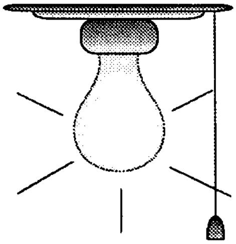
HP-28S Software Power Tools

Utilities



A Product of
Solve and Integrate Corporation

A GRAPEVINE PUBLICATION

HP-28S
Software Power Tools:

UTILITIES

A Product of
Solve and Integrate Corporation

Grapevine Publications, Inc.
P.O. Box 118
Corvallis, Oregon 97339-0118 U.S.A.

Acknowledgements

We extend our thanks once again to Hewlett-Packard for their top-quality products and documentation.

Printed in the United States of America
ISBN 0-931011-27-2

First Printing – November, 1989

CONTENTS

Chapter 0

Introduction To This Book

This book is primarily a toolkit of small HP-28S program routines that can help you build bigger and better programs of your own. These are not generally useful "all by themselves" (indeed, most of these tools are fairly boring and useless when invoked "one at a time," or "manually," from the keyboard). *They are meant to be combined within programs that you construct for your own purposes.*

So this book is mainly a reference source of "canned software." It's not a tutorial on programming itself (although you can learn a lot about that subject by following the examples and discussions here).

If you want a true tutorial on the HP-28S, then you should read An Easy Course In Using The HP-28S (see the last few pages in this book for more information on how to order this and other tutorial books).

So...

What You "Gotta" Do

- *You "gotta" know the basics of using and programming the HP-28S* This book is *not* a primer on the HP-28S.

- *You "gotta" key in, name, store and test some code (programs).* How much code? That depends on what you want to do with these tools.

- *You "gotta" invest a little time.* There are good reasons for the large number of pages you see here. It's just not realistic to expect to be able to look up your particular programming task in the index, flip to that page and instantly find the solution to your problem. You need to learn to program and learn how these program tools are meant to be used.

What You "Don't Gotta" Do

- *You "don't gotta" read everything in the book* (though it would be a good idea at least to look at the contents of each section (given on the opening pages of that section).

- *You "don't gotta" key in everything in the book;* you may never use some of this stuff. Only after you decide what you want to do will you know which routines will be helpful to get the job done.

- *You "don't gotta" be limited by this book.* If you're a proficient and interested programmer, then you can modify and expand upon these tools, inventing entirely new sets for your own use.

Reminders: Some HP-28S Basics

In case you need a refresher, here are a few reminders about the steps needed to key in, name, edit, store and use programs on your HP-28S:

"How Do I Load A Program?"

Consider the following program:

```
« → a b « a 'F' STO*
b F + →NUM "The answer
is " SWAP →STR + CLLCD
1 DISP » »
```

You *might* key it in this way:

```
[«] ■ [→] [LC] [A] [SPACE] [B] [«] [A] ['] [LC] [F] ['] ■ [STORE] STO*
[LC] [B] [SPACE] [LC] [F] [+] ■ [→NUM] ■ ["] [T] [LC] [H] [E] [SPACE] [A] [N] [S] [W] [E] [R] [SPACE]
[I] [S] [SPACE] ■ ["] ■ [SWAP] ■ [STRING] →STR [+] ■ [CONTRL] [NEXT] CLLCD
[1] DISP [ENTER]
```

You *might* key it in that way – or you might not – because there are many ways to do it.

Take a look now at some of the details here, by studying the first few keystrokes and what they mean:

« ■ → LC A SPACE B ...

The « keystroke signals the beginning of a program; it will always be the first key you press when entering a program. It also turns on the alpha cursor (■), so that certain typing aids will help to make loading the program easier.

For example, you don't need to *type* spaces in between the keystrokes « and ■ → – it's done automatically. This saves you many keystrokes, because *spaces must be keyed in exactly as they appear in a program listing*. Notice that most menu keys (e.g. STO) will also automatically put a space before and after the command names they type.

However, no spaces are needed around either the « or ' characters, because they are delimiters (like " , { , [, (, #, SPACE and NEWLINE), used by the HP-28S to delimit objects. The HP-28S puts spaces around the « simply to improve readability.

Finally, notice that *case is also significant*. The a and b *must* be lowercase, so you press LC before keying them in.

"How Do I Name It?"

Once you've keyed a program into the HP-28S (the example shown on page 8) so that it's on the stack, you'll need to give it a name by which you can call it and use it.

To give a program a name, the program must be on Level 1 of the stack. Then you need to put a unique and fairly descriptive name on the stack at Level 1 – thus pushing the program itself to Level 2. You must put single quotation marks (') around the name to prevent the HP-28S from trying to evaluate it when you ENTER it.

So, for example, to name the above program FRED (assuming the correct and complete program is now sitting at Level 1 of the stack), press ' F R E D STO

Keep in mind that this procedure will overwrite any object named FRED in the current directory, so you should take care that the name is unique!

"How Do I Change It?"

Now suppose you want to change the stored program (you've keyed it in wrongly or you want to enhance it). How can you do this?

You could recall the program (put it on the stack), edit it (with EDIT) and re-store it, or you can ■ VISIT it, which accomplishes all three of those things at once.

Type ' F R E D ■ VISIT. You can now move around the program with the cursor keys. And you can begin to edit right away, but keep in mind that typing anything new will *overwrite* (replace) the current contents unless you go into insert mode (INS), in which case what you type is inserted. This is often handier.

For example, you could use insert mode to add a NEWLINE to the end of the "The answer is " string. To do this, press ▼ ▼ ▼ ■ ▶ ◀. The cursor is over the quotation mark. Press ◆ to delete the space and INS ■ NEWLINE to insert the newline (notice that, while VISITing, NEWLINE characters actually cause a newline break in the program line).

Press ENTER now to accept the changes – or ON/ATTN will abort the edit without changing the program.

Then recall the program (press ' F R E D ■ RCL). Notice that the NEWLINE character is now represented by a ■. Notice also that this recalled program is a *copy* of the FRED program; changes to this copy won't affect the original unless you re-store it (' F R E D STO).

Press DROP to remove the program from the stack.

"How Do I Use It?"

Once you've loaded your program, to use it, all you need to do is call it by name: [F][R][E][D][ENTER].

If the stack was empty before you started, you'll get an error (Too Few Arguments) because this particular program needs input. The moral here is that you must always know the requirements of your program before you run it.

In this case, the program needs two real numbers on the stack and another real number named 'F'.

So start again: [0]['][F][STO][1][ENTER][2][ENTER][F][R][E][D][ENTER].

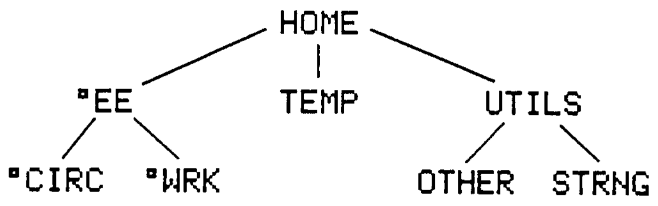
The answer is 2.

You should also notice that the name FRED appears on a menu key when you press [USER]. All things that you, the *user,* create are stored in *user* memory and showed to you by the *USER* menu.

Then, when you select FRED from that USER menu, this is the same as using any other command from any other menu: in immediate-execution mode (i.e. when you see either the □ cursor or no cursor at all), the name, FRED, is evaluated immediately; in alpha mode (the ■ cursor and α annunciator), the name is loaded into the command line –just as if you had typed it there. All this is exactly the way that built-in system commands work; a named program is quite literally an extension of your built-in catalog of commands!

"Where Else Can I Put It?"

For convenience, and organization, you can divide your USER menu (user memory) into *directories* – named areas partitioned off from the rest of user memory. The main directory is HOME, but you can create other sub-directories. A typical diagram of directories might be:



The directory you are in is the *current directory.* The directory containing your current directory is its *parent.* All directories sharing the same parent are called *sisters;* all the subdirectories of a directory are its *daughters.* If you were in directory UTILS above, the parent directory would be HOME; the sister directories, ▪EE and TEMP; the daughter directories, STRNG and OTHER. So, in this hypothetical set-up, you could put FRED into UTILS by first moving to UTILS (by pressing $\boxed{H}\boxed{O}\boxed{M}\boxed{E}\boxed{\text{ENTER}}\boxed{U}\boxed{T}\boxed{I}\boxed{L}\boxed{S}\boxed{\text{ENTER}}$) and then STOring FRED.

Here's why this matters: Typing the name of an object will evaluate that object only if it can be found either in the current directory or its parent (or grandparent or great-grandparent, etc.). *If your current directory is STRNG in the above diagram, you could successfully evaluate (run) your program, FRED, only if it were stored in STRNG, UTILS or HOME. If it were anywhere else, you wouldn't be able to "find" it.* Thus, since HOME is every directory's ultimate parent, an object "living at HOME" can be found and evaluated from *any* directory.

Notes On Using This Book

Before you key in anything, **read these important preliminaries:**

First, there are many ways to key things in on the HP-28S, and it's just impossible to show every method. This book simply cannot "read your mind" to know which menu or directory you're currently using when you want to call one of these tools – so it can't give you the most convenient set of keystrokes for your particular case.

In all programs and examples, therefore, rather than specifically telling you to press a *key* (e.g. ■ PURGE or DROP) or select a *menu item* (e.g. R→C P→R), you'll see all commands in generic form (spelled out as if you had typed them in): PURGE DROP R→C P→R, etc.

But keep in mind that, depending upon what you're doing, it might sometimes be more convenient to use special keys or menu items than to "type in" the commands character by character. That's up to you.

Secondly, a sample program description is shown on page 15. This is the general format for the description of each utility.

To make things easier to find, the routines are presented alphabetically (by NAME) *within their respective sections,* and there's also a complete index in the back, if you prefer.

Title:

A phrase that briefly tells you what the routine does.

Name (Checksum)

The name identifying the routine, followed by an integer to help you "proofread" the program after you have keyed it in and named it.

`« OBJECT »`

The program "code" itself, as it appears if you `RCL` it in `STD` display mode.

Summary:	A brief description of the routine's purpose and logic.
Example:	One or more simple examples to give you the general idea.
Inputs:	A list of acceptable types and locations of input objects.
Outputs:	A list of types and locations of output objects.
Errors:	A list of things that could go wrong due to machine conditions, bad input, etc.
Notes:	Other things you ought to know: Does this routine use (and therefore require) others from this book? How and when might you want to use it? Etc.

To help you check your accuracy when entering these program tools, each routine is listed with its *checksum,* a test value generated with the help of the CKSM routine (listed opposite, here).

Do This: Key in CKSM now (use the code listed on the opposite page).... Then STOre it: 'CKSM' STO.

Remember that the directory in which you store it (where you're "located" when you STO) will limit the memory locations from which you can "call" it – limited only to those locations "at" or "below" it on a directory tree.

Question: How do you know if a utility routine is keyed in properly?

Answer: You key in the routine's *name* (using ' marks), then use CKSM to test it. For example, to test whether or not you keyed in and modified FRED properly back on pages 8-11, you do this:

'FRED' CKSM <u>Correct Result</u>: 252400

If you get an *incorrect* result, you know there's a mistake in the routine. If so, then edit it ('FRED' ■VISIT), find and fix your typo(s), re-store the corrected version (ENTER), and repeat the test.

Important Conclusion: CKSM can and should be used after keying in and storing ***any*** program in this book. It's your best protection against typos!

Proofread A Named Object:

CKSM (1040278)

```
« 2 32 ^ RCLF → N F
S « N RCL STD HEX 64
STWS 45 SF 48 CF
→STR N →STR + 0 1 3
PICK SIZE FOR I OVER
I DUP SUB NUM I * +
DUP F MOD SWAP F /
IP + NEXT SWAP DROP
S STOF » »
```

Summary: CKSM (checksum) checks for "typos" by computing a unique integer for a named object.

Example: Problem: Test whether you can correctly key in CKSM:
Solution: Key in *and name* the CKSM routine...then use it "on itself:" 'CKSM' CKSM ENTER
Result: (if all is well) 1040278

Inputs: Level 1 – a name – the name of an object.

Outputs: Level 1 – an integer – the checksum.

Errors: Bad Argument Type will occur if the input is not a name (or Undefined Name if it's undefined). **Other errors** can occur if a typo in the CKSM program causes it to actually crash before returning a checksum.

Notes: CKSM is most generally useful in the HOME directory.

A few more details to bear in mind:

- Whenever you see the object types required for inputs and outputs, remember that symbolic expressions may also be allowable (to be sure, check the documentation for each routine).

  For example, a "real number" can mean either literally a real number value or any object (such as an algebraic expression, for example) that can be *reduced to a real number* with the →NUM or EVAL commands. A similar argument applies for complex numbers, etc.

  Remember that the states of flag 35 (constants mode) and flag 36 (results mode) will directly affect whether an object will be reduced to an actual value! See pages 206-207 of your Owner's Manual if you need to refresh your memory of these modes.

- You can store any or all of these utility tools in HOME or any other directory (and occasionally, as with CKSM, you'll read a recommendation as to where it might be most useful).

  Just bear in mind that when you invoke them, they must be in *the current directory or in a directory that is a "direct ancestor" (i.e. a parent, grandparent, great-grandparent, etc.) of the current directory* – anywhere in the direct pathway back to HOME, which is a direct ancestor of all directories. You won't be able to find these if they're stored in sister or daughter (or "aunt," "niece" or "cousin") directories.

- These utility programs are collected in chapters according to subject. For the most part, each routine is either independent or uses others from that same chapter. However, a few routines require the use of others from different chapters. Admittedly, this isn't necessarily the most convenient when you're keying in and testing specific routines, but it will lead to their better efficiency of execution and memory usage once they're properly stored. To make this easier, moreover, the ordering of the chapters has been arranged so that if you proceed through the book, keying in all the utilities in the order presented, no routine will require any other from any other chapter that you have not already keyed in.

  Furthermore, the ordering of the chapters makes some attempt to proceed logically – along the lines of increasing object complexity – beginning with mechanical stack manipulations, then to real numbers, then to complex numbers, etc. Hopefully, then, even the presentation of this book (as well as its contents, of course!) will help to reinforce and remind you once again of the idea of the HP-28S as a toolbox full of tools – to help you build even bigger and better tools!

Chapter 1

Stack Utilities

These routines provide quick and reliable ways to do certain manipulations, operations and tests on the HP-28S stack.

As shown in the following list, the 19 programs are organized into three logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they are presented together.

For a more in-depth discussion of the uses of these utilities, see page 40, immediately following these program listings.

Exchange Levels M And N

EXCH (333939)

```
« →NUM SWAP →NUM IF
DUP2 > THEN SWAP END
→ A B « A ROLL B
ROLL SWAP B ROLLD A
ROLLD » »
```

Summary: EXCH exchanges the contents of any two given stack Levels. The Level indices works like Level arguments for functions such as ROLL, indicating the stack Levels of objects *before* the arguments were placed on the stack, and the resulting modified stack assumes those Levels once again after the manipulation is complete. The Level indices may be given in either order. Any fractional portions of the indices are rounded before use. An index less than 1 causes no action to be taken.

Examples: STD 1 2 3 4 1 3 EXCH <u>Result</u>: 1 4 3 2

Inputs: Level *(n*+2) – any object – an object to be exchanged.
Level *(m*+2) – any object – an object to be exchanged.
Level 2 – any object that evaluates to a real number, *m* – one of the Levels to be exchanged.
Level 1 – any object that evaluates to a real number, *n* – the other Level to be exchanged.

Outputs: Levels 1 to *n* – the modified stack contents.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects or if the specified object Levels don't exist.
`Bad Argument Type` will occur if the Level-1 object does not evaluate to a real number.

Notes: None.

Perform A QuickSort Of Levels M – N

QSRT (5482233)

```
« → R L « L R + 2 /
IP PICK L R → X I J
« DO WHILE I PICK X
< REPEAT I 1 - 'I'
STO END WHILE J PICK
X > REPEAT J 1 + 'J'
STO END IF I J >
THEN J ROLL I ROLL
SWAP I ROLLD J ROLLD
END IF I J ≥ THEN I
1 - 'I' STO J 1 +
'J' STO END UNTIL I
J < END IF L J >
THEN J L QSRT END IF
I R > THEN R I QSRT
END » » »
```

Summary: QSRT sorts the specified stack levels. The objects in the stack must be orderable (i.e., they must be either real numbers, binary integers or strings). The resulting stack Levels are arranged in descending order (proceeding from lowest Level to highest Level).

Examples: `STD 6 4 7 5 3 8 1 1 7 QSRT`
<u>Result</u>: `1 3 4 5 6 7 8`

```
STD 5 4 3 2 1 1 5 QSRT
```
Result: `1 2 3 4 5`

```
STD 5 4 3 2 1 2 4 QSRT
```
Result: `5 2 3 4 1`

Inputs: Levels 3 to (*n*+2) – the objects to be sorted.
Level 2 – a real number, *m* – the lowest stack Level to be sorted.
Level 1 – a real number, *n* – the highest stack Level to be sorted.

Outputs: Levels 1 to (*n*-*m*+1) – the original stack with the specified levels sorted.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects or fewer objects than specified by input Levels 1 or 2, or if the Level-2 object is greater than the Level-1 object.
`Bad Argument Type` will occur if either the Level-1 or Level-2 object fails to reduce to a real number, or if any of the specified stack levels contain objects that are unorderable.

Notes: `QSRT` is useful for creating lists and arrays whose elements are arranged in ascending order.

Reverse The Order Of Levels 1 – 3

REV3 (11552)

```
« SWAP ROT »
```

Summary: REV3 reverses the order of the bottom three stack Levels (1, 2, and 3).

Examples: `STD 1 2 3 REV3`
Result: `3 2 1`

Inputs: Levels 1, 2 and 3 – any objects – the objects whose order is to be reversed.

Outputs: Level 3 – the previous Level-1 object.
Level 2 – the previous Level-2 object.
Level 1 – the previous Level-3 object.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects.

Notes: REV3 is generally useful for many stack manipulation needs in programming and in constructing larger data objects.

Reverse The Order Of Levels 1 – N

REVN (219414)

```
« →NUM → L « IF L 1
> THEN 1 L FOR I I
ROLL NEXT END » »
```

Summary: REVN reverses the order of the specified stack levels. The Level indicator number works like the level argument for functions such as ROLL: it indicates the stack Levels of objects *before* REVN's argument was placed on the stack, and the resulting modified stack assumes those Levels once again after the manipulation. Any fractional portion of the Level index is rounded before use. A rounded Level index less than 2 causes no action.

Examples: STD 1 2 3 4 4 REVN <u>Result</u>: 4 3 2 1

Inputs: Levels 2 to (n+1) – any objects.
Level 1 – any object that evaluates to a real number – the Level index.

Outputs: Levels 1 to n – the previous objects in reversed order.

Errors: Too Few Arguments will occur if the stack is empty or has fewer arguments than specified in Level 1. Bad Argument Type will occur if the Level 1 object does not evaluate to a real number.

Notes: None.

Roll Stack Levels 1 – N Down A Given Distance

ROLDN (356864)

```
« →NUM SWAP →NUM → N
L « IF N 1 > L 1 >
AND THEN 1 N START L
ROLLD NEXT END » »
```

Roll Stack Levels 1 – N Up A Given Distance

ROLLN (350463)

```
« →NUM SWAP →NUM → N
L « IF N 1 > L 1 >
AND THEN 1 N START L
ROLL NEXT END » »
```

Summary: ROLDN performs ROLLD the specified number of times. ROLLN performs ROLL the specified number of times.

Examples: STD 1 2 3 4 5 6 7 8 9 0 10 4
ROLDN <u>Result</u>: 7 8 9 0 1 2 3 4 5 6

STD 1 2 3 4 5 6 7 8 9 0 10 4
ROLLN <u>Result</u>: 5 6 7 8 9 0 1 2 3 4

Inputs: Levels 3 to $(n+2)$ – any objects – the objects to be rolled down or up.
Level 2 – any object that evaluates to a real number – the number of Levels, *n*, to be rolled.
Level 1 – any object that evaluates to a real number – the number of times to roll.

Outputs: Levels 1 to *n* – the rolled stack.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects or fewer objects than specified by the Level-2 input object.
`Bad Argument Type` will occur if the Level-1 and Level-2 objects do not reduce to real numbers.

Notes: You can use `ROLDN` and `ROLLN` in many useful ways. For example, `DEPTH DUP 2 / ROLDN`, or `DEPTH DUP 2 / ROLLN` swaps the upper and lower halves of the stack.

A general program for this might be `SWAPN` (`61426`):

```
« → N « N →NUM DUP 2
/ ROLDN » »
```

or

```
« → N « N →NUM DUP 2
/ ROLLN » »
```

`SWAPN` takes the argument at Level 1 to be the total number of Levels to be manipulated, then divides and swaps that much of the stack.

Combine Levels 1 – N With A Binary Operation

MERGE (118950)

```
« →NUM 1 - → F N « 1
N START F EVAL NEXT
» »
```

Summary: MERGE takes a binary operation and repeatedly applies it to stack Levels 1 and 2. The effect is to combine all of the specified stack Levels using the given function.

Example: `STD 1 2 3 4 5 « * » 5 MERGE`
Result: `120`

Inputs: Levels 3 to (n+2) – any objects.
Level 2 – a program or user-defined function – the procedure to be used to merge all the specified stack Levels.
Level 1 – any object that evaluates to a real number, n – specifying the top Level to be combined.

Outputs: Level 1 – an object – the result of the repeated operation.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects or if the Level-1 input object refers to a non-existent stack Level.
`Bad Argument Type` will occur if the Level-1 input object does not evaluate to a real number, or if the Level-2 program is incompatible with the specified argument.

Notes: MERGE is designed to use a *binary* operation – a program that takes two objects from the stack and returns only one. Other types of programs can be used, but the results are unpredictable.

Add An Object To Level N

STADD (70602)

```
« → N L « L « N + »
ST.OP » »
```

Divide Level N By An Object

STDIV (71905)

```
« → N L « L « N / »
ST.OP » »
```

Multiply Level N By An Object

STMUL (72277)

```
« → N L « L « N * »
ST.OP » »
```

Subtract An Object From Level N

STSUB (72146)

```
« → N L « L « N - »
ST.OP » »
```

Summary: STADD adds the given object to the specified Level. STDIV divides the specified Level by the given object. STMUL multiplies the specified Level by the given object. STSUB subtracts the given object from the specified Level. The Level index assumes the Levels of objects *before* the arguments were placed on the stack, and those Levels are restored again afterwards. If the Level index has a fractional portion, it is rounded before use. A Level index less than 1 causes no action.

Examples:

```
STD 1 2 3 6 3 STADD     Result: 7 2 3
STD 5 6 7 3 2 STDIV     Result: 5 2 7
STD 3 4 5 0 2 STMUL     Result: 3 0 5
STD 7 8 9 4 3 STSUB     Result: 3 8 9
```

Inputs: Level (*n*+2) – any object for which the operation is defined – the first operand.
Level 2 – any object for which the operation is defined – the second operand.
Level 1 – a real number, *n* – the Level of the operation.

Outputs: Level *n* – an object – the result of the operation

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects or the indexed Level is empty. `Bad Argument Type` will occur if the Level-1 object does not reduce to a real number, or if the objects at Levels 2 and (*n*+2) are incompatible for the operation.

Notes: None.

Perform An Operation On Level N

ST.OP (193463)

```
« →NUM → f.. l.. «
IF l.. 0 > THEN l..
PICK f.. EVAL l..
STOST END » »
```

Store An Object In Level N

STOST (595892)

```
« →NUM → N L « IF L
0 > THEN IF DEPTH L
1 - == THEN N L
ROLLD ELSE L ROLL
DROP N L ROLLD END
END » »
```

Summary: ST.OP performs the specified operation only on the given stack Level. STOST copies the contents of Level 2 to the given stack Level, overwriting the previous contents. The Level index works like the level argument for functions such as ROLL: it indicates the stack Levels of objects *before* the argument was placed on the stack. Any fractional portion of the Level index is rounded before use. A Level index less than 1 causes no action. STOST will not store into a non-existent stack Level except the first empty Level.

Examples: `STD 1 2 3 « 1 + » 3 ST.OP`
Result: `2 2 3`

`STD 'A' 'B' 'C' 1 3 STOST`
Result: `1 'B' 'C'`

Inputs: Level (*n*+2) – any object – the object to be operated upon or overwritten.
Level 2 – a program or user-defined function (for `ST.OP`) – the operation to be used, or (for `STOST`) any object – the object to be stored.
Level 1 – any object that evaluates to a real number, *n* – the Level index.

Outputs: Level *n* – the newly-modified or newly-stored object.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects or if the specified object Level doesn't exist, or (for `STOST`) if it is not the lowest-numbered empty Level.
`Bad Argument Type` will occur if the Level-1 object does not evaluate to a real number.

Notes: The operation used in `ST.OP` must take only one argument and return only one result, or else the stack may be hopelessly disordered. The local names, `f..` and `l..`, were chosen for `ST.OP` to reduce the chances of conflicts when operations such as `« STR→ »` are applied to strings. Therefore, avoid using `f..` and `l..` as global names in your own programming. `ST.OP` uses `STOST`.

Is Level N Equal To An Object?

STET? (215195)

```
« →NUM → N L « IF L
0 > THEN L PICK N ==
→NUM ELSE 0 END » »
```

Is Level N Not Equal To An Object?

STNE? (213356)

```
« →NUM → N L « IF L
0 > THEN L PICK N ≠
→NUM ELSE 0 END » »
```

Is Level N Greater Than Or Equal To An Object?

STGE? (212726)

```
« →NUM → N L « IF L
0 > THEN L PICK N ≥
→NUM ELSE 0 END » »
```

Is Level N Greater Than An Object?

STGT? (210262)

```
« →NUM → N L « IF L
0 > THEN L PICK N >
→NUM ELSE 0 END » »
```

Is Level N Less Than Or Equal To An Object?

STLE? (213092)

```
« →NUM → N L « IF L
0 > THEN L PICK N ≤
→NUM ELSE 0 END » »
```

Is Level N Less Than An Object?

STLT? (210579)

```
« →NUM → N L « IF L
0 > THEN L PICK N <
→NUM ELSE 0 END » »
```

Summary: STET?, STNE?, STGE?, STGT?, STLE?, and STLT? all compare the contents of Level N with the given object. In each of these tests, if the answer to the question posed is "yes," a 1 (true) is returned. Otherwise, a 0 (false) is returned.

The Level index for each of these tests works like the level argument for functions such as ROLL: it indicates the stack Levels of objects *before* the test's argument was placed on the stack, and the resulting modified stack assumes those Levels once again before returning the result of the test. If the Level index has a fractional portion, it is rounded before use. Level numbers less than 1 will cause the test to return 0.

Examples:

```
STD 1 2 3 2 3 STET?   Result: 1 2 3 0
STD 1 2 3 1 3 STET?   Result: 1 2 3 1

STD 1 2 3 2 3 STNE?   Result: 1 2 3 1
STD 1 2 3 1 3 STNE?   Result: 1 2 3 0

STD 1 2 3 4 3 STGE?   Result: 1 2 3 0
STD 2 3 2 2 STGE?     Result: 2 3 1

STD 1 2 3 0 3 STGT?   Result: 1 2 3 1
STD 2 3 2 2 STGT?     Result: 2 3 0

STD 1 2 3 0 3 STLE?   Result: 1 2 3 0
STD 2 3 2 2 STLE?     Result: 2 3 1
STD 1 2 3 4 3 STLT?   Result: 1 2 3 1
STD 2 3 2 2 STLT?     Result: 2 3 0
```

Inputs: Level (*n*+2) – any object for whose type the specified test is defined – one of the objects to be compared.
Level 2 – any object for whose type the specified test is defined – the other object to be compared.
Level 1 – a real number, *n*, the stack Level to be tested against the given object.

Outputs: Level 1 – a real number (either 1 or 0) – the result of the test.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects or if the specified stack Level does not exist.
`Bad Argument Type` will occur if the Level-1 object does not reduce to a real number, or if the objects at Levels 2 and (*n*+2) are incompatible arguments for the specific test being made.
`Undefined Name` will occur if either Level contains an undefined name.

Notes: None.

Stack Utilities: A Discussion

The Main Idea

The stack deserves its own set of tools for several reasons: First of all, these utilities are tools to help manipulate HP-28S data objects – and the stack *is* a data object.

Secondly, the stack is the intermediate for almost everything; it is *the* work area of the HP-28S. All manual calculations and most other operations affect or occur on the stack, and decomposed objects place their contents onto the stack for further manipulations with stack-related commands. Therefore, new stack commands also extend your ability to manipulate other objects.

Finally, because it's such a workhorse, the stack has been designed for high efficiency – it's fast. Thus, programs like QSRT use it – instead of directly accessing an array or list – to gain speed.

Where To Put These Programs

As always, to be accessible, these utilities must be in a directory that is listed in your current PATH. The easiest way to ensure that this is the case is to place each of the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

Tracking stack objects through long and/or complex operations can be a real chore. Sometimes the best strategy is to take an object from the stack, do an operation on the stack, then return the object afterward. In these cases, local variables are better solutions for managing and keeping track of objects, and so all of these stack utilities except REV3 use local variables to store their arguments so that they're out of the way when they're not needed but easily recallable when they are.

These routines are all quite straightforward – except for QSRT, which is rather large and logically complex. It is a classic implementation of a *recursive* Quicksort: the program sorts to a point, then checks if the complete data set has been sorted. If not, it simply adjusts the indices and then "calls itself" to sort some more. Use of recursion rather than iteration helped to keep the routine as small as it is, though an iterative implementation may further enhance its speed.

Errors And Error Recovery

Each of these tools is designed to generate an error when invalid input is entered – rather than continue and generate garbage outputs. When inputs are questionable (e.g., negative numbers for stack Levels), these utilities act similarly to the built-in stack commands (arguments are ignored or treated as 1, whichever makes more sense). When errors do occur, the stack is usually disrupted, and since the only way to restore it then is with the UNDO command, it's wisest to keep UNDO mode (in the MODES) menu) *active* whenever you these utilities.

How You Might Use These Utilities

These tools are extremely generic. That is, they are so basic as to be useful in many different situations.

QSRT, being relatively large and very generally useful, is called by several other programs in this book. Most of the rest of the routines are small enough that, rather than have many other programs call them, the actual program steps have been incorporated in the other programs.

The Manipulations routines are intended to extend the built-in stack manipulation commands of the HP-28S. EXCH is a generalization of SWAP, allowing you to swap any two stack levels. ROLDN and ROLLN extend ROLLD and ROLL, respectively, providing a method to repeat the action, thereby "scrolling" the stack in the specified direction.

Reversing stack elements can also be considered an extension of SWAP (which reverses the order of stack Levels 1 and 2). REV3 extends this idea to the bottom three stack Levels, and REVN reverses the bottom *n* Levels.

Finally, since you can consider all stack manipulations to be *ordering* the stack in some fashion, QSRT provides a method of sorting a portion of the stack into descending order. Note that QSRT is the only one of these manipulation routines that cares what the actual contents of the stack is; all the others simply move objects, but QSRT requires that the objects be orderable.

The Operations routines, with the exception of MERGE, consider the stack to be a collection of storage locations for which they provide storage and storage arithmetic operations. The basic four operations from the STORE menu (STO+, STO-, STO* and STO/) are reproduced for any arbitrary Level of the stack. Simple storage to any Level is provided by STOST (and note that you already have the generalized analog to RCL in the built-in PICK operation).

The Tests routines are the most straightforward: They simply extend the idea of the tests ==, ≠, ≥, >, ≤, and < to other Levels of the stack besides Levels 1 and 2.

Chapter 2

Real Number Utilities

These routines provide quick and reliable ways to do certain manipulations with real numbers.

As shown in the following list, the 9 programs are organized into four logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 60, immediately following these program listings.

Find The Greatest Common Divisor Of Two Positive Integers:

GCD (314696)

```
« → A B « A IP ABS
→NUM B IP ABS →NUM
WHILE OVER MOD SWAP
DUP REPEAT END DROP
» »
```

Find The Least Common Multiple Of Two Positive Integers:

LCM (123623)

```
« → A B « A IP ABS
→NUM B IP ABS →NUM
DUP2 GCD / * » »
```

Summary: GCD finds the Greatest Common Divisor of two positive integers. LCM returns the Least Common Multiple of two positive integers. The Greatest Common Divisor of two integers is the largest integer by which both numbers can be divided evenly. The Least Common Multiple of two integers is the smallest integer which is a multiple of both numbers. Negative arguments are converted to positive and fractional portions are truncated.

Examples:

`9 6 GCD`	Result: 3
`946 1462 GCD`	Result: 86
`1492 1776 GCD`	Result: 4
`8.5 -2 GCD`	Result: 2
`4 8 LCM`	Result: 8
`23 15 LCM`	Result: 345
`40 15 LCM`	Result: 120
`8.5 -2 LCM`	Result: 8

Inputs: Level 2 – a real number.
Level 1 – a real number.

Outputs: Level 1 – an integer – the GCD or LCM of the inputs.

Errors: `Too Few Arguments` will occur if there are fewer than 2 arguments on the stack.
`Bad Argument Type` will occur for inputs other than real numbers.
`Undefined Name` will occur if either of the arguments contains an undefined name.

Notes: Both `GCD` and `LCM` use local variables to allow their use within algebraic objects. `LCM` uses `GCD`.

Find The Remainder Of An Integer Division:

RMD (62702)

```
« → X Y « X IP Y IP
DUP2 / IP * - » »
```

Summary: RMD finds the remainder of an integer division.

Real numbers are allowed as inputs, but any fractional portions of those inputs are truncated.

Examples:

19 3 RMD	Result:	1
45 16 RMD	Result:	13
8.3 4.5 RMD	Result:	0
-6 4 RMD	Result:	-2

Inputs: Level 2 – a real number – the dividend.
Level 1 – a real number – the divisor.

Outputs: Level 1 – a real number – the remainder of the division.

Errors: Too Few Arguments will occur if there are fewer than 2 arguments on the stack.
Bad Argument Type will occur for inputs other than real numbers, names and algebraic objects.
Undefined Name will occur if an undefined name is input and symbolic results (flag 36) mode is not set.

Notes: While not strictly necessary, RMD uses local variables so that it can be used within algebraic objects.

RMD will return an algebraic expression if either of its inputs are symbolic and symbolic results mode is set. For example:

```
'A'  'B'  RMD
```

<u>Result</u>:

```
'IP(A)-IP(B)*IP(IP(A)/IP(B))'
```

Find The Prime Factors Of An Integer:

FCTR (3387925)

```
« ABS IP DEPTH «
WHILE DUP 3 PICK /
DUP FP NOT REPEAT
SWAP DROP OVER ROT
ROT END DROP » → L D
« IF DUP 3 > THEN 2
SWAP D EVAL 3 ROT
DROP SWAP WHILE DUP
1 ≠ OVER √ IP 4 PICK
≥ AND REPEAT D EVAL
2 ROT + SWAP END
SWAP DROP IF DUP 1
== THEN DROP END END
DEPTH L - 1 + →LIST
» »
```

Summary: FCTR finds the prime factors of a given positive integer. Negative arguments are converted to positive, and any fractional portions are truncated.

Examples:

8 FCTR	Result: { 2 2 2 }
144 FCTR	Result: { 2 2 2 2 3 3 }
83 FCTR	Result: { 83 }
1042 FCTR	Result: { 2 521 }

```
18.5 FCTR    Result: { 2 3 3 }
-100 FCTR    Result: { 2 2 5 5 }
```

Inputs: Level 1 – an integer or real number – the number to be factored.

Outputs: Level 1 – a list of one or more integers – the factors of the original number.

Errors: `Too Few Arguments` will occur if there are no arguments on the stack.
`Bad Argument Type` will occur for inputs other than real numbers.
`Undefined Name` will occur if an undefined name object is used in the input.

Notes: To regenerate the factored number (i.e. perform the inverse operation), a routine like this might be useful:

```
« LIST→ 2 SWAP START
* NEXT »
```

Generate A List Of Prime Numbers:

PRMS (1443948)

```
« ABS IP SWAP ABS IP
IF DUP2 < THEN SWAP
END DUP 2 MOD NOT +
SWAP DEPTH → D « FOR
I 3 I √ WHILE DUP2 <
I 4 PICK / FP AND
REPEAT 2 ROT + SWAP
END > 'I' IFT 2 STEP
DEPTH D - 2 + →LIST
» »
```

Summary: PRMS generates a list of all prime numbers within the two given limits. Negative limits are converted to positive and fractional portions are truncated. The limits are included in the range and may be supplied in either order.

Examples:

```
1 5 PRMS          Result: { 1 3 5 }
20 10 PRMS        Result: { 11 13 17 19 }
-23 50.1 PRMS
Result: { 23 29 31 37 41 43 47 }
```

Inputs: Level 2 – a real number.
Level 1 – a real number.

Outputs: Level 1 – a list of 0 or more prime numbers.

Errors: `Too Few Arguments` will occur if there are fewer than 2 arguments on the stack.
`Bad Argument Type` will occur for arguments other than real numbers.

Notes: None.

Generate A Random Integer Within Given Limits:

IRAND (244362)

```
« → H L « L IP H IP
IF DUP2 > THEN SWAP
END 1 + OVER - RAND
* + IP » »
```

Generate A Random Real Number Within Given Limits:

RRAND (182849)

```
« → H L « L H DUP2
IF > THEN SWAP END
OVER - RAND * + » »
```

Summary: IRAND randomly generates an integer whose value is between two given real-valued limits. Only the integer portions of the limits will be used, and these are included in the range of possible results. The limits may be supplied in either order.

RRAND randomly generates a real number whose value is between two given, real-valued limits, which may be supplied in either order. The lower limit is included in the range of possible results; the upper limit is excluded.

Examples: `STD -5 2.3 IRAND`
Result: `-1` (or any integer from -5 to 2)

`STD 8 -8 IRAND`
Result: `-7` (or any integer from -8 to 8)

`STD -5 2.3 RRAND`
Result: `1.44333879186` (or any real number from -5 to 2.29999999999)

`STD .25 10 RRAND`
Result: `1.62765141542` (or any real number from .25 to 9.99999999999)

Inputs: Level 2 – a real number.
Level 1– a real number.

Outputs: Level 1 – a real number – the random integer or real.

Errors: `Too Few Arguments` will occur if there are fewer than 2 arguments on the stack.
`Bad Argument Type` will occur for arguments other than real numbers.

Notes: Both `IRAND` and `RRAND` use local variables so that they can be used within algebraic objects.

Round A Real Number To The Nearest Integer:

IRND (58939)

```
« → N « N .5 + FLOOR
→NUM » »
```

Summary: IRND returns the integer value nearest the input value. Fractional portions of exactly 0.5 are rounded up.

Examples:

1.5 IRND	Result: 2
122.38 IRND	Result: 122
-5.5 IRND	Result: -5

Inputs: Level 1 – a real number – the number to be rounded.

Outputs: Level 1 – a real number – the integer nearest the input value.

Errors: Too Few Arguments will occur if there is no argument on the stack.
Bad Argument Type will occur if the argument is other than a real number.
Undefined Name will occur if the argument contains an undefined name object.

Notes: IRND uses local variables so that it can be used within algebraic objects.

Bear in mind that numbers with absolute values greater than 10^{11} don't have fractional portions on the HP-28S, because all twelve of the machine's available digits are required for the integer portions of such numbers.

Round A Real Number To The Specified Decimal Place:

RRND (108147)

```
« → X N « N IP ALOG
DUP X * .5 + FLOOR
SWAP / » »
```

Summary: RRND returns a real number rounded to the specified decimal place. Decimal places to the right of the decimal point are specified with a positive integer; those to the left of the decimal point are specified with a negative integer.

Examples:

```
STD π →NUM 6 RRND       Result: 3.141593
STD -.892664 2 RRND     Result: -.89
STD 122.38 1 RRND       Result: 122.4
STD 1492 -2 RRND        Result: 1500
```

Inputs: Level 2 – a real number, name or algebraic object – the number to be rounded.
Level 1 – an integer, name, or algebraic object – the value specifying the decimal places to be rounded.

Outputs: Level 1 – a real number or algebraic object – the rounded number.

Errors: `Too Few Arguments` will occur if there are fewer than 2 arguments on the stack.
`Bad Argument Type` will occur for inputs other than real numbers, names or algebraic objects.
`Undefined Name` will occur if an undefined name object is used in the input and symbolic results mode (flag 36) is not set.

Notes: `RRND` uses local variables so that it can be used within algebraic objects.

`RRND` will return an algebraic expression if either of its inputs are symbolic and symbolic results mode is set. For example:

`'A' 'B' RRND` <u>Result</u>:

```
'FLOOR(ALOG(IP(B))*A
+.5)/ALOG(IP(B))'
```

Real Number Utilities: A Discussion

The Main Idea

The main intention of these real number utility routines is to extend the basic real number functions of your HP-28S in clear and useful ways. You should be able to imagine these programs listed as commands in the REAL menu – and use them to build other, more sophisticated programs – just like the HP-28S' built-in real number operations. Arguments are entered similarly, and results return similarly; there should be no surprises.

Where To Put These Programs

As always, to be usable, these utility programs must be in a directory that is listed in your current PATH (i.e. in your current directory or one of its parent directories). Of course, the easiest way to ensure this is to put the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

The algorithms behind the routines are quite straight-forward (i.e. you should be able to follow what's going on and how they work) – except for FCTR and PRMS:

If a number is divisible by 2 or by any odd integer between 3 and the square-root of the number, it's not prime. Both FCTR and PRMS use this fact and perform successive divisions of the odd integers from the lower limit to the square-root of the upper limit.

Notice also that FCTR creates and uses a "local subroutine:" It creates two local variables, L and D. L is used to store the argument taken from the stack – the number to factor. D, on the other hand, is used to store a postfix program that FCTR itself has put on the stack. Later on in the program, the routine is evaluated (D EVAL) at two separate points. The alternatives would have been either to reproduce the same program steps twice within the FCTR program, or to create a separate, globally-named program and call it by name from within FCTR. The first option wastes space, the second creates an otherwise useless named object to clutter up the USER menu.

Another design point to notice is that the routines that take upper and lower limits will do so in either order, thanks to these program steps:

```
IF DUP2 > THEN SWAP
END
```

In other words, if the Level 2 value is greater than the Level 1 value, swap them (DUP2 is necessary because > consumes its arguments).

The same thing could also have been accomplished with:

```
DUP2 > « SWAP » IFT
```

or

```
DUP2 MIN ROT ROT MAX
```

The first routine is basically a rewrite of the original routine, using the IFT syntax (choosing IF...THEN vs. IFT is largely a matter of personal preference). The second routine abandons conditional statements entirely: performing a MIN and then a MAX on the same arguments effectively puts them in proper order on the stack (DUP2 is necessary to copy the arguments for both the MIN and MAX functions).

As you can tell, the HP-28S gives you nothing if it doesn't give you options. IF...THEN...END was used in the routines in this chapter simply because it is the easiest to read and understand – which can be an important consideration.

Errors And Error Recovery

Consistent with the behavior of the built-in real number commands, these programs do very little error-checking in input or output. So when an error condition occurs, the program halts, displaying the cause of the error (and probably a stack full of garbage).

The best way to deal with this is to be sure that UNDO is enabled *before* using the routines (check ■ MODE NEXT to see if UNDO is selected), then use the UNDO command after the error. You will come to admire the elegant simplicity – and life-saving ability – of an active UNDO.

How You Might Use These Utilities

Fractional Math

The programs FCTR, GCD, LCM, PRMS and RMD are chiefly useful unto themselves: if you need a prime number within a certain range, use PRMS; the remainder of a division is found with RMD, etc.

However, a routine like GCD can be used to easily construct another, very useful routine, called RDC (63757) – "ReDuCe":

```
« DUP2 GCD SWAP OVER
/ ROT ROT / SWAP »
```

Taking the real numbers on Levels 1 and 2 of the stack to be the denominator and numerator of a fraction, respectively, RDC divides both numbers by their Greatest Common Denominator (GCD) and thereby reduces the fraction.

For example:

5 20 RDC	Result: 1 4	(5/20 reduces to 1/4)
6 8 RDC	Result: 3 4	(6/8 reduces to 3/4)

This ability to easily reduce a fraction suggests further possibilities, such as fractional addition. FADD (71482) adds two fractions whose numerators and denominators are on the stack, then reduces the resulting sum:

```
« → A B C D « D A *
C B * + B D * RDC » »
```

Try adding 1/2 and 3/4: `1 2 3 4 FADD` <u>Result</u>: `5 4` (i.e. 5/4)

Add 1/4 to this result: `1 4 FADD` <u>Result</u>: `3 2` (i.e. 3/2)

Think about how FADD takes four arguments as the numerators and denominators of two fractions in the following order:

4:	< numerator_1 >
3:	< denominator_1 >
2:	< numerator_2 >
1:	< denominator_2 >

And then it returns a single numerator and denominator. So, how hard would it be to create a complete set of fractional math routines? For example:

`1 2 1 4 FSUB`	<u>Result</u>:	`1 4`	(1/2 - 1/4 = 1/4)
`1 2 1 4 FMUL`	<u>Result</u>:	`1 8`	(1/2 x 1/4 = 1/8)
`1 2 1 4 FDIV`	<u>Result</u>:	`2 1`	(1/2 ÷ 1/4 = 2/1)
`1 2 FINV`	<u>Result</u>:	`2 1`	(this is *too* easy!)
`1 2 8 SCALE`	<u>Result</u>:	`4 8`	(1/2 scaled to "eighths" = 4/8)

Random Numbers

Random number generators are generally useful for allowing a program to generate unpredictable results. Suppose, for example, that you want to test a program or other tool with an unpredictable, random set of circumstances. Or suppose that you're creating a game program where you don't want the program to play the same way all the time. So you want to simulate the occurrence of unpredictable events, like a toss of a die or the choice of a card from a deck.

TOSS (12974):

```
        « 1 6 IRAND »
```

CARD (253795):

```
        « STD 1 13 IRAND
        →STR " " {
        "HEARTS" "SPADES"
        "DIAMONDS" "CLUBS" }
        1 4 IRAND GET + + »
```

Both TOSS and CARD use IRAND because there are whole (integer) numbers of die faces, card suits, and cards per suit. But what if you're not so constrained? What if you're inventing a game, where you need to place a player randomly on a 100x100-unit playing field – where fractional units down to 1/100th are allowed? Try PUTIT (93197):

```
        « 0 100 RRAND 2 RRND 0
        100 RRAND 2 RRAND R→C »
```

100 is used for the maximum value with the assurance that it will be included within the range because RRND will round 99.995 and up to 100.

Now consider the following alternative to PUTIT, called PLACE (69150). Consider why it works:

```
« 0 10000 IRAND 0
10000 IRAND R→C 100
/ »
```

And you might even test PLACE or PUTIT with the following program, PTEST (175117):

```
« CLLCD (0,0) PMIN
(100,100) PMAX 1 100
START PLACE PIXEL
NEXT »
```

Rounding Notes

You might even think that once you've set the correct display format (say, `2 FIX`, for dollars and cents), all results are properly rounded for you. *Don't you believe it!* For example, if you pay $1000/year for three years – in 36 identical monthly payments – how much will you pay in total? Common sense says, "$3000;" so does the HP-28S display:

`2 FIX 1000 12 / 36 *`   Result: `3000.00`

But that's not right. In reality, each *monthly* payment is rounded to the nearest penny: `2 FIX 1000 12 / 2 RRND` Result: `83.33`
Now find the *real* total payment: `36 *`   Result: `2999.88`

Rounding is independent of the display format! You could round to two digits and yet have the display show you, say, one digit: `2999.9`

Notice also this feature: `STD 1492 -2 RRND` Result: `1500`
Digits to the *left* of the decimal point are rounded for negative arguments; `RRND` effectively does `1492 10E-2 * IRND 10E2 *`

One more thing: `IRND` is *not* the same as the `IP` command. `IP` truncates its argument to form an integer, while `IRND` rounds it (and the half-integer always rounds *up*: `-3.5` rounds *up* to `-3`).

value	IP	IRND	value	IP	IRND
3.14	3	3	-3.14	-3	-3
3.5	3	4	-3.5	-3	-3
3.75	3	4	-3.75	-3	-4

Chapter 3

Complex Number Utilities

These routines provide quick and reliable ways to use alternative formats when working with complex numbers.

As shown in the following list, there are 7 programs, generally presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 78, immediately following these program listings.

Convert Two Real Numbers To 'M*e^(i*α)':

R→e (266109)

```
« →NUM SWAP →NUM
SWAP R→C C→R RCLF 36
SF 'i' ROT * 'e'
SWAP ^ ROT SWAP *
SWAP STOF »
```

Summary: R→e converts two real numbers into an expression of the form 'M*e^(i*α)', where M is the magnitude and α is the angle of the complex vector.

Example: Problem: Key in the number $5e^{i0.93}$.
Solution: `2 FIX 5 .93 R→e`
Result: `'5*e^(i*0.93)'`

Inputs: Level 2 – a real number – the magnitude, M.
Level 1 – a real number – the angle, α, *in radians.*

Outputs: Level 1 – an algebraic object – the complex expression.

Errors: `Bad Argument Type` will occur for non-real input values.

Notes: The angle (α) is *always* expressed in radians.

Convert Two Real Numbers To 'Re+i*Im':

R→i (293698)

```
« →NUM SWAP →NUM
SWAP R→C C→R RCLF 36
SF ROT ROT 'i' OVER
SIGN * SWAP ABS * +
SWAP STOF »
```

Summary: R→i converts two real numbers into an expression of the form 'Re+i*Im', where Re is the real portion and Im is the imaginary portion of the complex vector.

Example: Problem: Key in the number 3+i4.
Solution: `2 FIX 3 4 R→i`
Result: `'3+i*4'`

Inputs: Level 2 – a real number – the real portion, Re.
Level 1 – a real number – the imaginary portion, Im.

Outputs: Level 1 – an algebraic object – the complex expression.

Errors: `Bad Argument Type` will occur for non-real input values.

Notes: None.

Convert Two Real Numbers To '▫(M,α)':

R→▫ (192694)

```
« →NUM SWAP →NUM
SWAP R→C RCLF STD
"'▫" ROT →STR + STR→
SWAP STOF »
```

Summary: R→▫ converts two real numbers to a complex expression in the polar form '▫(M,α)', where M is the magnitude and α is the angle of the complex vector.

Example: Problem: Key in the number 5∠53.13°.
Solution: `2 FIX 5 53.13 R→▫`
Result: `'▫(5,53.13)'`

Inputs: Level 2 – a real number – the magnitude, M.
Level 1 – a real number – the angle, α, *in degrees.*

Outputs: Level 1 – an algebraic object – the complex expression.

Errors: `Bad Argument Type` will occur for non-real input values.

Notes: The angle (α) is *always* in degrees.

Polar Format Function:

▫ (91127)

```
« → M A « RCLF DEG M
A R→C P→R SWAP STOF
» »
```

Summary: ▫ is an *auxiliary function* that converts two real numbers, M and α, where M is the magnitude and α is the angle of a vector (in degrees), into a rectangular complex number of the form (Re, Im). This function is used whenever a complex number that has been formatted in the polar notation '▫(M, α)' must be evaluated to an actual numerical value.

Example: None needed.

Inputs: Level 2 – a real number – the magnitude, M.
Level 1 – a real number – the angle, α, *in degrees.*

Outputs: Level 1 – a complex number – in rectangular format.

Errors: Bad Argument Type will occur for non-real input values.

Notes: The angle (α) is *always* in degrees.

Convert (Re, Im) To 'M*e^(i*α)':

→e (271112)

```
« (1,0) * →NUM C→R
R→C RCLF 36 SF RAD
SWAP R→P C→R 'i'
SWAP * 'e' SWAP ^ *
SWAP STOF »
```

Convert (Re, Im) To 'Re+i*Im':

→i (224825)

```
« (1,0) * →NUM C→R
RCLF 36 SF ROT ROT '
i' OVER SIGN * SWAP
ABS * +  SWAP STOF »
```

Convert (Re, Im) To '∡(M, α)':

→∡ (244542)

```
« (1,0) * →NUM C→R
R→C RCLF SWAP DEG
STD R→P →STR "'∡"
SWAP + STR→ SWAP
STOF »
```

Summary: →e converts a complex number from any evaluable format to this *exponential* format: `'M*e^i*α'`, where M is the magnitude of the vector in the complex plane and α is the directional angle, *in radians.*

→i converts a complex number from any evaluable format to a *rectangular algebraic* format: `'Re+i*Im'`, where Re is the real portion and Im is the imaginary portion of the complex vector.

→∡ converts a complex number from any evaluable format to this *polar* format: `'∡(M,α)'`, where M is the magnitude of the vector in the complex plane and α is the directional angle, in degrees.

Examples: Problem: Find $4e^{i\pi/4} + 1.5e^{i0.32}$ – in exponential format.

Solution:
```
2 FIX π 4 / i * EXP 4 * RAD
(1.5,.32) P→R + →e
```
Result: `'5.38*e^(i*0.66)'`

Problem: Find (-1.3+i0.5) + (4.4-i2.3). Again, express the answer in `'M*e^i*α'` format.

Solution:
```
2 FIX (-1.3,.5) '4.4-i*2.3'
+ →e
```
Result: `'3.58*e^(i*(-0.53))'`

Problem: Find $4e^{i\pi/4} + 1.5e^{i0.32}$ – in rectangular algebraic format.

Solution:
```
2 FIX π 4 / i * EXP 4 * RAD
(1.5,.32) P→R + →i
```
Result: `'4.25+i*3.30'`

Problem: Find (-1.3+i0.5) + (4.4-i2.3). Again, express the answer in `'Re+i*Im'` format.

Solution: `2 FIX (-1.3,.5) '4.4-i*2.3' + →i`

Result: `'3.10-i*1.80'`

Problem: Find $4e^{i\pi/4} + 1.5e^{i0.32}$ – in degree polar format.

Solution: `2 FIX π 4 / i * EXP 4 * RAD (1.5,.32) P→R + →∠`

Result: `'∠(5.38,37.82)'`

Problem: Find (-1.3+i0.5) + (4.4-i2.3). Again, express the answer in `'∠(M,α)'` format.

Solution: `2 FIX (-1.3,.5) '4.4-i*2.3' + →∠`

Result: `'∠(3.58,-30.14)'`

Inputs: Level 1 – any object that can be reduced to a real or complex number by →NUM.

Outputs: Level 1 – an algebraic expression – the formatted comoplex number expression. →e , →i , and →∠ generate algebraic expressions with no unevaluated variables ('e' and 'i' are symbolic constants). Thus these expressions behave as a *symbolic complex numbers.*

Certain conversions involving combinations of →e, →i and →∠ may produce odd results because of rounding errors.

For example, the sequence (0, 2) →e →i will return '8.7332384626 4E-12+i*2', instead of 'i*2', because →e gives '2*e^(i*1.57079632679)', instead of the more accurate '2*e^(i*π/2)'. Note that you can regard numbers smaller than 10^{-10} as 0; often, in fact, you can use RND and a proper display mode (2 FIX or 4 SCI, etc.) to actually round to 0.

Errors: Bad Argument Type will occur if an input is not reducible to a real or complex number.

Notes: R→P and P→R are other conversion commands that may be useful (and more familiar) to you in working with complex numbers. But be careful! HP-28S system commands (except for P→R) expect complex numbers to be in – or reduce to – *rectangular* form only. If you use any of those system commands on complex numbers in *polar* form, you'll get incorrect results!

Complex Number Utilities: A Discussion

The Main Idea

The main intention of these complex number utility routines is to offer some convenient conversions between commonly used real and complex number formats – formats that were not built into the HP-28S. You can do any sort of calculation with complex numbers in these formats that you can do with the built-in complex format (`(Re, Im)`).

Where To Put These Programs

As always, to be usable, these utility programs must be in a directory that is listed in your current PATH (i.e. in your current directory or one of its parent directories). Of course, the easiest way to ensure this is to put the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

Symbolic vs. Numeric Calculations

The HP-28S is not just a calculator. It is also a *symbolic expression solver*. For this reason, you must also consider not only how to *perform* a calculation, but what *form* your input is in and what *form* you want your output to take.

For example, it's clear what should happen when you do simple arithmetic (say, add 1 and 2, like so: 1 2 +). You should get one real-valued result, 3. But what if you want to add the number 5 to the number Z_A (like so: ZA 5 +)? What kind of answer do you want – a *numeric* or a *symbolic* result?

To more fully illustrate these questions, consider some examples:

- You want to build a symbolic expression 'A*B/(A+B)'. How?
- You have two variables 'A' and 'B', containing numbers. How would you calculate A x B / (A + B) and get a numeric result?
- You want to evaluate the expression, 'A*B/(A+B)', for many different values of 'A' and 'B'. What do you do?
- You want to find the numeric value of π+4. How?

The possibilities are endless. And these are all quite common and valid needs that you might have – *and they are perfectly reasonable things to expect your HP-28S to do.* You must simply "know the recipes...."

To work confidently with your HP-28S, you need to be familiar with two of its *system flags* (internal status indicators which you can vary back and forth between two opposite states). They are: **Flag 35** (constants mode) and **Flag 36** (results mode).

Flag 35 (constants mode) determines what kind of result you'll get when any of the system constants (i, e, MAXR, MINR and π) are evaluated.

For example, when flag 35 is *set* (which you do like this: 35 SF), and you perform 'π' EVAL, you'll get the *symbolic* result: 'π'. On the other hand, when flag 35 is *clear* (35 CF) performing 'π' EVAL will give you 3.14159265359, the *nearest numerical equivalent.*

Flag 36 (results mode) determines whether functions will reduce name objects to their numeric contents.

For example, when flag 36 is *set* (via 36 SF), the commands 'A' 1 + give the result 'A+1'. Setting flag 36 effectively tells functions to "leave all name objects alone." But when this flag is *clear* (36 CF), functions *will* evaluate all name objects so as to return numeric results: 2 'A' STO 'A' 1 + yields 3, for instance. Thus, if you're working with flag 36 clear, you should remember that name objects will function as places in which to store numbers – not as abstract variables. Indeed, names that *don't* contain numeric objects will cause errors when operated on under this flag setting.

Note also that symbolic constants are simply name objects with specially reserved names. Therefore, if flag 36 is clear, the machine "has permission" when using functions to reduce *all* names to their numeric equivalents – including the specially reserved names. In effect, then, when you clear flag 36, you are overriding the setting of flag 35.

The settings of these two flags is, of course, up to you, but *you'll have the most flexibility if you generally leave them both set,* thus preserving all names and constants in your results. Remember that when you have a symbolic expression that you need to reduce to its numeric equivalent, you can always do so easily with EVAL or →NUM.

Errors And Error Recovery

Each of the complex number format conversion routines is designed to cause an error when invoked with invalid arguments – rather than generating erroneous results. However, no provisions have been made to clean up the stack before the program halts at such an error, so "garbage" may be left on the stack at that point. In any event, the input argument(s) will almost certainly have been consumed and the stack disheveled. It's probably best, therefore, to use these tools with UNDO *active,* because UNDO is certainly the most convenient way to restore the stack after an error.

Complex Number Calculations

The six conversion routines extend the HP-28S complex number commands by allowing you to construct *complex number expressions.* They *always* return *symbolic* results; they were designed to do so – to allow you to generate complex numbers in alternate, *symbolic* formats. However, just as with any other arithmetic, the results mode (flag 36) does affect how these expressions are *combined.* Watch:

```
2 FIX 36 CF   (turns off symbolic results)
1 2 R→i       Result: '1+i*2'
5 53 R→∠      Result: '∠(5,53)'
+             Result: (4.01,5.99)
```

That's what you'd get with symbolic results turned *off* (and you would then need to use →i, →e, or →∠ to put it back into a symbolic format, if that's what you prefer). Now repeat this with symbolic results set:

```
36 SF         (turns on symbolic results)
1 2 R→i       Result: '1+i*2'
5 53 R→∠      Result: '∠(5,53)'
+             Result: '1+i*2+∠(5,53)'
```

Note that all symbolic expressions and constants are preserved – just as you'd expect with flag 36 set. All six routines produce results that behave this way. Then, if you want the numeric value, you use →NUM.

Keying In Complex Numbers

As you may know, there are several different complex number formats in widespread use among different disciplines. This is irrelevant, of course, if you never encounter any format other than the one you now know and love. But in case you do meet with a different format, it's good to be able to convert back and forth – and to combine numbers in different formats, reducing the results to your preferred format.

The first thing to realize with the HP-28S is that it has its own formats and conventions. The basic complex number on the HP-28S is the Cartesian-coordinate (also called *rectangular*) form: `(3,4)`, where the `3` is the real component and the `4` is the imaginary component. *All the HP-28S's calculations assume that a complex number is in this rectangular form* (except those operations specifically intended to convert from another form to this one).

All would be well if this rectangular form were the only one anybody ever used, but life is never that simple. There's also a *polar* form often used in engineering and it is sometimes acceptable to the HP-28S: `(5,53.13)`, where the `5` is the magnitude of the complex vector, and the `53.13` is the angle it makes with the real axis. This angle may be measured in degrees or radians (the HP-28S will assume one or the other according to the current angle mode).

The problem is, there's no way to tell by looking at your calculator's display in which form a complex number is being expressed. Indeed, experienced users often develop the habit of using one form or the other, converting between them only when necessary.

But the HP-28S makes life even easier than that. It can create and use *symbolic expressions* (including the symbolic constants `'e'` and `'i'`) as easily as numbers, thus allowing the creation of other common complex number formats: `'3+i*4'` and `'5*e^(i*0.93)'`

These are the machine's renderings of the *algebraic rectangular* form, R+iC, and the *exponential* form, $Me^{i\alpha}$; they will reduce to numeric values if you use `→NUM`. Note that the HP-28S uses the mathematician's **i**, rather than the engineer's **j**, to represent $\sqrt{-1}$. Note also that the exponential form is valid only for α in *radians.*

Unfortunately, there's yet *another* common complex number format – the *polar degree* format (the engineers' favorite), which is M∠ α, with α in degrees. The problem with this is that it's not at all convenient on the HP-28S, because the machine lacks the ∠ character. However, with a slightly different format, you can still present the same information: `'▪(5,53.13)'`. Here the `5` is the magnitude, and the `53.13` is the angle, α, *in degrees.* This is the polar degree format.

Of course, all three of these alternate complex number forms would be merely interesting novelties without some convenient methods for creating and using them on your HP-28S. So this chapter provides three commands that exactly parallel the HP-28S's `R→C` ("real-to-complex") command – corresponding to each of these three symbolic formats:

```
'Re+i*IM'     'M*e^(i*α)'     '▪(M,α)'

   R→i            R→e            R→▪
```

Just like `R→C`, each of these commands takes its components from stack levels 1 and 2 and leaves the resulting expression on level 1:

```
2 FIX 1 1 R→i      Result: '1+i'
2 √ π 4 / R→e      Result: '1.41*e^(i*0.79)'
2 √ 45 R→∠         Result: '∠(1.41,45)'
```

Of course, these commands are only for your convenience; you can always key in these expressions directly. For example, to create the expression `'1+i'`, you would press ['][1][+][LC][I][ENTER].

Math With Mixed Complex Formats

The real beauty of these three complex formats is that they're *fundamentally equivalent*. That is, they'll all reduce to the HP-28S's Cartesian rectangular form when you use `→NUM` (indeed, if you apply `→NUM` to the expressions on page 85, you'll get `(1.00,1.00)` for each one of them – try it)! *This means* that you can enter complex numbers in *any* of the three alternate formats, perform calculations on them, then reduce the final result with one simple `→NUM`!

For example:

`1 1 R→i` Result: `'1+i'`
`3 34 R→▪` Result: `'▪(3,34)'`
`+ 2 /` Result: `'(1+i+▪(3,34))/2'`
That's getting ugly.
`2 FIX →NUM` Result: `(1.74,1.34)`
That's much better.

Yes, but what's this result in polar-degrees format? After all, if you're used to working in a certain format, it would be ideal to get the final result in that format, too – no?

`→▪` Result: `'▪(2.20,37.52)'`

Note that this command is *not* `R→▪` but `→▪`. Keep in mind that `R→▪` (along with `R→i` and `R→e`) will always combine real numbers to *form* a complex expression where there was none before. This is useful mainly for *entering* complex numbers. By contrast, `→▪` (and `→i` and `→e`) will actually convert an *existing* complex number from any other allowable format to the desired format.

Thus, you have *four* basic complex number conversion routines: →i, →∠, →e, and →NUM. The built-in →NUM command is included, because after all, it too will convert any of the other formats to a certain desired format – the HP-28S's own Cartesian rectangular format!

So, if in that last example, you really wanted to see the results in, say, algebraic rectangular format, it wasn't necessary to use →NUM at all:

```
1 1 R→i 3 34 R→∠ +
2 /              Result: '(1+i+∠(3,34))/2'
→i               Result: '1.74+i*1.34'
```

Try another one:

```
1 45 R→∠         Result: '∠(1,45)'
LN               Result: 'LN(∠(1,45))'
→i               Result: '6.40E-13+i*0.79'
```

→i (like any of the other conversion routines) automatically evaluates the expression before converting it to Re+iIm format. So any function such as LN, that can take a complex expression as an argument, will be evaluated by the conversion routines.

Notice that the result of LN(1∠ 45°) has a very small real portion – small enough to be considered rounding error and replaced by 0. To do this, you could use IM to return the imaginary portion as a real number. Then reenter the result as the imaginary portion of a complex number with a real portion of 0:

```
IM               Result: 'IM(6.40E-13+i*0.79)'
0 SWAP
R→i              Result: 'i*0.79'
```

Solving A Complex Expression For A Complex Result

Sometimes you'll need to do algebra with complex numbers, forming complex expressions containing *variables:* `'A+i*B-∠(1,45)'`

You cannot build such an object just by using the conversion tools; you must key in the variable names by hand. Here's one method:

```
'A' 'i' 'B' * +      Result: 'A+i*B'
1 45 R→∠ -           Result: 'A+i*B-∠(1,45)'
```

Of course, you can always perform further operations to build a more complicated expression, but *only after the variables have been given values* can you get its numeric result or convert its format:

```
8 'A' STO 12 'B' STO →i   Result: '7.29+i*11.29'
```

You can also use the SOLV menu to conveniently store values into the variables in such an expression and then evaluate the expression (with the EXPR=, LEFT= or RT= commands). For example, to evaluate the expression, `'INV(INV(ZA)+INV(ZB))'`, with these values...

ZA	1	'2-i*3'	'∠(2,45)'
ZB	3	2.3	'∠(100,36)'

do this: `'INV(INV(ZA)+INV(ZB))'` SOLV STEQ SOLVR

Then: `1` ZA `3` ZB EXPR=   Result: `0.75`

Then: `'2-i*3'` `ZA` `2.3` `ZB` `EXPR=` `→i`
Result: `'1.47-i*0.58'`

Then: `'ZA' PURGE '∡(2,45)'` `ZA` `'∡(100,36)'` `ZB`
`EXPR=` `→∡` Result: `'∡(1.96,44.82)'`

Note, however, that since the "SOLVer" capability itself does not extend to complex numbers, you cannot generally use it directly to *solve* for the values of complex variables in complex expressions. Instead, you must use `ISOL` – and probably some other algebraic rearrangement tools.

Before using `ISOL`, you should become comfortable with the use of the solution mode flag (flag 34):

```
34 SF 'A^2=9'  'A'  ISOL     Result: 3.00
34 CF 'A^2=9'  'A'  ISOL     Result: 's1*3'

34 SF 'A^3=9'  'A'  ISOL     Result: 2.08
34 CF 'A^3=9'  'A'  ISOL
                    Result: 'EXP(2*π*i*n1/3)*2.08'
```

With flag 34 *clear,* the expression for the *general solution* in each case is given. Arbitrary integers are represented by `n1`, `n2`, etc. Thus, replacing `n1` above with any integer will yield a valid answer. Similarly, `s1`, `s2`, etc., represent arbitrary sign multipliers (±1), so that replacing `s1` above with either `1` or `-1` will yield valid results. With flag 34 set, you get the *principal value* as a result. This value is what you get when you substitute `0` for all arbitrary integers and `1` for all arbitrary signs.

Chapter 4

Vector Utilities

These routines provide quick and reliable ways to do certain type and dimension conversions and formatting of vectors.

As shown in the following list, the 8 programs are organized into three logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 100, immediately following these program listings.

Convert A Vector From 3-Dimensional To 2-Dimensional:

→2D (619608)

```
« → A « A EVAL IF
DUP SIZE { 3 } ≠
THEN { } 1 GET END {
2 } RDM IF A TYPE
DUP 6 == SWAP 7 ==
OR THEN A STO END »
»
```

Convert A Vector From 2-Dimensional To 3-Dimensional:

→3D (619788)

```
« → A « A EVAL IF
DUP SIZE { 2 } ≠
THEN { } 1 GET END {
3 } RDM IF A TYPE
DUP 6 == SWAP 7 ==
OR THEN A STO END »
»
```

Summary: →2D converts a 3-element vector into a 2-element vector (the third element is lost). →3D converts a 2-element vector into a 3-element vector (the third element is given the value of zero). If the vector is named, and the name is used, the result vector will be stored in it.

Examples:

`[ 1 2 3 ] →2D`	Result: `[ 1 2 ]`
`[ 1 2 ] →3D`	Result: `[ 1 2 0 ]`

Inputs: Level 1 – any object that evaluates to a 3-element/2-element vector – the vector to be converted.

Outputs:: Level 1 – if the input object was a name containing a 3-element/2-element vector, nothing is returned, but the resulting 2-element/3-element vector is stored in that name. Otherwise, that resulting 2-element/3-element vector is returned.

Errors: `Too Few Arguments` will occur for an empty stack. `Bad Argument Type` will occur if the input object does not reduce to a vector.
`Bad Argument Value` will occur if the input contains an undefined name or a vector of other than 3 elements (for →2D) or 2 elements (for →3D).

Notes: None.

Convert A Vector From 'I+J+K' Format:

IJK→ (312258)

```
« EVAL RCLF STD SWAP
→STR SWAP STOF
[ 1 0 0 ]
[ 0 1 0 ]
[ 0 0 1 ]
→ I J K « STR→ →NUM
» »
```

Convert A Vector To 'I+J+K' Format:

→IJK (658244)

```
« RCLF → f. « 36 SF
EVAL IF DUP SIZE { 3
} ≠ THEN { } 1 GET
END ARRY→ DROP 'K' *
ROT 'I' * ROT 'J' *
ROT + + COLCT f.
STOF »
```

Summary: IJK→ converts an algebraic expression, containing a linear combination of 'I', 'J' and 'K', into a vector. →IJK converts a 3-element vector into an algebraic expression that is a linear combination of 'I', 'J' and 'K'.

Examples: `STD 'I+J+K' IJK→` Result: `[ 1 1 1 ]`
`STD '(3.5*I-2.25*K)*2+15*I' IJK→`
Result: `[ 22 0 -4.5 ]`

`STD [ 1 2 1 ] →IJK` Result: `'I+2*J+K'`
`STD [ -3 0 1 ] →IJK` Result: `'-(3*I)+K'`

Inputs: Level 1 – an algebraic object/vector – the vector whose format is to be converted.

Outputs: Level 1 – a vector/algebraic object – the vector in the converted format.

Errors: `Too Few Arguments` will occur for an empty stack. `Undefined Name` will occur for `IJK→` if the argument is an expression containing an undefined name. `Invalid Dimension` will occur if an operation in the expression would involve multiplying two vectors. `Bad Argument Type` will occur if (for `IJK→`) there is any operation in the expression which is used on `'I'`, `'J'` or `'K'` and is undefined for vectors; or (for `→IJK`), if the input object does not evaluate to an array. `Bad Argument Value` will occur with `→IJK` if the input has a valid SIZE but is not a 3-element vector.

Notes: `IJK→` simply evaluates the Level-1 object after assigning vector values to the three variables, `'I'`, `'J'` and `'K'`. It is therefore not very sensitive to erroneous inputs. Expressions such as `'π'`, for example, are simply evaluated and return no vector at all.

Convert A One-Column Array To A Vector:

A→V (732375)

```
« → A « A EVAL DUP
SIZE 2 GET IF 1 ≠
THEN { } 1 GET END
DUP SIZE 1 1 SUB RDM
IF A TYPE DUP 6 ==
SWAP 7 == OR THEN A
STO END » »
```

Convert A Vector To A One-Column Array:

V→A (638695)

```
« → A « A EVAL DUP
SIZE DUP IF SIZE 1 ≠
THEN { } 1 GET END 1
+ RDM IF A TYPE DUP
6 == SWAP 7 == OR
THEN A STO END » »
```

Summary: `A→V` converts a one-column array (i.e., `[[ 1 ][ 2 ][ 3 ]]`) to a vector. `V→A` converts a vector to a one-column array. If the input array/vector is stored in a name and the name is used, the resulting vector/array will be stored in that name.

Examples: `STD [[ 1 ][ 2 ][ 3 ]] A→V`
Result: `[ 1 2 3 ]`

`STD [ 1 2 3 ] V→A`
Result: `[[ 1 ][ 2 ][ 3 ]]`

Inputs: Level 1 – any object that evaluates to a column array/vector – the column array/vector to be converted.

Outputs: Level 1– if the input object was a name that contained a column-array/vector, nothing is returned, but the resulting column-array/vector is stored in that name. Otherwise, that resulting vector/column-array is returned.

Errors: `Too Few Arguments` will occur if the stack is empty.
`Bad Argument Type` will occur if the argument does not reduce to an array/vector.
`Bad Argument Value` will occur if the input object is an array containing more than one column (for `A→V`) or is not a vector (for `V→A`).

Notes: None.

Convert A Complex Number To A Vector:

C→V (283497)

```
« → A « A EVAL C→R 2
→ARRY IF A TYPE DUP
6 == SWAP 7 == OR
THEN A STO END » »
```

Convert A Vector To A Complex Number:

V→C (681991)

```
« → A « A EVAL IF
DUP SIZE { 2 } ≠
THEN { } 1 GET END
ARRY→ DROP R→C IF A
TYPE DUP 6 == SWAP 7
== OR THEN A STO END
» »
```

Summary: C→V converts a complex number to a two-element vector. If the complex number is named and the name is used, the resulting vector is stored in that name. V→C converts a two-element vector to a complex number. If the vector is named and the name is used, the resulting complex number is stored in that name.

Examples: `STD (1,2) C→V` <u>Result</u>: `[ 1 2 ]`
`STD [ 1 2 ] V→C` <u>Result</u>: `(1,2)`

Inputs: Level 1 – any object that evaluates to a complex number/ 2-element vector – the complex number/2-element vector to be converted.

Outputs: Level 1 – if the input object was a name containing a complex number/2-element vector, nothing is returned, but the resulting 2-element vector/complex number is stored in that name. Otherwise, that resulting 2-element vector/complex number is returned.

Errors: `Too Few Arguments` will occur for an empty stack.
`Bad Argument Type` will occur if the argument does not reduce to a complex number/vector.
`Bad Argument Value` will occur with `V→C` if the input vector does not have exactly 2 elements.

Notes: None.

Vector Utilities: A Discussion

The Main Idea

The main purpose of these routines is to provide conversion utilities between the different vector formats available to the HP-28S. The "vectors" referred to here are the mathematically defined sort, and not simply the vector object type provided by the HP-28S.

For example, in two dimensions, these are mathematically equivalent for most operations, when used as vectors:

```
(1,2)     [ 1 2 ]     [[ 1 ] [ 2 ]]
```

And in three dimensions:

```
[ 1 2 3 ]

[[ 1 ] [ 2 ] [ 3 ]]

'I+2*J+3*K'
```

But be careful! Not all vector-type operations work with every object type. For example, although many common vector-type operations will work with complex numbers (e.g. +, −, ABS, NEG, scalar multiplication), not all will (e.g. CROSS and DOT). And there are more operations defined for complex numbers than for vectors (e.g. multiplication of two complex numbers).

Be warned also that the algebraic form (`'I+2*J+3*K'`) is a valid vector representation only if the symbols `'I'`, `'J'` and `'K'` have *no* associated values, *or* if those values are `[ 1 0 0 ]`, `[ 0 1 0 ]` and `[ 0 0 1 ]`, respectively

In the former case, the algebraic expressions may be combined to form mathematically correct expressions (with *symbolic* unit vectors). In the latter case, evaluation of the expressions will yield correct HP-28S vector objects – because those values are indeed the required unit vectors. *Any other values stored in either* `'I'`, `'J'`, *or* `'K'` *will yield invalid results when evaluated.*

Where To Put These Programs

As always, to be accessible, these utilities must be in a directory that is listed in your current PATH. The easiest way to ensure that this is the case is to place each of the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

Arrays Vs. Vectors

Both mathematically and as objects on the HP-28S, vectors can be considered to be a type of array: In math, a vector is a one-dimensional array and may either be a row-vector or a column-vector. However, on the HP-28S, a *vector object* is always a *column-vector,* represented by numbers within single brackets (`[ 1 2 3 ]`). The dimension of a vector object, as returned by the `SIZE` command, is `{ n }`, indicating that the vector is one-dimensional and has *n* elements.

This representation of a column-vector as a *vector object* is simply intended to make life easier for you. The alternative form of column vector is `[[ 1 ] [ 2 ] [ 3 ]]`, where the single column is represented as a list of one-element rows. However, on the HP-28S, this representation is an *array object* – not a vector object. Accordingly, it is represented as a list of numbers within *double* brackets (`[[ 1 2 3 ]]`), and its dimension is returned as `{ 1 n }` where *n* is the number of elements.

This would all be merely interesting trivia if it were not that *certain built-in HP-28S commands function only on vector objects and not on column-vector arrays* (`CROSS`, for example). On the other hand, certain "array-ish" commands refuse to take vector objects as arguments (e.g. `TRN`). For this reason, `A→V` and `V→A` have been included in these utilities to allow you to easily convert between these forms.

Calculations "In Place"

All of the vector utilities except →IJK and IJK→ allow you the option of providing a named object as the argument. In that case, the resulting object will be restored in that name object as the end of the calculation – and nothing will be returned to the stack. This feature will work either on global or local name objects and is intended to be analogous to the working of the storage arithmetic commands (see the HP-28S's STORE menu).

Although similar, these utilities lack one of the major advantages of the built-in storage math: Those built-in STORE menu commands will perform their calculations "in-place" – on top of the contents of the current array – thereby taking up less storage space than recalling the contents of the named objects, combining them, then overwriting the original named object. These utilities must use the latter method.

Errors And Error Recovery

Each of these tools is designed to generate an error when invalid input is entered – rather than continue and generate garbage outputs. When inputs are questionable (e.g., negative numbers for stack Levels), these utilities act similarly to the built-in stack commands (arguments are ignored or treated as 1, whichever makes more sense). When errors do occur, the stack is usually disrupted, and since the only way to restore it then is with the UNDO command, it's wisest to keep UNDO mode (in the MODES) menu) *active* whenever you these utilities.

How You Might Use These Utilities

All of the vector utilities provide convenient means to convert between equivalent (or nearly equivalent) forms of vectors – that's their purpose. One pair of routines, however, provides conversion between a *numeric* form and an *algebraic* form, and that algebraic form, in and of itself, opens up new vistas for vector operations.

The algebraic (or "symbolic") form of a vector is simply a linear combination of the symbolic unit vectors **i**, **j** and **k**. As such, all forms of mathematical and symbolic operations can be performed on the vector expression.

You must be careful, however, to perform only those mathematical operations *that are defined for vector-type objects.* The HP-28S will allow you to perform many operations on a symbolic expression (such as trigonometric and logarithmic functions) – operations which are in no way defined for vectors. And the resulting object will be algebraically correct if the names 'I', 'J' and 'K' are associated with real or complex objects, *but not for vector objects.* This fact will manifest itself when/if you apply IJK→ to a symbolic expression: Bad Argument Type will be the only result.

Try some examples:

```
STD '4*I' '2*I' '3*K' + +
                         Result: '4*I+(2*I+3*K)'

COLCT                    Result: '6*I+3*K'
IJK→                     Result: [ 6 0 3 ]
→2D                      Result: [ 6 0 ]
→3D                      Result: [ 6 0 0 ]
→IJK                     Result: '6*I'
(2,3) C→V →3D →IJK       Result: '2*I+3*J'
2 / COLCT                Result: '.5*(2*I+3*J)'
EXPAN COLCT              Result: 'I+1.5*J'
IJK→ →2D V→C             Result: (1,1.5)
```

Unfortunately, most of the vector-oriented commands of the HP-28S will *not* take symbolic arguments. Thus, you cannot "cross" two symbolic vectors using the built-in command, CROSS. You can, however, define *similar* commands, such as CROS (59588), like this:

```
« IJK→ SWAP IJK→
SWAP CROSS →IJK »
```

This version will take *either* numeric or symbolic arguments and return a symbolic cross product vector.

You can see how tempting it might be to define a whole set of similar commands to make your HP-28S a little more useful with symbolic expressions, no? Go ahead and do so on your own, as you wish....

Chapter 5

Array Utilities

These routines provide convenient, "canned" methods for building, editing and using arrays in the HP-28S.

As shown in the following list, the 30 programs are organized into three logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 152, immediately following these program listings.

Name	Function	Page
ASORT	Sort An Array By Element	140
ASRTC	Sort An Array By Column	142
ASRTR	Sort An Array By Row	142

Miscellaneous Operations

Name	Function	Page
AI→N	Convert An Array's Index List To A Numeric Index	144
AN→I	Convert A Numeric Index To An Array's Index List	144
AOP	Perform An Operation On Each Element Of An Array	146
AOPC	Perform An Operation On Each Column Of An Array	146
AOPR	Perform An Operation On Each Row Of An Array	147
APOS	Find The Position Of A Specified Real Element Within An Array	150

Get A Column From An Array:

AGETC (127941)

```
« →NUM → N « EVAL
DUP SIZE 2 2 SUB 0
CON N 1 PUT * » »
```

Get A Row From An Array:

AGETR (660007)

```
« →NUM → N « EVAL
DUP SIZE IF DUP SIZE
1 == THEN DROP N GET
{ 1 1 } →ARRY ELSE 1
1 SUB 1 SWAP + 0 CON
N 1 PUT SWAP * END »
»
```

Summary: AGETC extracts the specified column-vector from the given array. AGETR extracts the specified row-array from the given array or vector. If the row/column selector is beyond the dimensions of the source array, an error is reported. Any fractional portion of the row/column selector is rounded.

Examples: `STD [[ 1 2 3 ][ 4 5 6 ]] 3 AGETC`
Result: `[ 3 6 ]`

`'A' « 2 √ » AGETC`
Result: (assuming that array `'A'` is defined, you'll get a vector – column 1 of the array `'A'`)

`STD [[ 1 2 3 ][ 4 5 6 ]] 2 AGETR`
Result: `[[ 4 5 6 ]]`

`'B' « 2 √ » AGETR`
Result: (assuming that array `'B'` is defined, you'll get an array – row 1 of `'B'`)

Inputs: Level 2 – any object that evaluates to an array or vector. Level 1 – any object that evaluates to a real number – the row/column index.

Outputs: Level 1 – an array or vector – the row or column, respectively.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if the arguments don't evaluate to their required types.
`Bad Argument Value` will occur if the column index is out of range.

Notes: None.

Create An Array By Duplicating An Object

ARPT (462793)

```
« →NUM → A N « A
EVAL IF DUP TYPE NOT
THEN 1 →LIST END N
CON IF A TYPE 6 ==
THEN A STO END » »
```

Summary: ARPT creates an array by repeating a single number. The dimensions of the resulting array are specified either by an integer index, a list index, or an array. The index determines the type of array/vector object returned: Integers and single element lists return vectors, while 2-element lists specifying numbers of rows and columns will return the corresponding arrays. An array-type index returns an array of the same dimensions. All real-number indices are rounded before use.

Examples: `STD { 2 3 } 0 ARPT`

Result: `[[ 0 0 0 ][ 0 0 0 ]]`

`STD 5 10 ARPT`

Result: `[ 10 10 10 10 10 ]`

`STD 1.5 1 ARPT`

Result: `[ 1 1 ]`

Inputs:	Level 2 – any object that evaluates to a real number, list, array or vector – the dimensions of the desired array. Level 1 – any object that evaluates to a real number – the value to be repeated throughout the array being created.
Outputs:	Level 1 – If the Level-2 object was a name containing a valid number, list, vector or array, nothing is returned, but the resulting array or vector is stored in that name. Otherwise the resulting array or vector is returned.
Errors:	`Too Few Arguments` will occur if the stack contains fewer than 2 objects. `Bad Argument Type` will occur if either argument is not of its prescribed type. `Undefined Name` will occur if the Level-2 object is an undefined name.
Notes:	None.

Get A Subarray From An Array:

ASUB (4745605)

```
« EVAL { } + ROT
EVAL ROT EVAL { } +
ROT LIST→ 1 == 1 IFT
ROT LIST→ 1 == 1 IFT
→ M C D A B « M DUP
SIZE IF DUP SIZE 1
== THEN 1 + RDM ELSE
DROP END DUP DUP 'M'
STO { A B } GET DROP
{ C D } GET DROP IF
C A < D B < OR THEN
[ 1 ]
2 GET END A C FOR I
B D FOR J M { I J }
GET NEXT NEXT { 'C-A
+1' } D B - 1 + IF
DUP 1 ≠ THEN + ELSE
DROP END →ARRY » »
```

Summary: ASUB extracts a sub-array from the given array. Two indices are required: the upper left element of the sub-array, and the lower right. A real number may also be used as an index for a vector, or for a 2-dimensional array, in which latter case, it will be taken to mean the first column of that row in the array.

Examples: `STD [[ 1 2 3 ][ 4 5 6 ]] { 1 2 } { 2 3 } ASUB`

Result: `[[ 2 3 ][ 5 6 ]]`

`STD [ 1 2 3 4 5 ] 3 4 ASUB`

Result: `[ 3 4 ]`

Inputs: Level 3 – any object that reduces to an array or vector – the source array.
Level 2 – an object that reduces to a list or real number – the index of the upper left corner of the sub-array.
Level 1 – an object that reduces to a list or real number – the index of the lower right corner of the sub-array.

Outputs: Level 1 – an array or vector – the extracted sub-array.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects.
`Bad Argument Type` will occur if any of the stack objects do not evaluate to their prescribed types.
`Bad Argument Value` will occur if either of the indices is out of bounds.

Notes: Of course, `ASUB` can be used to extract individual rows and columns from an array, but `AGETR` and `AGETC` are probably more convenient for those specific tasks.

Decompose An Array Into Columns:

ARY→C (32254l)

```
« EVAL TRN ARRY→
LIST→ DROP → R C « 1
R FOR I C →ARRY R I
- C * I + ROLLD NEXT
R » »
```

Summary: ARY→C decomposes the given array into its component column arrays (vectors), which are left on the stack (in order), along with a count of these vectors.

Example: `STD [[ 1 2 ][ 3 4 ][ 5 6 ]] ARY→C`
Result: `[ 1 3 5 ] [ 2 4 6 ] 2`

Inputs: Level 1 – any object that evaluates to an array – the array to be decomposed.

Outputs: Levels 2 to (*n*+1) – vectors – the array's columns.
Level 1 – a real number, *n* – the number of columns.

Errors: `Too Few Arguments` will occur for an empty stack.
`Bad Argument Type` will occur if the argument does not evaluate to an array (or `Invalid Dimension` will occur for a vector argument).
`Undefined Name` will occur if the argument contains an undefined name.

Notes: None.

Compose An Array By Columns:

C→ARY (4885879)

```
« →NUM .5 + IP → N «
N →LIST → L « L 1
GET EVAL SIZE LIST→
IF 2 == THEN DROP
END → R « { } 1 N
FOR I L I GET EVAL
DUP SIZE LIST→ 1 ==
1 IFT IF SWAP R ≠
THEN
[ 1 ]
TRN END IF 1 == THEN
ARRY→ LIST→ DROP
ELSE TRN ARRY→ LIST→
DROP * END →LIST +
NEXT LIST→ R / R 2
→LIST →ARRY TRN » »
» »
```

Summary: C→ARY creates an array from the given column-arrays, vectors, and/or arrays – combined column-wise (symbolic arguments will be evaluated). Input arrays/vectors must all have the same number of rows. An integer index must also be given to indicate how many stack items to combined. Any fractional portion of the index is rounded.

Examples: `STD [ 1 2 ] [ 3 4 ] [ 5 6 ] 3 C→ARY`
Result: `[[ 1 3 5 ][ 2 4 6 ]]`

`STD [[ 1 2 ][ 3 4 ]] DUP 2 C→ARY`
Result: `[[ 1 2 1 2 ][ 3 4 3 4 ]]`

Inputs: Levels 2 to (n+1) – any object that reduces to an array or vector – the objects to be combined.
Level 1 – any object that evaluates to a real number, n – the number of objects to be combined.

Outputs: Level 1 – an array – the newly-created array.

Errors: `Too Few Arguments` will occur if the stack is empty or there are fewer objects on the stack than are specified in Level 1.
`Bad Argument Type` will occur if any of the stack levels don't reduce to their respective object types.
`Bad Argument Value` will occur if the (rounded) index value is less than 1.
`Invalid Dimension` will occur if the given arrays and/or vectors do not all have the same row dimension.

Notes: None.

Decompose An Array Into Rows:

ARY→R (371489)

```
« EVAL ARRY→ LIST→ 1
== 1 IFT → R C « 1 R
FOR I { 1 C } →ARRY
R I - C * I + ROLLD
NEXT R » »
```

Summary: ARY→R decomposes the given vector or array into its component 1-row arrays, left on the stack in order, along an integer representing the total number of these rows.

Examples: `STD [[ 1 2 ][ 3 4 ]] ARY→R`
Result: `[[ 1 2 ]] [[ 3 4 ]] 2`

`STD [ 1 2 3 ] ARY→R`
Result: `[[ 1 ]] [[ 2 ]] [[ 3 ]] 3`

Inputs: Level 1 – any object that evaluates to a vector or array.

Outputs: Levels 2 to (*n*+1) – the array's component rows.
Level 1 – a real number, *n* – the number of components.

Errors: `Too Few Arguments` will occur if the stack is empty or if the argument is a vector.
`Bad Argument Type` will occur if the argument does not evaluate to a vector or array.

Notes: None.

Compose An Array By Rows:

R→ARY (2191223)

```
« →NUM .5 + IP → N «
N →LIST → L « L 1
GET EVAL SIZE 2 GET
→ C « { } 1 N FOR I
L I GET EVAL DUP
SIZE IF 2 GET C ≠
THEN
[ 1 ]
TRN END ARRY→ LIST→
DROP * →LIST + NEXT
LIST→ C / C 2 →LIST
→ARRY » » » »
```

Summary: R→ARY will create an array from the given arrays (symbolic arguments will be evaluated). The component arrays will be combined row-wise (input arrays must all have the same number of columns). An integer index must also be given to indicate how many stack items are to be combined. Any fractional portion of the index is rounded.

Examples: STD [[1 2]] [[3 4]] [[5 6]] 3 R→ARY
<u>Result</u>: [[1 2][3 4][5 6]]

```
STD [[ 1 2 ][ 3 4 ]] DUP 2 R→ARY
```
Result: `[[ 1 2 ][ 3 4 ][ 1 2 ][ 3 4 ]]`

Inputs: Levels 2 to (n+1) – any objects that reduce to arrays; the objects to be combined.
Level 1 – any object that evaluates to a real number, n – the number of objects to be combined.

Outputs: Level 1 – an array – the newly-composed array.

Errors: `Too Few Arguments` will occur if the stack is empty or there are fewer objects on the stack than are specified in Level 1.
`Bad Argument Type` will occur if any of the stack levels don't reduce to their respective object types.
`Invalid Dimension` will occur if the given arrays do not have the same column dimension.

Notes: None.

Convert An Array To A List:

A→L (183691)

```
« EVAL ARRY→ → D « D
LIST→ IF 2 == THEN *
END →LIST D » »
```

Summary: A→L converts the given array or vector into a list of its elements, in row-major order. A second list will also be returned, containing the size information from the original array so that the array can be reconstructed.

Examples: `STD [[ 1 2 ][ 3 4 ]] A→L`
Result: `{ 1 2 3 4 } { 2 2 }`

`STD [ 1 2 3 4 ] A→L`
Result: `{ 1 2 3 4 } { 4 }`

Inputs: Level 1 – Any object that evaluates to an array or vector – the array to be converted.

Outputs: Level 2 – a list – the elements of the original array.
Level 1 – a list – the original dimensions of the array.

Errors: `Too Few Arguments` will occur for an empty stack. `Bad Argument Type` will occur if the argument does not evaluate to an array or vector.

Notes: None.

Convert A List To An Array:

L→A (592057)

```
« EVAL SWAP EVAL → D
L « L SIZE D LIST→
IF 2 == THEN * END
IF ≠ THEN { } 1 GET
END L LIST→ DROP D
→ARRY » »
```

Summary: L→A converts the given list of numbers into an array or vector of the elements from the list (in row-major order). A second list must be given, containing the size information of the resulting array, i.e., `{ rows columns }`.

Examples: `STD { 1 2 3 4 } { 2 2 } L→A`
Result: `[[ 1 2 ][ 3 4 ]]`

`STD { 1 2 3 4 } { 4 } L→A`
Result: `[ 1 2 3 4 ]`

Inputs: Level 2 – any object that evaluates to a list of real numbers – the list to be converted.
Level 1 – any object that evaluates to a list – the list containing the dimensions of the desired array.

Outputs: Level 1 – an array or vector, depending on the specification – the object just converted from the input list.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if the arguments do not evaluate to lists.
`Bad Argument Value` will occur if the dimensions of the size list do not correspond to the number of elements in the element list.

Notes: None.

Delete A Column From An Array:

ADELC (32258З)

```
« → A R « A EVAL TRN
R ADELR TRN IF A
TYPE DUP 6 == SWAP 7
== OR THEN A STO END
» »
```

Delete A Row From An Array:

ADELR (2327790)

```
« →NUM .5 + IP → A R
« A EVAL DUP SIZE 1
R PUT GET DROP A
EVAL ARRY→ LIST→ 1
== 1 IFT → N M « N R
- M * →LIST M 1 +
ROLLD M DROPN LIST→
DROP N 1 - IF M 1 ≠
THEN M 2 →LIST END
→ARRY IF A TYPE DUP
6 == SWAP 7 == OR
THEN A STO END » » »
```

Summary: ADELC deletes the specified column from a given array. ADELR deletes the specified row from a given array or vector. The column/row number is rounded before use. If that rounded number is less than 1 or greater than the number of column/rows in the array, an error will occur. If the name of an array variable is used, the modified array is restored in the given name.

Examples:

```
STD [[ 1 2 3 ][ 4 5 6 ][ 7 8 9 ]] 2
ADELC
```

Result: `[[ 1 3 ][ 4 6 ][ 7 9 ]]`

```
'A' 5 ADELC
```

Result: (assuming the array 'A' is defined, it loses its fifth column but nothing is left on the stack.)

```
STD 3 IDN 'π' ADELC
```

Result: `[[ 1 0 ][ 0 1 ][ 0 0 ]]`

```
STD [[ 1 2 3 ][ 4 5 6 ][ 7 8 9 ]] 2
ADELR
```

Result: `[[ 1 2 3 ][ 7 8 9 ]]`

```
'A' 4 ADELR
```

Result: (assuming the array 'A' is defined, it loses its fourth row but nothing is left on the stack.)

```
STD [ 1 2 3 ] 2 ADELR
```

Result: `[ 1 3 ]`

Inputs: Level 2 – any object that evaluates to an array – the array to be edited.
Level 1 – any object that evaluates to a real number – the column/row number.

Outputs: Level 1 – if the input array is a name, nothing is returned, but the modified array is stored in that name. Otherwise the modified array is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than two objects.
`Invalid Dimension` will occur with `ADELC` if the Level-2 object is a vector.
`Bad Argument Type` will occur if the Level-2 object does not evaluate to an array (a vector is also OK for `ADELR`), or if the Level-1 object does not evaluate to a real number.
`Undefined Name` will occur if the Level-1 object contains an undefined name.
`Bad Argument Value` will occur with `ADELR` if you try to delete the last remaining row or if the row number is out of bounds.

Notes: `ADELC` uses `ADELR`.

Exchange Elements Within An Array:

AEX (695868)

```
« →NUM SWAP →NUM → A
M N « A EVAL DUP DUP
N GET SWAP M GET ROT
N ROT PUT M ROT PUT
IF A TYPE DUP 6 ==
SWAP 7 == OR THEN A
STO END » »
```

Summary: AEX exchanges any two elements of the given vector or array. The indices for the two elements may be either integers or lists. Any fractional portions of integer indices are rounded. If either index is beyond the valid dimensions of the array, an error occurs. If the array is named and the name is used, the modified array will be restored in the given name.

Examples: `STD [[ 1 2 ][ 3 4 ]] 1 2 AEX`
Result: `[[ 2 1 ][ 3 4 ]]`

`[[ 1 2 ][ 3 4 ]] { 1 1 } { 2 2 } AEX`
Result: `[[ 4 2 ][ 3 1 ]]`

`'A' 'X-1' { 2 Z } AEX`
Result: (assuming that X and Z contain real values, the array, 'A', is modified, but nothing is left on the stack.)

```
STD [ 1 2 3 ] 1 3 AEX
```
Result: `[ 3 2 1 ]`

Inputs: Level 3 – the array or vector – any object that evaluates to either an array or a vector.
Level 2 – any object that evaluates to a real number or a list of two real numbers – the index of one of the elements to be exchanged.
Level 1 – any object that evaluates to a real number or a list of two real numbers – the index of the other element to be exchanged.

Outputs: Level 1 – if the input array or vector is a name, nothing is returned, but the modified array or vector is stored in that name. Otherwise the modified array or vector is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than three objects.
`Bad Argument Type` will occur if the Level-2 object doesn't evaluate to an array or vector, or if the Level-1 object doesn't evaluate to a real number or a list.
`Undefined Name` will occur if the Level-3 object contains an undefined name.

Notes: None.

Exchange Columns Within An Array:

AEXC (411743)

```
« → A N M « A ARY→C
→LIST N M AEX LIST→
C→ARY IF A TYPE DUP
6 == SWAP 7 == OR
THEN A STO END » »
```

Exchange Rows Within An Array:

AEXR (414458)

```
« → A N M « A ARY→R
→LIST N M AEX LIST→
R→ARY IF A TYPE DUP
6 == SWAP 7 == OR
THEN A STO END » »
```

Summary: AEXC exchanges any two columns of the given array. AEXR exchanges any two rows of the given array. The indices for the two columns/rows may either be integers or (single-element) lists. Any fractional portions of integer indices are rounded. If either of the indices is beyond the dimensions of the array, an error occurs. If the array is named and the name is used, the modified array will be restored in the given name.

Examples: `STD [[ 1 2 ][ 3 4 ]] 1 2 AEXC`
<u>Result</u>: `[[ 2 1 ][ 4 3 ]]`

`STD [[ 1 2 3 ]] 1 3 AEXC`
<u>Result</u>: `[[ 3 2 1 ]]`

`STD [[ 1 2 ][ 3 4 ]] 1 2 AEXR`
<u>Result</u>: `[[ 3 4 ][ 1 2 ]]`

`STD [ 1 2 3 ] 1 3 AEXR`
<u>Result</u>: `[[ 3 2 1 ]]`

Inputs: Level 3 – any object that evaluates to an array.
Level 2 – any object that evaluates to a real number – the index of one of the columns/rows to be exchanged.
Level 1 – any object that evaluates to a real number – the index of the other column/row to be exchanged.

Outputs: Level 1 – if the input array is a name, nothing is returned, but the modified array is stored in that name. Otherwise the modified input array is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than three objects.
`Bad Argument Type` will occur if the Level-3 object does not evaluate to an array or if the Level-1 and Level-2 objects do not evaluate to real numbers or lists.
`Undefined Name` will occur if the Level-3 object contains an undefined name.

Notes: `AEXC` uses `ARY→C`, `C→ARY`, and `AEX`. `AEXR` uses `ARY→R`, `R→ARY`, and `AEX`.

Insert A Column Into An Array:

AINSC (626491)

```
« → A R V « A EVAL
TRN R V V SIZE LIST→
1 == 1 IFT 2 →LIST
RDM AINSR TRN IF A
TYPE DUP 6 == SWAP 7
== OR THEN A STO END
» »
```

Insert A Row Into An Array:

AINSR (2238455)

```
« EVAL TRN TRN SWAP
→NUM .5 + IP → A V R
« A EVAL ARRY→ LIST→
1 == 1 IFT → N M « N
R - 1 + M * →LIST V
ARRY→ DROP M 1 +
ROLL LIST→ DROP N 1
+ IF M 1 > THEN M 2
→LIST END →ARRY IF A
TYPE DUP 6 == SWAP 7
== OR THEN A STO END
» » »
```

Summary: AINSC inserts the given column into the given array. AINSR inserts the given row into the given array. If the column/row number is less than 1 an error is generated. A column/row index greater than the column/row-size of the array will cause the new columnrow to be added to the end of the array. The number of columns/rows in the inserted array and destination array must be equal. If the name of an array variable is used, the modified array is restored in the given name.

Examples: `[[ 1 3 ][ 4 6 ]] 2 [ 2 5 ] AINSC`
Result: `[[ 1 2 3 ][ 4 5 6 ]]`

`STD [ 1 2 3 ] 4 [[ 4 ]] AINSR`
Result: `[ 1 2 3 4 ]`

Inputs: Level 3 – any object that evaluates to an array.
Level 2 – any object that evaluates to a real number – the column/row index.
Level 1 – any object that evaluates to a vector or array – the column/row to be inserted.

Outputs: Level 1 – if the input array was a name, nothing is returned, but the modified array is stored in that name. Otherwise the modified array is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than two objects.
`Bad Argument Type` will occur if the input objects don't evaluate to their required types.

Notes: AINSC uses AINSR.

Overwrite A Subarray Onto An Array:

APUTA (6923463)

```
« EVAL SWAP EVAL →
A1 A2 B « A2 ARRY→
LIST→ 1 == 1 IFT 2
→LIST →ARRY DUP SIZE
{ 1 1 } SWAP LIST→
DROP R→C B { } +
LIST→ 1 == 1 IFT R→C
SWAP OVER + (1,1) -
SWAP C→R ROT C→R ROT
SWAP A1 EVAL ARRY→
LIST→ 1 == 1 IFT 2
→LIST →ARRY → N M A3
« FOR I N M FOR J
GETI A3 { I J } ROT
PUT 'A3' STO NEXT
NEXT DROP2 A3 » DUP
SIZE LIST→ DROP IF 1
== THEN { } + RDM
ELSE DROP END IF A1
TYPE DUP 6 == SWAP 7
== OR THEN A1 STO
END » »
```

Summary: APUTA puts the given sub-array into the given array, overwriting the contents of the array with the contents of the sub-array. An index specifies the position in the array at which the upper left corner of the sub-array will be located after the operation. This index may be a real number if the destination array is a vector. If the destination is a 2-dimensional array and the index is a real number rather than a list, it is taken to mean the first column of that row. The entire sub-array must fit into the destination array or an error will occur.

Examples: `STD [[ 1 2 3 ][ 4 5 6 ][ 7 8 9 ]] { 2 2 } [[ 0 0 ][ 0 0 ]] APUTA`
Result: `[[ 1 2 3 ][ 4 0 0 ][ 7 0 0 ]]`

`STD [ 1 2 3 4 ] 2 [ 0 0 ] APUTA`
Result: `[ 1 0 0 4 ]`

Inputs: Level 3 – any object that evaluates to an array – the destination array.
Level 2 – any object that evaluates to a list or real number – the index.
Level 1 –an object evaluating to an array– the sub-array.

Outputs Level 1 – if the Level-3 input was a name, nothing is returned, but the modified array is stored in that name. Otherwise, the modified array is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects.
`Bad Argument Type` will occur if any of the arguments fail to reduce to their prescribed values.
`Bad Argument Value` will occur if the index does not fall within the destination array.

Notes: None.

Overwrite A Column In An Array:

APUTC (1454002)

```
« → A C V « A EVAL
TRN C V EVAL ARRY→
LIST→ 1 == 1 IFT 2
→LIST →ARRY TRN
APUTR IF DUP SIZE
DUP SIZE 1 == THEN 1
+ RDM ELSE DROP END
TRN IF A TYPE DUP 6
== SWAP 7 == OR THEN
A STO END » »
```

Summary: APUTC will overwrite the specified column in the destination array with the given column array or vector. The column index is rounded before use and must then be a real number between 1 and the number of columns in the destination array. The column array must contain only 1 column and have the same number of rows as the destination array. If the destination array is named and its name is used, the resulting array will be stored in that name. A vector is allowable as the column array, since it is mathematically equivalent. For example, `[ 1 2 3 ]` and `[[ 1 ][ 2 ][ 3 ]]` are both valid.

Examples: `STD [[ 1 2 ][ 3 4 ][ 5 6 ]] 2 [ 0 0 0 ] APUTC`

Result: `[[ 1 0 ][ 3 0 ][ 5 0 ]]`

`STD [[ 1 2 3 4 ]] 3 [ 0 ] APUTC`

Result: `[[ 1 2 0 4 ]]`

Inputs: Level 3 – any object that evaluates to an array – the destination array.

Level 2 – any object that evaluates to a real number – the column index.

Level 1 – any object that evaluates to a column array or vector – the column vector.

Outputs: Level 1 – if the Level-3 input object was a name, nothing is returned, but the modified array is stored in that name. Otherwise, the modified array is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects.

`Bad Argument Type` will occur if any of the input objects fail to eveluate to their prescribed types.

`Bad Argument Value` will occur if the column index is out of bounds or if the number of rows in the column array does not equal the number of rows in the destination array.

Notes: `APUTC` uses `APUTR`.

Overwrite A Row In An Array:

APUTR (4415836)

```
« EVAL SWAP →NUM .5
+ IP → A V R « A
EVAL ARRY→ LIST→ 1
== 1 IFT → N M « N M
* →LIST V ARRY→
LIST→ IF 2 ≠ ROT 1 ≠
OR R N > OR OVER M ≠
OR R 1 < OR THEN
[ 1 ]
2 GET END →LIST OVER
R M * 1 + N M * SUB
+ SWAP 1 R 1 - M *
SUB SWAP + LIST→
DROP { N } IF M 1 ≠
THEN M + END →ARRY
IF A TYPE DUP 6 ==
SWAP 7 == OR THEN A
STO END » » »
```

Summary: APUTR overwrites the specified row in the destination array with the given row array. The row index is rounded before use and must then specify an existing row in the destination array. The row array must have the same number of columns as the destination array. If the destination array is a name, the resulting array will be stored in that name. A vector is allowed for the destination array, since it is equivalent to column array. But in such cases, only the first representation will be *returned* by APUTR, despite the input format.

Examples:
```
STD [[ 1 2 ][ 3 4 ][ 5 6 ]]
2 [[ 0 0 ]] APUTR
```
Result: `[[ 1 2 ][ 0 0 ][ 5 6 ]]`

```
STD [ 1 2 3 4 ] 2 [[ 99 ]] APUTR
```
Result: `[ 1 99 3 4 ]`

Inputs: Level 3 – any object that evaluates to an array or vector – the destination array.
Level 2 – any object that evaluates to a real number – the row index.
Level 1 – any object that evaluates to a row array.

Outputs: Level 1 – if the Level-3 object was a name, nothing is returned, but the modified array is stored in that name. Otherwise, the modified array is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects.
`Bad Argument Type` will occur if any of the input objects fail to evaluate to their prescribed types.
`Bad Argument Value` will occur if the row index is out of bounds, if the numbers of columns in the row array and destination array do not match, or if the row array is given as a vector.

Notes: None.

Reverse The Order Of The Elements In An Array:

AREV (972352)

```
« → A « A EVAL ARRY→
DUP LIST→ 2 == « * »
IFT → S N « 1 N FOR
I I ROLL NEXT S
→ARRY » IF A TYPE
DUP 6 == SWAP 7 ==
OR THEN A STO END »
»
```

Sort An Array By Element:

ASORT (1046789)

```
« → A « A EVAL ARRY→
LIST→ 1 == 1 IFT → R
C « 1 R C * QSRT { R
} IF C 1 ≠ THEN C +
END →ARRY » IF A
TYPE DUP 6 == SWAP 7
== OR THEN A STO END
» »
```

Summary: AREV reverses the order of the elements of the specified array. ASORT sorts the elements of the specified array in row-major and ascending order. If the array is named, and the name is used, the resulting array is restored in that name. Since a vector is equivalent to a column array, its format is valid also for the input array. In such cases, only the first format will be returned by `ASORT`, regardless of the input format.

Examples: `STD [[ 1 2 ][ 3 4 ]] AREV`
Result: `[[ 4 3 ][ 2 1 ]]`

`STD [ 5 1 4 2 3 ] ASORT`
Result: `[ 1 2 3 4 5 ]`

Inputs: Level 1 – any object that reduces to an array or vector – the array or vector whose elements are to be reversed or sorted.

Outputs: Level 1 – if the input object was a name, nothing is returned, but the modified array or vector is stored in that name. Otherwise, an array or vector is returned – the modified array or vector.

Errors: `Too Few Objects` will occur if the stack is empty. `Bad Argument Type` will occur if the argument does not evaluate to an array or vector.

Notes: Sorting a `{ 10 10 }` array of random integers takes about a minute. Sorting in descending order can be accomplished by applying `AREV` after sorting. Sorting in column major order can be accomplished by transposing the array both before and after sorting.

Sort An Array By Column:

ASRTC (2421252)

```
« →NUM → A C « A
EVAL ARY→R → D « 1
CF DO 1 D 1 - START
IF DUP2 C GET SWAP C
GET SWAP > THEN SWAP
1 SF END D ROLLD
NEXT D ROLLD UNTIL 1
FC?C END D R→ARY »
IF A TYPE DUP 6 ==
SWAP 7 == OR THEN A
STO END » »
```

Sort An Array By Row:

ASRTR (2422032)

```
« →NUM → A C « A
EVAL ARY→C → D « 1
CF DO 1 D 1 - START
IF DUP2 C GET SWAP C
GET SWAP > THEN SWAP
1 SF END D ROLLD
NEXT D ROLLD UNTIL 1
FC?C END D C→ARY »
IF A TYPE DUP 6 ==
SWAP 7 == OR THEN A
STO END » »
```

Summary: ASRTC sorts the rows of the given array in ascending order by the indexed column. ASRTR sorts the columns in ascending order by the specified row. If the name of an array is used, the resulting array is restored in that name. The column/row index is rounded before use.

Examples: `STD [[ 5 1 ][ 4 2 ][ 3 6 ]] 1 ASRTC`
Result: `[[ 3 6 ][ 4 2 ][ 5 1 ]]`

`STD [[ 5 1 ][ 4 2 ][ 3 6 ]] 1 ASRTR`
Result: `[[ 1 5 ][ 2 4 ][ 6 3 ]]`

Inputs: Level 2 – any object that evaluates to an array – the array to be sorted.
Level 1 – any object that evaluates to a real number – the column/row specifier.

Outputs: Level 1 – If the Level 2 object was a name containing an array, nothing is returned to the stack, but the sorted array is stored in that name. Otherwise it is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects, or if the Level-2 object is a vector or row array.
`Bad Argument Type` will occur if the input objects do not evaluate to their prescribed types.
`Invalid Dimension` will occur if the Level-2 object is a row-array or vector.

Notes: ASRTC uses ARY→R and R→ARY. ASRTR uses ARY→C and C→ARY. ASRTC and ASRTR both use user flag 1 to indicate a sorted array.

Convert An Array's Index List To A Numeric Index:

AI→N (676494)

```
« EVAL LIST→ 1 == 1
IFT ROT EVAL LIST→ 1
== 1 IFT 4 DUPN SWAP
4 ROLL < ROT ROT >
OR →NUM « { } 2 GET
» IFT SWAP DROP ROT
1 - * + →NUM »
```

Convert A Numeric Index To An Array's Index List:

AN→I (1440106)

```
« →NUM IP SWAP EVAL
LIST→ IF 2 == THEN 3
DUPN * > →NUM « { }
2 GET » IFT SWAP
DROP →NUM → N C « N
C / CEIL C OVER 1 -
* N SWAP - 2 » ELSE
→NUM OVER < « { } 2
GET » IFT 1 END
→LIST »
```

Summary: AI→N generates an integer index for the given array size, equivalent to the given index list. AN→I does the converse, generating an index list from an integer index. A vector's (a single-column array's) index may be either in the form of { row } or { row 1 }.

Examples:

`{ 4 4 } { 2 3 } AI→N`	Result: `7`
`{ 3 } { 2 } AI→N`	Result: `2`
`{ 2 2 } 3 AN→I`	Result: `{ 2 1 }`
`{ 16 } 8 AN→I`	Result: `{ 8 }`

Inputs: Level 2 – any object that reduces to a list – the dimensions of the array in question.
Level 1 – any object that reduces to a list (for AI→N) or a real number (for AN→I) – the index to be converted to a real number (for AI→N) or a list (for AN→I).

Outputs: Level 1– a real number (for AI→N) – the (row-major) single-value index equivalent; or a list (for AN→I) – the row-column list index equivalent.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if either Level 2 contains an object that is not a list or if Level 1 contains an object type other than that of the input requirement.
`Bad Argument Value` will occur if the specified index is out of bounds for the specified array.

Notes: None.

Perform An Operation On Each Element Of An Array:

AOP (1361863)

```
« → A F « A EVAL
ARRY→ → S « S LIST→
2 == « * » IFT → N «
1 N START F →NUM N
ROLL NEXT » S →ARRY
IF A TYPE DUP 6 ==
SWAP 7 == OR THEN A
STO END » » »
```

Perform An Operation On Each Column Of An Array:

AOPC (718786)

```
« → A F « A ARY→C →
N « 1 N START F →NUM
N ROLL NEXT N C→ARY
IF A TYPE DUP 6 ==
SWAP 7 == OR THEN A
STO END » » »
```

Perform An Operation On Each Row Of An Array:

AOPR (722581)

```
« → A F « A ARY→R →
N « 1 N START F →NUM
N ROLL NEXT N R→ARY
IF A TYPE DUP 6 ==
SWAP 7 == OR THEN A
STO END » » »
```

Summary: AOP performs a given operation on every element of the given array, replacing each element with the element resulting from the operation. AOPC performs a given operation on every column of the given array, replacing each column array with the column array resulting from the operation. AOPR performs a given operation on every row of the given array, replacing each row array with the row array resulting from the operation. If the name of an array is used, the resulting array will be re-stored in that name.

Examples: `STD [[ 1 2 ][ 3 4 ]] « 1 - » AOP`
Result: `[[ 0 1 ][ 2 3 ]]`

`STD [ 1 2 3 4 ] « → X 'X^2-1' » AOP`
Result: `[ 0 3 8 15 ]`

```
'A' 'B' AOP
```

Result: (assuming that `'A'` contains an array and that `'B'` contains an operation, `'A'` is modified, but nothing is left on the stack.)

```
'Q' « IF DUP 4 < THEN DROP 4 END »
AOP
```

Result: All elements of the array, `'Q'`, with values greater than 4 will be changed to 4.

```
[[ 1 2 ][ 3 4 ]] « DUP ABS / »
AOPC 2 FIX
```

Result: `[[ 0.32 0.45 ][ 0.95 0.89 ]]`

```
STD [[ 1 2 ][ 3 4 ][ 5 6 ]] « ARRY→
→ Q « ROT Q » →ARRY » AOPC
```

Result: `[[ 3 4 ][ 5 6 ][ 1 2 ]]`

```
[[ 1 2 ][ 3 4 ]] « DUP ABS / »
AOPR 2 FIX
```

Result: `[[ 0.45 0.89 ][ 0.60 0.80 ]]`

```
STD [[ 1 2 3 ][ 4 5 6 ]] « DUP 1 GET
/ » AOPR
```

Result: `[[ 1 2 3 ][ 1 1.25 1.5 ]]`

Inputs: Level 2 – any object that evaluates to an array – the array to be operated upon.
Level 1 – any object that evaluates to a program or user-defined function – the operation to be used.

Outputs: Level 1 – if the Level-2 input was a name, nothing is re-turned, but the modified array is stored in that name. Otherwise, the modified array is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if either of the arguments is not of the prescribed type, or if the operation does not produce a real number (for `AOP`), a vector (for `AOPC`) or a row array (for `AOPR`).

Notes: Operations on array columns or rows that don't produce array columns or rows (respectively) would better be served by use of `LOP` (see Chapter 7):

```
ARY→C →LIST « op » LOP      or
ARY→R →LIST « op » LOP
```

The results form a list and thus do not need to conform to an array structure. This list can then be transformed into other data objects if appropriate: `LIST→ →ARRY` Note that this would produce a column vector as a result (which is not necessarily what you want), and require that the results of `« op »` be numeric.

`AOPC` uses `C→ARY` and `ARY→C`. `AOPR` uses `R→ARY` and `ARY→R`.

Find The Position Of A Specified Real Element Within An Array:

APOS (717463)

```
« →NUM 2 * 2 / →NUM
→ N « EVAL ARRY→ → S
« S LIST→ 2 == « * »
IFT →LIST N POS IF
DUP THEN S SWAP AN→I
END » » »
```

Summary: APOS finds the position of the first occurrence of the specified real number within the given array or vector (searching in row-major order). If found, the position is returned as a list-index ({ row column }). Otherwise, 0 is returned.

Examples: `STD [[ 3 4 ][ 5 6 ]] 5 APOS`
Result: `{ 2 1 }`

`STD [ 5 4 3 2 ] 4 APOS`
Result: `{ 2 }`

`'A' 'C/SQ(D)' APOS`
Result: `{ 9 14 }` (for example – if 'A', 'C', and 'D' are defined)

`STD [ 1 4 2 3 ] 8 APOS` Result: 0

Inputs:	Level 2 – any object that will evaluate to an array or vector. Level 1 – any object that will reduce to a real number.
Outputs:	Level 1 – if found, a list – the position of the target value; otherwise, `0`.
Errors:	`Too Few Arguments` will occur if there are fewer than 2 objects on the stack. `Bad Argument Type` will occur if either argument does not reduce to its prescribed value. `Undefined Name` will occur if the Level-1 object contains an undefined name.
Notes:	`APOS` uses `AN→I`.

Array Utilities: A Discussion

The Main Idea

Arrays as data objects – as opposed to as mathematically defined matrices – are rather under-represented on the HP-28S, judging from the tools provided to manipulate them. For instance, arrays of numbers are usually thought of in terms of rows and/or columns of data, but the HP-28S gives you no built-in commands with which to build or decompose arrays in either a column-wise or row-wise fashion.

The utilities in this chapter are intended to remedy this situation: their main emphasis to allow manipulation of arrays as data objects, not as matrices. Thus there are utilities to insert, delete, extract and overwrite (i.e. GET and PUT) rows and columns, exchange, sort by, or operate on elements, rows and columns, build and decompose by row or column, extract a subarray, etc.

Where To Put These Programs

As always, to be accessible, these utilities must be in a directory that is listed in your current PATH. The easiest way to ensure that this is the case is to place each of the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

Arrays vs. Vectors

Both mathematically and as objects on the HP-28S, vectors can be considered to be a type of array: In math, a vector is a one-dimensional array and may either be a row-vector or a column-vector. However, on the HP-28S, a *vector object* is always a *column-vector,* represented by numbers within single brackets (`[ 1 2 3 ]`). The dimension of a vector object, as returned by the `SIZE` command, is `{ n }`, indicating that the vector is one-dimensional and has *n* elements.

This representation of a column-vector as a *vector object* is simply intended to make life easier for you. The alternative form of column vector is `[[ 1 ] [ 2 ] [ 3 ]]`, where the single column is represented as a list of one-element rows. However, on the HP-28S, this representation is an *array object* – not a vector object. Accordingly, it is represented as a list of numbers within *double* brackets (`[[ 1 2 3 ]]`), and its dimension is returned as `{ 1 n }` where *n* is the number of elements.

This would all be merely interesting trivia if it were not that *certain built-in HP-28S commands function only on vector objects and not on column-vector arrays* (`CROSS`, for example). On the other hand, certain "array-ish" commands refuse to take vector objects as arguments (e.g. `TRN`).

For this reason, as far as possible, the array utilities make no distinction between vectors and column arrays. And, in most cases, when the result of an array utility would be a column array, it is returned as a vector because vectors are most convenient. It may behoove you, therefore, simply to forget that the HP-28S's column-array format even exists, using instead the vector form, because these array utilities allow for that.

Calculations "In Place"

Many of the array utilities allow you the option of providing a named object as the argument. In that case, the resulting object will be restored in that name object as the end of the calculation – and nothing will be returned to the stack. This feature will work either on global or local name objects and is intended to be analogous to the working of the storage arithmetic commands (see the HP-28S's STORE menu).

Although similar, these utilities lack one of the major advantages of the built-in storage math: Those built-in STORE menu commands will perform their calculations "in-place" – on top of the contents of the current array – thereby taking up less storage space than recalling the contents of the named objects, combining them, then overwriting the original named object. These utilities must use the latter method.

Errors And Error Recovery

Each of these tools is designed to generate an error when invalid input is entered – rather than continue and generate garbage outputs. When inputs are questionable (e.g., negative numbers for stack Levels), these utilities act similarly to the built-in stack commands (arguments are ignored or treated as 1, whichever makes more sense). When errors do occur, the stack is usually disrupted, and since the only way to restore it then is with the UNDO command, it's wisest to keep UNDO mode (in the MODES) menu) *active* whenever you these utilities.

How You Might Use These Utilities

When you use an array simply as a convenient object in which to store and manipulate data (rather than a *mathematically significant* object), you might, for example, want to consider each row or column as a coherent data set. In that case, it's very useful to be able to manipulate each such data set as a unit.

For example, suppose you're collecting data on a population. The data for each individual is sex (0=male, 1=female), age in years, height in inches, and weight in pounds.

First, you would enter the data in that order, as rows in the ΣDAT array, using the Σ+ command in the STAT menu. Then you can perform the following computations:

To segregate the males and females, try this: `'ΣDAT' 1 ASRTC`. Since 0 is less than 1, the males will be first in the array (the lower numbered rows).

Likewise, to sort by age, you could do this: `'ΣDAT' 2 ASRTC`.

To find the first female entry in the already gender-segregated data, you could do this: `'ΣDAT' 1 AGETC 1 APOS`.

This finds the first occurrence of 1 (female) in the first column (sex).

Or, to find the median weight, you could use this short program:

```
            'MEDW' (137916)

      « 'ΣDAT' 4 AGETC
      ASORT NΣ 2 / DUP2
      GET ROT ROT IP GET +
      2 / »
```

To convert the height data from inches to meters, this would work:

```
'ΣDAT' DUP 3 AGETC .0254 * 3 SWAP APUTC
```

To add a column for marital status and insert it as the new column 2, you could do it this way: `'ΣDAT' 2 ROT AINSC`.

To delete an erroneous entry: `'ΣDAT'` <number> `ADELR`.

As you can see, these array utilities provide you with many possib-lilites for data management within the structure of an array. You can easily imagine and create other operations on other sorts of data.

Chapter 6

Character String Utilities

These routines provide convenient, "canned" methods for building/decomposing, editing, and formatting character strings in the HP-28S.

As shown in the following list, the 21 programs are organized into three logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 186, immediately following these program listings.

Convert An Integer To A String:

SIP (89682)

```
« →NUM IP →NUM RCLF
SWAP STD →STR SWAP
STOF »
```

Summary: SIP evaluates the object at stack Level 1, takes the integer portion, then converts that to a string.

Examples:

```
123.45 SIP       Result: "123"
-15.902 SIP      Result: "-15"
1E-12 SIP        Result:  "0"
```

Inputs: Level 1– a real number.

Outputs: Level 1 – a character string – the character representation of the integer portion of the input real number.

Errors: Too Few Arguments will occur if the stack contains no objects.
Bad Argument Type will occur if the input object does not reduce to a real number.
Undefined Name will occur if the input object contains an undefined name.

Notes: None.

Generate The LCD Pattern Of A String:

SPAT (162168)

```
« →STR LCD→ SWAP DUP
1 DISP SIZE 6 * LCD→
1 ROT SUB SWAP →LCD
»
```

Summary: SPAT takes a string version of the object in stack Level 1 and creates a pattern string suitable for →LCD, DPAT, or PRPAT. If the original object string is longer than 23 characters, the resulting pattern string will contain only its first 22 characters, plus an ellipsis (...).

Example: "A" ENTER SPAT ATTN
Result: "∿■■■∿■"

"123" ENTER SPAT ATTN
Result: "■B※@■■bQIIF■"III6■"

Inputs: Level 1 – the object whose character string representation is to be used to make a character pattern.

Outputs: Level 1 – the resulting character pattern string.

Errors: Too Few Arguments will occur for an empty stack.

Notes: None.

Form A String By Repetition Of A String:

SRPT (796841)

```
« →NUM ABS IP →NUM
IF DUP NOT THEN
DROP2 "" ELSE SWAP
EVAL SWAP OVER SIZE
OVER * ROT ROT LN 2
LN / IP 0 SWAP START
DUP + NEXT 1 ROT SUB
END »
```

Summary: SRPT creates a character string by concatenating copies of the given character string.

Examples: `"Ha" 12 SRPT`
Result: `"HaHaHaHaHaHaHaHaHaHaHaHa"`

`"7" 7 SRPT`   Result: `"7777777"`

`"π" 'π'`   Result: `"πππ"`

Inputs: Level 2 – any object that evaluates to a character string – the string to be the repeat pattern.
Level 1 – any object that evaluates to a real number – the number of repetitions.

Outputs: Level 1 – the resulting character string.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.

`Bad Argument Type` will occur if the arguments are not reducible to their prescribed types.

`Undefined Name` will occur if the Level-1 object is an undefined name.

Notes: Caution should be observed when using something other than a real number for the repeat value since its absolute value is taken. Complex numbers and arrays in particular will probably cause undesirable results.

Split A String Into Characters:

STG→ (312398)

```
« →STR DEPTH → D « 1
OVER SIZE FOR I DUP
I DUP SUB SWAP NEXT
DROP DEPTH D - 1 + »
»
```

Summary: STG→ converts the input object to a string and then breaks that down into characters, placing them in order on the stack. The SIZE of the original string conversion is also placed on the stack. Some string conversions may contain NEWLINE characters at certain points.

Examples: "123" STG→ Result: "1" "2" "3" 3

Inputs: Level 1 – the object whose character-string representation is to be decomposed into its component characters, which will then be placed onto the stack.

Outputs: Levels 2 to (n+1) – the n characters of the input string. Level 1 – a real number, n – the number of characters in the input string.

Errors: Too Few Arguments will occur for an empty stack.

Notes: A STG→-broken string can be recomposed with →STG.

Combine A Stack Of Objects Into A String:

→STG (204916)

```
« →NUM IP ABS →NUM
SWAP →STR 1 ROT 1 -
START SWAP →STR SWAP
+ NEXT »
```

Summary: →STG forms a composite string out of a number of items from the stack. All stack items are converted to strings before being added to the resultant string. The number of items to be used is taken from stack Level 1.

The fractional portion and sign of the item count at Level 1 are ignored. No spaces are placed around stack objects before they are added to the resulting string; if delimiter characters are necessary, they must be placed explicitly on the stack in their appropriate positions.

Examples: `STD 1 2 3 3 →STG` Result: `"123"`

`STD "« " 1 " " 1 " + " "»" 6 →STG`
Result: `"« 1 1 + »"`

`"A" "B" "C" 'π' →STG`
Result: `"ABC"`

Inputs: Levels 2 to $(n+1)$ – n objects which, after being converted to strings, will be appended together.
Level 1 – any object that evaluates to a real number, n – the count of objects to be taken from the stack and combined into a character string.

Outputs: Level 1 – a character string – the resulting composite string.

Errors: `Too Few Arguments` will occur either if the stack is empty or if the number in Level 1 is greater than the number of other items on the stack.
`Bad Argument Type` will occur if the Level-1 object is not a real number.
`Undefined Name` will occur if the Level-1 object contains an undefined name.

Notes: Caution should be observed when using something other than a real number for the number of objects since its absolute value is taken. Complex numbers and arrays in particular may cause undesirable results.

Split A String At A Specified Character:

SCUT (168035)

```
« →NUM SWAP EVAL
SWAP DUP2 1 SWAP 1 -
SUB ROT ROT OVER
SIZE SUB »
```

Summary: SCUT cuts a character string into two sub-strings. The break will occur to the left of the position specified.

Examples: `"HI THERE" 3 SCUT 2 SCUT`
Result: `"HI" " " "THERE"`

Inputs: Level 2 – any object that reduces to a character string – the string to split.
Level 1 – any object that reduces to a real number – the position after the cut.

Outputs: Level 2 – a string – the characters to the left of the cut.
Level 1– a string – the characters to the right of the cut.

Errors: `Too Few Arguments` will occur if there are fewer than 2 objects on the stack.
`Bad Argument Type` will occur if the Level-2 object does not reduce to a string or if the Level-1 object does not reduce to a real number.

Notes: None.

Delete A Substring:

SDEL (26677)

```
« →NUM ROT EVAL ROT
→NUM ROT → N M « DUP
1 N 1 - SUB SWAP M 1
+ OVER SIZE SUB + »
»
```

Summary: SDEL deletes a sub-string from the string in stack Level 3. The sub-string is specified with indices to its first and last characters (at stack Levels 2 and 1, respectively). The indexed characters are included in the deletion. If the sub-string's starting index is less than 1, 1 is used. If the sub-string's ending index is greater than the size of the source string, the size of the source string is used. If the starting index is greater than the ending index, no characters are deleted.

Examples: `"DELIBERATE" 1 2 SDEL`
Result: `"LIBERATE"`

`"TEN CHARS." 8 14 SDEL`
Result: `"TEN CHA"`

`"123456789" 3 6 SDEL`
Result: `"12789"`

Inputs:	Level 3 – any object that evaluates to a character string – the original string. Level 2 – any object that evaluates to a real number – the position of the start of the substring. Level 1 – any object that evaluates to a real number – the position of the end of the substring.
Outputs:	Level 1 – a character string – the modified string.
Errors:	`Too Few Arguments` will occur if the stack contains fewer than 3 objects. `Bad Argument Type` will occur if the arguments do not reduce to the specified types.
Notes:	None.

Insert A Substring:

SINS (365327)

```
« EVAL ROT EVAL ROT
→NUM 1 - DUP2 SWAP
SIZE IF > THEN SLJ
SWAP + ELSE 1 + SCUT
ROT SWAP + + END »
```

Put A Substring:

SPUT (866206)

```
« EVAL ROT EVAL ROT
→NUM 1 MAX IP 1 -
→NUM DUP2 SWAP SIZE
IF > THEN OVER SIZE
- SPADR SWAP + ELSE
1 + SCUT 3 PICK SIZE
1 + OVER SIZE SUB
ROT SWAP + + END »
```

Summary: SINS inserts a string immediately before the indexed character in the destination string. SPUT replaces (overwrites) a portion of one string with another, beginning at the indexed position. If the index is less than 1, 1 is used. If the index or resulting string exceeds the

SIZE of the destination string, the destination string is padded with spaces or extended.

Examples: "ABCDEFGHIJKL" 3 "44" SINS
Result: "AB44CDEFGHIJKL"

"ABCDEFGHIJKL" 3 "44" SPUT
Result: "AB44EFGHIJKL"

Inputs: Level 3 – any object that reduces to a character string – the original string.
Level 2 – any object that reduces to a real number – the character position after the insertion point or at the start of the replacement.
Level 1– any object that reduces to a character string – the string to be inserted or "put" into the original.

Outputs: Level 1 – a character string – the newly edited string.

Errors: Too Few Arguments will occur if there are fewer than 3 objects on the stack.
Bad Argument Type will occur if the Level-2 object cannot be reduced to a real number.
Undefined Name will occur if the Level-2 object contains an undefined name.

Notes: SINS uses SCUT and SLJ. SPUT uses SCUT and SPADR. The order of the inputs is similar to the HP-28SPUT command. Use caution when using something other than a real number for the index value. Complex numbers and arrays in particular will probably cause undesirable results.

Convert Uppercase Letters To Lowercase:

SLC (508688)

```
« EVAL
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
→ S A « "" 1 S SIZE
FOR I S I DUP SUB A
SWAP POS 32 0 IFTE
CHR + NEXT S OR » »
```

Summary: SLC converts a string's uppercase characters to lowercase. Lowercase and non-alphabetic characters are unaltered.

Example: "HI THERE" SLC Result: "hi there"

Inputs: Level 1 – any object that reduces to a character string – the string to be converted.

Outputs: Level 1 – a character string – the converted string.

Errors: Too Few Arguments will occur for an empty stack.
Bad Argument Type will occur if the Level-1 object does not reduce to a string.
Undefined Name will occur if the Level-1 object contains an undefined name.

Notes: None.

Convert Lowercase Characters To Uppercase:

SUC (550128)

```
« EVAL
"abcdefghijklmnopqrstuvwxyz"
→ S A « "" 1 S SIZE
FOR I S I DUP SUB A
SWAP POS 95 255 IFTE
CHR + NEXT S AND » »
```

Summary: SUC converts all lowercase characters in the given string to uppercase. Uppercase characters and nonalphabetic characters remain unaltered.

Example: `"hi there" SUC` <u>Result</u>: `"HI THERE"`

Inputs: Level 1 – any object that evaluates to a character string – the string to be converted to all uppercase characters.

Outputs: Level 1 – a character string – the converted string.

Errors: `Too Few Arguments` will occur for an empty stack. `Undefined Name` will occur if the Level-1 object is an undefined name. `Bad Argument Type` will occur if the Level-1 object does not reduce to a string.

Notes: None.

Reverse The Characters In A String:

SREV (200689)

```
« EVAL "" SWAP 1
OVER SIZE FOR I DUP
I DUP SUB ROT + SWAP
NEXT DROP »
```

Summary: SREV evaluates the Level-1 object, then reverses the order of the resulting string's characters.

Examples: STD "12345" SREV Result: "54321"
"HI THERE" SREV Result: "EREHT IH"

Inputs: Level 1 – any object that evaluates to a character string.

Outputs: Level 1 – a character string – the reversed string.

Errors: Too Few Arguments will occur for an empty stack. Bad Argument Type will occur if the argument does not evaluate to a character string.

Notes: None.

Rotate The Characters Of A String

SROT (165882)

```
« →NUM IP NEG →NUM
SWAP EVAL SWAP OVER
SIZE MOD 1 + SCUT
SWAP + »
```

Summary: SROT rotates a string by the specified number of characters. A positive rotation index rotates to the right, a negative to the left (fractional parts of the index are truncated).

Examples: "12345" 1 SROT <u>Result</u>: 51234

Inputs: Level 2 – any object that evaluates to a character string – the string to be rotated.
Level 1 – any object that evaluates to a real number – the characters to be rotated.

Outputs: Level 1 – the rotated string.

Errors: Too Few Arguments will occur if the stack contains fewer than 2 objects.
Bad Argument Type will occur if the arguments cannot be reduced to their appropriate types.
Undefined Name will occur if the Level-1 object contains an undefined name.

Notes: SROT uses SCUT.

Replace All Occurrences Of A Substring:

SRPL (1248335)

```
« EVAL SWAP EVAL
DEPTH → B A N « IF
DUP A POS THEN WHILE
DUP A POS DUP REPEAT
SCUT A SIZE 1 + OVER
SIZE SUB SWAP B +
SWAP END DROP N
DEPTH START + NEXT
END » »
```

Summary: SRPL searches the object string for every occurrence of the pattern string, substituting the replacement string.

Examples: "123123" "3" "0" SRPL Result: "120120"

Inputs: Level 3 – the object string.
Level 2 – the pattern string.
Level 1 – the replacement string.

Outputs: Level 1 – the modified string.

Errors: Too Few Arguments will occur if there are fewer than 3 objects on the stack.
Bad Argument Type will occur if the arguments do not reduce to character strings.

Notes: SRPL uses SCUT.

Remove All Occurrences Of A Substring:

SZAP (46136)

```
« EVAL SWAP EVAL
SWAP "" SRPL »
```

Summary: SZAP deletes all occurrences of a substring from another string.

Examples:

"12345" "3" SZAP	Result: "1245"
"XYXYXY" "Y" SZAP	Result: "XXX"
"ABCDEF" "Q" SZAP	Result: "ABCDEF"

Inputs: Level 2 – any object that evaluates to a character string – the original string to be edited.
Level 1 – any object that evaluates to a character string – the substring to be deleted from the original string.

Outputs: Level 1 – a character string – the modified string.

Errors: Too Few Arguments will occur if the stack contains fewer than 2 objects.
Bad Argument Type will occur if either of the arguments fails to reduce to a character string.

Notes: SZAP uses SRPL.

Remove Characters From The Left End:

SZAPL (37926θ)

```
« EVAL NUM CHR SWAP
EVAL SWAP → S «
WHILE DUP NUM CHR S
== REPEAT 2 OVER
SIZE SUB END » »
```

Remove Characters From The Right End:

SZAPR (5θ76θθ)

```
« EVAL NUM CHR SWAP
EVAL SWAP → S «
WHILE DUP SIZE DUP2
DUP SUB S == REPEAT
1 SWAP OVER - SUB
END DROP » »
```

Summary: SZAPL repeatedly removes the specified character from the beginning of a string until there are none remaining. SZAPR repeatedly removes the specified character from the end of a string until there are none remaining. Only the first character of the pattern string is used as a delete pattern.

Examples:

`"8" " " SZAPL`	Result: `"8"`
`"5551212" "5" SZAPL`	Result: `"1212"`
`"8" " " SZAPR`	Result: `"8"`
`"5.0000" "0" SZAPR`	Result: `"5."`

Inputs: Level 2 – any object that evaluates to a character string – the string to be "trimmed."
Level 1 – any object that evaluates to a character string – the string whose first character is to be deleted repeatedly from the beginning or end of the target string.

Outputs: Level 1 – the modified string.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if either of the arguments fails to evaluate to a character string.

Notes: None.

Center A String In A Field Of Spaces:

SCTR (57764)

```
« →NUM SWAP →STR DUP
SIZE ROT DUP ROT - 2
/ IP →NUM IF DUP 0 ≥
THEN " " SWAP SRPT
ROT + SWAP SLJ ELSE
DROP 1 SWAP SUB END
»
```

Left-Justify A String In A Field Of Spaces:

SLJ (421808)

```
« →NUM ABS IP →NUM
SWAP →STR DUP SIZE
ROT SWAP IF DUP2 <
THEN DROP 1 SWAP SUB
ELSE - " " SWAP SRPT
+ END »
```

Right-Justify In A Field Of Spaces:

SRJ (463907)

```
« →NUM ABS IP →NUM
SWAP →STR DUP SIZE
ROT SWAP IF DUP2 <
THEN DROP 1 SWAP SUB
ELSE - " " SWAP SRPT
SWAP + END »
```

Summary: SCTR converts the object from stack Level 2 into a string and centers it within a specified field of spaces. Similarly, SLJ left-justifies and SRJ right-justifies the object within the field. Any fractional portion of the field width value is truncated before use. If the field width is smaller than the object, the object string is truncated to fit within the specified field size. If the field size is zero or less, the resulting string will be empty. NEWLINE characters are counted when determining the length of the object. A centered object may be placed one character to the left of center, as necessary.

Examples:

```
STD 8 7 SCTR     Result: "   8   "
STD 8 7 SLJ      Result: "8      "
STD 8 7 SRJ      Result: "      8"
```

Inputs: Level 2 – the object to be centered or justified.
Level 1 – any object that evaluates to a real number – the"field-width."

Outputs: Level 1 – a character string – the object as a string centered or justified within a field of spaces.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if the Level-1 object does not evaluate to a real number.
`Undefined Name` will occur if the Level-1 object contains an undefined name.

Notes: `SCTR` uses `SLJ`. `SCTR`, `SLJ` and `SRJ` all use `SRPT`. Caution should be observed when using something other than a real number for the field-width value, since its absolute value is taken. In particular, complex numbers and arrays will probably cause undesirable results.

Pad A String On The Left With Spaces:

SPADL (45229)

```
« " " SWAP SRPT SWAP
→STR + »
```

Pad A String On The Right With Spaces:

SPADR (58673)

```
« " " SWAP SRPT SWAP
→STR SWAP + »
```

Summary: SPADL adds the specified number of spaces to the left side of a character string. SPADR adds the specified number of spaces to the right side. If the object to be padded is not a string, it is converted before padding.

Examples:

```
"HI" 5 SPADL        Result: "     HI"
7 7 SPADL           Result: "       7"
"HI" 5 SPADR        Result: "HI     "
7 7 SPADR           Result: "7       "
```

Inputs: Level 2 – the object whose string equivalent is to be padded with spaces.
Level 1 – a real number – the number of spaces to be added as "padding."

Outputs: Level 1 – the padded string.

Errors: Too Few Arguments will occur if the stack contains fewer than 2 objects.
Bad Argument Type will occur if the Level-1 object is not reducible to a real number.
Undefined Name will occur if the Level-1 object contains an undefined name.

Notes: SPADL and SPADR use SRPT. Caution should be observed when using something other than a real number for the "padding number," since its absolute value is taken. In particular, complex numbers and arrays may cause undesirable results.

Character String Utilities: A Discussion

The Main Idea

Character strings are the most versatile data objects provided by the HP-28S; they can be converted from and to *any* other HP-28S data object. They can contain the representations of one or many objects, and once assembled, their contained object(s) can be sequentially evaluated using STR→, just like a program. And of course, strings have the unique ability to present information to you in your own language: words.

Such versatility and ability might suggest that a rather large collection of string-related commands is surely built into the HP-28S. Not so – there are very few. *However,* with these powerful few, all of the possibilities inherent in character strings can be realized by writing a handful of relatively straightforward programs – and that's what this collection of utilities is all about.

Where To Put These Programs

As always, to be accessible, these utilities must be in a directory that is listed in your current PATH. The easiest way to ensure that this is the case is to place each of the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

As shown in the list of contents (page 159), each of the string utilities belongs basically to one of several different functional groups:

Building/Decomposition: building and tearing down strings from and to characters and objects of other types – including one command (SPAT) that gives you the ability to build a display pattern from a character string (the HP-28S provides a method of capturing and redisplaying its LCD using character strings, so manipulation of the pattern string can allow you to build interesting displays).

Editing: These are the commands that perform actual physical modifications to a string object: adding and deleting characters, splitting and concatenating strings, inserting and overwriting characters in a string, replacing and removing substrings, trimming excess characters, reversing and rotating a string.

Formatting: Any object type can be converted to a character string, and because the content of a string is virtually unrestricted, you can create representations of the objects which would otherwise be impossible. Objects can be labelled, positioned within fields of spaces, special characters can be added, extraneous characters removed, etc.

Notes About Conversions

Conversions of objects to strings depends on current system modes, among other factors. The conversion of real numbers (or compound objects containing real numbers) to strings uses the current display format. Thus `2 FIX 1.2345 →STR` gives `"1.23"` and then `STR→` returns `1.23` regardless of the current display format. In other words, *information is lost.*

Other objects, like binary integers, are converted using the current base and word size, and these forms are static regardless of how these system states may have been altered since their conversion.

Another less obvious artifact of conversion is that large objects like programs are converted to strings using the form they would take during an EDIT or VISIT, *including embedded NEWLINE characters.* Thus, `[[ 1 ][ 2 ]]` would become `"[[ 1 ]▪ [ 2 ]]"`, for example, when converted to a string in `STD` display mode (regardless of the current multiline mode.)

Errors And Error Recovery

Each of these tools is designed to generate an error when invalid input is entered – rather than continue and generate garbage outputs. When errors do occur, the stack is usually disrupted, and since the only way to restore it then is with the `UNDO` command, it's wisest to keep UNDO mode (in the MODES) menu) *active* whenever you these utilities.

How You Might Use These Utilities

Sometimes the best explanation is simply a set of examples. Here is such a set (you'll notice that in some cases, it's most convenient to use these String utilities in concert with some of the List utilities from chapter 7):

How many E's are in a given string? To find out, you could do this (the LSORT routine used here is from chapter 7):

```
STG→ →LIST LSORT LIST→ →STG
DUP "E" POS SCUT
SREV DUP "E" POS SCUT SIZE
```

To convert a string to a list of character codes, try this (the LOP routine is from chapter 7):

```
STG→ →LIST « NUM » LOP
```

To convert "6 ft 2 3/4 in" to a real number of inches, here's one method:

```
" ft" "*12" SRPL " in" "" SRPL
" " "+" SRPL "'" SWAP + STR→ EVAL
```

To convert 6.023E23 to "6.023*10^(23)", you could do this:

```
STD →STR
"E" "*10^(" SRPL ")" +
```

This could be a handy little program to have, if you wanted to name it, right?

To convert every occurrence of →NUM in an object to EVAL try this:

```
→STR "■" " " SRPL
" →NUM " " EVAL " SRPL STR→.
```

The ■ you see here is the NEWLINE character. Notice that you use spaces to bracket the match string, so that patterns like 'X→NUM' don't match.

Notice also that all NEWLINE's (■'s) are converted to spaces first because →NUM might occur next to one, in which case you would otherwise need to search for and replace " →NUM ", "■→NUM ", " →NUM■" and "■→NUM■" as special cases.

You can break up an arbitrary string into individual "words," with a little routine such as BREAK (717843):

```
« "■" " " SRPL WHILE
DUP "  " POS REPEAT
"  " " " SRPL END
" " 34 CHR DUP +
SRPL 34 CHR SWAP
OVER + + DEPTH → D «
STR→ D » DEPTH SWAP
- »
```

Or, suppose you wanted to display a set of eight numbers in two columns in the display. Here's a routine, DISP8 (187922), to do that:

```
« 1 4 START 8 ROLL
12 SLJ 8 ROLL 11 SRJ
"■" + NEXT + + + + +
+ + 1 DISP »
```

Note that not all display formats will work here (you might also try replacing SLJ and SRJ with SCTR).

Chapter 7

List Utilities

These routines provide convenient, "canned" methods for building/ decomposing, editing, and operating on lists in the HP-28S.

As shown in the following list, the 19 programs are organized into three logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 224, immediately following these program listings.

Convert An Array To A List:

A→L (183691)

```
« EVAL ARRY→ → D « D
LIST→ IF 2 == THEN *
END →LIST D » »
```

Summary: A→L converts the given array or vector into a list of its elements, in row-major order. A second list will also be returned, containing the size information from the original array so that the array can be reconstructed.

Examples: `STD [[ 1 2 ][ 3 4 ]] A→L`
Result: `{ 1 2 3 4 } { 2 2 }`

`STD [ 1 2 3 4 ] A→L`
Result: `{ 1 2 3 4 } { 4 }`

Inputs: Level 1 – Any object that evaluates to an array or vector – the array to be converted.

Outputs: Level 2 – a list – the elements of the original array.
Level 1 – a list – the original dimensions of the array.

Errors: `Too Few Arguments` will occur for an empty stack. `Bad Argument Type` will occur if the argument does not evaluate to an array or vector.

Notes: None.

Convert A List To An Array:

L→A (592057)

```
« EVAL SWAP EVAL → D
L « L SIZE D LIST→
IF 2 == THEN * END
IF ≠ THEN { } 1 GET
END L LIST→ DROP D
→ARRY » »
```

Summary: L→A converts the given list of numbers into an array or vector of the elements from the list (in row-major order). A second list must be given, containing the size information of the resulting array, i.e., `{ rows columns }`.

Examples: `STD { 1 2 3 4 } { 2 2 } L→A`
Result: `[[ 1 2 ][ 3 4 ]]`

`STD { 1 2 3 4 } { 4 } L→A`
Result: `[ 1 2 3 4 ]`

Inputs: Level 2 – any object that evaluates to a list of real numbers – the list to be converted.
Level 1 – any object that evaluates to a list – the list containing the dimensions of the desired array.

Outputs: Level 1 – an array or vector, depending on the specification – the object just converted from the input list.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if the arguments do not evaluate to lists.
`Bad Argument Value` will occur if the dimensions of the size list do not correspond to the number of elements in the element list.

Notes: None.

Form A List By Repetition Of An Element:

LRPT (29276Ø)

```
« →NUM .5 + FLOOR →
E N « { } IF N 1 ≥
THEN 1 N START E +
NEXT END » »
```

Summary: LRPT creates a new list through an indexed repetition of a given element or list. Any fractional portion of the repetition index is rounded before use. If a list is used as the repeated object, the resulting list is formed by repeating all of the objects in the repeat list, in order.

Examples:

```
STD 'A' 3.6 LRPT     Result: { A A A A }
STD "HI" Ø LRPT      Result: { }
STD { 1 2 3 } 2 LRPT
                     Result: { 1 2 3 1 2 3 }
```

Inputs: Level 2 – any object – the object to be repeated.
Level 1 – a real number value – the repetition index.

Outputs: Level 1 – a list – that formed by repetition of the object.

Errors: Too Few Arguments will occur if the stack contains fewer than 2 objects.
Bad Argument Type will occur if the Level-1 input object does not evaluate to a real number.

Notes: None.

Filter A List With A Procedure:

FLTR (1987293)

```
« → l.. t.. « { } 1
l.. EVAL SIZE IF DUP
THEN FOR i.. l..
EVAL i.. GET IF DUP
t.. EVAL THEN 1
→LIST + ELSE DROP
END NEXT ELSE DROP2
END IF l.. TYPE DUP
6 == SWAP 7 == OR
THEN l.. STO END » »
```

Summary: FLTR will filter all objects out of a list that fail a user-defined test procedure. Only those elements that pass the test (i.e. return a 1 rather than a 0 to stack Level 1) will be used to form the return list. If the list is stored in a name and that name is used, the resulting list will be stored in that name.

Examples: `STD { 1 2 3 4 5 6 7 } « 4 < » FLTR`
Result: `{ 1 2 3 }`

`STD { 5 6 7 8 9 10 } « → X 'IFTE`
`(X^2<50,1,0)' » FLTR`
Result: `{ 5 6 7 }`

```
STD { A 4 { 1 } (2,2) }
« TYPE 5 == » FLTR
```
Result: `{ { 1 } }`

Inputs: Level 2 – any object that will evaluate to a list – the list to be filtered.
Level 1 – either a program or a user-defined function that takes one argument from the stack and returns either a 1 or 0 to the stack – the filtering test.

Outputs: Level 1 – if a name containing a list was given as the Level-2 argument, the resulting list will be restored in that name. Otherwise, the filtered list is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if either the Level-2 object is not a list or the test procedure in Level 1 is incompatible with an object in the list.
Unpredictable errors will occur if the Level-1 object is not a program or user-defined function, if the test takes more than one object from the stack, or if the test returns more than one object to the stack.

Notes: The local names, `l..`, `t..` and `i..` were chosen to reduce the chances of conflicts when operations such as `« STR→ »` are applied to lists of strings. Therefore, avoid using `l..`, `t..` and `i..` as global names in your own programming.

Split A List At A Specified Point:

LCUT (165719)

```
« →NUM → L E « L
EVAL DUP 1 E 1 - SUB
SWAP E OVER SIZE SUB
» »
```

Summary: LCUT cuts a list from stack Level 2 into two sub-lists. The point of the break is specified by the real number in Level 1. The list will be split between the specified element and the element to its left.

Examples: STD { 1 2 3 4 5 } 4 LCUT
Result: { 1 2 3 } { 4 5 }

Inputs: Level 2 – any object that evaluates to a list – the list to be split.
Level 1 – any object that evaluates to a real number – the point, *n*, in the list before which the split is to be made.

Outputs: Level 2 – a list – the (*n*-1) elements of the original list which were to the left of the cut.
Level 1 – a list – the (SIZE-*n*+1) elements of the original list which were to the right of the cut.

Errors: `Too Few Arguments` will occur if there are fewer than 2 objects on the stack.
`Bad Argument Type` will occur if the Level-2 object is not a list or the Level-1 object is not a real number.

Notes: None.

Delete The Specified Sublist:

LDEL (700391)

```
« → L N M « L EVAL
DUP 1 N →NUM 1 -
→NUM SUB SWAP M →NUM
1 + →NUM OVER SIZE
SUB + IF L TYPE DUP
7 == SWAP 6 == OR
THEN L STO END » »
```

Summary: LDEL will delete the specified element or sub-list of elements from the given list. Two indices are required: the first element to delete and the last element to delete. All elements between and including these indexed elements are deleted. If the beginning index is less than 1, 1 is used. Likewise, if the ending index is greater than the size of the list, the size of the list is used. If either index is non-integer, the value is rounded. If the list is named and the name is used, the modified list is restored in the name.

Examples: `STD { 1 2 3 4 5 6 } 1 3 LDEL`
Result: `{ 4 5 6 }`

`{ A B C D E F } 3 5 LDEL`
Result: `{ A B F }`

STD { "HI" A 4 } 'L' STO 'L' 3 3 LDEL L Result: { "HI" A }

STD { 1 2 3 4 5 6 } 4 10 LDEL Result: { 1 2 3 }

Inputs: Level 3 – any object that evaluates to a list – the list to be edited.
Level 2 – any object that evaluates to a real number – the index of the beginning of the sublist to be deleted.
Level 1 – any object that evaluates to a real number – the index of the end of the sublist to be deleted.

Outputs: Level 1 – if the Level-3 object was a name that contained a list, the result is stored in that name. Otherwise a list is returned – the newly-edited list.

Errors: Too Few Arguments will occur if the stack contains fewer than 3 objects.
Bad Argument Type will occur if the input objects do not evaluate to their prescribed types.

Notes: None.

Exchange Elements Within A List:

LEX (700543)

```
« →NUM SWAP →NUM → L
M N « L EVAL DUP DUP
N GET SWAP M GET ROT
N ROT PUT M ROT PUT
IF L TYPE DUP 6 ==
SWAP 7 == OR THEN L
STO END » »
```

Summary: LEX exchanges the positions of the two specified elements within the given list. If either index is non-integer, the value is rounded before use. If the name of a list is used, the resulting list is restored in that name.

Examples: `{ A B C D E F } 1 6 LEX`
Result: `{ F B C D E A }`

`STD { 4 3 6 } 'L' STO 'L' 2 1`
`LEX L` Result: `{ 3 4 6 }`

Inputs: Level 3 – any object that evaluates to a list – the list to be edited.
Level 2 – any object that evaluates to a real number – the index of one of the elements to be exchanged.
Level 1 – any object that evaluates to a real number – the index of the other element to be exchanged.

Outputs: Level 1 – if the Level-3 object was a name containing a list, the result is stored in that name. Otherwise, a list is returned – the newly-edited list.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects.
`Bad Argument Type` will occur if the input objects do not evaluate to their prescribed types.
`Bad Argument Value` will occur if either of the indices is out of bounds for the list.

Notes: None.

Insert An Object Into A list:

LINS (309935)

```
« → L N S « L N LCUT
S SWAP + + IF L TYPE
DUP 6 == SWAP 7 ==
OR THEN L STO END »
»
```

Put An Object Into A List:

LPUT (513346)

```
« → L N S « L N LCUT
S SWAP OVER SIZE 1 +
LCUT SWAP DROP + +
IF L TYPE DUP 6 ==
SWAP 7 == OR THEN L
STO END » »
```

Summary: LINS inserts an object into a list at the specified location. If the inserted object is another list, all of that list's objects will be inserted, in order, at the indexed location. LPUT overwrites a list with the contents of another list, starting at the indexed location. If the index has a fractional portion, it will be rounded before being used. If the index is less than 1, 1 will be used. If the index is greater than the size of the list, the size of the list is used. If the objects being inserted or placed would write past the end of the destination list, the destination list is extended. If the destination list's name is used, the resulting list will be restored in that name.

Examples:

```
STD { 1 2 3 } 2 15 LINS
```
Result: `{ 1 15 2 3 }`

```
STD { 1 2 3 } 8 15 LINS
```
Result: `{ 1 2 3 15 }`

```
STD { 1 2 3 } 2 { 4 5 6 } LINS
```
Result: `{ 1 4 5 6 2 3 }`

```
STD { 1 2 3 } 2 { { 15 } } LINS
```
Result: `{ 1 { 15 } 2 3 }`

```
STD { 1 2 3 4 5 } 0 { A B C } LPUT
```
Result: `{ A B C 4 5 }`

```
STD { 1 2 3 4 5 } 1 { A B C } LPUT
```
Result: `{ A B C 4 5 }`

```
STD { 1 2 3 4 5 } 3 { A B C } LPUT
```
Result: `{ 1 2 A B C }`

```
STD { 1 2 3 4 5 } 4 { A B C } LPUT
```
Result: `{ 1 2 3 A B C }`

```
STD { 1 2 3 4 5 } 6 { A B C } LPUT
```
Result: `{ 1 2 3 4 5 A B C }`

Inputs: Level 3 – any object that evaluates to a list – the list to be edited.
Level 2 – any object that evaluates to a real number – the insertion/replacement point.
Level 1 – (for `LINS`) any objects – the objects to be inserted, or (for `LPUT`) any object that evaluates to a list – the replacement list.

Outputs: Level 1 – if the Level-3 object was the name of a list, the result is stored in that name and nothing is returned to the stack. Otherwise, the newly-edited list is returned.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects.
`Bad Argument Type` will occur if the input objects fail to evaluate to their prescribed types.

Notes: `LINS` and `LPUT` both use `LCUT`.

Reverse The Order Of The Elements:

LREV (958659)

```
« → L « L EVAL LIST→
→ N « IF N THEN 1 N
FOR I I ROLL NEXT
END N →LIST » IF L
TYPE DUP 6 == SWAP 7
== OR THEN L STO END
» »
```

Summary: LREV reverses the order of the elements within a list. If the name of a list is specified, the resulting list is restored in that name.

Example: STD { 1 2 3 } LREV   Result: { 3 2 1 }

Inputs: Level 1 – any object that evaluates to a list – the list whose elements are to be reversed.

Outputs: Level 1 – if the input object was a name containing a list, the result is stored in that name, and nothing is returned to the stack. Otherwise, a list is returned – the input list with its elements reversed.

Errors: Too Few Arguments will occur for an empty stack.

Notes: None.

Rotate The Positions Of The Elements:

LROT (624930)

```
« → L N « L EVAL N
→NUM .5 + FLOOR NEG
→NUM OVER SIZE MOD 1
+ LCUT SWAP + IF L
TYPE DUP 6 == SWAP 7
== OR THEN L STO END
» »
```

Summary: LROT rotates the positions of elements of a list to the left or right by the specified number of elements. A positive rotation index specifies rotation to the right; a negative index specifies rotation to the left. If the index has a fractional portion, it is rounded before use. If the list is named and the name is used, the resulting list is stored in that name.

Examples: `STD { 1 2 3 4 5 } 1 LROT`
Result: `{ 5 1 2 3 4 }`

`STD { 1 2 3 4 5 } -1 LROT`
Result: `{ 2 3 4 5 1 }`

`STD { 1 2 3 4 5 } 3 LROT`
Result: `{ 3 4 5 1 2 }`

Inputs: Level 2 – any object that evaluates to a list – the list whose elements are to be rotated.
Level 1 – any object that evaluates to a real number – the index specifying the extent and direction of the rotation.

Outputs: Level 1 – if the Level-2 input object was a name containing a list, the result is stored in that name and nothing is returned to the stack. Otherwise, a list is returned – the input list with its elements properly rotated.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if either argument does not reduce to its prescribed type.

Notes: `LROT` uses `LCUT`.

Replace All Occurrences Of An Element:

LRPL (980117)

```
« → L A B « L EVAL
IF A B SAME NOT THEN
WHILE DUP A POS DUP
REPEAT B PUT END
DROP END IF L TYPE
DUP 6 == SWAP 7 ==
OR THEN L STO END »
»
```

Summary: LRPL replaces every occurrence of a given object within a list with a second object. If a list name is used, the resulting list is stored in that name.

Examples: `STD { 5 5 5 1 2 1 2 } 5 6 LRPL`
<u>Result</u>: `{ 6 6 6 1 2 1 2 }`

`STD { 1 2 3 4 5 } 6 7 LRPL`
<u>Result</u>: `{ 1 2 3 4 5 }`

`STD { A B C D E F } 'C' "HI" LRPL`
<u>Result</u>: `{ A B "HI" D E F }`

Inputs: Level 3 – any object that evaluates to a list – the list to be edited.
Level 2 – any object – the target object to be replaced.
Level 1 – any object – the replacement object.

Outputs: Level 1 – if the Level-3 object was a name containing a list, the result is stored in that name and nothing is returned to the stack. Otherwise, a list is returned – the newly-edited list.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 3 objects.
`Bad Argument Type` will occur if the Level-3 input object does not evaluate to a list.

Notes: None.

Sort A List By Element:

LSORT (750384)

```
« → L « L EVAL LIST→
→ N « IF N 1 > THEN
1 N QSRT END N →LIST
» IF L TYPE DUP 6 ==
SWAP 7 == OR THEN L
STO END » »
```

Summary: LSORT will sort the given list so that the elements are arranged in ascending order. The elements of the list must be orderable (i.e., they must be either real numbers, binary integers, or strings) or an error will occur. If a list name is used, the resulting list will be stored in that name.

Examples: `STD { 8 4 0 3 2 } LSORT`
Result: `{ 0 2 3 4 8 }`

`{ Z H T F Y } « →STR » LOP LSORT`
`« STR→ » LOP` Result: `{ F H T Y Z }`

Inputs: Level 1 – any object that evaluates to an orderable list – the list to be sorted.

Outputs: Level 1 – if the input object was a name, the result will be stored in that name. Otherwise a list is returned – the newly-sorted list.

Errors: Too Few Arguments will occur for an empty stack. Bad Argument Type will occur if the input list does not evaluate to a list or if the list is not orderable.

Notes: LSORT assumes that the elements of the list are orderable. As shown in the second **example** (opposite), if the elements of the list are unorderable as is, then perform « →STR » LOP before the sort and « STR→ » LOP after the sort to effectively sort the elements based on their *decompiled* ("character-string") representations. LOP is described on page 220.

To sort the elements of a list in descending order, use LREV on the sorted list. To sort the evaluated values of a list, use « →NUM » LOP or « EVAL » LOP before sorting. LSORT uses QSRT.

Remove All Occurrences Of An Element:

LZAP (61082l)

```
« → L E « L EVAL
WHILE DUP E POS DUP
REPEAT DUP LDEL END
DROP IF L TYPE DUP 6
== SWAP 7 == OR THEN
L STO END » »
```

Summary: LZAP deletes all occurrences of the specified object from the given list. If the list has a name and the name is used, the result is restored in that name.

Examples: `STD { 5 5 5 1 2 1 2 } 1 LZAP`
Result: `{ 5 5 5 2 2 }`

`{ A B C } 'C' LZAP` Result: `{ A B }`

Inputs: Level 2 – any object that evaluates to a list – the list to be edited.
Level 1 – any object – the target object all of whose occurrences are to be deleted from the list.

Outputs: Level 1 – if the Level-2 input object was a name containing a list, the result is stored in that name and nothing is returned to the stack. Otherwise, a list is returned to the stack – the newly-edited list.

Errors: Too Few Arguments will occur if the stack contains fewer than 2 objects.
Bad Argument Type if will occur if the Level-2 input object does not evaluate to a list.

Notes: LZAP uses LDEL.

Add An Element To The End Of A Queue:

ENQ (326692)

```
« → Q N « N Q EVAL
LIST→ 1 + →LIST IF Q
TYPE DUP 6 == SWAP 7
== OR THEN Q STO END
» »
```

Remove The First Element From A Queue:

UNQ (521908)

```
« → Q « Q EVAL LIST→
SWAP → N « 1 - →LIST
N IF Q TYPE DUP 6 ==
SWAP 7 == OR THEN
SWAP Q STO END » » »
```

Summary: ENQ adds a given element to the given queue. UNQ removes a given element from the given queue (a queue is a list whose elements are accessed on a first-in-first-out basis; the first object put into a queue will be the first object taken out). If the queue is stored in either a local or global name and the name is used, the resulting queue is restored in that name.

Examples: `{ } 1 ENQ 2 ENQ 3 ENQ`
Result: `{ 3 2 1 }`

`{ 4 3 2 1 } UNQ` Result: `{ 4 3 2 } 1`

Inputs: Level 2 (for ENQ only) – any object that evaluates to a list – the queue to be added to.
Level 1 – the object to be added to the queue (for ENQ) or the queue to be edited (for UNQ).

Outputs: Level 1 – the unqueued object (for UNQ), or (for ENQ) if the Level-2 input object was a name, the result is stored in that name and no object is returned to the stack. Otherwise, a list is returned – the modified queue.
Level 2 (for UNQ only) – a list – the modified queue.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects (for ENQ), or (for UNQ) if the stack contains no objects or the input list is empty.
`Bad Argument Type` will occur for ENQ if the Level-2 object does not evaluate to a list.

Notes: Coordinated use of ENQ and UNQ will allow you to maintain a named or unnamed queue. The commands `LIST→ 1 + →LIST` in ENQ are less efficient than the equivalent `+` but provide the benefit of generating an error if the Level-1 object (`'Q'`) is not a list.

Perform An Operation On Each Element Of A List:

LOP (1497943)

```
« → l.. f.. « l..
EVAL LIST→ IF DUP
THEN → n.. « 1 n..
START n.. ROLL f..
EVAL NEXT n.. →LIST
» IF l.. TYPE DUP 6
== SWAP 7 == OR THEN
l.. STO END ELSE
DROP END » »
```

Summary: LOP performs the specified operation on each element of the given list. The operation must take exactly one object from the stack and return exactly one object to the stack, replacing the element operated on. If a list name is used, the resulting list will be stored in that name.

Examples: `STD { 1 2 3 4 } « SQ » LOP`
Result: `{ 1 4 9 16 }`

`STD { 1 2 3 4 } « → X 'X^2' » LOP`
Result: `{ 1 4 9 16 }`

```
STD { 2 4 3 -7 } 0 'I' STO « 'X' I
^ * 1 'I' STO+ » LOP
```
Result: `{ 2 '4*X' '3*X^2' '-(7*X^3)' }`

```
STD { 5 6 7 8 9 } « IF DUP 7 ≤ THEN
SQ END » LOP
```
Result: `{ 25 36 49 8 9 }`

Inputs: Level 2 – any object that evaluates to a list – the list to be operated upon.
Level 1 – a program or user-defined function – the operation to be performed on each element in the list.

Outputs:: Level 1 – if the Level-2 object was a name containing a list, the result is stored in the name and nothing is returned to the stack. Otherwise, a list is returned – with the newly-modified elements.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if the Level-2 object does not evaluate to a list or if the operation is not valid for an element of the list.
The stack will fill with garbage if the Level 1 object is not a program or user-defined function.
Various unpredictable errors can occur if the operation is not valid over the whole range of list elements.

Notes: The local names, `l..`, `f..` and `n..` were chosen to reduce the chances of conflicts when operations such as `« STR→ »` are applied to lists of strings. Therefore, avoid using `l..`, `f..` and `n..` as global names in your own programming.

Remove The Last Element From A Stack:

POP (521242)

```
« → Q « Q EVAL LIST→
SWAP → N « 1 - →LIST
N IF Q TYPE DUP 6 ==
SWAP 7 == OR THEN
SWAP Q STO END » » »
```

Add An Element To The Bottom Of A Stack:

PUSH (369591)

```
« → S N « S EVAL
LIST→ N SWAP 1 +
→LIST IF S TYPE DUP
6 == SWAP 7 == OR
THEN S STO END » »
```

Summary: POP removes the next element from the given stack. PUSH adds the given element to the given stack (a stack is a list whose elements are accessed on a first-in-last-out basis; the first object put into a stack is the last object taken out – as with the HP-28S' own internal stack). If a stack name is used, the resulting stack is stored in that name.

Example: STD { 3 2 1 } POP Result: { 3 2 } 1
STD { 3 2 } 1 PUSH Result: { 3 2 1 }

Inputs: Level 2 (for PUSH only) – any object that evaluates to a list – the stack to be amended.
Level 1 – (for POP) any object that evaluates to a list – the stack to be edited, or (for PUSH) any object – the object to be added to the list.

Outputs: Level 2 (for POP only) – if the input object was a name, the modified stack is stored in that name. Otherwise it returns here.
Level 1 – (for POP) an object – the object just removed ("popped") from the input stack, or (for PUSH) if the Level-2 input object was a name, the modified stack is stored in that name. Otherwise, it returns here.

Errors: Too Few Arguments will occur (for POP) if the stack is empty or if the given list is empty, or (for PUSH) if the stack contains fewer than 2 objects.
Bad Argument Type will occur (for POP) if the input object does not evaluate to a list, or (for PUSH) if the Level-2 input object does not evaluate to a list..

Notes: Coordinated use of POP and PUSH will allow you to maintain a named or unnamed stack. POP is identical to UNQ.

List Utilities: A Discussion

The Main Idea

Lists are the most general purpose data objects provided by the HP-28S. They may be any size from 0 to the limits of memory – containing *any* HP-28S data objects in any combination – including other lists. This flexibility gives you tremendous control over what you put into lists and how you use them.

On the other hand, because lists are so generic, there are very few commands built into the HP-28S to manipulate them. Yet with these powerful few, many of the possibilities of lists can be realized with a handful of programs. The tools in this section are such a handful – a set of some of the more generally useful list operations. They're still very generic – because only you will know the specifics of manipulating the actual lists you've created – but they can still help you manipulate your lists regardless of how you've organized their information.

Where To Put These Programs

As always, to be accessible, these utilities must be in a directory that is listed in your current PATH. The easiest way to ensure that this is the case is to place each of the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

Lists have properties that make them similar to arrays (and vectors) and characters strings. You can see this in that the first "page" of the LIST menu is similar to that of the first page of the ARRAY menu, while the second page of the LIST menu is similar to that of the STRING menu. The + operation is also analogous between strings and lists, as it appends the two objects in stack Levels 1 and 2.

The ability to GET and PUT objects from and to a list allows you to create and maintain ordered sets of objects in the same way that arrays do. But unlike arrays, the dynamic length (SIZE) of a list and the ability to search a list by content – as well as by index – allows you to create dynamic data structures, like stacks or queues, that shrink or grow based on their current information.

You'll therefore find utilities here that exploit both these characteristics of lists, and these routines are indeed very analogous to string and array tools found elsewhere in this book.

Errors And Error Recovery

Each of the tools is designed to generate an error for invalid input, rather than continue and possibly generate garbage outputs. When errors do occur, the stack is almost invariably disrupted, and since the only way to restore a disrupted stack is with the UNDO command, it's wisest to activate UNDO mode (in the MODES menu) and leave it active throughout your use of these utilities.

How You Might Use These Utilities

One of the advantages of lists is their ability to contain different types of objects. You can thus create different data aggregates that are effectively new data types. And once you define such a data type, you can then create the tools you need to operate on it.

Consider, for example, a list in which each entry contains a person's name, birthdate and telephone number:

```
{ "Smith, John" { 2 6 1959 } 5553426 }
```

You could easily create a list of such objects and then (not quite as easily) create new commands to do things such as these:

1. Add an element (an entry).
2. Delete an element, given its index.
3. Display an entry, based on its index.
4. Sort the list by last name.
5. Sort the list by some other attribute.
6. Search for a name and return that element.
7. Find all of the birthdates that fall on today's date.

To implement each of these seven commands, here's how you might proceed:

1. Create the new element and add it to the end of the list (with `+`) or insert it somewhere among the current entries (with `LINS`).

   Note that since each element is itself a list, you must put it into another list before adding. That is,

```
{ "Doe, Jane" { 6 2 1960 } 5559812 }
{ "Smith, John" { 2 6 1959 } 5553426 }
+
```

gives

`{ "Doe, Jane" { 6 2 1960 } 5559812 "Smith, John" { 2 6 1959 } 5553426 }`, which is incorrect; the information from both entries has been combined into a single list.

However,

```
{{ "Doe, Jane" { 6 2 1960 } 5559812 }}
{{ "Smith, John" { 2 6 1959 } 5553426 }}
+
```

gives

`{ { "Doe, Jane" { 6 2 1960 } 5559812 } { "Smith, John" { 2 6 1959 } 5553426 } }`, which is correct.

2. Use `LDEL`.

3. Use GET to get the element and a routine something like the following to display it: DSP (165651)

```
« « →STR » LOP LIST→
DROP "■" + ROT "■" +
ROT "■" + ROT + + 1
DISP »
```

(The ■ characters are NEWLINE characters and should be keyed in as such.)

4. Fortunately, the last name is the first thing in the object. So a procedure such as

```
« →STR » LOP LSORT « STR→ » LOP
```

will do the job.

5. Generally, any time you want to sort the list, you'll need to convert its objects into strings so that they can be compared (and keep in mind that in the general case, the object to be sorted by is not necessarily the first object in the element). You'll therefore need a conversion routine to transform the element both before and after the sort. To sort by birthdate (or rather, birth-*month*), for example, the simplest procedure would be:

```
« 1 2 LEX →STR » LOP LSORT « STR→ 1 2 LEX
» LOP
```

6. What you want to do is filter the list, returning only those elements that contain the search string. So:

```
« →STR "Smith" POS » FLTR
```

The `"Smith"` is the search string – whatever name you're searching for.

7. Something like this will work:

```
« 2 GET 1 2 SUB { 2 6 } == » FLTR,
```

where `{ 2 6 }` is an example list of today's month and day.

Chapter 8

Directory Utilities

These routines provide quick and reliable ways to edit, test and traverse directory structures in your HP-28S.

As shown in the following list, the 10 programs are organized into three logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 248, immediately following these program listings.

Alphabetically Sort The Contents Of The Current Directory:

DSORT (366834)

```
« VARS IF DUP SIZE
THEN « →STR » LOP
LSORT « STR→ » LOP
ORDER ELSE DROP END
»
```

Summary: DSORT reorders the contents of the current directory so that the USER menu is displayed in alphabetical order.

Example: DSORT

Inputs: None.

Outputs: None.

Errors: None.

Notes: DSORT uses LOP and LSORT from Chapter 7. DSORT can take a few minutes to run if the current directory is a large one.

Remove A Directory And Its Contents:

KILLD (1620927)

```
« → n.. « IF n..
DIR? THEN n.. EVAL
VARS 'DIR?' FLTR IF
DUP SIZE THEN → d..
« 1 d.. SIZE FOR i..
d.. i.. GET KILLD
NEXT » ELSE DROP END
CLUSR DU END n..
PURGE » »
```

Summary: KILLD removes (purges) a directory and its contents. If it is used on a named object, the name is purged. It cannot be used on the HOME directory.

Example: 'Q' KILLD

Inputs: Level 1 – a directory or name – the directory to be purged.

Outputs: None.

Errors: Too Few Arguments will occur for an empty stack. Bad Argument Type will occur if the input object is not a name or directory.

Notes: `KILLD` uses `DIR?`, `DU` and `FLTR` (from Chapter 7). *Caution:* `KILLD` provides no margin for error and can quickly destroy huge amounts of data!

Move And/Or Rename An Object:

MOVE (6293966)

```
« 0 → a.. b.. p.. «
IF b.. TYPE 5 ==
THEN b.. 1 OVER SIZE
1 - SUB b.. DUP SIZE
GET 'b..' STO ELSE
PATH END 'p..' STO
PATH  p.. GOTO IF
b.. DIR? b.. 1 →LIST
{ HOME } == OR THEN
p.. b.. + 'p..' STO
a.. 'b..' STO END
DUP GOTO IF a.. DIR?
THEN DEPTH → d.. «
a.. STACKEM DROP p..
GOTO b.. DUP CRDIR
EVAL DO EVAL UNTIL
DEPTH d.. == END »
ELSE a.. RCL a..
PURGE p.. GOTO b..
STO END GOTO » »
```

Summary: MOVE will move the named object from the current directory to the specified directory. The object to be moved may be either a named object or a directory. The destination may be either a directory path or name. If the destination name or the last name in the directory path is not a directory, the object to be moved will also be renamed using that name.

Examples:

```
'PETE' 'FRED' MOVE
'PETE' { HOME JANE FRED } MOVE
```

Inputs: Level 2 – a name object – the name of the object to be moved from the current directory.
Level 1 – a name object or list – the destination to which to move the Level-2 object.

Outputs: None.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if the Level-2 object is not a name or directory.
`Undefined Name` (and maybe a stackful of garbage) will occur if the Level-2 name is empty, or if the destination directory is a sub-directory of the directory to be moved, or if any of the names in the destination list except the last one are undefined.

Notes: `MOVE` is a sophisticated command with potentially destructive effect. It should therefore be used only by experienced HP-28S users. **DO NOT** try to move a parent directory to one of its descendents. The remains of the parent directory will be left on the stack with little chance of restoring it. Having a corrupt or otherwise incorrect destination path is perhaps the worst of all possible errors, because the source directory will have already been removed and placed on the stack before an attempt is made to move to the destination. Therefore, the best procedure is to move to the target directory and invoke the PATH command to get the correct destination path.

MOVE uses GOTO, DIR? and STACKEM. MOVE, GOTO, DIR? and STACKEM *must* be in the PATHs of both the source and destination directories. The only reasonable place for these four tools, therefore, is in the HOME directory.

Place The Contents Of A Directory Onto The Stack

STACKEM (3152576)

```
« → n.. «IF n.. DIR?
THEN 'DU' n.. EVAL
VARS IF DUP SIZE
THEN → l.. « 1 l..
SIZE FOR i.. l.. i..
GET STACKEM NEXT »
ELSE DROP END "«"
n.. →STR " CRDIR "
OVER " EVAL»" + + +
+ STR→ DU n.. PURGE
ELSE n.. RCL n.. n..
PURGE « STO » END »
»
```

Summary: STACKEM places the contents of the name object onto the stack, along with the name and any commands necessary to recreate the object. The previous contents of the name object are PURGE'd as if with KILLD. The stacked information can then be restored by repeated execution of EVAL until all of the stacked objects have been removed.

Examples: `1 'A' STO 'A' STACKEM`

Result: `1 'A' « STO »`

```
'A' CRDIR A 'B' CRDIR B 6 'C' STO DU
DU 'A' STACKEM
```

Result: (on the stack)

```
'DU'
'DU'
6
'C'
« STO »
« 'B' CRDIR 'B' EVAL »
« 'A' CRDIR 'A' EVAL »
```

Inputs: Level 1 – a name object – the name of the object to be stacked.

Outputs: Levels 1 to *n* – the contents of the object, along with the commands necessary to recreate it.

Errors: `Too Few Arguments` will occur if there are no arguments on the stack.
`Bad Argument Type` will occur if the Level-1 object is not a name or directory.

Notes: `STACKEM` uses `DU` and `DIR?`

Test Whether An Object Is A Directory:

DIR? (365895)

```
« RCLF → D F « 31 CF
IFERR D RCL THEN 64
STWS ERRN # 12Ah ==
ELSE DROP 0 END F
STOF » »
```

Summary: DIR? tests the given object to see if it is a directory. It returns 1 if the object is a directory and 0 if not.

Examples:

1 DIR?	Result:	0
'FRED' DIR?	Result:	1

```
IF J DIR? THEN YES ELSE NO END
```

Result: (this example program segment will evaluate the routine YES if the object, J, is indeed a directory; or the routine, NO if it is not.)

Inputs: Level 1 – the object to be tested.

Outputs: Level 1 – a real number – either 1 or 0 (true or false).

Errors: Too Few Arguments will occur if the stack is empty.

Notes: The binary integer in the program listing is shown in hexadecimal. It will appear differently if the current binary mode is other than HEX. The results of `DIR?` are intended to be compatible with other logical tests in the HP-28S (as illustrated with the `IF` statement in the **examples**).

Test For An Empty Name:

MT? (280133)

```
« RCLF SWAP 31 CF
IFERR RCL THEN 64
STWS ERRN #204h ==
ELSE DROP 0 END SWAP
STOF »
```

Summary: MT? tests a named object to determine whether or not it is empty. All non-name objects are considered to be non-empty and therefore return a 0.

Examples:

`'FRED' PURGE 'FRED' MT?`	Result: 1
`1 'PETE' STO 'PETE' MT?`	Result: 0

Inputs: Level 1 – an object – the object to be tested.

Outputs: Level 1 – a real number – either 0 or 1 (false or true).

Errors: None.

Notes: None.

Find The Type Of The Named Object:

NTYPE (420162)

```
« → N « IF N MT?
THEN -1 ELSE IF N
DIR? THEN 11 ELSE N
RCL TYPE END END » »
```

Summary: NTYPE tests the named object and returns its type (consistent with that returned by the built-in command, TYPE, plus added type values of -1 and 11): -1 = Empty; 0 = Real; 1 = Complex; 2 = String; 3 = Real array; 4 = Complex array; 5 = List; 6 = Global name; 7 = Local name; 8 = Program; 9 = Algebraic; 10 = Binary integer; 11 = Directory.

Example: `{ 5 } 'A' STO 'A' NTYPE` <u>Result</u>: 5

Inputs: Level 1 – a name – the object whose type is to be tested.

Outputs: Level 1 – an integer from -1 to 11 – the type of the input.

Errors: Too Few Arguments will occur for an empty stack. Bad Argument Type will occur if the Level-1 input object is neither a name object nor a directory object.

Notes: NTYPE uses MT? and DIR?.

Move Up One Directory (Directory Up):

DU (142235)

```
« PATH DUP SIZE 1 -
IF DUP THEN GET EVAL
ELSE DROP2 END »
```

Summary: DU moves from the current directory up to the one immediately above it. If the current directory is already the HOME directory, no action is taken.

Example: DU

Inputs: None.

Outputs: None.

Errors: None.

Notes: None.

Find A Name In The Directory Tree:

FIND (1376855)

```
« → N « IF VARS DUP
N POS THEN PATH SWAP
END 'DIR?' FLTR IF
DUP SIZE THEN → D «
1 D SIZE FOR I D I
GET DUP 1 DISP EVAL
N FIND DU NEXT »
ELSE DROP END » CLMF
»
```

Summary: `FIND` recursively traverses the directory tree – starting at the current directory – looking for the given name object. It returns the path (a list of directories) of the directory in which the named object is found. If the name exists in more than one directory, the path to each occurence will be returned. If the name is not found, no path is returned. Since it can take some time to search all subdirectories, `FIND` displays the name of the directory it's currently searching to show how far it has progressed.

Example: `'FRED' FIND`
Result: `{ HOME PETE JOE EMILY }`

Inputs: Level 1 – a name object – the object being sought.

Outputs: Levels 1 to *n* – the paths to each of *n* occurences of the name. Nothing is returned if the name is not found.

Errors: None.

Notes: If the Level 1 object is not a name, `FIND` will still search the directory tree for a match, but of course it won't find one.

`FIND` uses `DIR?`, `DU`, and `FLTR` (from Chapter 7).

Go To A Directory By Using A Path List:

GOTO (1218662)

```
« { HOME } LIST→
DROP → H « LIST→ IF
DUP THEN 1 FOR I I
ROLL IF DUP DIR?
OVER H == OR THEN
EVAL ELSE 1 LIST→
END -1 STEP ELSE
HOME DROP END » »
```

Summary: GOTO moves to the directory specified by the given PATH list. The PATH list need not start from HOME – just some directory in the path of the current directory.

Example: `{ HOME A B C } GOTO`

Inputs: Level 1 – a list – the PATH list of directory entries.

Outputs: None.

Errors: Too Few Arguments will occur for an empty stack. Bad Argument type will occur either if the argument is not a list or it contains a non-directory element.

Notes: GOTO uses DIR?.

Directory Utilities: A Discussion

The Main Idea

Directories are great for organizing and partitioning user memory. Logical groups of data and programs can be created and named within their own directory. Then moving between directories and calling routines in other directories is as easy as calling their names – as long as they can be found in the current directory's PATH (i.e. as long as they're somewhere between where you're "calling from" and HOME).

These directory tools enhance the usefulness of directories with their often-needed functions. You can: MOVE to any directory; FIND any object within the subtree of the current directory; DSORT the names in the current directory; purge an entire directory sub-tree; move up to the parent directory; and do some useful directory-related tests.

Where To Put These Programs

Unlike most of the other utilities in this book, the directory utilities are almost useless unless they're placed in the HOME directory. This is because many of them can "move you" out of the current PATH, thus preventing you any further access to commands that exist only in that PATH. *The only directory that is always a member of all PATHs is the HOME directory, so put these utilities in the HOME directory.*

Some Observations

The HP-28S' directory structure is a fairly standard, multi-way tree. That bears a bit of explaining: The "tree" is the overall structural pattern, starting with a main (HOME) level, or "root node." Every point of branching is called a node. At a node there might be one or more "leaves" (items with actual evaluable data in it) and/or "branches" (paths leading to further nodes).

In some tree structures, there are rules about how many leaves and branches can be attached to any one node. The good news is that a multi-way tree (as in the HP-28S) has no limit to these numbers: you can put as many leaves (programs or data objects) and branches (sub-directories in any directory level (node) as you want.

Recursive programming techniques are one of the best ways to perform tree traversals and access the data in the nodes and leaves. Therefore, all of the routines that either move or remove data from node to node do so recursively.

If you have some difficulty understanding what the previous sentence is saying, you get the point: The routines in this chapter are probably the most sophisticated routines in this book, and it would take far more space than is available here to fully explain recursive programming techniques.

However, if you want to begin to explore them, the best way is to study how they are used here – in the programs that "call themselves:" FIND, KILLD, and STACKEM. You might also look at QSRT in Chapter 1.

Some of these routines are useful mainly as keyboard commands, but several are particularly handy in writing your own programs (and you can get some idea of their relative usefulness by noting the frequency with which they occur within other programs in this section): DIR?, DU, GOTO, MT? and NTYPE.

STACKEM was developed as a subprogram for MOVE, but you may find it convenient as a tree "pruner" and "grafter" in your own programs – without the additional overhead of MOVE.

Errors And Error Recovery

For the most part, these tools make every effort to cause errors *before* much movement of data has occurred. But to keep the routines relatively small, not all conceivable precautions have been implemented. It is very possible to destroy a lot of information with these routines. *In most cases UNDO will not help you either, because data movement and/or destruction has occurred outside of the stack.*

How You Might Use These Utilities

Tests

Unfortunately, the built-in TYPE command isn't particularly consistent when dealing with directories. If you create a directory, then place its name on the stack and invoke TYPE, you'll get 6, telling you that the object was a name. This is "sort of" correct but not really: You can't store into or recall from a directory as you can with a name.

Therefore, the tools in this section include some functions that allow you to test whether or not an object is specifically a directory. DIR? asks the question, "Is this object a directory?" NTYPE actually extends the idea of object types to return the type of the given named object, including type number -1 for an empty name and 11 for a directory. MT? tests to see if the Level-1 name is empty and returns 0 – false – if it contains a directory.

Going And Coming Back

The information returned by the PATH command is nice to tell you where you are within the directory tree, but you can't do anything else with it. GOTO remedies this by allowing you to go to the directory specified by the PATH list.

Then, when you go somewhere, it's nice to know how to get back. Use the following methods to "remember" where you've been and get back:

```
PATH whereto GOTO dosomething GOTO
```

or

```
PATH → whereiwas « whereto GOTO
dosomething whereiwas GOTO »
```

`whereto` is the PATH-list of your (temporary) destination and `dosomething` is what you want to do while you're there. You use PATH before GOTO to get the location of the current directory. Then, after the task is completed, you recall this previous PATH and use GOTO to get back there once again.

FIND And GOTO

GOTO is also useful after the FIND command – to go to the directory containing the object you just found. For example:

```
« PATH → thing
whereiwas « thing
FIND GOTO thing EVAL
whereiwas GOTO » »
```

The program takes the name of an object to be sought (`thing`) and the current directory PATH from the stack. FIND finds `thing` in the directory tree and GOTO goes to it. `thing` is evaluated and GOTO then returns to the previous directory (this assumes that FIND will actually find a `thing` and that it will find only one of them).

STACKEM As An Alternative To MOVE

You can think about the MOVE utility as a mess of preparation to call the STACKEM program. MOVE checks and corrects the inputs, then uses STACKEM, then moves to the destination directory and repeatedly invokes EVAL to placed the stacked objects in the new directory.

If you're uncomfortable with this level of automation (and with your own hard-won data and programs at stake, this is quite understandable), you can invoke STACKEM manually, then move to the target directory and press EVAL repeatedly to restore the information. If you're unsure of yourself or the program, watching it work in this way can be reassuring.

KILLD Versus PURGE And CLUSR

PURGE deletes either a single object or a list of named objects from the current directory. CLUSR deletes evey named object in the current directory. But neither PURGE nor CLUSR will delete a non-empty directory.

KILLD, on the other hand, deletes a single named object from the current directory – *even if that object is a non-empty directory.* Thus KILLD "rounds out" your ability to remove objects from memory by allowing you to delete a directory in one fell swoop. As such, KILLD is a very destructive command and should be used with extreme caution, much as you would CLUSR – only more so!

Chapter 9

Output Utilities

These routines provide convenient, "canned" methods for formatting output to the HP-28S display or printer – both with character and graphic information.

As shown in the following list, the 20 programs are organized into five logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 280, immediately following these program listings.

Name	Function	Page
	Printer Positioning Routines	
PRCTR	Print An Object Centered	274
PRLJ	Print An Object Left-Justified	274
PRRJ	Print An Object Right-Justified	274
PRPUT	Print An Object At A Specified Column	276
	Printer Special Effects	
PRDW	Print An Object Double-Wide	277
PRINV	Print An Object In Inverse (White On Black)	277
PRUDL	Print An Object Underlined	277
PRPAT	Print A Character Pattern	279

Center An Object In A Display Line:

DCTR (35294)

```
« SWAP 23 SCTR SWAP
DISP »
```

Left-Justify An Object In A Display Line:

DLJ (6109)

```
« DISP »
```

Right-Justify An Object In A Display Line:

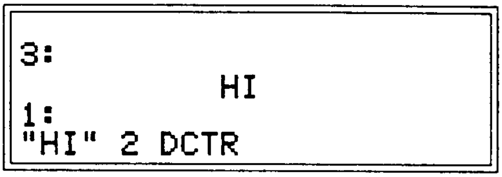
DRJ (30630)

```
« SWAP 23 SRJ SWAP
DISP »
```

Summary: DCTR displays the Level-2 object, centering it on the display line specified in Level 1. DLJ displays the Level-2 object left-justified, starting on the display line specified in Level 1. DRJ displays the Level-2 object right-justified on the display line specified in Level 1.

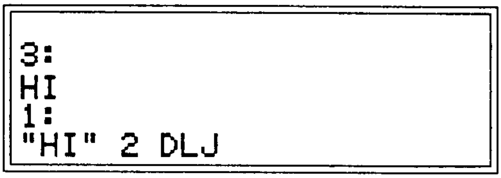
If the line specified is less than 1, then line 1 is used. If the line specified is greater than 4, then line 4 is used. The line specifier is rounded before use. Since the display's width is an odd number of characters (23), centered objects with an even number of characters will be spaced one character farther to the left than to the right. If an object contains NEWLINE characters, DLJ and DRJ will begin its display on the line specified, then continue on subsequent lines as directed by the NEWLINE's.

Examples: `"HI" 2 DCTR` Result:



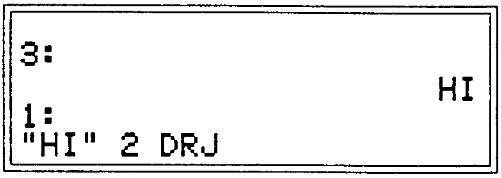
```
3:
          HI
1:
"HI" 2 DCTR
```

`"HI" 2 DLJ` Result:



```
3:
HI
1:
"HI" 2 DLJ
```

`"HI" 2 DRJ` Result:



```
3:
                     HI
1:
"HI" 2 DRJ
```

Inputs: Level 2 – the object to be displayed.
Level 1 – a real number – the line on which that object is to be displayed.

Outputs: (Displays such as in the **Examples**.)

Errors: `Too Few Arguments` will occur if the stack contains fewer then 2 objects.
`Bad Argument Type` will occur if the Level-1 input object is not a real number.
`DCTR` will fail if the object to be displayed contains a NEWLINE character or leading or trailing spaces.

Notes: `DCTR` uses `SCTR`. `DRJ` uses `SRJ`. `DLJ` is an alias for the HP-28S command, `DISP`, so as to be consistent with the naming of `DRJ` and `DCTR`.

Put An Object Into A Display Line, Beginning At A Specified Column:

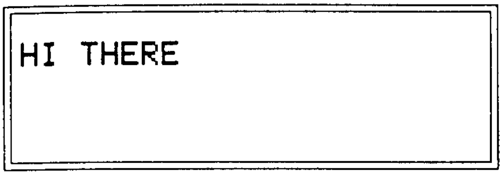
DPUT (64769)

```
« 1 MAX .5 + FLOOR
23 MIN SWAP 1 MAX .5
+ FLOOR 4 MIN → C R
« →STR 1 24 C - SUB
LCD→ SWAP SPAT 137 R
1 - * C 1 - 6 * +
SWAP SPUT →LCD » »
```

Summary: DPUT displays the specified object, starting at the given line and column, without clearing the rest of that line (unlike DISP). If the line specified is less than 1 or greater than 4, the object is displayed on line 1 or 4, respectively. If the column number is less than 1, it is treated as 1. If the specified column is greater than 23, nothing is displayed. If the displayed object extends beyond the end of the display or contains a NEWLINE character, it is truncated at that point. Line and column numbers are rounded before use.

Example: `CLLCD "HI" 1 DISP "THERE" 1 4 DPUT`

Result:



```
HI THERE
```

Inputs: Level 3 – the object to be displayed.
Level 2 – a real number – the line on which the object is to be displayed.
Level 1 – a real number – the column at which the object's display is to begin.

Outputs: (A display such as in the **Example**.)

Errors: `Too Few Arguments` will occur if the stack contains fewer then 3 objects.
`Bad Argument Type` will occur if the Level-1 and Level-2 objects are not both real numbers.

Notes: `DPUT` uses `SPAT` and `SPUT`.

Display A Character Pattern:

DPAT (930121)

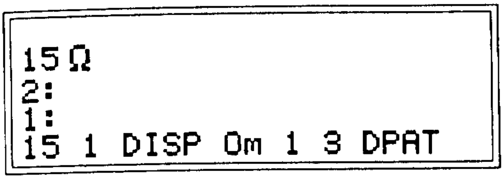
```
« 1 MAX .5 + FLOOR
23 MIN 1 - 6 * SWAP
1 MAX .5 + FLOOR 4
MIN → C R « C OVER
SIZE + 137 SWAP -
OVER SIZE + 1 SWAP
SUB LCD→ SWAP 137 R
1 - * C + 1 + SWAP
SPUT →LCD » »
```

Summary: DPAT takes a character string from Level 3 of the stack (which is of the form returned by LCD→) and displays it (as →LCD) starting at the display line as given by the number in Level 2 and at the character column within that line, as given by the number in Level 1. If the line number specified is less than 1, 1 is used. If the line specified is greater than 4, 4 is used. If the column number is less than 1, 1 is used. If the column number is greater than 23, 23 is used. Line and column numbers are rounded before use.

Example:

```
94 CHR 97 CHR 1 CHR
97 CHR 94 CHR 0 CHR
+ + + + + 'Om' STO
15 1 DISP Om 1 3 DPAT
```

Result:



```
15Ω
2:
1:
15 1 DISP Om 1 3 DPAT
```

Inputs: Level 3 – a character string – the character pattern to be displayed.
Level 2 – a real number – the line on which the character is to be displayed.
Level 1 – a real number – the column at which the character's display is to begin.

Outputs: A display such as in the **Example**.

Errors:: `Too Few Arguments` will occur if the stack contains fewer then 3 objects.
`Bad Argument Type` will occur if the Level-3 object is not a character string, or if the Level-1 and Level-2 objects are not both real numbers.

Notes: `DPAT` uses `SPUT`. The typical HP-28S character is 6 columns wide, with each character in the pattern building a column. The 6th column is usually blank (character 0) so as to leave space between adjacent characters.
`DPAT` is useful for building and displaying user-created special characters (e.g. greek math symbols).

Invert A Display Line (To Inverse Video):

DINV (234106)

```
« LCD→ SWAP 1 - 3
MIN .5 + FLOOR 137 *
1 + SCUT 138 SCUT
SWAP NOT SWAP + +
→LCD »
```

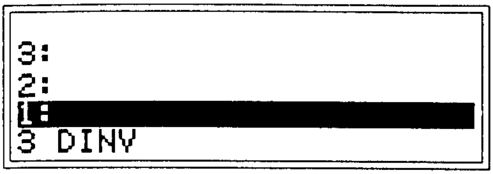
Underline A Display Line:

DUDL (398305)

```
« 1 MAX .5 + FLOOR 4
MIN →NUM LCD→ SWAP 1
- 137 * 1 + SCUT 138
SCUT SWAP 128 CHR
137 SRPT OR SWAP + +
→LCD »
```

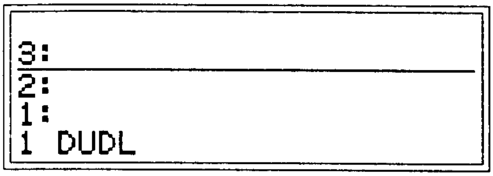
Summary: DINV inverts the specified display line (black to white and vice versa). DUDL underlines the specified display line. The line specifier defaults to 1 and 4 for inputs outside of those limits. Fractional portions of line specifiers are rounded.

Examples: `3 DINV` Result:



```
3:
2:
1:
3 DINV
```

`1 DUDL` Result:



```
3:
2:
1:
1 DUDL
```

Inputs: Level 1 – a real number – the display line to be inverted or underlined.

Outputs: (A display such as in the **Example** above.)

Errors: `Too Few Arguments` will occur for an empty stack. `Bad Argument Type` will occur if the Level-1 object is not a real number. `Undefined Name` will occur if the Level-1 object is an undefined name.

Notes: `DINV` uses `SCUT`. `DUDL` uses `SCUT` and `SRPT`.

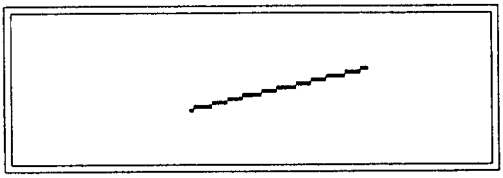
Draw A Line Between Two Points:

LINE (1472131)

```
« →NUM SWAP →NUM → R
L « L PIXEL IF L R ≠
THEN PXDM C→R R L -
DUP C→R 4 ROLL / ABS
SWAP 4 ROLL / ABS
MAX PPAR 4 GET /
SWAP OVER / L 1 4
ROLL START OVER +
DUP PIXEL NEXT DROP2
R PIXEL END » »
```

Summary: LINE draws a line in the display between the two given points (complex number objects). The current PPAR values (PMIN, PMAX, and RES) are used. If RES is greater than 1, the line is drawn to the specified resolution. If the variable PPAR does not exist, it is created with default values.

Example: `(0,0) PMIN (3,3) PMAX 1 RES`
`(1,1) (2,2) CLLCD LINE` <u>Result</u>:



Inputs: Level 2 – an object that reduces to a complex number – the coordinates of the "from" point.
Level 1 – An object that reduces to a complex number – the coordinates of the "to" point.

Outputs: A line is drawn in the display.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if the input objects do not reduce to complex numbers.

Notes: `LINE` does not test to see if the points it is plotting are out of bounds for the display. `LINE` uses `PXDM`.

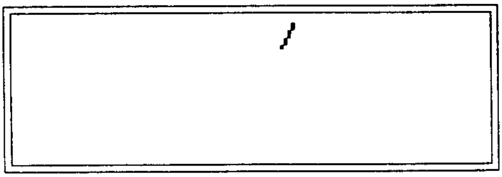
Draw A Line From The Current Plot Position To The Specified Position:

PLOT (373667)

```
« →NUM → N « IF
coord DUP TYPE 6 ==
THEN (0,0) DUP ROT
STO END N LINE N
'coord' STO » »
```

Summary: PLOT draws a line from the point established by PSET to the specified point. The current plot position – the value of 'coord' – is updated to be that of the specified endpoint. If 'coord' has no value, it is given a value of (0, 0). If the variable PPAR does not exist, it is created with default values.

Example: 'PPAR' PURGE (1,1) PSET (2,3) CLLCD PLOT <u>Result</u>:



Inputs: Level 1– an object that reduces to a complex number.

Outputs: A line is drawn in the display.

Errors: Too Few Arguments will occur for an empty stack. Bad Argument Type will occur if the input value or the contents of 'coord' is not a complex number.

Notes: PLOT does not test whether the points being plotted are out of bounds for the display. PLOT uses LINE.

Plot A Series Of Connected Points:

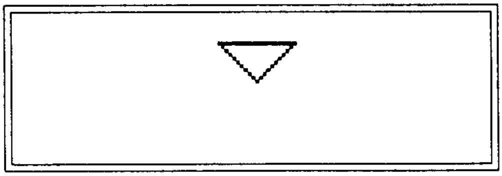
POLYL (874768)

```
« EVAL → L « L 1 GET
→NUM 2 L SIZE IF DUP
TYPE 5 == THEN LIST→
2 == « * » IFT END
FOR I L I GET →NUM
SWAP OVER LINE NEXT
DROP » »
```

Summary: POLYL ("poly-line") takes a list of complex numbers or a complex array object and plots lines between the points it contains. The first point is taken to be the origin, so the first line is drawn from that point. Arrays are traversed in row-major order. If RES is greater than 1, the line is drawn to the specified resolution. If the variable PPAR does not exist, it is created with default values.

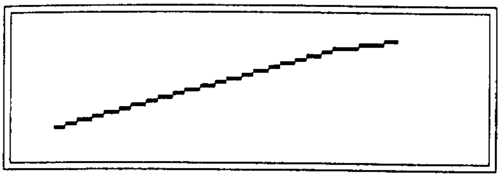
Examples: `'PPAR' PURGE { (0,0) (1,1) (-1,1) (0,0) } CLLCD POLYL`

Result:



```
(-10,-3) PMIN (10,3) PMAX
[ (5.3,2.4) (3,2) (-8,-2.1) ]
CLLCD POLYL
```

Result:



Inputs: Level 1 – any object that evaluates to a list, array or vector – the series of point coordinates to be plotted.

Outputs: A plot is generated.

Errors: `Too Few Arguments` will occur for an empty stack. `Bad Argument Type` will occur if the input object does not evaluate to a list, array or vector, or if one or more of its components is not a complex number.

Notes: `POLYL` does not test to see if the points it is plotting are out of bounds for the display. `POLYL` uses `LINE`.

Move To A New Point In A Plot:

PSET (44268)

```
« →NUM C→R R→C
'coord' STO »
```

Summary: PSET establishes a point in the plotting area from which a subsequent line can be drawn with the PLOT program.

Example: (2,2) PSET
<u>Result</u>: (nothing changes in the display)

Inputs: Level 1 – a complex number; the coordinates of the point to which to move.

Outputs: None.

Errors: Too Few Arguments will occur if the stack is empty.
Bad Argument Type will occur if the input object is not a complex number.

Notes: PSET creates the variable 'coord' in the current directory, overwriting any variable of the same name.

Get The Dimensions Of A Pixel

PXDM (156215)

```
« PPAR 1 2 SUB LIST→
DROP SWAP - C→R 31 /
SWAP 136 / SWAP R→C
»
```

Summary: PXDM determines the height and width of a display pixel (a single dot), given the current plotting parameters (i.e. the contents of the PPAR variable). The dimensions are returned as a complex number. PXDM assumes that the PPAR variable exists.

Examples: `STD 'PPAR' PURGE (0,0) PIXEL PXDM`
Result: `(.1,.1)`

Inputs: None.

Outputs: Level 1 – a complex number – the dimensions of a display pixel under the current plotting parameters.

Errors: `Bad Argument Type` will occur if `'PPAR'` does not exist or does not contain a valid plotting parameter list.

Notes: None.

Print An Object Centered:

PRCTR (25977)

```
« 24 SCTR PR1 DROP »
```

Print An Object Left-Justified

PRLJ (44143)

```
« DEPTH « PR1 » IFT
DROP »
```

Print An Object Right-Justified

PRRJ (22066)

```
« 24 SRJ PR1 DROP »
```

Summary: PRCTR prints the given object, centered on the printer paper. PRLJ prints it left-justified; PRRJ prints it right-justified. If the object contains NEWLINE characters, PRRJ and PRLJ will begin displaying the object on the line specified, then continue on subsequent lines as directed by the NEWLINE's in the object.

Examples: "HI" PRCTR Result: (printed on the paper):

```
            HI            
```

"Whoop" PRLJ Result: (printed on the paper):

```
Whoop                     
```

"hi" PRRJ Result: (printed on the paper):

```
                        hi
```

Inputs: Level 1 – the object to be printed.

Outputs: (print-outs such as in the **examples**.)

Errors: Too Few Arguments will occur for an empty stack. PRCTR will fail to center its output if the object contains a NEWLINE character or leading or trailing spaces. PRLJ and PRRJ will fail if the print buffer is not empty before the command is invoked (the print buffer is emptied by CR or any printing command executed with flag 33 clear), or if the object contains leading spaces.

Notes: PRCTR uses SCTR. PRRJ uses SRJ.

Print An Object At A Specified Column:

PRPUT (123506)

```
« " " SWAP 1 MAX
→NUM 1 - SRPT SWAP
→STR + PR1 DROP »
```

Summary: PRPUT prints the given object at the given printer column. A column number less than 1 is treated as 1. Column numbers greater than 24 will print the object on line number (n DIV 24), column number (n MOD 24) + 1.

Example: `"HI" 20 PRPUT` Result:

```
                   HI
```

Inputs: Level 2 – the object to be printed.
Level 1 – a real number – the printer column on which to begin printing the object.

Outputs: (A print-out such as in the **Example**.)

Errors: `Too Few Arguments` will occur if the stack contains fewer then 2 objects.
`Bad Argument Type` will occur if the Level-1 object is not a real number.
`Undefined Name` if the Level 1 object is an undefined name.

Notes: PPUT uses SRPT.

Print An Object Double-Wide:

PRDW (140624)

```
« 27 CHR DUP 253 CHR
+ ROT →STR + SWAP
252 CHR + + PR1 DROP
»
```

Print An Object In Inverse (White On Black)

PRINV (1306107)

```
« RCLF → S « →STR
"■" + 33 SF WHILE
DUP "■" POS DUP
REPEAT SCUT 2 SCUT
SWAP DROP SWAP DO 23
SCUT SWAP SPAT NOT
PRPAT UNTIL DUP SIZE
NOT END DROP CR END
DROP2 S STOF » »
```

Print An Object Underlined:

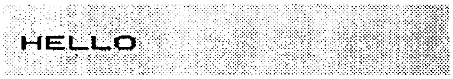
PRUDL (145490)

```
« 27 CHR DUP 251 CHR
+ ROT →STR + SWAP
250 CHR + + PR1 DROP
»
```

Summary: PRDW prints the given object double-wide on the printer paper. PRINV prints it in inverse (white on black). PRUDL prints it underlined.

Examples: "HELLO" PRDW

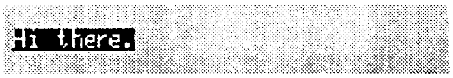
Result: (printed on the paper):



```
HELLO
```

"Hi there." PRINV

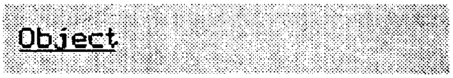
Result: (printed on the paper):



```
Hi there.
```

"Object" PRUDL

Result: (printed on the paper):



```
Object
```

Inputs: Level 1 – the object to be printed.

Outputs: (A print-out such as in the **Example** above.)

Errors: Too Few Arguments will occur for an empty stack.

Notes: PRINV uses SCUT, SPAT and PRPAT.

Print A Character Pattern:

PRPAT (386107)

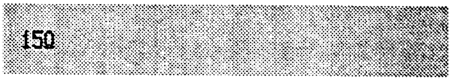
```
« DO DUP 1 166 SUB
27 CHR OVER SIZE CHR
+ SWAP + PR1 DROP 1
166 SDEL UNTIL DUP
SIZE NOT END DROP »
```

Summary: PRPAT takes a character string from Level 1 of the stack (of the form returned by LCD→) and prints the corresponding character pattern.

Example:

```
94 CHR 97 CHR 1 CHR
97 CHR 94 CHR 0 CHR
+ + + + + 'Om' STO
33 SF 15 PRLJ 33 CF Om PRPAT
```

Result: (printed on the paper):



Inputs: Level 1 – a character string – the character pattern to be printed.

Outputs: (A print-out such as in the **Example.**)

Errors: Too Few Arguments will occur for an empty stack.

Notes: PRPAT uses SDEL.

Output Utilities: A Discussion

The Main Idea

The HP-28S is not intended to be a general purpose computer. It is intended to be a very competent calculator. For that reason (as you have certainly noticed), very little emphasis is placed on sophisticated, built-in input and output capabilities. Indeed, the only way to put information into the machine is with your own fingers; and the only ways to get information out of it are through the display and through the (optional) infrared printer.

Even so, sometimes a bit of output formatting becomes important. Displayed results of complicated or data-intensive programs may be quite confusing – and you'll often need to refer to printouts long after the actual calculations have been performed – so you'll certainly need some intelligible organization and labelling for both the display and printer. The tools in this section let you do just that.

Where To Put These Programs

As always, to be accessible, these utilities must be in a directory that is listed in your current PATH. The easiest way to ensure that this is the case is to place each of the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

Most of the printer and display commands are applications of related string commands, including the ability to make general, graphical patterns. Underlining and inverse video (white on black printing) and are common display/printing enhancements that are also included.

There is also a handful of plotting extensions, including a general purpose line-drawing routine (LINE) that allows you to draw a line between any two points in the plotting window. The other routines basically extend this idea by allowing you to plot several lines consecutively.

Errors And Error Recovery

Each of the tools is designed to generate an error with invalid input, rather than continue and potentially generate garbage outputs. When errors do occur, the stack is almost invariably disrupted, and since the only way to restore a disrupted stack is with the UNDO command, it's wisest to activate UNDO mode (in the MODES menu) and leave it active throughout your use of these utilities.

How You Might Use These Utilities

Connect Data Points With DRWΣ

Suppose that the array ΣDAT contains data that you've just entered and sorted by the independent variable's column (you know how to do this with the Array utilities from Chapter 5, right?).

Suppose also that ΣPAR contains valid COLΣ data. You might then use POLYL to connect the points in the DRWΣ scatter plot, thereby outlining a trend (if any), with the help of this short program, called CNCTΣ (360196):

```
« 0 PREDV DROP RCLΣ
ΣPAR 1 GET AGETC
RCLΣ ΣPAR 2 GET
AGETC (0,1) * + V→A
SCLΣ CLLCD DRWΣ
POLYL »
```

Labelling Printouts

Suppose you have an array of data that you want to print out column by column, with each column labelled. You might use the following program, called PCOL (1346283):

```
« → A « CRON CR
"COLUMN DATA" PRCTR
"=" 24 SRPT PRLJ CR
A EVAL SIZE LIST→
DROP → R C « 1 C FOR
J "Column " J SIP +
PRLJ 1 R FOR I A
EVAL { I J } GET
PRRJ NEXT CR NEXT »
» »
```

Extending Display Graphics

Suppose you want to be able to draw certain patterns and shapes in the display – arcs, boxes, arrows, etc. How would you go about building your own extended set of display graphics utilities?

The first thing to do, of course, is to decide what capabilities you want, then recognize the core routines you would need to make them work. Just to get you started, here's a couple of possibilities (the programming is left up to you):

ARC1	Given a center of curvature, one endpoint of the arc, and an angle (current angular mode), ARC1 plots the arc, using ARC2, PXDM and 'PPAR' as needed.
ARC2	Given a center of curvature and the two endpoints of an arc, ARC2 plots the corresponding arc, using PXDM and 'PPAR' as needed.

Chapter 10

Programming Utilities

These routines provide convenient, "canned" methods for conducting various object and system tests, controlling system parameters, and for allowing dynamic program control and evaluation.

As shown in the following list, the 41 programs are organized into ten logical groups, presented alphabetically. Within each group, the programs are also usually presented alphabetically (by NAME), although in some cases, certain sets of programs may be complementary or otherwise so similar that they may be presented together.

For a more in-depth discussion of the uses of these utilities, see page 306, immediately following these program listings.

Name	Function	Page
	Tests For Angle Modes	
ANG?	What Is The Angle Mode?	295
DEG?	Is The Angle Mode DEGrees?	295
RAD?	Is The Angle Mode RADians?	295
	Tests For Array Dimensionality	
AR1D?	Is The Given Array 1-Dimensional?	296
AR2D?	Is The Given Array 2-Dimensional?	296
DIM?	What Is The Dimensionality Of The Given Array?	296
	Tests For Binary Modes	
BASE?	What Is The Binary Mode?	298
BIN?	Is The Binary Mode BINary?	298
DEC?	Is The Binary Mode DECimal?	298
HEX?	Is The Binary Mode HEXadecimal?	298
OCT?	Is The Binary Mode OCTal?	299
	Tests For Display Formats	
DIGS?	How Many Digits Are Being Displayed?	300
ENG?	Is The Display Format ENGineering?	300

Evaluate The Object Corresponding To A Test Value:

CASE (32291I)

```
« → I V R O « IF V
EVAL I POS DUP THEN
R SWAP GET EVAL ELSE
DROP O EVAL END » »
```

Summary: The CASE conditional allows you to evaluate one of several objects based on the value of the supplied object In other words, in the case that the test object has such and such a value, such and such object will be evaluated. Its value lies in replacing "nested" IF statements (IF ... THEN IF ... THEN IF ...).

Examples:

```
0 { 0 1 2 } { "REAL" "CPLX" "STR" }
"INVALID" CASE
```
Result: "REAL"

```
2 { 0 1 2 } { "REAL" "CPLX" "STR" }
"INVALID" CASE
```
Result: "STR"

```
5 { 0 1 2 } { "REAL" "CPLX" "STR"}
"INVALID" CASE
```
Result: "INVALID"

Inputs: Level 4 –any object – the object of the test.
Level 3 – a list – the choice of possible objects.
Level 2 – a list – the outcomes corresponding to the object choices.
Level 1 – any object – a default object to be evaluated if the test object does not match any of the choices.

Outputs: The possible outputs dependent on which of the objects are evaluated and what that evaluation yields.

Errors: `Too Few Arguments` will occur if there are fewer than 4 objects on the stack.
`Bad Argument Type` will occur if the Level 2 and 3 objects do not evaluate to lists.

Notes: None.

Evaluate The Object Indexed By A Test Value:

ON (63953)

```
« → C L « L EVAL C
→NUM GET EVAL » »
```

Summary: ON provides a means of selecting an action based on the value of an object, which is used directly as an index to a list of options. The selected option is then evaluated. The index value is rounded to an integer before use. The index *must* select a valid option or an error will occur.

Examples: `2 { "A" "B" "C" } ON` Result: `"B"`
`STD 4 1 { « SQ 1 - » « SQ 2 * 2 - » } ON` Result: `15`

Inputs: Level 2 – any object that evaluates to a real number – the index.
Level 1 – any object that evaluates to a list – the options.

Outputs: The output is dependent on the option evaluated.

Errors: `Too Few Arguments` will occur if the stack contains fewer than 2 objects.
`Bad Argument Type` will occur if the objects don't evaluate to their prescribed types.
`Bad Argument Value` will occur if the index is out of range for the list.

Notes: Various unpredictable errors can occur based on the evaluation of the selected list object.

Generate An Error Beep:

ERRBP (20649)

```
« 1400 .075 BEEP »
```

Summary: ERRBP generates a beep of the same pitch and duration as the standard error beep. It can be used to signal errors in user-created programs.

Example: ERRBP

Inputs: None.

Outputs: None.

Errors: None.

Notes: Flag 51 controls the status of the HP-28S tone generator. If this flag is clear, a BEEP will produce a tone; if the flag is set, no action is taken.

Find The Percentage Of Total System Memory Still Available:

MEM% (40031)

```
« MEM 32397.5 SWAP
%T 2 RRND »
```

Summary: MEM% returns the percent of memory free. A maximum available memory of 32397.5 bytes is assumed and the result is rounded to 2 decimal places.

Examples: STD MEM% Result: 71.51

Inputs: None.

Outputs: Level 1 – A real number.

Errors: None.

Notes: MEM% uses RRND.

Test And Controls For The Beeper:

BPON (7800)

```
« 51 CF »
```

BPOFF (8873)

```
« 51 SF »
```

BEEP? (9633)

```
« 51 FC? »
```

Summary:	BPON and BPOFF turn the internal beeper on and off respectively, thus enabling/disabling subsequent executions of BEEP. BEEP? tests to see whether the beeper is enabled, returning a 1 (true) if it is and a 0 otherwise.
Examples:	`BPON STD BEEP?` Result: 1 `BPOFF STD BEEP?` Result: 0
Inputs:	None.
Outputs:	Level 1 (for BEEP? only) – a real number, either a 0 or 1 – the result of the test
Errors:	None.
Notes:	None.

Test And Controls For The Printer's Automatic Carriage Return

CR? (7446)

```
« 33 FC? »
```

CROFF (8910)

```
« 33 SF »
```

CRON (7837)

```
« 33 CF »
```

Summary: CRON and CROFF enable and disable, respectively, the printer's automatic carriage return. CR? tests whether it is enabled, returning a 1 (true) if so and a 0 otherwise.

Examples:

```
CRON STD CR?     Result: 1
CROFF STD CR?    Result: 0
```

Inputs: None.

Outputs: Level 1 (for CR? only) – a real number, either a 0 or 1 – the result of the test.

Errors: None.

Notes: None.

Tests For Angle Modes:

ANG? (36022)

```
« { DEG RAD } RAD? 1
+ GET »
```

DEG? (8382)

```
« 60 FC? »
```

RAD? (8568)

```
« 60 FS? »
```

Summary: These routines test the current angle mode. DEG? and RAD? return a 1 (true) or 0 (false), indicating whether the tested-for mode is active. ANG? returns the current mode's name: DEG or RAD.

Examples: DEG RAD? Result: 0 DEG DEG? Result: 1
RAD ANG? Result: RAD

Inputs: None.

Outputs: DEG? and RAD? return either a 1 or 0 to Level 1. ANG? returns a program object, either DEG or RAD.

Errors: None.

Notes: ANG? returns *evaluable* program objects. It uses RAD?.

Tests For Array Dimensionality:

AR1D? (12552)

```
« DIM? 1 == »
```

AR2D? (12577)

```
« DIM? 2 == »
```

DIM? (91400)

```
« SIZE LIST→ 1 == 1
IFT 2 →LIST 1 POS 1
2 IFTE »
```

Summary: These routines all test the dimensions of the given array. DIM? returns the dimension of the array: 1 or 2 (HP-28S vector objects are considered to be one-dimensional arrays). AR2D? returns true or false, based on whether the given array is two-dimensional or not. AR1D? performs a similar test for one-dimensional arrays.

Examples: STD [1 2 3] DIM? Result: 1

```
STD [ 1 2 3 ] AR2D?
```
Result: 0

```
STD [ 1 2 3 ] AR1D?
```
Result: 1

```
STD [[ 1 2 ] [ 3 4 ]] AR2D?
```
Result: 1

```
STD [[ 1 2 3 ]] AR1D?
```
Result: 1

Inputs: Level 1 – an array or vector object.

Outputs: Level 1 – `DIM?` returns either 1 or 2, `AR2D?` and `AR1D?` return either 0 or 1.

Errors: `Too Few Arguments` will occur if the stack is empty.
`Bad Argument Type` will occur if the input object is not an array or vector.

Notes: None.

Tests For Binary Modes:

BASE? (96471)

```
« { DEC BIN OCT HEX
} 43 FS? 2 * 44 FS?
+ 1 + GET »
```

BIN? (23273)

```
« 43 FC? 44 FS? AND
»
```

DEC? (22724)

```
« 43 FC? 44 FC? AND
»
```

HEX? (23677)

```
« 43 FS? 44 FS? AND
»
```

OCT? (23466)

```
« 43 FS? 44 FC? AND
»
```

Summary: These routines all return information about the current binary integer format.
All the other tests return either a 1 (true) or 0 (false), depending upon whether or not the tested-for mode is active. BASE? returns the current mode's name: BIN, DEC, HEX or OCT.

Examples:

BIN BASE?	Result: BIN
HEX BIN?	Result: 0
OCT OCT?	Result: 1

Inputs: None.

Outputs: BIN?, DEC?, HEX? and OCT? all return either a 1 or 0 to Level 1. BASE? returns one of the following program objects: BIN, DEC, HEX, OCT.

Errors: None.

Notes: The objects returned by BASE? are actually evaluatable programs.

Tests For Display Formats:

DIGS? (83403)

```
« 53 FS? 54 FS? 2 *
+ 55 FS? 4 * + 56
FS? 8 * + »
```

ENG? (23389)

```
« 49 FS? 50 FS? AND
»
```

FIX? (23493)

```
« 49 FS? 50 FC? AND
»
```

FMT? (93483)

```
« { STD SCI FIX ENG
} 49 FS? 2 * 50 FS?
+ 1 + GET »
```

SCI? (23385)

```
« 49 FC? 50 FS? AND
»
```

STD? (23444)

```
« 49 FC? 50 FC? AND
»
```

Summary: These tests all return information about the current numerical display format. FIX?, SCI?, ENG? and STD? all return either a 1 (true) or 0 (false), indicating whether the tested-for mode is active. DIGS? returns the number of displayed digits in the current mode (0 is returned for STD mode). FMT? returns the current mode's name: FIX, SCI, ENG or STD.

Examples:

`4 FIX DIGS?`	Result: `4.0000`
`STD DIGS?`	Result: `0`
`11 SCI FMT?`	Result: `SCI`
`8 ENG STD?`	Result: `0.00000000E0`
`3 SCI SCI?`	Result: `1.000E0`

Inputs: None.

Outputs: Level 1 – FIX?, SCI?, ENG? and STD? return a real number, either a 1 or 0 – the result of the test. DIGS? returns a real number, between 0 and 11 – the number of display digits. FMT? returns a program object, either FIX, SCI, ENG, or STD – the current display format.

Errors: None.

Notes: The evaluable objects returned by FMT? take the number of display digits from the stack (except for STD).

Tests For Object Types:

ALGB? (13021)

```
« TYPE 9 == »
```

ARRY? (35513)

```
« TYPE { 3 4 } SWAP
POS »
```

BNRY? (14804)

```
« TYPE 10 == »
```

CARY? (13436)

```
« TYPE 4 == »
```

CPLX? (13532)

```
« TYPE 1 == »
```

LIST? (13634)

```
« TYPE 5 == »
```

LOCL? (13330)

```
« TYPE 7 == »
```

NAME? (13172)

```
« TYPE 6 == »
```

PRGM? (13532)

```
« TYPE 8 == »
```

RARY? (13653)

```
« TYPE 3 == »
```

REAL? (13170)

```
« TYPE 0 == »
```

STR? (12260)

```
« TYPE 2 == »
```

Summary: These routines are all tests that return true or false (1 or 0) based on whether or not the object is of the type for which the test is being made:

Routine	Object Type Being Tested For
ALGB?	Algebraic Object
ARRY?	Array or Vector
BNRY?	Binary Integer
CARY?	Complex-Valued Array
CPLX?	Complex Number
LIST?	List
LOCL?	Local Name
NAME?	Global Name
PRGM?	Program Object
RARY?	Real-Valued Array
REAL?	Real Number
STR?	Character String

Examples:

```
"HI" STR?              Result: 1
STD (1,0) CPLX?        Result: 1
« STD » REAL?          Result: 0
```

Inputs: Level 1 – any object

Outputs: Level 1 – a real number, either a 1 or 0 – the result of the test.

Errors: `Too Few Arguments` will occur for an empty stack.

Notes: None.

Waits For Keystrokes:

GETK (32655)

```
« DO UNTIL KEY END »
```

KEYWAIT (53950)

```
« DO UNTIL KEY END
DROP »
```

Summary:	GETK pauses program execution to get a key. Once a key is pressed, the key name is returned and the program continues. KEYWAIT waits for a keystroke before continuing, but does not return a value.	
Examples:	GETK [9]	Result: "9"
	KEYWAIT [7]	Result: (wait)
Inputs:	None. After the routines are invoked, any keystroke other than [ATTN] will be accepted.	
Outputs:	Level 1 – (GETK only) a character string – the name of the key pressed.	
Errors:	None.	
Notes:	Pressing [ATTN] will interrupt either routine, potentially leaving a 0 on the stack.	

Programming Utilities: A Discussion

The Main Idea

Most of these tools are in the Programming section because they are useless in manual calculations. For instance, how often would you need a program to tell you the type of object in stack Level 1 when you can look right at it and see for yourself? On the other hand, this sort of tool is very useful within a program to give it the intelligence to determine such conditions – what the Level 1 object is, or what the current display mode is, etc.

There are also some tools here that give you new ways to control program evaluation, control calculator states and gather user input.

Where To Put These Programs

As always, to be accessible, these utilities must be in a directory that is listed in your current PATH. The easiest way to ensure that this is the case is to place each of the programs in the HOME directory – the ultimate parent of all other directories.

Some Observations

Perhaps here more than in any other chapter of this book the use of the tools is governed by your own preferences. Most of the tests and flag controlling commands are very short programs, barely necessary unless you can't remember the number of the beeper control flag, or the type number of a character string. Their (hopefully) meaningful names and one-keystroke entries are timesavers. And if you design programs that use them heavily, you might save some memory, also.

You'll probably find that in most situations, the programs you write are independent of the current calculator state. It is easy to save the current state, set a new state for the duration of the program, then reset the previous state before leaving. Take `SIP` as an example:

```
« →NUM IP →NUM RCLF
SWAP STD →STR SWAP
STOF »
```

`RCLF` recalls the states of all the system and user flags. Then various other modes may be safely set by the program (in this case `STD`), since they will be undone afterward, by a restoration of the original flag settings with `STOF`. This is how you can make any program mode independent – so that it will work the same regardless of what modes it encounters as it begins execution.

Note, however, that you can also write programs that work *differently* with different system settings. For example, you might have a program calculation whose accuracy varies according to the current display setting, or which gets different results in degrees or radians mode. With that in mind, some of the more useful routines in this section are those that allow you to direct program control dynamically. For example, GETK and KEYWAIT allow programs to interact with the user, taking actions based on the identity of a single keystroke.

ON (after the HP-BASIC ON...GOTO... statement) simply evaluates an argument and uses the integer result to select and evaluate an object from a list. Since the objects in the list may be programs or names of procedure objects, ON is a conditional evaluation based on an index.

The CASE command is similar, but takes an argument and compares it with a list of values. If it matches one of the values, the returned index is used to select an object to be evaluated from a second list. If the argument matches none of the options, a catch-all object is evaluated. Thus, CASE is similar to the following Pascal construct:

```
CASE object IN
     1 : first_condition;
     2 : second_condition;
     3 : third_condition;
     OTHERWISE not_1_2_or_3;
END;
```

Errors And Error Recovery

Each of these tools is designed to generate an error with invalid input, rather than to continue and possibly generate bad output. When errors do occur, the stack is usually disrupted, and since the only way to restore a disrupted stack is with the UNDO command, it's wisest to activate UNDO mode (in the MODES menu) when using these utilities.

GETK

Because GETK returns a string object to the stack, a program that uses GETK can use the return value in several ways to provide conditional action. First, using CASE, you could do this:

```
GETK { "LEFT" "RIGHT" "UP" "DOWN" }
{ GOLEFT GORIGHT GOUP GODOWN } 'IDLE' CASE
```

But consider this simpler and faster approach:

```
GETK "GO" SWAP + STR→
```

Or, you could provide the 'IDLE' option in the following way:

```
GETK "GO" SWAP + STR→ DROP IDLE
```

Thus, for example, if you press [9], STR→ evaluates to 'GO9' which (hopefully) does not name an object and therefore the name is left on the stack. The program continues, DROPping the name and running the IDLE routine. Beware of possible errors though: After STR→ evaluates the named routine, DROP expects to drop something and IDLE is evaluated regardless.

Now, if all of the called routines were designed to return a 1 to indicate their completion, the following routine could be used:

```
GETK "GO" SWAP +
STR→ IF 1 SAME NOT
THEN IDLE END
```

or:

```
GETK "GO" SWAP +
STR→ 1 SAME NOT
'IDLE' IFT
```

(As you can see, the only real difference here is the form of the IF test being used.)

Type Testing

ON and CASE provide convenient means of *branching* – choosing at "run-time" among several program options, based upon values arrived at during the program. A common example of this is testing a program's arguments to see what TYPE they are and processing differently based on the object. For example:

```
« DUP TYPE { 0 1 2 }
{ Real Complex String }
"Bad Type" CASE »
```

Here is a routine that decomposes "decomposable" objects, but does nothing for other types of objects. Note that there is one list element for every possible value returned by TYPE.

```
OBJ→ (804254)

« DUP TYPE 1 + { « »
« C→R » « STR→ » «
ARRY→ DROP » « ARRY→
DROP » « LIST→ DROP
» « » « » « » « » «
» } ON »
```

Or, to avoid having to provide a list element for every TYPE value, you can use an IF test to restrict the range:

```
OBJ2→ (983461)

« DUP TYPE IF DUP 0
> OVER 6 < AND THEN
{ « C→R » « STR→ » «
ARRY→ DROP » « ARRY→
DROP » « LIST→ DROP
» } ON ELSE DROP END
»
```

Utilities Index

HP-28S Software Power Tools: **Utilities**

Here's a must-have collection of HP-28S tools – program routines to make your own programming go more quickly and smoothly!

You get ten different sets of "canned" solutions, ready for "off the shelf" use: Utilities for **the stack, real numbers, complex numbers, vectors, arrays, strings, lists, directories, output,** and **program development.**

Each routine is documented, and there's a discussion at the end of each set, to give you some examples of how to use those utilities. You'll find this book to be a great collection of advice, good habits and sound programming principles. Whether you're just starting to program or are experienced already, these all-purpose tools belong in your HP-28S and under your fingers – don't miss them!

FROM THE PRESS AT
GRAPEVINE PUBLICATIONS, INC.
P.O. Box 118 • Corvallis, Oregon 97339-0118 • U.S.A. • (503) 754-0583

ISBN 0-931011-27-2

0 12841 00027 9

HP Part # 92220Y