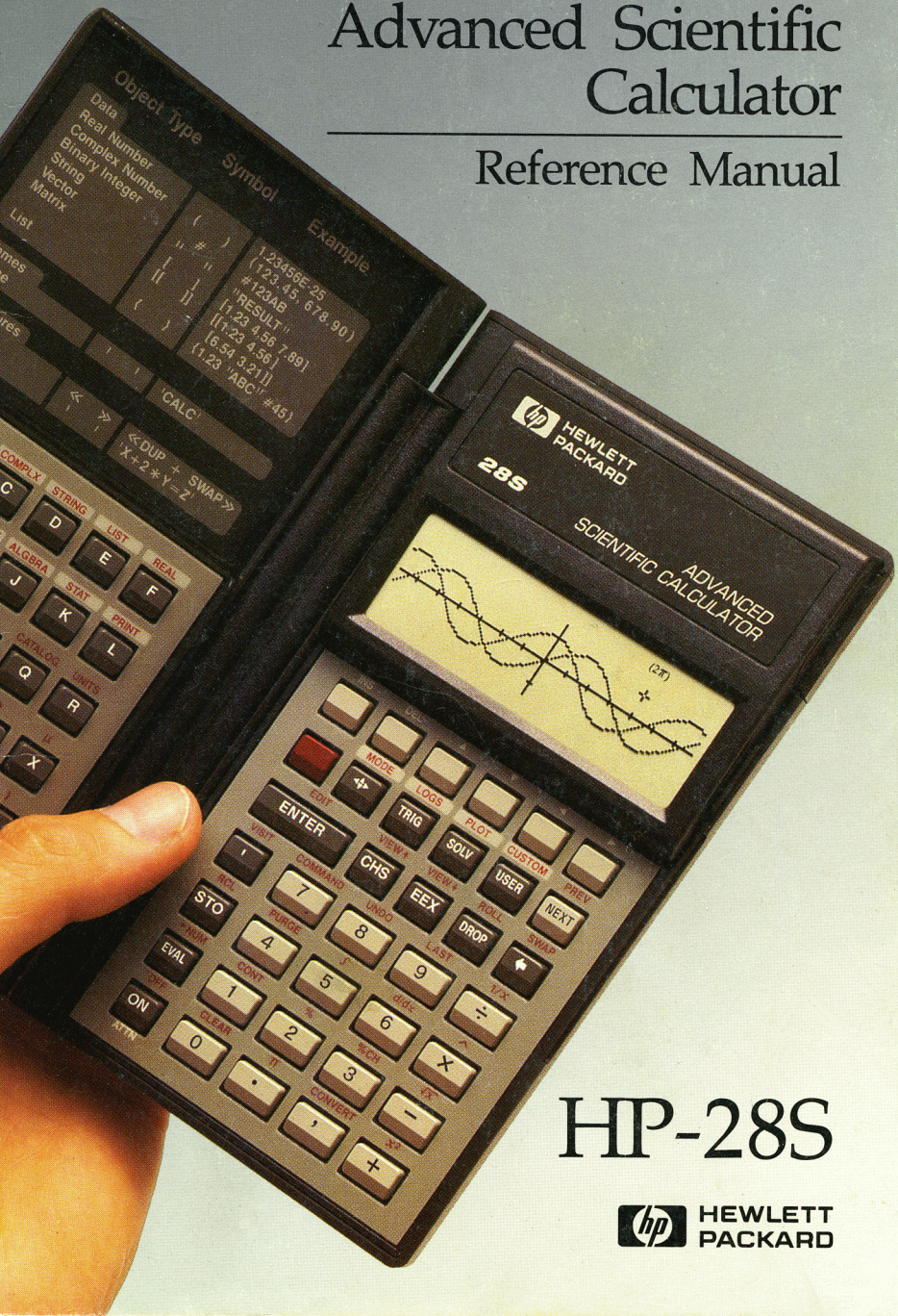
HEWLETT-PACKARD

Advanced Scientific Calculator

Reference Manual



HP-28S
Advanced Scientific Calculator

Reference Manual

HEWLETT PACKARD

Edition 4 November 1988
Reorder Number 00028-90068

Notice

This manual and any keystroke programs contained herein are provided **"as is"** and are subject to change without notice. **Hewlett-Packard Company makes no warranty of any kind with regard to this manual or the keystroke programs contained herein, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.** Hewlett-Packard Co. shall not be liable for any errors or for incidental or consequential damages in connection with the furnishing, performance, or use of this manual or the keystroke programs contained herein.

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

Printing History

Edition 1	October 1987	Mfg. No. 00028-90069
Edition 2	April 1988	Mfg. No. 00028-90129
Edition 3	June 1988	Mfg. No. 00028-90131
Edition 4	November 1988	Mfg. No. 00028-90148

Welcome to the HP-28S

Congratulations! With the HP-28S you can easily solve complicated problems, including problems you couldn't solve on a calculator before. The HP-28S 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-28S 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 HP Solve on the HP-28S, 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-28S Owner's Manual* contains three parts. Part 1, "Fundamentals," demonstrates how to work some simple problems. Part 2, "Summary of Calculator Features," builds on part 1 to help you apply those examples to your own problems. Part 3, "Programming," describes programming features and demonstrates them in a series of programming examples.

The *HP-28S Reference Manual* (this manual) gives detailed information about commands. It is a dictionary of menus, describing the concepts and commands for each menu.

We recommend that you first work through the examples in part 1 of the Owner's Manual to get comfortable with the calculator, and then look at part 2 to gain a broader understanding of the calculator's operation. When you want to know more about a particular command, look it up in the Reference Manual. When you want you learn about programming, read part 3 of the Owner's Manual.

These manuals show you how to use the HP-28S 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-28S effectively, you should know elementary calculus.

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

Contents

How To Use This Manual

This manual contains general information about how the HP-28S 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 323). 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.	The Dictionary (page 15). All menus are listed alphabetically.
What a displayed message means.	Appendix A, "Messages" (page 298).
What an unfamiliar term means.	The Glossary (page 310).

How This Manual is Organized

The 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.)

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

Appendix B, "User Flags," describes the choices and default setting for user flags 31 through 64.

The Glossary defines terms used in this manual.

The Operation Index is an alphabetical listing of all operations, commands, and functions in the HP-28S. 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 (rounded if non-integer).
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	Real number specifying an element in a list or array; or list with one real number (or object that evaluates to a number) specifying an element in a list or vector; or list with two real numbers (or objects that evaluates to numbers) specifying an element in a matrix.
{ *dim* }	List of one or two real numbers specifying the dimension(s) of an array.
' *name* '	Global name or local name.
' global '	Global name.
' local '	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
$\square array_1 \square$	$\square array_2 \square$	➧	$\square array_1+array_2 \square$
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 $\square array_1 \square$ and $\square array_2 \square$ returns a third array $\square array_1+array_2 \square$.
- Adding a real or complex number z and a symbolic object '*symb*' returns a symbolic object '$z+(symb)$'.

Dictionary

The Dictionary is organized around the menus in the HP-28S. It also includes additional topics not related to specific menus:

- Arithmetic
- Calculus
- Evaluation
- Programs
- UNITS

Not included are menus that don't contain a fixed set of commands:

- Cursor menu
- Custom menu
- USER menu
- Catalog of 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-28S 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.

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-28S 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.

...ALGEBRA

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*):

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

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-28S, = 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.

...ALGEBRA

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-28S, 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-28S is in Symbolic Result 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-28S is in Numerical Result mode (flag 36 clear), the result is the numerical difference of the two arguments. In effect, = acts as the − operator in Numerical Result mode.

Functions of Symbolic Arguments

Result Mode

Symbolic Result Mode (flag 36 set). In Symbolic Result mode, functions return symbolic results if their arguments are symbolic. This is the default 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)'.
```

Numerical Result Mode (flag 36 clear). In Numerical Result 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 evalulated, 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 Result mode (flag 36 set) or Numerical Result mode (flag 36 clear).

† Applies only when the angle mode is radians.

...ALGEBRA

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-28S 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 Result mode (flag 36) is set, so that functions will accept symbolic arguments.

...ALGEBRA

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 '`7+(3+T)`'. Further evaluation makes no additional changes, since T has no value.

The values 7 and 3 obtained are not arguments to the same + operator in the expression, and hence are not combined. If you want to combine the 7 and the 3, 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 Constants mode (flag 35) as well as by the Results mode (flag 36).

- If flag 35 or flag 36 is clear, these objects will evaluate to their numeric values. For example:

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

- If flag 35 and flag 36 are both 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-28S Symbolic Constants

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

...ALGEBRA

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-28S.

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'`.

...ALGEBRA

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)."

...ALGEBRA

OBSUB — *Object Substitute* — Command

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

OBSUB substitutes a number, name, or function 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'.
```

TAYLR ISOL QUAD SHOW OBGET EXGET

TAYLR is described in "Calculus," along with ∂ and $\int$. 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.

ALGEBRA (FORM)

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.

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.

...ALGEBRA (FORM)

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.

...ALGEBRA (FORM)

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.

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.

...ALGEBRA (FORM)

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))

(*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)

...ALGEBRA (FORM)

`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))

`-()` **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))

...ALGEBRA (FORM)

`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`

`^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→ .

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)

...ALGEBRA (FORM)

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))

(*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)

...ALGEBRA (FORM)

(*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)
	-(A*B)	(-(A)*B)
	-(A/B)	(-(A)/B)
	-(LOG(A))	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)

...ALGEBRA (FORM)

Antilogarithm (ALOG)

Operation	Before	After
D→	ALOG(A+B) ALOG(A-B)	(ALOG(A)*ALOG(B)) (ALOG(A)/ALOG(B))
1/()	ALOG(A) ALOG(-(A))	INV(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/B)	(LN(A)+LN(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-B)	(EXP(A)*EXP(B)) (EXP(A)/EXP(B))
1/()	EXP(A) EXP(-(A))	INV(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	' $symb$ '	➡	' $z+\langle symb \rangle$ '
' $symb$ '	z	➡	' $symb+z$ '
' $symb_1$ '	' $symb_2$ '	➡	' $symb_1+\langle symb_2 \rangle$ '
$\langle list_1 \rangle$	$\langle list_2 \rangle$	➡	$\langle list_1 list_2 \rangle$
$\langle list \rangle$	obj	➡	$\langle list\ obj \rangle$
obj	$\langle list \rangle$	➡	$\langle obj\ list \rangle$
" $string_1$ "	" $string_2$ "	➡	" $string_1 string_2$ "
# 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.

...Arithmetic

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.

A list and a non-list object. The result is a list obtained by treating the non-list object as a one-element list and concatenating the two lists.

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.

...Arithmetic

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	➧	# $n_1 n_2$

* 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.

...Arithmetic

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*$^{-1}$ ⟧
⟦ *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

/ ([÷]) 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 inverse of the matrix in level 1 with the array in level 2. 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.

An array and a number. The result is a new array, with each new element the quotient of the corresponding old element and the number.

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.

...Arithmetic

^ *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*)'

√ (■ √x) 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

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

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 (■ x²) 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.

...Arithmetic

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.

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-28S, 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-28S 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.

...ARRAY

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-28S 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 $(x, 0)$.

...ARRAY

* *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**]

/ (⌹) applied to array arguments solves the system of equations $\mathbf{AX} = \mathbf{B}$ for **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**.

If flag 59 (Infinite Result) is clear, the HP-28S will arrive at a 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.

...ARRAY

Improving the Accuracy of System Solutions

Because of rounding errors during calculation, a numerically calculated solution **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 $\|\Delta\mathbf{A}\| \leq \epsilon\|\mathbf{A}\|$ and $\|\Delta\mathbf{B}\| \leq \epsilon\|\mathbf{B}\|$, where ϵ is a small number and $\|\mathbf{A}\|$ is the *norm* of **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 **A** and **B**.

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

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

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

where n is the dimension of **A**. For the HP-28S, which carries 12 accurate digits,

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

In many applications, this accuracy may be adequate. When additional accuracy is desired, the computed solution **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 **Z** to the original system **AX** = **B**. Then **Z** is treated as an approximation to **X**, in error by **E** = **X** − **Z**. Then **E** satisifies the linear system

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

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

For **F** + **Z** to be a better approximation to **X** than is **Z**, the residual **R** = **B** − **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 **B** and **A** from the stack, the same as /, and returns the result array **Z**, which will be an improved approximation to the solution **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.

...ARRAY

SQ — *Square* — Analytic

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

SQ ([x²]) 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).

→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
	$x_1 \ldots x_n$	n	→	[vector]
	$x_1 \ldots x_n$	$\{n\}$	→	[vector]
	$x_{11} \ldots x_{nm}$	$\{n\ m\}$	→	[matrix]

→ ARRY takes a list (or, for vectors, a number) representing the size of the result array from level 1:

Vectors. If level 1 contains an integer n or a list consisting of 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]	➡	$x_1 \ldots x_n$	$\{n\}$
[matrix]	➡	$x_{11} \ldots x_{nm}$	$\{n\ m\}$

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*$_2$ ⌋
' *global* '	*index*	*x*	➡	
⌈ *C-array*$_1$ ⌋	*index*	*z*	➡	⌈ *C-array*$_2$ ⌋
' *global* '	*index*	*z*	➡	
{ *list*$_1$ }	*index*	*obj*	➡	{ *list*$_2$ }
' *global* '	*index*	*obj*	➡	

PUT replaces an object in the specified position in an array or list. This section describes its use with arrays; see "LIST" for its use with lists.

PUT takes three arguments from the stack:

- From level 3, an array or the name of an array.
- From level 2, a one-element list (specifying position in a vector), a two-element list (specifying row and column in a matrix), or a real number (specifying an element in row order in a vector or a matrix).
- From level 1, the number to be put in the array. If this number is complex, the array must also be complex.

If the argument in level 3 is an array, PUT returns the altered array to the stack. If the argument in level 3 is a name, PUT alters the array variable and returns nothing to the stack.

GET — *Get Element* — Command

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

GET gets an object from the specified position in an array or list. This section describes its use with arrays; see "LIST" for its use with lists.

GET takes two arguments from the stack:

- From level 2, an array or the name of an array.
- From level 1, a one-element list (specifying position in a vector), a two-element list (specifying row and column in a matrix), or a real number (specifying an element in row order in a vector or a matrix).

GET returns the specified object to the stack.

PUTI — *Put and Increment Index* — Command

Level 3	Level 2	Level 1		Level 2	Level 1
[$array_1$]	$index_1$	*x*	➡	[$array_2$]	$index_2$
' *global* '	$index_1$	*x*	➡	' *global* '	$index_2$
[*C-*$array_1$]	$index_1$	*z*	➡	[*C-*$array_2$]	$index_2$
' *global* '	$index_1$	*z*	➡	' *global* '	$index_2$
{ $list_1$ }	$index_1$	*obj*	➡	{ $list_2$ }	$index_2$
' *global* '	$index_1$	*obj*	➡	' *global* '	$index_2$

PUTI replaces an object in the specified position in an array, returning the array (or name) and the next position. You can then put an object in the next position simply by putting the object on the stack and executing PUTI again.

You can specify the position by a one-element list (specifying position in a vector), by a two-element list (specifying row and column in a matrix), or by a real number (specifying an element in row order in a vector or a matrix).

Generally, after putting an object in position *n* (in row order), PUTI returns *n* + 1 as the next position and clears flag 46. However, when *n* is the last position in the list, PUTI returns 1 as the next position and sets flag 46. (If you're using lists rather than row-order numbers to specify position, the next position is { 1 } or { 1, 1 }.)

The following example uses PUTI and flag 46 to put the contents of a variable X in an array, from the initially specified position (not shown) to the last position.

```
...DO X PUTI UNTIL 46 FS? END...
```

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* ⟩	$index_1$	➡	⟨ *list* ⟩	$index_2$	*obj*
' *name* '	$index_1$	➡	' *name* '	$index_2$	*obj*

GETI gets an object from the specified position in an array, also returning the array (or name) and the next position. You can then get the object in the next position simply by removing the object from level 1 and executing GETI again.

Generally, after getting an object from position *n* (in row order), GETI returns *n* + 1 as the next position and clears flag 46. However, when *n* is the last position in the list, GETI returns 1 as the next position and sets flag 46. (If you're using lists rather than row-order numbers to specify position, the next position is { 1 } or { 1, 1 }.)

The following example uses GETI and flag 46 to add array elements, from the initially specified position (not shown) to the last position, to a variable X.

```
...DO GETI 'X' STO+ UNTIL 46 FS? END...
```

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.

...ARRAY

RDM *Redimension* Command

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

TRN returns the (conjugate) transpose of its argument. That is, an $n \times m$ matrix **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 **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-*$array_2$ ⌋
' *global* '	*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.

...ARRAY

IDN *Identity Matrix* Command

Level 1		Level 1
n	➧	[*R-identity matrix*]
[*matrix*]	➧	[*identity matrix*]
' *global* '	➧	

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 **C** = **A** × **B** of the vectors ⊏a_1 a_2 a_3⊐ and ⊏b_1 b_2 b_3⊐, where

$$\begin{aligned} c_1 &= a_2b_3 - a_3b_2 \\ c_2 &= a_3b_1 - a_1b_3 \\ c_3 &= a_1b_2 - a_2b_1 \end{aligned}$$

The arguments must be two- or three-element vectors. A two-element argument [d_1 d_2] is converted to a three-element argument [d_1 d_2 0].

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 $1 \times 4 + 2 \times 5 + 3 \times 6$, 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-28S 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	➧	|*z*|
⌈*array*⌋	➧	‖*array*‖
'*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.

...ARRAY

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.

...ARRAY

CONJ — *Conjugate* — Analytic

Level 1		Level 1
x	➡	*x*
(*x*,*y*)	➡	(*x*,−*y*)
[*R-array*]	➡	[*R-array*]
[*C-array*$_1$]	➡	[*C-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-28S 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 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). Binary integers are displayed with a *base marker* b, o, d, or h, indicating the current base. 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.

You can enter a binary integer in any base if you also enter the base marker; you can enter one in the current base by omitting the base marker.

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.

...BINARY

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.

...BINARY

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 $n - 1$.

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.

...BINARY

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 (hexadecimal) (truncated to the current wordsize).

...BINARY

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 functions AND, OR, XOR, and NOT can be applied to binary integers, strings, or flags (real numbers or algebraics). This section describes their use with binary integers and strings; see "PROGRAM TEST" for their use with flags.

These functions treat binary integers and strings as sequences of bits (0's and 1's).

- A binary integers is treated as a sequence of length n, where n is the current wordsize. The bits correspond to the 0's and 1's in the binary integer's representation in base 2.

- A string is treated as a sequence of length $8n$, where n is the number of characters in the string. Each set of eight bits corresponds to the binary representation of one character code. For AND, OR, and XOR, the two string arguments must be the same length.

AND *And* Function

Level 2	Level 1		Level 1
# n_1	# n_2	➧	# n_3
"$string_1$"	"$string_2$"	➧	"$string_3$"

AND returns the logical AND of two arguments. Each bit in the result is determined by the corresponding bits (bit_1 and bit_2) in the two arguments, according to the following table:

bit_1	bit_2	bit_1 AND bit_2
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
"$string_1$"	"$string_2$"	➧	"$string_3$"

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

bit_1	bit_2	bit_1 OR bit_2
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
"$string_1$"	"$string_2$"	➧	"$string_3$"

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

bit_1	bit_2	bit_1 XOR bit_2
0	0	0
0	1	1
1	0	1
1	1	0

NOT *Not* Function

Level 1		Level 1
# n_1	➧	# n_2
$string_1$	➧	$string_2$

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

bit	NOT *bit*
0	1
1	0

Calculus

The HP-28S 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
'$symb_1$'	'*global*'	➡	'$symb_2$'

∂ ([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-28S 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-28S 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))'
```

...Calculus

Integration

∫ **Integrate** **Command**

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

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* (global 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.

...Calculus

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 global name, and where each limit is a real number or an object that evaluates to a number.

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,

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

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.

...Calculus

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 ∫:

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

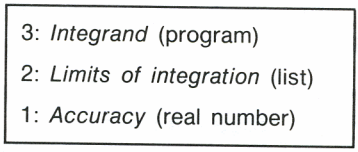
Numerical integration returns two numbers to the stack. The value of the integral is returned to level 2. The error returned to level 1 is an upper limit to the fractional error of the computation, where normally

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

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, ∫ 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:



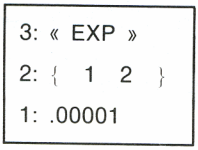
The *integrand* must be a program that takes one real number from the stack, and returns one real number. ∫ evaluates the program at each of the sample points between the limits of integration. For each evaluation ∫ places the sample value on the stack. The program takes that value, and returns the value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



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
' $symb_1$ '	' *global* '	*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:

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

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^.5' '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.

...Calculus

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-28S. 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.

COMPLEX

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

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

Complex number objects in the HP-28S 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-28S 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 82.

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-array*$_1$]	➧	[*C-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):

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

...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

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

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

$$\theta = \begin{cases} \text{arc tan } y/x & \text{for } x \geqslant 0, \\ \text{arc tan } y/x + \pi \text{ sign } y & \text{for } x < 0, \text{ radians mode,} \\ \text{arc tan } y/x + 180 \text{ sign } y & \text{for } x < 0, \text{ degrees mode.} \end{cases}$$

The current angle mode determines whether θ is expressed as degrees or radians.

...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-28S 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-28S 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 $\text{ASIN}(-z) = -\text{ASIN}(z)$.

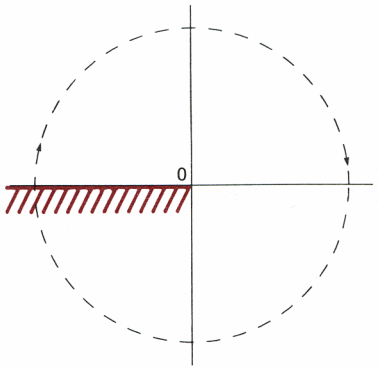
The illustrations below show the principal branches for √, LN, ASIN, ACOS, ATAN, ACOSH. The graphs of the domains show where the cuts occur: the solid color or black lines are on one side of the cut, and the shaded color or black regions are on the other side. The graphs of the principal branches show 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.

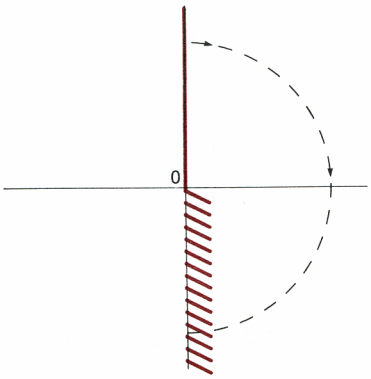
The functions LOG, ^, ASINH, and ATANH are closely related to the illustrated functions. You can determine principal values for LOG, ^, ASINH, and ATANH by extension from the illustrations. Also given are the general solutions for these functions.

Principal Branch for $\sqrt{Z}$

Domain: `Z = (x,y)`



Principal Value: `W = (u,v) = √(x,y)`

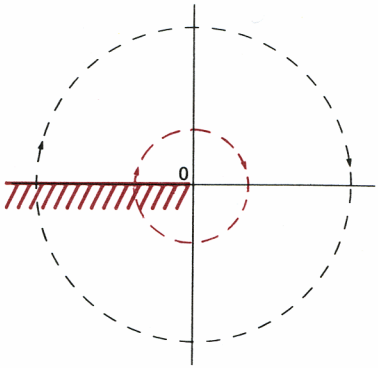


General Solution: `'SQ(W)=Z' 'W' ISOL` returns `'⍵1*√Z'`.

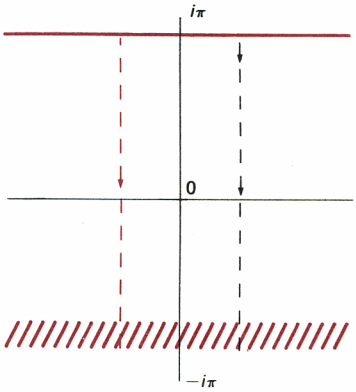
...COMPLEX

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 $\text{asinh}\, z = -i\, \text{asin}\, iz$.

General Solution: `'SINH(W)=Z' 'W' ISOL` returns
`'ASINH(Z)+2*π*i*n1'`

Principal Branch for ATANH(Z)

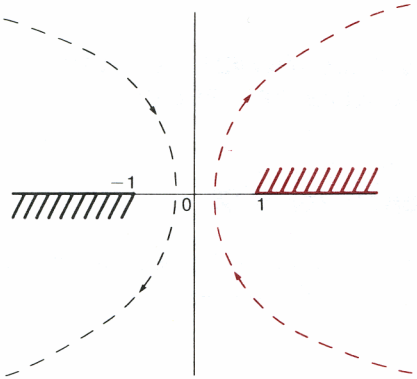
You can determine the principal branch for ATANH from the illustrations for ATAN (on page 122) and the relationship $\text{atanh}\, z = -i\, \text{atan}\, iz$.

General Solution: `'TANH(W)=Z' 'W' ISOL` returns
`'ATANH(Z)+π*i*n1'`

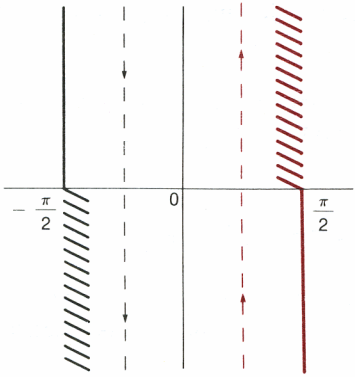
...COMPLEX

Principal Branch for ASIN(Z)

Domain: Z = (x,y)



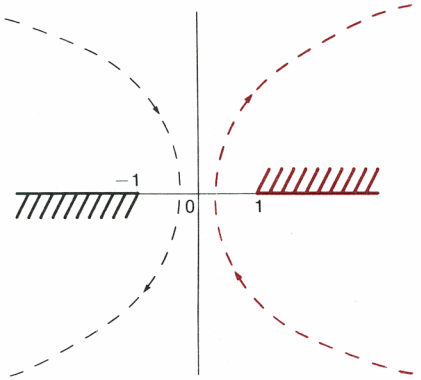
Principal Value: W = (u,v) = ASIN(x,y)



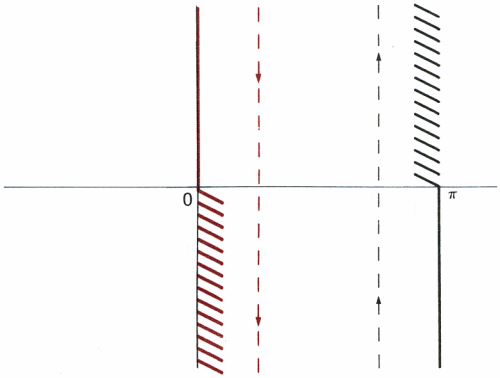
General Solution: 'SIN(W)=Z' 'W' ISOL returns 'ASIN(Z)*(-1)^n1+π*n1'.

Principal Branch for ACOS(Z)

Domain: Z = (x,y)



Principal Value: W = (u,v) = ACOS(x,y)

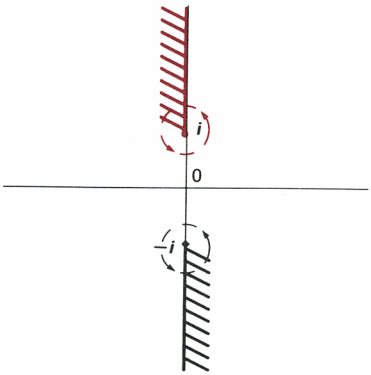


General Solution: 'COS(W)=Z' 'W' ISOL returns 's1*ACOS(Z)+2*π*n1'

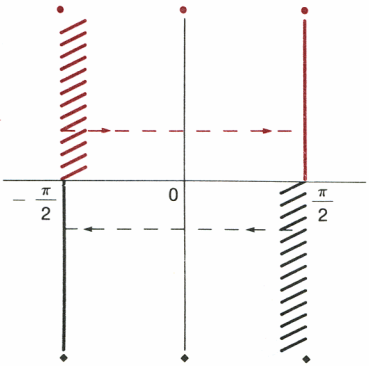
...COMPLEX

Principal Branch for ATAN(Z)

Domain: Z = (x,y)



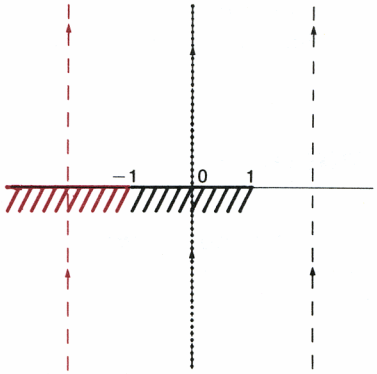
Principal Value: W = (u,v) = ATAN(x,y)



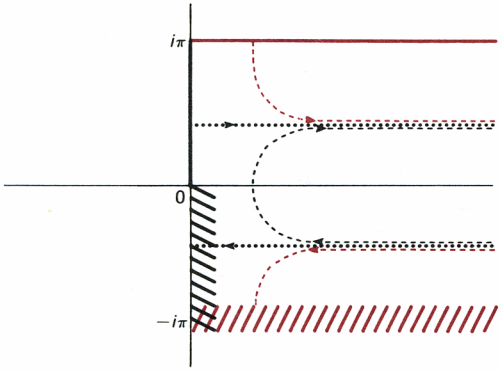
General Solution: 'TAN(W)=Z' 'W' ISOL returns 'ATAN(Z)+π*n1'

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'

Evaluation

Evaluation occurs in all aspects of calculator use, but most evaluation occurs automatically. This section describes commands for explicitly evaluating an object on the stack.

For information about the result of evaluating a particular type of object, see chapter 23, "Evaluation," in the Owner's Manual.

Keyboard Commands

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.

The evaluation of functions is affected by the current Result mode (see page 21). In Numerical Result mode, EVAL and →NUM have the same effect.

→NUM — *Evaluate to Number* — Command

Level 1	Level 1
obj ➡	*z*

→NUM evaluates the object in level 1, temporarily selecting Numerical Result mode (see page 21) to ensure that functions return numerical results. The current Result mode is restored when →NUM is completed.

Automatic Evaluation of Programs and Algebraics

The following commands take programs or algebraics as arguments, evaluating those arguments in the course of execution.

∂	COLCT	ISOL
∫	DRAW	EVAL
TAYLR	ROOT (and Solver)	→NUM

If you execute one of these commands and then execute LAST, LAST returns the arguments to commands in the evaluated program or algebraic, *not* the arguments to the original command.

Automatic Evaluation of Arguments in a List

The following commands can automatically evaluate the contents of a list and use the results as arguments.

- GET, GETI, PUT, and PUTI accept a list specifying the index. Evaluating the contents of the list must produce one or two real numbers.
- →ARRY accepts a list specifying the dimensions. Evaluating the contents of the list must produce one or two real numbers.
- ROOT (or the Solver) accepts a list specifying the initial guess. Evaluating the contents of the list must produce one, two, or three real numbers.
- ∫ accepts a list specifying the variable of integration (optionally) and the limits of integration. Evaluating the last two objects in the list must produce two real numbers.

Evaluation of System Objects

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 and copyright message of your HP-28S by executing `# 10d SYSEVAL`.

LIST

→LIST	LIST→	PUT	GET	PUTI	GETI
POS	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*$_2$ }
{ *list* }	*obj*	➡	{ *list obj* }
obj	{ *list* }	➡	{ *obj list* }

The + function concatenates two lists, or one list and an object treated as a single-element list.

A complete stack diagram for + is given in the "Arithmetic" section.

...LIST

→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.

...LIST

PUT — *Put Element* — Command

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

PUT replaces an object in the specified position in an array or list. This section describes its use with lists; see "ARRAY" for its use with arrays.

PUT takes three arguments from the stack:

- From level 3, a list or the name of a list.
- From level 2, a real number (by itself or in a list) specifying a position in the list.
- From level 1, the object to be put in the list.

If the argument in level 3 is a list, PUT returns the altered list to the stack. If the argument in level 3 is a name, PUT alters the list variable and returns nothing to the stack.

GET — *Get Element* — Command

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

...LIST

GET gets an object from the specified position in an array or list. This section describes its use with lists; see "ARRAY" for its use with arrays.

GET takes two arguments from the stack:

- From level 2, a list or the name of a list.
- From level 1, a real number (by itself or in a list) specifying the position in the list.

GET returns the specified object to the stack.

PUTI ***Put and Increment Index*** **Command**

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

Like PUT, PUTI replaces an object in the specified position in a list. In addition, PUTI returns the list (or name) and the next position. You can then put an object in the next position simply by putting the object on the stack and executing PUTI again.

Generally, after putting an object in position n, PUTI returns $n + 1$ as the next position and clears flag 46. However, when n is the last position in the list, PUTI returns 1 as the next position and sets flag 46.

The following example uses PUTI and flag 46 to put the contents of a variable X in a list, from the initially specified position (not shown) to the last position.

```
...DO X PUTI UNTIL 46 FS? END...
```

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* »	$index_1$	➡	« *list* »	$index_2$	*obj*
' *name* '	$index_1$	➡	' *name* '	$index_2$	*obj*

Like GET, GETI gets an object from the specified position in a list. In addition, GETI returns the list (or name) and the next position. You can then get the object in the next position simply by removing the last-gotten object from level 1 and executing GETI again.

Generally, after getting an object from position *n*, GETI returns $n + 1$ as the next position and clears flag 46. However, when *n* is the last position in the list, GETI returns 1 as the next position and sets flag 46.

The following example uses GETI and flag 46 to add list elements, from the initially specified position (not shown) to the last position, to a variable X.

```
...DO GETI 'X' STO+ UNTIL 46 FS? END...
```

...LIST

POS SUB SIZE

POS — *Position* — Command

Level 2	Level 1		Level 1
"*string*$_1$"	"*string*$_2$"	➡	*n*
⟨ *list* ⟩	*obj*	➡	*n*

POS returns the position of *obj* within ⟨ *list* ⟩. If there is no match for *obj*, POS returns 0.

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 returns a list containing elements n_1 through n_2 of the original list. If $n_2 < n_1$, SUB returns an empty list.

SIZE — *Size* — Command

Level 1		Level 1
"*string*"	➡	*n*
⟨ *list* ⟩	➡	*n*
[*array*]	➡	⟨ *list* ⟩
' *symb* '	➡	*n*

SIZE returns a number *n* that is the number of elements in the list.

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.

...LOGS

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.

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.

MEM	MENU	ORDER	PATH	HOME	CRDIR
VARS	CLUSR				

The MEMORY menu contains commands that relate to variables, directories, and user memory in general.

Keyboard Commands

STO — Store — Command

Level 2	Level 1	
obj	'*global*'	➧
obj	'*local*'	➧
obj	'*global*⟨*index*⟩'	➧

STO stores an object in a global variable, in a local variable, or as an element in a list variable or array variable.

In a Global Variable. If *name* is a global name, *obj* is stored in a variable of that name in the current directory. If no variable of that name exists in the current directory, a new variable is created; if a variable of that name exists, its contents are replaced by *obj*.

In a Local Variable. Local variables are created only by the program-structure commands → and FOR. A program that creates a local variable can use STO to change the contents of that variable.

An Element in a List Variable or Array Variable. When a list or array is stored in a variable, you can replace an element by using the variable name as a user function and the index to the list or array as an argument. For example, 'A(3)' acts as the "name" of the third element in a list or vector A; you could store a value of 5 there by executing

```
5 'A(3)' STO
```

Similarly, 'A(3, 5)' acts as the "name" of the element in the third row and fifth column of a matrix A.

RCL — *Recall* — Command

Level 1	Level 1
'name' ➡	*obj*

RCL returns the contents of the specified variable. The object returned is not evaluated. RCL searches the entire current path, starting with the current directory.

PURGE — *Purge* — Command

Level 1	
'global' ➡ *{ globals }* ➡	

PURGE deletes one or more variables and empty directories from the current directory. You must purge the contents of a directory before you can purge the directory itself.

MEM MENU ORDER PATH HOME CRDIR

MEM *Memory Available* Command

	Level 1
	➡ *x*

MEM returns the number of bytes of currently unused memory. This number is only a rough indicator of usable available memory, since recovery features (COMMAND, UNDO, LAST) consume or release varying amounts of memory with each operation.

MENU *Create Custom Menu* Command

Level 1	
{ *names and commands* }	➡
{ STO *names* }	➡
n	➡

MENU creates a custom menu specified by a list of names, or it displays a standard menu specified by a real number.

Custom User Menu. You can combine built-in commands and your own variables in one custom user menu. For example, after creating user functions CSC (*cosecant*), SEC (*secant*), and COT (*cotangent*), you could combine them in a menu with SIN, COS, and TAN by executing:

```
{ SIN CSC COS SEC TAN COT } MENU
```

...MEMORY

Custom Input Menu. You can create a custom menu that automatically stores values in variables. The first element in the list must be STO; the other elements must be names. (You can't include names of built-in commands.) For example, you could make an input menu for the variables A, B, and C by executing

```
« STO A B C » MENU
```

Then executing 10 [A] 20 [B] 30 [C] stores 10, 20, and 30 in variables A, B, and C.

Standard Menu. You can programmatically select a standard menu by using MENU with a numerical argument. The menus are numbered in the order in which they appear on the keyboard.

Number	Standard Menu	Number	Standard Menu
1	ARRAY	13	PROGRAM CONTROL
2	BINARY	14	PROGRAM BRANCH
3	COMPLEX	15	PROGRAM TEST
4	STRING	16	MODE
5	LIST	17	LOGS
6	REAL	18	PLOT
7	STACK	19	CUSTOM
8	STORE	20	Cursor
9	MEMORY	21	TRIG
10	ALGEBRA	22	SOLVE
11	STAT	23	USER
12	PRINT	24	Solver

ORDER — *Order USER Menu* — Command

Level 1	
{ *names* } ➡	

ORDER rearranges the current directory so that variables appear in the USER menu in the same order as specified in the list. Variables not specified in the list remain in their previous order, appearing after the reordered variables.

If the list includes the name of a large directory, there may be insufficient memory to execute ORDER. In this case, execute System Halt ([ON][▲]) and try again.

PATH — *Current Path* — Command

	Level 1
	➡ { HOME *directory-names* }

PATH returns a list containing the sequence of directory names that identifies the path to the current directory. The first directory is always HOME, and the last directory is always the current directory (which may also be HOME).

HOME — *Switch to HOME Directory* — Command

➡

HOME makes the HOME directory the current directory.

...MEMORY

CRDIR *Create Directory* Command

Level 1	
' *name* ' ➡	

CRDIR creates a subdirectory in the current directory, giving the new directory the specified name. Executing CRDIR doesn't change the current directory; you must evaluate the name of the new subdirectory to make it the current directory.

VARS CLUSR

VARS *Variables* Command

	Level 1
➡ { names }	

VARS returns a list of all variables and subdirectories in the current directory.

CLUSR *Clear User Variables* Command

➡	

CLUSR purges all variables and empty subdirectories in the current directory.

Pressing `CLUSR` always writes the command name to the command line, even in immediate or algebraic entry mode, to help prevent accidental execution. To then execute CLUSR, press ENTER.

MODE

STD	FIX	SCI	ENG	DEG	RAD
CMD	UNDO	LAST	ML	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: a small box in a menu label indicates that the mode is selected.

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 $1 \leq x < 10$, 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.

...MODE

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-28S 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 $1 \leq mantissa < 10$.

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.)

...MODE

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 $x \leq 1$ 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 $x \leq 1$ are always expressed in radians.)

Executing RAD turns on the (2π) annunciator, and sets user flag 60.

CMD UNDO LAST ML RDX, PRMD

The operations `CMD`, `UNDO`, `LAST`, `ML`, and `RDX,` enable and disable the following modes. When one of these menu labels shows a small square, the corresponding mode is enabled.

Mode	Effect When Mode is Enabled
CMD▪	Command lines are saved. You can recover previous command lines by pressing ■ [COMMAND].
UNDO▪	The stack is saved each time you press [ENTER]. You can recover the previous stack by pressing ■ [UNDO] (to "undo" changes to the stack).
LAST▪	Arguments are saved. You can recover the arguments to the last command by pressing ■ [LAST]. To select this mode programmatically, set flag 31.
ML▪	The object in level 1 is displayed in "multi-line" format. To select this mode programmaticaly, set flag 45.
RDX,▪	The radix mark is defined to be the comma—that is, the comma is used as the decimal point. To select this mode programmatically, set flag 48.

PRMD *Print Modes* Command

▪

PRMD displays and prints a listing of current HP-28S 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	DGTIZ	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, and digitize information from plots.

The Display

The HP-28S 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.

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-28S.)

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.

...PLOT

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.

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-28S 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:
   - [INS] returns the coordinates as a complex number (x, y).
   - [DEL] returns a string representing the current display. This action is equivalent to the LCD→ command (page 269).
   - The four rightmost menu keys move the cursor up, down, left or right by one pixel, or by several pixels if you hold down the key, or to the edge of the display if you first press [▪].
   - [◂▸] displays the coordinates in line 4 while you hold down the key.

You can digitize several points by moving the cursor and pressing [INS], moving the cursor again and pressing [INS] again, and so on. As always, you can print the display by pressing [ON][L] at the same time. To terminate interactive plot mode, press [ON].

To activate interactive plot mode from a program, follow the DRAW or DRWΣ command by DGTIZ (*digitize*). After plotting, the program will halt while you digitize; when you press [ON] the program will continue.

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 global 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 equivalent 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 in the current directory. To recall a variable EQ from a parent directory (when EQ doesn't exist in the current directory), execute `'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.

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	
' *global* ' ➡	

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-28S display. If you execute DRAW by pressing the DRAW menu key, an interactive plot is produced, as described in "Interactive Plots" on page 155.

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.

...PLOT

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* }

Pressing PPAR 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 156.

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	
factor ➡	

*W adjusts x_{min} and x_{max}, changing the horizontal scale but not the center:

$$x_{max}' - x_{min}' = factor \times (x_{max} - x_{min})$$

$$\frac{x_{max}' + x_{min}'}{2} = \frac{x_{max} + x_{min}}{2}$$

*H — Multiply Height — Command

Level 1	
factor ➡	

*H adjusts y_{min} and y_{max}, changing the vertical scale but not the center:

$$y_{max}' - y_{min}' = factor \times (y_{max} - y_{min})$$

$$\frac{y_{max}' + y_{min}'}{2} = \frac{y_{max} + y_{min}}{2}$$

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-28S.

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 contents of the variable ΣDAT from the current directory. To recall the statistics matrix ΣDAT from a parent directory (when ΣDAT doesn't exist in the current directory), execute `ΣDAT`.

...PLOT

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 155.

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 DGTIZ 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-28S display (except the annunciators) and sets the system message flag.

...PLOT

DGTIZ — *Digitize* — Command

➡

DGTIZ enables programs to activate the interactive plot mode. Use DRAW DGTIZ to make a mathematical function plot and then digitize points, or use DRWΣ DGTIZ to make a statistical scatter plot and then digitize points. When you're done digitizing, press [ON] to continue the program.

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

➡

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	CR	TRAC
PRSTC	PRUSR	PRMD			

The HP-28S 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-28S case. Before printing, check that the printer can receive the infrared beam from the HP-28S. 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-28S 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 and decrease battery life.

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-28S 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-28S. This loss of data corrupts printed output and can cause the printer to change its configuration.

Double-Space Printing

You can select double-space printing (one blank line between text lines) by setting flag 47. To return to normal printing, clear flag 47.

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
```

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 ($n \leq 166$) 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

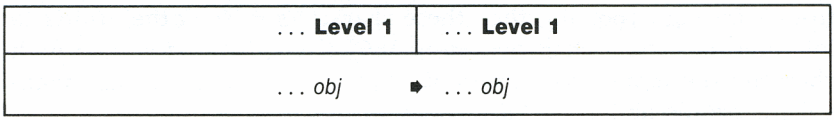
You can print any combination of text, graphics, and objects on a single print line by accumulating data in the printer. The printer stores the data in a part of its memory called a *buffer*.

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**



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	
'*global*'	➡
{ *global*$_1$ *global*$_2$ . . . }	➡

PRVAR searches the current path for the specified variables and prints the name and contents of each variable, using multi-line printer format.

PRLCD — *Print LCD* — Command

➡

PRLCD prints a pixel-by-pixel image of the current HP-28S 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.

CR — *Carriage Right* — Command

➡

CR prints the contents, if any, of the printer buffer.

...PRINT

TRACE Mode

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 TRAC. The menu label then shows a box, indicating that TRACE mode is enabled. You can enable TRACE mode within a program by setting flag 32.

To disable TRACE mode, press TRAC a second time. 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

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.

PRUSR — *Print User Variables* — Command

PRUSR prints a list of all names of variables in the current directory; it is equivalent to `VARS PR1`. 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.

Programs

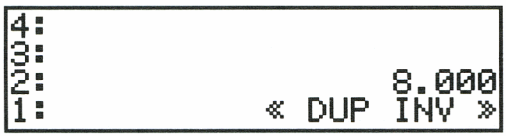
A *program* is a procedure object delimited by « » characters containing a series of commands, objects, and *program structures*, that are executed in sequence when the program is evaluated. Certain program structures, such as those described in "PROGRAM BRANCH" or those specifying local names, must 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-28S, you are extending its programming language.

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:



pressing [EVAL] yields:

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.

...Programs

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.

As you add length and complexity to a program, it can grow beyond a size that is conveniently readable on the HP-28S 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 have 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.

...Programs

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 ($4 \times 1 + 3 \times 2 + 2 \times 3$). 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.

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-28S 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-28S.

- 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? 02 GTO 01 03 GHI 04 JKL 05 GTO 02 06 LBL 01 07 ABC 08 DEF 09 LBL 02 ⋮	10 IF X>0 THEN GOTO 50 20 GHI 30 JKL 40 GOTO 70 50 ABC 60 DEF ⋮

Here is an HP-28S 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.

...PROGRAM BRANCH

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	⋮
⋮	

In the HP-28S, 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-28S 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 (non-zero), 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.

***trap-clause* IFERR *error-clause* THEN *normal-clause* ELSE 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 (CONTRL) 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-28S 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. If you alter the stack, resume program execution, and then execute UNDO when the program is completed, the stack is restored to its state before you executed the program.

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.

...PROGRAM CONTROL

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 O 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-28S 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	Key	String
INS	"INS"	◀▶	"CURSOR"
DEL	"DEL"	◀	"BACK"
▲	"UP"	SPACE	" "
▼	"DOWN"	LC	"1"
◀	"LEFT"		
▶	"RIGHT"		

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).

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.

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.

...PROGRAM CONTROL

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-28S 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-28S 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*≠*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 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*'	➡	'$x \leq symb$'
'*symb*'	*x*	➡	'$symb \leq x$'
'$symb_1$'	'$symb_2$'	➡	'$symb_1 \leq symb_2$'

≥ 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*'	➡	'$x \geq symb$'
'*symb*'	*x*	➡	'$symb \geq x$'
'$symb_1$'	'$symb_2$'	➡	'$symb_1 \geq symb_2$'

The following description refers to the four stack diagrams above.

Each of the four commands $<$, $>$, $\leq$, and $\geq$ 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.

$<$, $>$, $\leq$, and $\geq$, because they imply an ordering, apply to fewer object types than $\neq$, $==$, 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 $1 \leq n \leq 64$.

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).

...PROGRAM TEST

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 $1 \leq n \leq 64$. 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), to binary integers, and to strings. In the first case, the commands act as logical operators that combine true or false truth values into result flags. In the other cases, 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 strings.

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 $symb_2$'

...PROGRAM TEST

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.

...PROGRAM TEST

Following a memory reset, RCLF will return the value `# 2882523502781399O4d`, 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-28S *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 **Analytic**

	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 27.

e

e **Analytic**

	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 27.

NEG FACT RAND RDZ MAXR MINR

NEG

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 [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 $x \geq 253.1190554375$ or x a negative integer, FACT causes an `Overflow` exception; for $x \leq -254.1082426465$, 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-28S uses a linear congruous method and a seed value to generate a random number x, which always lies in the range $0 \leq x < 1$. 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* Analytic

	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 27.

MINR *Minimum Real* Analytic

	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 27.

...REAL

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-28S.

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.

...REAL

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
z_1	➡	z_2
[$array_1$]	➡	[$array_2$]
'*symb*'	➡	'RND(*symb*)'

RND rounds a real number, or each real number in a complex number or array, according to the current display mode:

- In STD display mode, no rounding occurs.
- In *n* FIX display mode, the number is rounded to *n* decimal places.
- In *n* SCI or *n* ENG display mode, the number is rounded to *n* + 1 significant digits.

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.

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.

...SOLVE

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 from the current directory. To recall a variable EQ from a parent directory (when EQ doesn't exist in the current directory), execute `'EQ' RCL`.

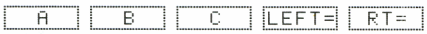
Activating the Variables Menu

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:



results if A, B, and C do not contain procedures. But if we store 'D+E' in C, the menu will become

(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.)

...SOLVE

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.

...SOLVE

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 are the points where the root-finder is sampling the procedure values. The root-finder first searches the domain of the procedure for two points where the procedure values have opposite signs; during this stage, the search region may grow. Once it finds a sign reversal, the root-finder tries to narrow the search region to a point where the procedure value is zero. By watching whether the search region is growing or shrinking, you can track the root-finder's progress.

Interpreting Results

The HP-28S 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`	The Solver found a point where the procedure value is zero.
`Sign Reversal`	The Solver found two points where the procedure values have opposite signs, but it can't find an intermediate point where the procedure value is zero because (a) the two points are neighbors or (b) the procedure is not real-valued between the two points. The Solver returns the point where the procedure value is closer to zero. If the procedure is a continuous real function, this point is the calculator's best approximation to an actual root.
`Extremum`	The Solver found a point where the procedure value approximates a local minimum (for positive values) or maximum (for negative values), or it stopped searching at the point $\pm$9.99999999999E499 because there are no larger machine-representable numbers.

...SOLVE

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»	'global'	guess	➡	root
«program»	'global'	{ guesses }	➡	root
'symb'	'global'	guess	➡	root
'symb'	'global'	{ 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-28S 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
'$symb_1$'	'*global*'	➡ '$symb_2$'

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-28S 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-28S can compute an algebraic inverse are identified as *analytic functions* in this manual.

QUAD *Quadratic Form* Command

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

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
'$symb_1$'	'*global*'	➡	'$symb_2$'

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'.
```

...SOLVE

General Solutions

HP-28S 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 $\sqrt{}$ 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-28S equation `'X=√2'` cannot represent correctly both solutions—the $\sqrt{}$ 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.

General Solution Mode

When the HP-28S 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 four unique values, corresponding to n1 = 0, 1, 2, and 3. 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.

...SOLVE

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 $\pm$, 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, ■ LAST 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.

...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.

LAST — *Last Arguments* — Command

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

LAST returns copies of the arguments to the most recently executed 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 (∂, $\int$, TAYLR, COLCT, DRAW, ROOT, ISOL, EVAL, or →NUM), the last arguments saved are from the procedure, not from the original command.

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.

...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).

...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	COMB	PERM

HP-28S 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	X_{11}	X_{12}	...	X_{1m}
2	X_{21}	X_{22}	...	X_{2m}
⋮	⋮	⋮		⋮
n	X_{n1}	X_{n2}	...	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.

...STAT

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	➡
$[x_1\ x_2\ \ldots\ x_m]$	➡
$[[x_{11}\ x_{12}\ \ldots\ x_{1m}]$ ⋮ $[x_{n1}\ x_{n2}\ \ldots\ x_{nm}]]$	➡

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 $\Sigma+$ in several ways:

Entering one data point with a single coordinate value. The argument for $\Sigma+$ is a real number.

Entering one data point with multiple coordinate values. The argument for $\Sigma+$ is a vector of m real coordinate values.

Entering several data points. The argument for $\Sigma+$ 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, $\Sigma+$ 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 $\Sigma+$ doesn't match the number of columns in ΣDAT.

$\Sigma-$ *Sigma Minus* Command

	Level 1
➡	x
➡	$[x_1\ x_2\ \ldots\ x_m]$

$\Sigma-$ 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 $\Sigma+$ into the statistics matrix ΣDAT. The last row of the statistics matrix is deleted.

The vector returned by $\Sigma-$ can be edited or replaced, then restored to the statistics matrix by $\Sigma+$.

...STAT

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	
obj ➡	

STOΣ takes an object from the stack and stores it in the variable ΣDAT.

RCLΣ *Recall Sigma* Command

	Level 1
	➡ *obj*

RCLΣ returns the contents of the variable ΣDAT from the current directory. To recall the statistics matrix ΣDAT from a parent directory (when ΣDAT doesn't exist in the current directory), execute `ΣDAT`.

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 ith coordinate value in a column, and n is the number of data points.

...STAT

SDEV — *Standard Deviation* — Command

	Level 1
➡	x
➡	$[x_1\ x_2\ \ldots\ x_m]$

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
➡	$[x_1\ x_2\ \ldots\ x_m]$

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

$$\frac{1}{n-1}\sum_{i=1}^{n}\left(x_{in_1}-\bar{x}_{n_1}\right)\left(x_{in_2}-\bar{x}_{n_2}\right)$$

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:

$$predicted\ value = (x \times slope) + intercept.$$

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 COMB PERM

The HP-28S 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 $1 - F(x)$, where $F(x)$ 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 $100\% - 90\% = 10\%$.

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

$$1 - P = \text{utpn}\ (M, V, X),$$

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.

COMB *Combinations* Command

Level 2	Level 1		Level 1
n	*m*	➡	$C_{n, m}$

COMB returns the number of combinations of n items taken m at a time:

$$C_{n, m} = \frac{n!}{m!\ (n - m)!}$$

The arguments n and m must be less than 10^{12}.

PERM *Permutations* Command

Level 2	Level 1		Level 1
n	*m*	➡	$P_{n, m}$

PERM returns the number of permutations of n items taken m at a time:

$$P_{n, m} = \frac{n!}{(n - m)!}$$

The arguments n and m must be less than 10^{12}.

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 variables in the current directory—you cannot use storage arithmetic for variables in other directories or for local variables.

STO+ STO− STO* STO/ SNEG SINV

STO+ *Store Plus* Command

Level 2	Level 1	
z	'*global*'	➡
'*global*'	*z*	➡
[*array*]	'*global*'	➡
'*global*'	[*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	'*global*'	➡
'*global*'	*z*	➡
⌊*array*⌋	'*global*'	➡
'*global*'	⌊*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.

...STORE

STO* — *Store Times* — Command

Level 2	Level 1	
z	'*global*'	➡
'*global*'	*z*	➡
[*array*]	'*global*'	➡
'*global*'	[*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	'*global*'	➧
'*global*'	*z*	➧
⊏*array*⊐	'*global*'	➧
'*global*'	⊏*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.

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	
' *global* ' ➧	

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	
' *global* ' ➧	

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	
' *global* ' ➧	

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	→LCD	LCD→
POS	SUB	SIZE	DISP		

A *string* object consists of a sequence of characters delimited by double-quote marks " at either end. Any HP-28S 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.

...STRING

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** **→LCD** **LCD→**

→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.

...STRING

CHR — *Character* — Command

Level 1		Level 1
n	➡	"*string*"

CHR returns a one-character string containing the HP-28S 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-28S 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-28S as the default character ▪.

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	▒

...STRING

Character Codes (128–255)

NUM	CHR	NUM	CHR	NUM	CHR	NUM	CHR
128		160	¿	192	à	224	Á
129	÷	161	À	193	è	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	^	202	ò	234	ő
139	≠	171	¨	203	ù	235	š
140	α	172	~	204	ä	236	ÿ
141	→	173	Ù	205	ë	237	ú
142	←	174	Û	206	ö	238	ÿ
143	μ	175	£	207	ü	239	ÿ
144	↳	176	¯	208	Ã	240	Þ
145	▫	177	ý	209	î	241	þ
146	«	178	ÿ	210	ø	242	·
147	»	179	▫	211	Æ	243	µ
148	⊢	180	Ç	212	ã	244	¶
149	ı	181	ç	213	í	245	¼
150	₂	182	Ñ	214	≠	246	–
151	²	183	ñ	215	æ	247	¼
152	³	184	¡	216	Ä	248	½
153	‹	185	¿	217	ì	249	¾
154	›	186	¤	218	Ö	250	º
155	▪▪	187	£	219	Ü	251	«
156	‹	188	¥	220	É	252	▮
157	›	189	§	221	ï	253	»
158	k	190	ƒ	222	ß	254	±
159	n	191	¢	223	Ô	255	

→LCD — *String to LCD* — Command

Level 1	
"*string*" ➡	

→LCD takes a string from the stack and, interpreting each character as a graphics code, displays the graphics string. The process is equivalent to the following steps:

1. Each character in the string is converted to an eight-bit binary number equal to its character code.
2. Each binary integer is converted to an eight-high column of pixels, where ones represent black pixels and zeros represent white pixels. The leading digit in the binary integer corresponds to the lowest pixel in the column.
3. Each column of pixels is displayed, starting at the upper-left corner of the display.

A string of 548 characters covers the entire display: the first 137 characters cover line 1 (the top line), the next 137 characters cover line 2, and so on.

LCD→ — *LCD to String* — Command

	Level 1
➡	"*string*"

LCD→ returns a 548-character string that represents the current display. You can later recreate the current display by returning the string to level 1 and executing →LCD.

...STRING

You can use the logical functions AND, OR, XOR, and NOT to combine and modify such strings before executing →LCD. Strings are treated as binary numbers, eight bits for each character in the string. The logical functions have the following effects:

- OR returns the union of two strings. Displayed, this is the superpositioning of the two images.
- AND returns the intersection of two strings. Displayed, this is the shared pixels of the two images.
- XOR returns the symmetric difference of two strings. Displayed, this is the superpositioning of the two images less the shared pixels of the two images.
- NOT returns the inverse of a string. Displayed, this is the inverse of the original image.

POS **SUB** **SIZE** **DISP**

POS

Position **Command**

Level 2	Level 1		Level 1
"*string*$_1$"	"*string*$_2$"	➡	*n*
{ *list* }	*obj*	➡	*n*

POS returns the position of *string*$_2$ within *string*$_1$ or the position of *obj* within { *list* }. If there is no match for *string*$_2$ or *obj*, POS returns 0.

For strings, POS searches for a substring within *string*$_1$ that matches *string*$_2$, returning the position of the first character of the matching substring.

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.

...STRING

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.

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."

...TRIG

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 $-1 \leq x \leq 1$ 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*'	➡	'ACOS(*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 $-1 \leq x \leq 1$ 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:

$$\text{arc tan } z = \frac{i}{z} \ln\left(\frac{i + z}{i - z}\right)$$

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:

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

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.

...TRIG

You can represent a vector in spherical coordinates as [$r\ \phi\ \theta$], 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 [$r\ \theta$], representing polar coordinates, where:

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

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 [$\rho\ \theta\ z$] 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.

...TRIG

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

$$\theta = \begin{cases} \text{arc tan } y/x & \text{for } x \geq 0, \\ \text{arc tan } y/x + \pi \text{ sign } y & \text{for } x < 0, \text{ radians mode,} \\ \text{arc tan } y/x + 180 \text{ sign } y & \text{for } x < 0, \text{ degrees mode.} \end{cases}$$

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-28S 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-28S's unit conversion system is based upon the International System of Units (SI). There are 120 units included in the HP-28S'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-28S 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.

...UNITS

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.

The two unit strings must represent a dimensionally consistent unit conversion. For example, you can convert "1" (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"	➧	1234500	"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.

Dimensionless Units of Angle

Plane and solid angles are called *dimensionless* because they involve no physical dimensions. You can use the following *dimensionless units* as constants in your unit strings; however, the calculator can't check for dimensional consistency in dimensionless units.

Dimensionless Unit	Abbreviation	Value
Arcmin	arcmin	1/21600 unit circle
Arcsec	arcs	1/1296000 unit circle
Degree	°	1/360 unit circle
Grade	grad	1/400 unit circle
Radian	r	1/2π unit circle
Steradian	sr	1/4π unit sphere

Some photometric units are defined in terms of steradians. These units include a factor of 1/4π in their numerical values. Because this factor is dimensionless, the calculator can't check for its presence or absence. Therefore, to convert between photometric units that include this factor and photometric units that don't, you should include the dimensionless unit "sr". The following table lists photometric units according to whether their definition includes steradians.

Photometric Units

Includes Steradians	Doesn't Include Steradians
Lumen (lm)	Candela (cd)
Lux (lx)	Footlambert (flam)
Phot (ph)	Lambert (lam)
Footcandle (fc)	Stilb (sb)

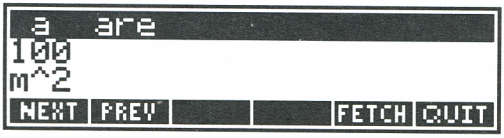
To convert between photometric units in the same column, no "sr" factor is required. However, to convert between photometric units in different columns, you must divide the unit in the left column by "sr" or multiply the unit in the right column by "sr". *Be sure to do so, because the calculator can't check that your units are consistent.* Some examples of consistent photometric units are:

"lm"	is consistent with	"cd*sr"
"fc/sr"	is consistent with	"flam"
"lm/sr*m^2"	is consistent with	"lam"

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-28S, 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-28S display is superceded by the UNITS display:



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-28S 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 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.
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-28S 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	4.62962962963E−5
arcs	Second of arc	Plane angle	7.71604938272E−7
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	Speed of light	Velocity	299792458 m/s
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

HP-28S Units (Continued)

Unit	Full Name	Description	Value
erg	Erg	Energy	.0000001 Kg*m^2/s^2
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	.856564774909
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	.0025
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-28S 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	7.95774715459E−2 cd
lx	Lux	Illuminance	7.95774715459E−2 cd/m^2
lyr	Light year	Length	9.46052840488E15 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

HP-28S Units (Continued)

Unit	Full Name	Description	Value
mmHg	Millimeter of mercury	Pressure	133.322368421 Kg/m*s^2
mol	Mole	Amount of substance	1 mol
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	795.774715459 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	.159154943092
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

HP-28S Units (Continued)

Unit	Full Name	Description	Value
s	Second	Time	1 s
S	Siemens	Electric conductance	1 A^2*s^3/Kg*m^2
sb	Stilb	Luminance	10000 cd/m^2
slug	Slug	Mass	14.5939029372 Kg
sr	Steradian	Solid angle	7.95774715459E−2
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	31556925.9747 s
°	Degree	Angle	2.77777777778E−3

...UNITS

HP-28S Units (Continued)

Unit	Full Name	Description	Value
°C	Degree Celsius	Temperature	1 °K
°F	Degree Fahrenheit	Temperature	.555555555556 °K
°K	Degree Kelvin	Temperature	1 °K
°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 global 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 $\times 10^{-3}$. 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-28S correspond with standard SI notation. The one exception is "deka", indicating an exponent of +1, which is "D" in HP-28S 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.

A

Messages

This appendix lists all error and status messages given by the HP-28S. 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.

Messages Listed Alphabetically

Message	Error Number		Meaning
	Hex	Decimal	
Bad Argument Type	202	514	A command required a different object type or types as arguments.
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.
Directory Not Allowed	12A	299	The name of an existing directory was used as an argument.
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-28S 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 nonexistent variable ΣDAT.
Non-Empty Directory	12B	300	An attempt was made to purge a non-empty directory.
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.

Messages Listed Alphabetically (Continued)

Message	Error Number		Meaning
	Hex	Decimal	
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*$_2$.
Out of Memory			You must purge one or more objects to continue calculator operation.
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.
Power Lost	6	6	Memory may have been corrupted by low power.
Sign Reversal	Status		The Solver found an approximation to an actual root or a discontinuity in the procedure values. (See page 231.)
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.

Messages Listed Alphabetically (Continued)

Message	Error Number		Meaning
	Hex	Decimal	
Unable to Isolate	120	288	The specifed 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.
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
006	006	Power 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
12A	299	Directory Not Allowed
12B	300	Non-Empty Directory
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

Error Messages Listed by Error Number (Continued)

Hex	Decimal	Message
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
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

User Flags

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

Default Settings. For each flag or set of flags listed below, the setting described first is the default setting that occurs at Memory Lost.

Flag	Description
31	**Last Arguments mode** Set: Last Arguments on. Last arguments are saved for recovery by LAST or in case of error. Clear: Last Arguments off.
32	**Printer Trace mode** Clear: Printer Trace off. Set: Printer Trace on. Each time you press an immediate-execute key, the calculator prints the contents of the command line, the immediate-execute operation, and the result in level 1.
33	**Auto CR mode** Clear: Auto CR on. Print commands send Carriage Right at end of transmission. Set: Auto CR off. Data from print commands accumulate in printer buffer.

Flag	Description
34	**Solution mode** Clear: General Solution mode. Solutions returned by ISOL and QUAD include variables for arbitrary signs and integers. Set: Principal Value mode. ISOL and QUAD take arbitrary signs to be 1 and arbitrary integers to be 0.
35	**Constants Mode** Set: Symbolic Constants mode. Evaluating a symbolic constant returns its symbolic form, unchanged. Clear: Numerical Constants mode. Evaluating a symbolic constant returns its numerical value.
36	**Results Mode** Set: Symbolic Results mode. Functions accept symbolic arguments and return symbolic results. Clear: Numerical Results mode. Functions repeatedly evaluate symbolic arguments, accepting only numerical arguments and returning only numerical results. Symbolic constants return numerical values, regardless of flag 35.
37–42	**Binary integer wordsize** These flags are interpreted as a binary integer $0 \leqslant n \leqslant 63$, flag 42 being the most significant bit; wordsize is $n + 1$, default value 64.
43–44	**Binary integer base** 43 clear, 44 clear: Decimal base. 43 clear, 44 set: Binary base. 43 set, 44 clear: Octal base. 43 set, 44 set: Hexadecimal base.
45	**Multi-line Display mode** Set: Multi-line on. Object in level 1 displayed on more than one line if appropriate. Clear: Multi-line off.
46	**Index Wraparound indicator** Clear: Last execution of GETI or PUTI didn't increment index to first position. Set: Last execution of GETI or PUTI incremented index to first position.

Flag	Description
47	**Double-Space Printing mode** Clear: Double-Space Printing off. Set: Double-Space Printing on. One blank line printed between every two text lines.
48	**Radix mark** Clear: Period is radix mark, comma is separator. Set: Comma is radix mark, period is separator.
49–50	**Number format** 49 clear, 50 clear: Standard number format. 49 clear, 50 set: Scientific number format. 49 set, 50 clear: Fixed number format. 49 set, 50 set: Engineering number format.
51	**Beeper mode** Clear: Beeper on. Set: Beeper off.
52	**Print speed** Clear: Normal print speed. Set: Faster print speed.
53–56	**Number of decimal digits** These flags interpreted as a binary integer $0 \leqslant n \leqslant 11$, flag 50 being the most significant bit; default value 0.
57	**Underflow action** Clear: Underflow exceptions return 0 and set flag 61 or 62. Set: Underflow exceptions are errors.
58	**Overflow action** Clear: Overflow exceptions return $\pm$9.99999999999E499 and set flag 63. Set: Overflow exceptions are errors.

Flag	Description
59	**Infinite Result action** Set: Infinite Result exceptions are errors. Clear: Infinite Result exceptions return ±9.99999999999E499 and set flag 64.
60	**Angle mode** Clear: Degrees angle mode. Set: Radians angle mode.
61	**Underflow− indicator** Clear: No Underflow− exception occurred since this flag last cleared. Set: Underflow− exception occurred since this flag last cleared.
62	**Underflow+ indicator** Clear: No Underflow+ exception occurred since this flag last cleared. Set: Underflow+ exception occurred since this flag last cleared.
63	**Overflow indicator** Clear: No Overflow exception occurred since this flag last cleared. Set: Overflow exception occurred since this flag last cleared.
64	**Infinite Result indicator** Clear: No Infinite Result exception occurred since this flag last cleared. Set: Infinite Result exception occurred since this flag last cleared.

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-28S 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-28S 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 directory: The directory currently available in the USER menu. To make a directory the current directory, evaluate its name. Following Memory Lost or System Halt, the current directory is HOME. Most commands that accept global names as arguments search only the current directory. (EVAL, RCL, and PRUSR search the current path).

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+.

current path: The sequence of directories leading from the HOME directory to the current directory.

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.

directory: A named set of global variables. A directory can also contain other directories.

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-28S operations, which are procedure objects stored in ROM.

exponent: In scientific notation, the power of 10 that is multiplied by a number between 1 and 10.

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-28S 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.

global variable: The combination of a name object (the variable name) and any other object (the variable value) stored together in user memory.

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.

HOME directory: The default directory; the current directory following Memory Lost or System Halt.

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 program-structure commands → or FOR. Local variables are automatically purged when the program structure that created them completes execution.

machine singular: Describes a numerical value that is too large to be represented by an HP-28S floating-point number.

mantissa: In scientific notation, the number between 1 and 10 that is multiplied by a power of 10.

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-28S 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.

parent directory: When one directory contains another directory, the first is called the *parent* directory; the second is called a *subdirectory*.

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. The commands delimit *clauses*, which 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: A sequential ordering of the elements in an array. Row order starts with the first element (first row, first column); proceeds left to right along each row, from the first row to the last row; and ends with the last element (last row, last column).

RPN: Reverse Polish Notation.

scatter plot: A plot of data points from the statistics matrix, produced by DRWΣ.

scientific notation: The representation of a number as a signed mantissa (between 1 and 10) and signed exponent (a power of 10).

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-28S 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.

subdirectory: When one directory contains another directory, the second is called a *subdirectory;* the first is called the *parent* directory.

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-28S 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-28S. 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. 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.

Out. How many objects are returned to the stack.

Where. Where the command is described in this manual.

HP-28S Operation Index

Name	Description	Type	In	Out	Where	
ABORT	Aborts program execution.	C	0	0	PROGRAM CONTROL	195
ABS	Absolute value.	F	1	1	ARRAY COMPLEX REAL	79 114 218
ACOS	Arc cosine.	A	1	1	TRIG	273
ACOSH	Arc hyperbolic cosine.	A	1	1	LOGS	136
▒▒	Adds fractions.	O*			ALGEBRA (FORM)	46
■ ALGBRA	Selects the ALGEBRA menu.	O*			ALGEBRA	16
ALOG	Antilogarithm (10 to a power).	A	1	1	LOGS	133
AND	Logical or binary AND.	F	2	1	BINARY PROGRAM TEST	92 206
ARG	Argument.	F	1	1	COMPLEX TRIG	114 277
■ ARRAY	Selects the ARRAY menu.	O*			ARRAY	63
ARRY→	Replaces an array with its elements as separate stack objects.	C	1	$n+1$	ARRAY	70
ASIN	Arc sine.	A	1	1	TRIG	273
ASINH	Arc hyperbolic sine.	A	1	1	LOGS	136

ASR	Arithmetic shift right.	C	1	1	BINARY	91
ATAN	Arc tangent.	A	1	1	TRIG	273
ATANH	Arc hyperbolic tangent.	A	1	1	LOGS	136
ATTN (ON)	Aborts program execution; clears the command line; exits catalogs, FORM, plot displays.	O*				
AXES	Sets intersection of axes.	C	1	0	PLOT	160
A→	Associates to the right.	O*			ALGEBRA (FORM)	39
BEEP	Sounds a beep.	C	2	0	PROGRAM CONTROL	198
BIN	Sets binary base.	C*	0	0	BINARY	87
BINARY	Selects the BINARY menu.	O*			BINARY	85
BRANCH	Selects the PROGRAM BRANCH menu.	O*			PROGRAM BRANCH	183
B→R	Binary-to-real conversion.	C	1	1	BINARY	89
CATALOG	Starts the command catalog.	O*				
CEIL	Next greater integer.	F	1	1	REAL	219
CENTR	Sets center of plot display.	C	1	0	PLOT	160
CF	Clears a user flag.	C	1	0	PROGRAM TEST	204
CHR	Makes a one-character string.	C	1	1	STRING	264
CHS	Changes the sign of a number in the command line or executes NEG.	O*			Arithmetic	62
CLEAR	Clears the stack.	C	*n*	0	STACK	240

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where
CLLCD	Blanks the display.	C	0	0	PLOT 165 PROGRAM CONTROL 198
CLMF	Clears the system message flag.	C	0	0	PLOT 165 PROGRAM CONTROL 198
CLUSR	Clears all user variables.	C†	0	0	MEMORY 144
CLΣ	Purges the statistics matrix.	C	0	0	STAT 246
CMD	Enables or disables CMD mode.	O*			MODE 150
COMPLX	Selects the COMPLEX menu.	O*			COMPLEX 110
CNRM	Computes a column norm.	C	1	1	ARRAY 79
COLCT	Collects like terms.	C	1	1	ALGEBRA 28
COLCT	Collects like terms in a subexpression.	O*			ALGEBRA (FORM) 37
COLΣ	Selects statistics matrix columns.	C	2	0	PLOT 163 STAT 251
COMB	Combinations	C	2	1	STAT 254
COMMAND	Recovers previous contents of command line (if saved).	O			MODE 151
CON	Creates a constant matrix.	C	2	0, 1	ARRAY 75

CONJ	Complex conjugate.	F	1	1	ARRAY COMPLEX	82 111
■ CONT	Continues a halted program.	O			PROGRAM CONTROL	194
CONVERT	Performs a unit conversion.	C	3	2	UNITS	283
CORR	Correlation coefficient.	C	0	1	STAT	251
COS	Cosine.	A	1	1	TRIG	273
COSH	Hyperbolic cosine.	A	1	1	LOGS	136
COV	Covariance.	C	0	1	STAT	251
CR	Prints a carriage-right.	C	0	0	PRINT	171
CRDIR	Creates a directory.	C	1	0	MEMORY	141
CROSS	Cross product of two- or three-element vectors.	C	2	1	ARRAY	79
■ CONTRL	Selects the PROGRAM CONTROL menu.	O*			PROGRAM CONTROL	193
■ CUSTOM	Selects the last-displayed custom menu.	O*				
C→R	Complex-to-real conversion.	C	1	2	ARRAY COMPLEX TRIG	82 111 277
■ d/dx	Derivative (∂ function).	F	2	1	Calculus	96
DEC	Sets decimal base.	C*	0	0	BINARY	87
DEG	Sets degrees mode.	C*	0	0	MODE	145
DEL	Deletes character at cursor; returns graphics string representing current display.	O*	0	0, 2	PLOT	155

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
DEL	Deletes character at cursor and all characters to the right.	O*				
DEPTH	Counts the objects on the stack.	C	0	1	STACK	243
DET	Determinant of a matrix.	C	1	1	ARRAY	79
DGTIZ	Activates interactive plot mode	C	0	0	PLOT	165
DINV	Double inverts.	O*			ALGEBRA (FORM)	43
DISP	Displays an object.	C	2	0	PROGRAM CONTROL STRING	198 270
DNEG	Double negates.	O*			ALGEBRA (FORM)	42
DO	Part of DO...UNTIL...END.	C†			PROGRAM BRANCH	192
DOT	Dot product of two vectors.	C	2	1	ARRAY	79
DRAW	Creates a mathematical function plot.	C	0	0	PLOT	157
DRAX	Draws axes.	C	0	0	PLOT	165
DROP	Drops one object from the stack.	C	1	0	STACK	239
DROPN	Drops $n+1$ objects from the stack.	C	$n+1$	0	STACK	243
DROP2	Drops two objects from the stack.	C	2	0	STACK	241

DRWΣ	Creates a statistics scatter plot.	C	0	0	PLOT	163
DUP	Duplicates one object on the stack.	C	1	2	STACK	241
DUPN	Duplicates *n* objects on the stack.	C	*n*+1	2*n*	STACK	243
DUP2	Duplicates two objects on the stack.	C	2	4	STACK	241
D→	Distributes to the right.	O*			ALGEBRA (FORM)	40
D→R	Degrees-to-radians conversion.	F	1	1	TRIG	280
e	Symbolic constant *e*.	A†	0	1	ALGEBRA REAL	27 215
■ EDIT	Copies the object in level 1 to the command line for editing.	O	1	1		
EEX	Enters exponent in command line.	O*				
ELSE	Begins ELSE clause.	C†			PROGRAM BRANCH	186
END	Ends program structures.	C†	0,1	0	PROGRAM BRANCH	186 192
ENG	Sets engineering display format.	C	1	0	MODE	145
ENTER	Parses and evaluates the command line or executes DUP.	O				
ERRM	Returns the last error message.	C	0	1	PROGRAM CONTROL	198
ERRN	Returns the last error number.	C	0	1	PROGRAM CONTROL	198
EVAL	Evaluates an object.	C	1	0	Evaluation	124

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
EXGET	Gets a subexpression.	C	2	1	ALGEBRA	33
EXGET	Gets a subexpression.	O*		2	ALGEBRA (FORM)	37
EXP	Natural exponential.	A	1	1	LOGS	133
EXPAN	Expands an algebraic.	C	1	1	ALGEBRA	28
EXPAN	Expands a subexpression.	O*			ALGEBRA (FORM)	37
EXPM	Natural exponential minus 1.	A	1	1	LOGS	133
EXPR=	Evaluates the current equation.	O	0	1	SOLVE	227
EXSUB	Substitutes a subexpression.	C	3	1	ALGEBRA	28
E^	Replaces power-of-product with power-of-power.	O*			ALGEBRA (FORM)	46
E()	Replaces power-of-power with power-of-product.	O*			ALGEBRA (FORM)	46
FACT	Factorial or gamma function.	F	1	1	REAL	215
FC?	Tests a user flag.	C	1	1	PROGRAM TEST	204
FC?C	Tests and clears a user flag.	C	1	1	PROGRAM TEST	204
FETCH	Exits CATALOG or UNITS, writes the current command or unit in the command line.	O*			UNITS	287
FIX	Sets FIX display format.	C	1	0	MODE	145

FLOOR	Next smaller integer.	F	1	1	REAL	219
FOR	Begins definite loop.	C†	2	0	PROGRAM BRANCH	188
FORM	Changes the form of an algebraic.	C	1	1, 3	ALGEBRA ALGEBRA (FORM)	28 34
FP	Fractional part.	F	1	1	REAL	219
FS?	Tests a user flag.	C	1	1	PROGRAM TEST	204
FS?C	Tests and clears a user flag.	C	1	1	PROGRAM TEST	204
GET	Gets an element from an object.	C	2	1	ARRAY LIST	70 128
GETI	Gets an element from an object and increments the index.	C	2	3	ARRAY LIST	70 128
HALT	Suspends program execution.	C†			PROGRAM CONTROL	195
HEX	Sets hexadecimal base.	C*	0	0	BINARY	87
HMS+	Adds in HMS format.	C	2	1	TRIG	280
HMS−	Subtracts in HMS format.	C	2	1	TRIG	280
HMS→	Converts from HMS format.	C	1	1	TRIG	280
HOME	Selects the HOME directory.	C	0	0	MEMORY	141
i	Symbolic constant *i*.	A†	0	1	ALGEBRA	27
IDN	Creates an identity matrix.	C	1	0, 1	ARRAY	75

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
IF	Begins IF clause.	C†	0	0	PROGRAM BRANCH	186
IFERR	Begins IF ERROR clause.	C†	0	0	PROGRAM BRANCH	186
IFT	If-then command.	C	2	0	PROGRAM BRANCH	188
IFTE	If-then-else function.	F	3	0	PROGRAM BRANCH	188
IM	Returns the imaginary part of a number or array.	F	1	1	ARRAY COMPLEX	82 111
INDEP	Selects the plot independent variable.	C	1	0	PLOT	157
INS	Switches between insert and replace modes; digitizes point.	O*	0	0, 1	PLOT	155
■ INS	Deletes all characters to the left of the cursor.	O*				
INV	Inverse (reciprocal).	A	1	1	Arithmetic ARRAY	60 69
IP	Integer part.	F	1	1	REAL	219
ISOL	Solves an expression or equation.	C	2	1	ALGEBRA SOLVE	33 234
KEY	Returns a key string.	C	0	1, 2	PROGRAM CONTROL	195
KILL	Aborts all suspended programs.	C	0	0	PROGRAM CONTROL	195

LAST	Returns last arguments (if saved).	C	0	1, 2, 3	MODE STACK	151 240
LAST	Enables or disables LAST mode.	O*			MODE	150
LC	Switches between upper-case and lower-case modes.	O*				
LCD→	Returns display image as a graphics string.	C	0	1	STRING	264
LEFT=	Evaluates the left side of the current equation.	O	0	1	SOLVE	227
LEVEL	Displays the level of the selected subexpression.	O*			ALGEBRA (FORM)	37
LIST	Selects the LIST menu.	O*			LIST	127
LIST→	Moves list elements to the stack.	C	1	$n+1$	LIST STACK	128 241
LN	Natural logarithm.	A	1	1	LOGS	133
LNP1	Natural logarithm of (*argument*+1).	A	1	1	LOGS	133
LOG	Common (base 10) logarithm.	A	1	1	LOGS	133
LOGS	Selects the LOGS menu.	O*			LOGS	133
LR	Computes a linear regression.	C	0	2	STAT	251
L( )	Replaces product-of-log with log-of-power.	O*			ALGEBRA (FORM)	45
L*	Replaces log-of-power with product-of-log.	O*			ALGEBRA (FORM)	45
MANT	Returns the mantissa of a number.	F	1	1	REAL	218

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
MAX	Returns the maximum of two numbers.	F	2	1	REAL	221
MAXR	Symbolic constant maximum real.	A	0	1	ALGEBRA REAL	27 215
MAXΣ	Finds the maximum coordinate values in the statistics matrix.	C	0	1	STAT	249
MEAN	Computes statistical means.	C	0	1	STAT	249
MEM	Returns available memory.	C	0	1	MEMORY	141
MENU	Selects the specified built-in or custom menu.	C	1	0	MEMORY	141
■ MENUS	Switches shifted action and unshifted action of letter keys A through R.	O				
MIN	Returns the minimum of two numbers.	F	2	1	REAL	221
MINR	Symbolic constant minimum real.	A	0	1	ALGEBRA REAL	27 215
MINΣ	Finds the minimum coordinate values in the statistics matrix.	C	0	1	STAT	249
ML	Enables or disables ML mode.	0*			MODE	150
MOD	Modulo.	F	2	1	REAL	221
■ MODE	Selects the MODE menu.	O*			MODE	145

M→	Merges right factor.	O*			ALGEBRA (FORM)	42
NEG	Negates an argument.	A	1	1	Arithmetic ARRAY COMPLEX REAL	62 82 114 215
NEXT	Ends definite loop.	C†	0	0	PROGRAM BRANCH	188
[NEXT]	Displays the next row of menu labels.	O*				57
NEXT	Advances to next command or unit in a catalog.	O*			UNITS	287
NEXT	Advances to next argument option in USAGE.	O*				
NO	Choose not to purge during `Out of Memory.`	O*				
NOT	Logical or binary NOT.	F	1	1	BINARY PROGRAM TEST	92 206
NUM	Returns character code.	C	1	1	STRING	264
NΣ	Returns the number of data points in the statistics matrix.	C	0	1	STAT	246
OBGET	Extracts an object from an algebraic.	C	2	1	ALGEBRA	33
OBSUB	Substitutes an object into an algebraic.	C	3	1	ALGEBRA	28
OCT	Sets octal base.	C*	0	0	BINARY	87
[ON] ([ATTN])	Turns the calculator on; aborts program execution; clears the command line; exits catalogs, FORM, plot displays.	O*				

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where
ON DEL	Cancels system halt or memory reset if pressed before ON is released.	O*			
ON INS ▶	(Memory Reset) Stops program execution, clears local and user variables, clears the stack, resets user flags.	O*			
ON L	Print LCD.	O*			PRINT
ON NEXT	Starts keyboard test.	O*			
ON +	Increases the display contrast.	O*			
ON −	Decreases the display contrast.	O*			
ON ▲	(System Halt) Stops program execution, clears local variables, clears the stack.	O*			
ON ◀	Starts continuous system test.	O*			
■ OFF	Turns the calculator off.	O*			
OR	Logical or binary OR.	F	2	1	BINARY 92 PROGRAM TEST 206
ORDER	Rearranges the user menu.	C	1	0	MEMORY 141
OVER	Duplicates the object in level 2.	C	2	3	STACK 241
PATH	Returns a list showing the current path.	C	0	1	MEMORY 141
PERM	Permutations.	C	2	1	STAT 254

PICK	Duplicates the *n*th object.	C	*n*+1	*n*+1	STACK	243
PIXEL	Turns on a display pixel.	C	1	0	PLOT	165
PLOT	Selects the PLOT menu.	O*			PLOT	152
PMAX	Sets the upper-right plot coordinates.	C	1	0	PLOT	157
PMIN	Sets the lower-left plot coordinates.	C	1	0	PLOT	157
POS	Finds a substring in a string or an object in a list.	C	2	1	LIST STRING	132 270
PPAR	Recalls the plot parameters list in the current directory.	O	0	1	PLOT	160
PREDV	Predicted value.	C	1	1	STAT	251
PREV	Displays the previous row of menu labels.	O*				
PREV	Displays the previous command or unit in a catalog.	O*			UNITS	287
PREV	Displays the previous argument option in USAGE.	O*				
PRINT	Selects the PRINT menu.	O*			PRINT	168
PRLCD	Prints an image of the display.	C	0	0	PLOT PRINT	165 171
PRMD	Prints and displays current modes.	C	0	0	MODE PRINT	150 174
PROGRAM						
BRANCH	Selects the PROGRAM BRANCH menu.	O*			PROGRAM BRANCH	183
CONTRL	Selects the PROGRAM CONTROL menu.	O*			PROGRAM CONTROL	193
TEST	Selects the PROGRAM TEST menu.	O*			PROGRAM TEST	201

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
PRST	Prints the stack.	C	0	0	PRINT	171
PRSTC	Prints the stack in compact format.	C	0	0	PRINT	174
PRUSR	Prints a list of variables in the current directory.	C	0	0	PRINT	174
PRVAR	Prints the name and contents of one or more variables.	C	1	0	PRINT	171
PR1	Prints the level 1 object.	C	0	0	PRINT	171
PURGE	Purges one or more variables.	C	1	0	MEMORY	140
PUT	Replaces an element in an array or list.	C	3	0, 1	ARRAY LIST	70 128
PUTI	Replaces an element in an array or list, and increment the index.	C	3	2	ARRAY LIST	70 128
P→R	Polar-to-rectangular conversion.	F	1	1	COMPLEX TRIG	114 227
QUAD	Solves a quadratic polynomial.	C	2	1	ALGEBRA SOLVE	33 235
QUIT	Exits CATALOG or UNITS.	O*			UNITS	287
QUIT	Exits USAGE display.	O*				

RAD	Sets radians mode.	C*	0	0	MODE	145
RAND	Returns a random number.	C	0	1	REAL	215
RCEQ	Recalls the current equation.	C	0	1	PLOT SOLVE	157 226
RCL	Recalls the contents of a variable, unevaluated.	C	1	1	MEMORY	140
RCLF	Returns a binary integer representing the user flags.	C	0	1	PROGRAM TEST	211
RCLΣ	Recalls the current statistics matrix.	C	0	1	PLOT STAT	163 246
RCWS	Recalls the binary integer wordsize.	C	0	1	BINARY	87
RDM	Redimensions an array.	C	2	0, 1	ARRAY	75
RDX,	Enables or disables RDX, mode.	O*			MODE	150
RDZ	Sets the random number seed.	C	1	0	REAL	215
RE	Returns the real part of a complex number or array.	F	1	1	ARRAY COMPLEX	82 111
REAL	Selects the REAL menu.	O*			REAL	213
REPEAT	Part of WHILE...REPEAT...END.	C†	1	0	PROGRAM BRANCH	192
RES	Sets the plot resolution.	C	1	0	PLOT	160

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
RL	Rotates left by one bit.	C	1	1	BINARY	89
RLB	Rotates left by one byte.	C	1	1	BINARY	89
RND	Rounds according to real number display mode.	F	1	1	REAL	219
RNRM	Computes the row norm of an array.	C	1	1	ARRAY	79
ROLL	Moves the level $n+1$ object to level 1.	C	$n+1$	n	STACK	240
ROLLD	Moves the level 2 object to level n.	C	$n+1$	n	STACK	243
ROOT	Finds a numerical root.	C	3	1, 3	SOLVE	233
ROT	Moves the level 3 object to level 1.	C	3	3	STACK	241
RR	Rotates right by one bit.	C	1	1	BINARY	89
RRB	Rotates right by one byte.	C	1	1	BINARY	89
RSD	Computes a correction to the solution of a system of equations.	C	3	1	ARRAY	75
RT=	Evaluates the right side of the current equation.	O	0	1	SOLVE	227
R→B	Real-to-binary conversion.	C	1	1	BINARY	89
R→C	Real-to-complex conversion.	C	2	1	ARRAY COMPLEX TRIG	82 111 279
R→D	Radians-to-degrees conversion.	F	1	1	TRIG	280

R→P	Rectangular-to-polar conversion.	F	1	1	COMPLEX TRIG	114 277
SAME	Tests two objects for equality.	C	2	1	PROGRAM TEST	206
SCI	Sets scientific display format.	C	1	0	MODE	145
SCLΣ	Auto-scales the plot parameters according to the statistical data.	C	0	0	PLOT	163
SCONJ	Conjugates the contents of a variable.	C	1	0	STORE	262
SDEV	Computes standard deviations.	C	0	1	STAT	249
SF	Sets a user flag.	C	1	0	PROGRAM TEST	204
SHOW	Resolves all references to a name implicit in an algebraic.	C	2	1	ALGEBRA SOLVE	33 235
SIGN	Sign of a number.	F	1	1	COMPLEX REAL	111 218
SIN	Sine.	A	1	1	TRIG	273
SINH	Hyperbolic sine.	A	1	1	LOGS	136
SINV	Inverts the contents of a variable.	C	1	0	STORE	258
SIZE	Finds the dimensions of a list, array, string, or algebraic.	C	1	1	ALGEBRA ARRAY LIST STRING	28 75 132 270

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
SL	Shifts left by one bit.	C	1	1	BINARY	91
SLB	Shifts left by one byte.	C	1	1	BINARY	91
SNEG	Negates the contents of variable.	C	1	0	STORE	258
SOLV	Selects the SOLVE menu.	O*			SOLVE	224
SOLVR	Selects the Solver variables menu.	O			SOLVE	225
SQ	Squares a number or matrix.	A	1	1	Arithmetic ARRAY	61 70
SR	Shifts right by one bit.	C	1	1	BINARY	91
SRB	Shifts right by one byte.	C	1	1	BINARY	91
SST	Single-steps a suspended program.	O			PROGRAM CONTROL	195
STACK	Selects the STACK menu.	O*			STACK	239
START	Begins definite loop.	C†	2	0	PROGRAM BRANCH	188
STAT	Selects the STAT menu.	O*			STAT	245
STD	Sets standard display format.	C*			MODE	145
STEP	Ends definite loop.	C†	1	0	PROGRAM BRANCH	188
STEQ	Stores the current equation.	C	1	0	PLOT SOLVE	157 226

STO	Stores an object in a variable.	C	2	0	MEMORY	139
STOF	Sets all user flags according to the value of a binary integer.	C	1	0	PROGRAM TEST	211
■ STORE	Selects the STORE menu.	O*			STORE	258
STO∗	Storage arithmetic multiply.	C	2	0	STORE	258
STO+	Storage arithmetic add.	C	2	0	STORE	258
STO−	Storage arithmetic subtract.	C	2	0	STORE	258
STO/	Storage arithmetic divide.	C	2	0	STORE	258
STOΣ	Stores the current statistics matrix.	C	1	0	PLOT STAT	163 246
■ STRING	Selects the STRING menu.	O*			STRING	263
STR→	Parses and evaluates the commands defined by a string.	C	1	0	STRING	264
STWS	Sets the binary integer wordsize.	C	1	0	BINARY	87
SUB	Extracts a portion of a list or string.	C	3	1	LIST STRING	132 270
SWAP	Swaps the objects in levels 1 and 2.	C	2	2	STACK	239

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
SYSEVAL	Executes a system object.	C	1	0	Evaluation	126
TAN	Tangent.	A	1	1	TRIG	273
TANH	Hyperbolic tangent.	A	1	1	LOGS	136
TAYLR	Computes a Taylor series approximation.	C	3	1	ALGEBRA Calculus	33 106
[TEST]	Selects the PROGRAM TEST menu.	O*			PROGRAM TEST	201
THEN	Begins THEN clause.	C†	1	0	PROGRAM BRANCH	186
TOT	Sums the coordinate values in the statistics matrix.	C	0	1	STAT	249
TRAC	Enables or disables printer Trace mode.	O*			PRINT	171
[TRIG]	Selects the TRIG menu.	O*			TRIG	273
TRN	Transposes a matrix.	C	1	0, 1	ARRAY	75
TYPE	Returns the type of an object.	C	1	1	PROGRAM TEST	211
[UNDO]	Recovers previous stack contents (if saved).	O*			MODE	151
UNDO	Enables or disables UNDO mode.	O*			MODE	150
[UNITS]	Selects the units catalog.	O*			UNITS	287
UNTIL	Part of BEGIN...UNTIL...END.	C†			PROGRAM BRANCH	192
USE	Displays USAGE for current command in CATALOG.	O*				

USER	Selects the USER menu.	O*				
UTPC	Upper-tail Chi-Square distribution .	C	2	1	STAT	254
UTPF	Upper-tail F-distribution.	C	3	1	STAT	254
UTPN	Upper-tail normal distribution.	C	3	1	STAT	254
UTPT	Upper-tail t-distribution.	C	2	1	STAT	254
VAR	Computes statisical variances.	C	0	1	STAT	249
VARS	Returns a list of variables in the current directory.	C	0	1	MEMORY	144
VIEW↑	Moves the display window up one line.	O*				
VIEW↓	Moves the display window down one line.	O*				
VISIT	Copies an object to the command line for editing.	O	1	0		
WAIT	Pauses program execution.	C	1	0	PROGRAM CONTROL	195
WHILE	Begins WHILE...REPEAT...END.	C†	0	0	PROGRAM BRANCH	192
XOR	Logical or binary XOR.	F	2	1	BINARY PROGRAM TEST	92 206
XPON	Returns the exponent of a number.	F	1	1	REAL	218
x^2	Executes function SQ.	A	1	1	Arithmetic ARRAY	61 70
YES	Chooses to purge during Out of Memory.	O*				

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
[1/x]	Executes function INV.	A	1	1	Arithmetic ARRAY	69 69
1/()	Double invert and distribute.	O*			ALGEBRA (FORM)	44
+	Adds two objects.	A	2	1	Arithmetic ARRAY LIST STRING	53 65 127 264
+1-1	Adds and subtracts 1.	O*			ALGEBRA (FORM)	45
−	Subtracts two objects.	A	2	1	Arithmetic ARRAY	55 65
-()	Double negates and distributes.	O*			ALGEBRA (FORM)	43
*	Multiplies two objects.	A	2	1	Arithmetic ARRAY	56 66
*H	Adjusts the height of a plot.	C	1	0	PLOT	160
*W	Adjusts the width of a plot.	C	1	0	PLOT	160
*1	Multiplies by 1.	O*			ALGEBRA (FORM)	44
/	Divides two objects.	A	2	1	Arithmetic ARRAY	58 66

⁄1	Divides by 1.	O*			ALGEBRA (FORM)	44
%	Percent.	F	2	1	REAL	214
%CH	Percent change.	F	2	1	REAL	214
%T	Percent of total.	F	2	1	REAL	221
^	Raises a number to a power.	A	2	1	Arithmetic	60
^1	Raises to the power 1.	O*			ALGEBRA (FORM)	45
$\sqrt{\ }$	Takes the square root.	A	1	1	Arithmetic	61
[■] [$\sqrt{x}$]	Executes the function $\sqrt{\ }$.	A	1	1	Arithmetic	61
$\int$	Definite or indefinite integral.	C	3	1, 2	Calculus	100
∂	Derivative.	F	2	1	Calculus	96
$<$	Less-than comparison.	F†	2	1	PROGRAM TEST	202
$\leq$	Less-than-or-equal comparison.	F†	2	1	PROGRAM TEST	203
$=$	Equals operator.	A†	2	1	ALGEBRA	21
$==$	Equality comparison.	F	2	1	PROGRAM TEST	206
$\neq$	Not-equal comparison.	F†	2	1	PROGRAM TEST	201
$\geq$	Greater-than-or-equal comparison.	F†	2	1	PROGRAM TEST	203
$>$	Greater-than comparison.	F†	2	1	PROGRAM TEST	202
■	Shift key.	O*				
◆	Selects cursor menu or restores last menu; displays coordinates.	O*			PLOT	155

HP-28S Operation Index (Continued)

Name	Description	Type	In	Out	Where	
▲	Moves cursor up.	O*				
■ ▲	Moves cursor up all the way.	O*				
▼	Moves cursor down.	O*				
■ ▼	Moves cursor down all the way.	O*				
◀	Moves cursor left.	O*				
■ ◀	Moves cursor left all the way.	O*				
▶	Moves cursor right.	O*				
■ ▶	Moves cursor right all the way.	O*				
[←]	Moves FORM cursor left.	O*			ALGEBRA (FORM)	37
[→]	Moves FORM cursor right.	O*			ALGEBRA (FORM)	37
⌫	Backspace.	O*				
α	Switches entry mode from Immediate to Alpha; from Alpha to Algebraic; or from Algebraic to Immediate.	O*				
π	Symbolic constant π.	A†	0	1	ALGEBRA REAL	27 215
Σ+	Adds a data point to the statistics matrix.	C	1	0	STAT	246
Σ−	Deletes the last data point from the statistics matrix.	C	0	1	STAT	246

←A	Associates to the left.	O*			ALGEBRA (FORM)	38
←D	Distributes to the left.	O*			ALGEBRA (FORM)	40
←M	Merges left factors.	O*			ALGEBRA (FORM)	41
←→	Commutes arguments.	O*			ALGEBRA (FORM)	38
→	Creates local variables.	C†	*n*	0	Programs	178
→ARRY	Combines numbers into an array.	C	*n*+1	1	ARRAY	70
→HMS	Converts a number to HMS format.	C	1	1	TRIG	280
→LCD	Displays graphics string.	C	1	0	STRING	264
→LIST	Combines objects into a list.	C	*n*+1	1	LIST STACK	128 243
→NUM	Evaluates an object in numerical mode.	C	1	0	Evaluation	124
→STR	Converts an object to a string.	C	1	1	STRING	264
→( )	Distributes prefix operator.	O*			ALGEBRA (FORM)	39

Subject Index

For index entries with multiple references, page numbers in **bold** type indicate primary references.

Q

R

S

T

U

V

W, X, Y, Z

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 (rounded if non-integer).
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	Real number specifying an element in a list or array; or list with one real number (or object that evaluates to a number) specifying an element in a list or vector; or list with two real numbers (or objects that evaluates to numbers) specifying an element in a matrix.
{ *dim* }	List of one or two real numbers specifying the dimension(s) of an array.
'*name*'	Global name or local name.
'*global*'	Global name.
'*local*'	Local name.
« *program* »	Program.
'*symb*'	Expression, equation, or a name treated as an algebraic.

Contents

HEWLETT PACKARD

Reorder Number
00028-90068

00028-90148 English
Printed in U.S.A. 11/88