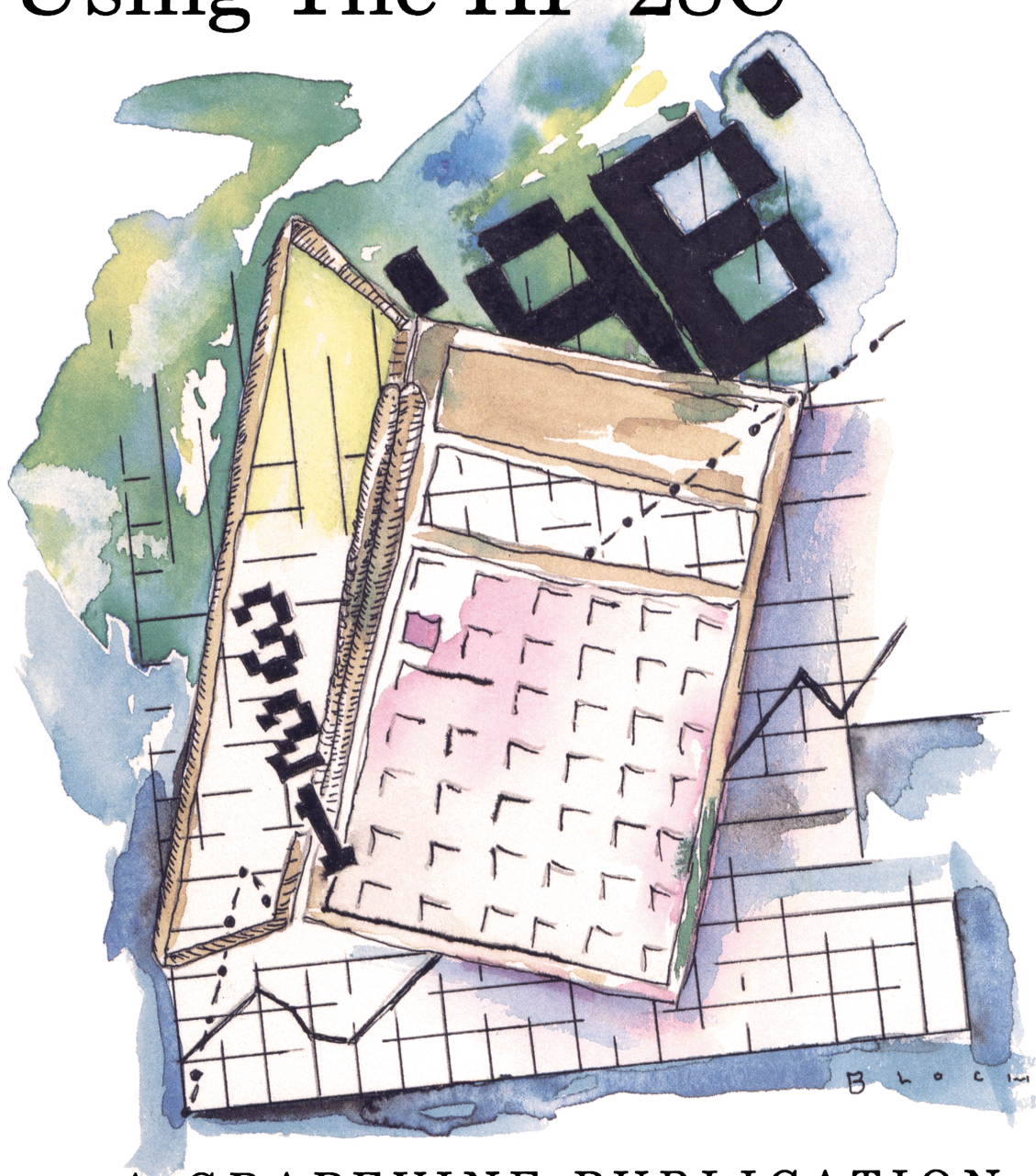


# An Easy Course In

---

# Using The HP-28C



A GRAPEVINE PUBLICATION

By John W. Loux and Chris Coffin

Illustrated by Robert L. Bloch





# AN EASY COURSE IN USING THE HP-28C

By John W. Loux  
and Chris Coffin

Illustrated by Robert L. Bloch

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

## Acknowledgement

Thanks and appreciation go once again to the Hewlett-Packard Company for continuing to produce such top-quality products and documentation. For the sake of brevity, the full name of this calculator, "HP-28C," has been shortened to "HP-28" throughout this book.

© 1987, Grapevine Publications, Inc. All rights reserved. No portion of this book or its contents, nor any portion of the programs contained herein, may be reproduced in any form, printed or mechanical, without written permission from Grapevine Publications, Inc.

Printed in The United States of America

First Printing – August, 1987

ISBN 0-931011-17-5

**DISCLAIMER:** Neither the authors nor Grapevine Publications, Inc. make any express or implied warranty with regard to the keystroke procedures and program material herein offered, nor to their merchantability nor fitness for any particular purpose. These keystroke procedures and program material are made available solely on an "as is" basis, and the entire risk as to their quality and performance is with the user. Should the keystroke procedures or program material prove defective, the user (and not Grapevine Publications, Inc., nor the authors, nor any other party) shall bear the entire cost of all necessary correction and all incidental or consequential damages in connection with, or arising out of, the furnishing, use, or performance of these keystroke procedures or program material.

# CONTENTS

<b>Introducing...The Introduction</b>	<b>7</b>
What Is This Tool?	9
What Is This Book?	10
What's In This Book – and What's Not?	12
 <b>How to Picture Your HP-28</b>	 <b>13</b>
The Display	16
The Keyboards	17
Posting Memos: Interactions Between the Keyboards and the Display	18
The Menu Keys: Your Command Card File	26
Immediate Execution ("Do-It-Now") Keys	32
Messages From the System – Memos From Your Staff	35
Status Messages: The Annunciator Area	37
A Tricorder Reading	38
Quickie Quiz	39
Quickie Answers	40
 <b>Making Your HP-28 Work For You: The Command Line</b>	 <b>41</b>
Typing Characters Into the Command Line	42
Changing A Character in the Command Line	43
Adding and Removing Characters	47
$\blacksquare$ INS, $\blacksquare$ DEL, and $\square$ ATTN	50
$\blacksquare$ NEWLINE, $\blacktriangle$ and $\blacktriangledown$	53
The $\square$ LC Key	56
$\square$ and $\blacksquare$ $\alpha$ LOCK	57
Item Delimiters and $\square$ ENTER	60
$\blacksquare$ EDIT and $\blacksquare$ COMMAND	62
Command Line Quiz	65
Command Line Answers	66
 <b>Real Numbers, the Stack, and Postfix Notation</b>	 <b>67</b>
Real Numbers – and the Real World	68
Representing Real Numbers On the HP-28	70
Scientific Notation on Your HP28	70
12-Digit Accuracy: Rounding Error	71



Magnitude: How Big (or Small) Can You Get?	73
Posting Real Numbers: <b>[CHS]</b> , <b>[EEX]</b> and Display Modes	76
Display Formats	78
The Stack and Postfix Notation	81
Real Number Commands: 0-, 1-, and 2- Number Operations	82
Arithmetic Practice	88
Arithmetic Practice Solutions	90
STACK Operations	91
<b>[ENTER]</b> 's Second Job	92
The <b>[SWAP]</b> Function	93
How to <b>[CLEAR]</b> the Stack	93
Strenuous But Practical Stack Practice Problems	105
S.B.P.S.P.P. Solutions	106
 <b>The "Stuff" Upon Which the HP-28 Works</b>	 <b>109</b>
An Equal Opportunity Calculator	110
The HP-28's Philosophy of Information	111
Real Numbers	112
Complex Numbers	114
Pop Quiz: Simple Questions About Complex Numbers	121
Simple Answers to Simple Questions About Complex Numbers	122
Vectors	123
A Visit With Vectors	133
Results of A Visit With Vectors	134
Arrays	136
Array Aptitude Test	141
A.A.T. Results	142
Characters	145
Character Strings	146
Character String Query	153
C.S.Q. Answers	154
Names	155
Name Games	161
Name Game Winners	162
Bits	164
Binary Integers	165
Binary Integer Test	167
B.I.T. Answers	168

A Pause For the Cause	169
Lists	170
List Lessons	175
List Lessons Learned	176
Procedures: (a) Postfix Programs	177
Program Problems	181
Program Problem Solutions	182
Procedures: (b) Algebraic Expressions	185
Algebraic Aptitude Test	193
A.A.T. Scores	194
Procedures: (c) User-Defined Functions	197
User-Defined Function Fun	201
U.D.F.F. Consequences	202
<b>Appendices</b>	<b>205</b>
Introduction to the Appendices	206
Appendix A: Algebra	207
Appendix B: Using the Equation Solver	215
Appendix C: Plotting	221
Appendix D: Postfix Programming	225
Appendix E: Keyboard Error Recovery	230
Editorial	234







**INTRODUCING...THE INTRODUCTION**

## Well now...so you have an HP-28, eh?

Now the questions are, "What do you want to do with it?" and (mostly), "*How?*"

These are the right questions to ask, of course. And you may have heard that once you've decided *what* you want to do, the *how* should be intuitively obvious – even to the most casual observer.

That's just not true. There's nothing wrong with your intuition or your personal casualness index. It's simply that this machine is not all that simple. Even if you're experienced with other HP calculators, this one is so *radically* different that you may find yourself "starting over" in many respects. You may still recognize some familiar HP stack-oriented arithmetic, but that's about where the similarities end. For the most part, the HP-28 will probably be a "brave new world."

Of course, as with all calculators, the HP-28 is only a tool, a problem-solving tool. So is a hammer. And though it's fairly obvious (even to extremely casual hammer-observers) that a hammer is good for pounding, it takes more than casual observation to use it effectively in building a house. It takes time and practice.

So it is with the HP-28. Being just a bit more complex than a hammer, it does require more effort on your part to use it effectively. But once you make that effort, you'll be amazed at the "houses" you can build with it.

That's the purpose of this book – to help you learn to use a tool. Just be sure to remember that it *is* just a tool, not a magic box that gives you the answer to your every question. It can't check to see if you've given it the right numbers to "crunch," nor can it catch you when you're attacking a problem altogether wrongly. It's an inanimate mechanical aid – not a replacement for your understanding of the problem. You *must* understand both your tool *and* your problem in order to use the one on the other.

## What Is This Tool?

Before you begin to use the HP-28 as such a problem-solving tool, you'd better have at least some idea that it's actually the *right* tool for the job.

This calculator is not omnipotent. It does some things very well and other things not so well. It is flexible, but for some tasks, it may cost you more effort to bend it to your will than it's worth. In those cases, you would come out ahead by choosing another, more appropriate tool.

So what is the HP-28 really "good at?"

Mostly, it's a math engine. It provides you with an extensive set of mathematical operations. And it uses these operations on a fairly comprehensive set of mathematical "things": real numbers, vectors, arrays, complex numbers, and algebraic expressions, to name a few. So if a lot of your problems involve such math, then the HP-28 is probably as good a "hammer" as any you will find.

But it's *not* a generalized computer. It doesn't have "bigabytes" of memory, nor the means to save your calculations anywhere else (i.e., on magnetic tape or disc). It can't run a very large variety of "user-friendly" software. For example, you wouldn't want to try to type your doctoral thesis on it.

Of course, with some considerable effort you *could* coerce it into doing many different things, but don't be surprised (or upset) if the results are not the best. After all, you *can* drive a screw with a hammer, but if you do and then things don't turn out very well, don't go blaming the hammer for not being a screw driver. It just wasn't built for that.



## What Is This Book?

There are (at least) four ways to approach your learning about the HP-28:

1. Be Joe Computer-Whiz, for whom it is either intuitively obvious or the essence of joy to play with such a machine until it yields all of its secrets;
2. Apply brute force – not knowing where to start, but pressing buttons anyway, hoping that something meaningful will result;
3. Resort to tears and despair (usually as a result of method 2);
4. Ask for some help and explanation (usually as a result of method 3).

If you're now using method 4, then this book is meant for you. And there's absolutely nothing wrong with that. It carries no shame or stigma to say "I don't understand this yet." You just haven't yet seen it explained in a way that "clicked" for you. This book is merely a different way to explain the HP-28.

Admittedly, it doesn't appeal to everybody. You may find the pace too slow or the explanations too meticulous. But chances are, there is *something* presented here that could "shed more light on your HP-28" for you. So relax and browse, if nothing else. A lot of people discover the same thing – that such a slow, classroom-style approach seems to work better than the "brute force" method.

Above all, please don't feel that you're being "talked down to" by this Easy Course.

Just because the printing is large and spread out and the wording is simple and "folksy," you shouldn't take this as any commentary on your technical expertise or vocabulary. The subjects here are *not* trivial, nor is your intellect being trivialized by seeing them presented in this fashion. The only reason for all this is to communicate to you the skills and knowledge you need to make the best use of a very sophisticated tool. Apparently, this method of communication often helps.

And how does this method go? Here are a few things to know about the book and its classroom approach:

1. Every so often, you'll come across a little set of quiz problems. These are just some exercises to help you make sure you've "got things under control" before you move on. If you have any major difficulties with the questions, you'll find the answers immediately following, along with page numbers so you can go back and review if you wish.
2. At certain points along the way, you'll be given the option to skip ahead if you feel that you already know the material being discussed. If you do skip, it will usually be to the quiz at the end of that section. This way, you can be sure you're really as up to speed as you thought.
3. There's no race, no time limit, no clock, no exam proctor or #2-pencil-grading-machine breathing down your neck. This is *your* Course, to be taken at *your* speed. Who cares if you go back and reread something a couple of times? The idea is to learn about your calculator, not to break a speed record for doing so.

## What's In This Book – and What's Not?

This book is *not* a re-packaging of the manuals that came with your HP-28. Many keys, features and functions on your calculator just don't appear anywhere in this book – and this is no accident. Why should anyone try to document every last aspect of the machine? That's what the HP manuals do so well; why try to improve on them?

Instead, what you're going to see here are the fundamental concepts and principles of the HP-28. Of course, you'll need to learn the mechanics of the keyboard and the display first, but the real idea here is to orient yourself and move around in the generalized data-manipulation world of the HP-28. By the time you finish this Easy Course, you should feel quite comfortable in using and combining the different available data "objects" to help solve your math problems.

But all the while, keep in mind that this is only an *introduction* to the HP-28.

Why just an introduction? Two reasons:

First of all, this book is meant to help you get enough "calculator savvy" to begin building a more exhaustive understanding of this tool on your own terms – and in your own way. Hopefully, it's enough to get you on your way down that road, without unduly burdening you with a load of preferences and biases as to how you ought to actually apply this tool to your everyday tasks. At some point *you* must take over and decide for yourself exactly when and how to use your HP-28.

Secondly, there's only so much room in one book!





**HOW TO PICTURE YOUR HP-28**

Before you can really do anything in the world of the HP-28, you'll need to know how to move around in it. Learning these mechanical skills isn't always a whole lot of thrills – but it is necessary.

So you need to realize right up front that these next three chapters are really just a set of lessons in controlling and communicating with your calculator. Boring as that may sound, don't underestimate the importance of these skills. OK?

Now then: A picture is still worth a thousand words, so it makes some sense to have a picture of the HP-28's world to help you understand it.

Of course, the picture you'll need isn't exactly an 8-by-10 color glossy of the calculator (with circles and arrows and writing on the back).

First of all, who needs such a photo when you have the real thing? And anyway, you may already know from personal experience that you can stare at the physical HP-28 until drops of blood bead up on your forehead and you still won't come any closer to understanding how the machine works.

So, because that's what you really want to know – *how* the thing works and not what it looks like – you'll need instead a picture of something that doesn't physically exist, the *logic* of the calculator.

Unfortunately, the camera that can take that picture hasn't been invented. But you might try another method instead: mind games....

The name of this particular mind game is "Easy-Course-Warmer-Upper-HP-28-Mental-Picture-Of-Its-Logic." (Sort of catchy, don't you think?) It's for all ages and requires only one player and one mind (and since you seem to "have a mind" to use your HP-28, you'll do quite nicely, thank you).

As you might suspect, the object of the game is to paint for yourself a mental picture you can use as a map to explore the unfamiliar world of the HP-28. But there are no rules; you just make it up as you go along.

So, if you're ready, flex your mental muscle, and sure enough, a picture begins to form in your mind's eye....

- - - - -

The first thing you see is, not surprisingly, the HP-28 (see it there?). Of course, as you've already observed, this doesn't get you very far (especially if, in your mental picture, you've forgotten to open the calculator).

So you concentrate even harder, focusing in on its two most obvious features, the display and the keyboards, and slowly but surely, a better picture forms....

- - - - -

You're the newly-elected president of a very talented little company of mathematicians who make their livings by solving problems for others. By prior arrangement, these mathematicians have their offices inside the HP-28.

Of course, as president, your job is to properly delegate and assign tasks, so that the overall results are those requested by your clients. You're the go-between, understanding and translating your clients' needs into terms that your staff can understand and act upon.

## The Display

Think of the display of your HP-28 as a bulletin board, and picture it that way. It's how you communicate with your staff (the calculator).

Upon reflection, you'll find that this makes quite a bit of sense because the display is *interactive*. That is to say, it changes as you and the calculator do things to change it.

As with a real bulletin board, messages are posted in the display by *you* (for the machine) – problems to solve, numbers to store or "crunch", etc.

And messages are posted by the *machine* (for you) – status reports, information and graphs for your inspection and correction.

You'll see as you go along that your bulletin board is quite well organized with different messages from different departments posted in different areas on the board. And you'll also find that, very much like a real-world bulletin board, your display bulletin board can become cluttered. New messages can obscure or even "bump off" old messages.

Not to worry though. There are ways to tell your "staff" that whenever they need to post messages that would "bump off" other ones, they should save such bumped messages, just in case you want to look at them again. Your staff will obey this – and all your instructions – if you make them clear.

Actually, all things considered, you have a fairly well-organized, imaginary math-problem-solving business here.

# The Keyboards

The next areas to notice are the keyboards (and you should probably continue to think of them as two distinct keyboards rather than as two halves of one keyboard because there are some significant differences in how each is used).

Continue with your mental picture: If the display is your bulletin board, then you can envision the keyboards as your typewriter or dictation recorder. After all, as president you need some way of creating memos and messages for posting on the bulletin board, right? OK, draw it in your mind as a typewriter (got it?), and look at how this typewriter is arranged.

First, look at the left-hand keyboard. If you pay attention only to the white letters on the keys, the left-hand keyboard really does look like a typewriter with its keys rearranged. And indeed, these keys are used for typing words and phrases (ignore the other, less obvious things on the keyboard for now. You'll come back to them later as you need them).

Likewise, if you look at the right-hand keyboard and notice only the white keys with black lettering, you see what appears to be a simple, four-function calculator. Again, this is how it ought to appear; that's exactly what those keys are for (and again, ignore for now the other, less self-explanatory keys.)

-----

So, as a new president, you're beginning to at least find your way around the office. Review your picture up to this point:

You have a bulletin board (the display) through which you communicate with your staff (the HP-28 system). You also have a simple, desktop calculator and a typewriter to use in writing memos for posting. Not bad.

Next thing to figure out: How do you actually post memos?

## Posting Memos: Interactions Between the Keyboards and the Display

As you might expect, in order to get any work out of your staff, you need to tell them what to do – i.e. post a memo, after composing it on your typewriter.

Of course, right now is when you realize that your typewriter actually types directly onto the bulletin-board (quite a high-tech office, really).

Unlikely? Well, yes, it's true that things don't work exactly like this in the real world, but it doesn't stretch your imagination too much to picture it this way nevertheless.

Now then, it's time for everyone's favorite game (yep – even company presidents like to play):

"Press the Pretty Buttons and See What Happens."

But before you do that, stop and think for a minute: Whatever memos are posted on that bulletin board now are from the previous administration (heaven forbid).

Better clean up the bulletin board so that everyone will know exactly who to blame (you) for anything that appears hereafter.

*Be forewarned*, however: As the new president, you're really going to clean house here. Nothing whatsoever will remain of any numbers, programs, or other valuables that may now be in your HP-28. This will be a total reset of the machine. If you have already stored something meaningful in there, you'd better make a note of it now, for you'll need to re-key it in later.

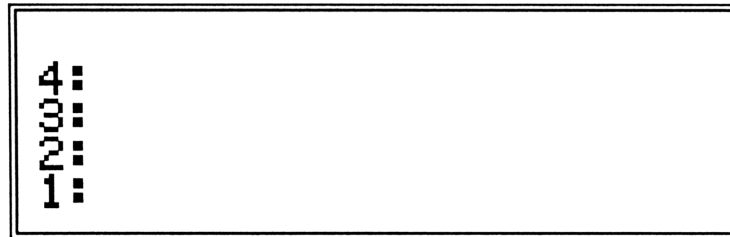
Ready?

---

**Scrub And Dust:** Reset your HP-28 to the status it had when it left the factory.

**Here's How:** Because this is such an all-powerful destructive action to take, the keystrokes are not at all simple; there's very little danger of your doing this by accident. Press and hold down the **ON** key. While holding it down, press and hold down the **INS** key (upper left of the right-hand keyboard). While holding both those keys down, press and release the **▶** key (upper right of the right-hand keyboard). Now release the **INS** key. Now release the **ON** key.

You'll hear a beep and see the **Memory Lost** message in the display. Press the **ON** key once again to clear the message, and you're finished. After doing all this, your display will look like this:



---

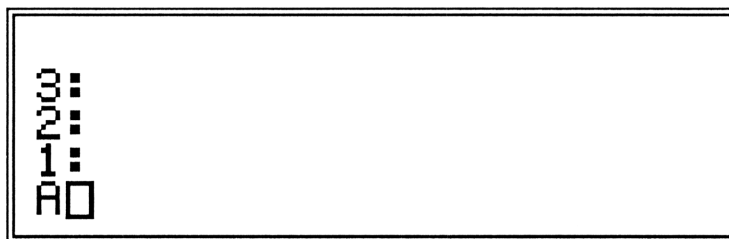
OK, now everyone in your company should be ready to receive instructions from the new chief.

And now you're ready to try your typewriter to see its effects on the bulletin board. Of course, you'll notice that the board (the display) isn't totally bare. Don't be too concerned about what those remaining numbers and colons mean. For now, just watch them move around as you begin to use your typewriter....

(At this point, if you already know how to type and post message and command memos, how to use menus and immediate-execute keys, then you can probably skip ahead now to page 39. Otherwise, stick around.)

---

**Go:** Find the **[A]** key in the upper left-hand corner of the left-hand keyboard. Press it once, and then look at the display. You should see this:



---

"Whoa!" (you undoubtedly say), "that's quite impressive!" Not at all (shucks). Actually, here are the really important things to notice:

Firstly, notice that almost everything that was already in the display was pushed up one line – to make room for the newcomer on the bottom line.

The space opened up at the bottom is lovingly known as the *command line*. In your mental picture, this is where the things you type on your typewriter are first put onto the bulletin board.

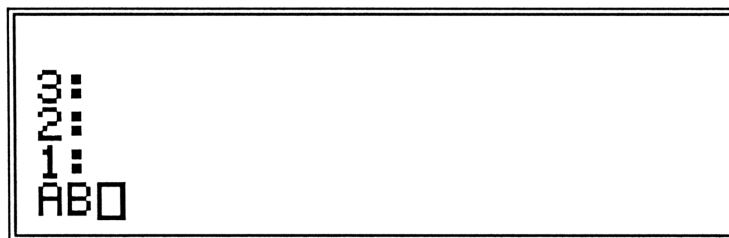
Secondly, there's a flashing, empty box immediately to the right of the **A**.

This box, called the *cursor*, shows where the next character will be placed if you type another one (notice that your typewriter does indeed produce *characters* – not just letters; it can type other things, such as numerals and special symbols. All of these things are collectively called characters).



---

**Next:** To prove to yourself what this cursor (the flashing box) is for, find **B** on the left-hand keyboard and press it.



The **B** is placed where the box was, and the box is moved one space to the right – to where the *next* character will be placed. And so on.

Important point: You will see the cursor *only* when typing in the command line.

---

By the way, if you press the wrong letter key while using the command line, use **←** to correct it. **←** is the same as the backspace key on a typewriter keyboard. That is, by pressing it, you move the cursor one space to the left *and* remove the character that was there.

If you use **←** to remove the last remaining character, the command line goes away. If you keep pressing **←** after that, nothing more will happen.

Play with it, if you wish (then restore your display to the way you see it above).

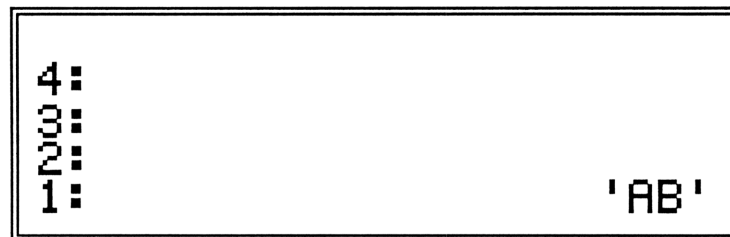
OK, since you're a new chief executive trying to learn the ropes around here, do a trial run: Pretend that what you've typed so far is actually something meaningful that you'd like to post on the bulletin board.

---

**Give It A Whirl:** Seeing that the cursor is still blinking merrily, you press **ENTER**.

The command line goes away and the message is posted.

You see:



---

What can you learn from this?

1. The message was posted at the bottom of the bulletin board. It was put in the first spot, indicated by the **1:**.

That's what those numbers on the left-hand side of the bulletin board are – level markers. They just tell you the age of each message on the board, the youngest (most recent) ones going on at the bottom.

And as in any normal office, those newest postings are always the most interesting. If anything is to be done by your staff, therefore, they will look first at that bottom (the last) memo you've posted.

2. You're already seeing the work of the "memo poster" – that loyal "office boy" on your staff, whose job it is to make sure that your posted memos are given the space and attention they merit.

Of course, this memo poster has worked here longer than you have, so he knows enough to do certain things without being told all the details. After all, weren't you wondering just who was actually cleaning and rearranging the bulletin board to make room for the command line? And who was putting that cursor up there?

And note that the memo poster has put single quotation marks around your message. Why? Because he didn't recognize the memo as anything but a message to be remembered (e.g. "softball practice today at 6:00"). Therefore he didn't do anything special to or with the message; he just posted it.

3. The command line then went away. By posting something with the ENTER key, you've told the memo poster that you don't need the command line anymore, so he clears it away – to leave more space on the bulletin board for messages.
4. The cursor went away, too. As you know, that cursor will appear only when you're typing in the command line – and the command line is gone now.

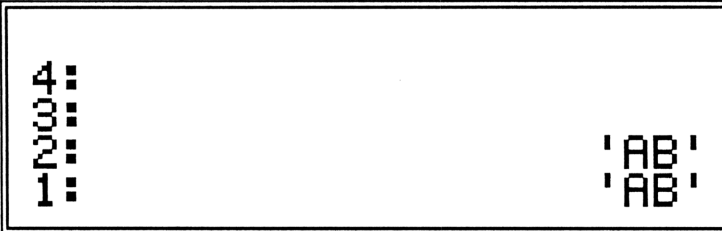
Now for the real test: Post something that really *is* a command – a memo that someone in your staff knows is an explicit request to *do* something.

---

**Try This One:** Type, from the left-hand keyboard, `DUP`.

The display at this point shows nothing that you haven't seen before (just different characters). And everything was pushed up to make room for the command line, and the cursor is sitting there, telling you where you are. No surprises, right?

Now press `ENTER`. Here's what you should see:



```
4:  
3:  
2:  
1:  
  
'AB'  
'AB'
```

---

And here's why you see it:

DUP is indeed a command that someone in your staff understands. In this case, that someone is the memo poster himself; he recognizes it as a command intended for him and, without hesitation, he does what it tells him to do. He doesn't even bother to post it – he just does it.

DUP is shorthand for "DUPLICATE the last message on the bulletin board."

The memo poster reads this and quickly makes a copy of the bottommost memo (i.e., the memo at Level 1). Then he pushes the old memo ( 'AB' ) up the board, and posts the new memo (the copy of 'AB' ) as the last message on the board. No real mystery, right?

And keep this in mind: To you, there's no real difference between posting a command memo and posting any other kind of memo. Either way, you can just type it in and press ENTER.

The difference to your staff is whether or not someone knows what to do with it. In this case, your office clerk – the memo poster – was the person responsible for carrying out the command, and he did so immediately.

- - - - -

Now stop and recap for a minute: What all do you now know about this HP-28 "staff" you have working for you?

You've seen basically how you and your office/business work together – how the common language currency is the memo. You also know how to write and post these memos, *and* you know that there are basically "information" memos and "command" memos.

Are you starting to feel more at home in your new position (a couple of ferns and some pictures of the family ought to just about do it, then, eh)?

Weeell...unfortunately, being somewhat new at the job of president, you don't yet quite know all the commands you might need for working with your staff.

But all is not lost. You do have a command card file.



## The Menu Keys: Your Command Card File

The key to any efficient office is organization. And though you may not realize it yet, your office is organized "to the max."





You have a command card file, a file containing virtually every command that your staff can execute. Not only that, being a card file, it has index tabs (those little category names that stick up out of the card file, effectively dividing the file into sections). It's about time to explore this card file and see how it works, but before you do that, you should first know about this:




### The ("Shift") Key

See that red key on the right-hand keyboard? Now notice that most of the keys on both keyboards have red words or symbols written above them. This is not a coincidence.

Up to now, you've assumed that when you press a key, it will produce the action or character written on the key face (e.g., pressing  causes an  to be placed into the command line).



Well, by pressing the red key and then any key with a red word or symbol over it, you'll produce whatever action or character that's written in red over that key.

For example, press  . What happens? A  is written into the command line. And < just happens to be what's written in red above the  key. The red key is called the *shift* key, because it shifts the operation of the keys to a second set of operations – just like the shift key on a typewriter.

(If you have indeed pressed  , then press  now, before you go on.)

Back to this card file you were going to explore. Notice the top three rows of keys on the left-hand keyboard. Most of these keys have red words above them – as do the keys in the second row on the right-hand keyboard. These keys are the index tabs for your command card file. As in a real card file, if you select one of these index tabs, you should find a logically-related group of "things" under it.


---

**Try One:** Press  (which is really ). You should see:



---

As you look at this, you should realize:

1. The words in the black boxes at the bottom of the display are all commands. What's more, they're all related – they're all *array* commands. **ARRAY** is the red word over the  key, which you just pressed. In other words, you selected the **ARRAY** index tab, so you're given this set of array commands.
2. A set of commands such as this is called a *menu*, because it's a list of items from which one chooses, just as in a restaurant.
3. A menu's appearance in the display moves everything else on the bulletin board up one line. Notice that this doesn't make the memos any older; it only moves everything up out of the way – just as the command line does.

Now then: You've seen how the command line will take the bottom line of the display. But what happens if there's a menu already there when you activate the command line?

---

**One Way To Find Out:** Type `STD`.

You'll see:



---



As you can see, the menu stays on the bottom line, and the command line takes the next line, pushing everything else up one line farther than usual.

Why does the menu remain? Because you might actually want to use one of its commands in the command line.

Now press `ENTER` to execute what you've just keyed in (apparently it was a command that was meaningful to someone on your calculator staff. You can see this by the fact that it wasn't simply put up on the bulletin board as a message. Instead, someone recognized it and did it – immediately).



---

**Try Another Menu:** Pick another index tab from the card file, say,  **REAL** ( **F**).

You should see:



---

This is the REAL number menu. Because it's a menu, you should be able to pick and use an item from it.

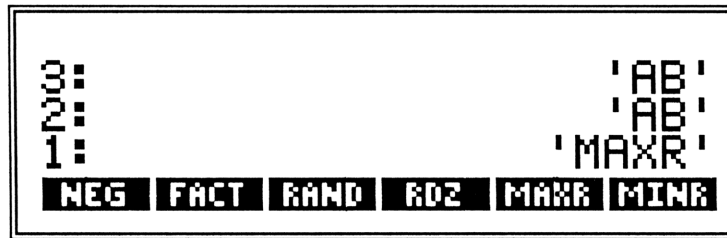
To do so, first notice that the black boxes around the items are lined up over the top row of keys on the right-hand keyboard – and those keys are blank.

Another non-coincidence.

Whenever a menu is shown in the display, the keys in that top row take on the meanings of the names in the menu. By pressing the key under an item, you will be choosing that item from the menu.

---

**Order From This Menu:** For example, press the key under **MAXR** and see:



---

MAXR, in this case, is another command to the message poster. It just says "Post this name (MAXR) as the message." As you'll see later, posting such names can be very useful in certain situations.

But here's an important point: Menus are a convenience feature, *not* a vital necessity. You could have *typed* in the name, MAXR, to get the same result. In fact,

---

**Try It:** Type **M****A****X****R** **ENTER**.



You accomplished the same thing on your typewriter as you did with your card file! You can therefore think of your card file as your stock of ready-typed command memos.

---

In case you were wondering, it's true that many menus have more than 6 items. To see the other items, you simply need to flip to the next "page" of the menu by using the **NEXT** key. And to flip pages in the other direction, use **PREV**. Practice now with these two keys by looking through the entire REAL number menu.

Once again, tick off the things you now know:

You know how your memo poster obeys your keyboard by posting or acting upon memos. And you know that you can change the meanings of keys with the ■ ("shift") key.

You also know how to pull out various collections of related commands from your command card file. Each such collection is called a menu, and when you want to, you can select from it by using the blank keys on the top of the right-hand keyboard.

But those menus just give you easy access to the names of the commands. What if you forget the particular rules for using them?

Ask your office boy. If you press ■ CATALOG, you can get him to show you the details for each command – rules and limitations that might appear on the bottom of each card in a real, paper card file. You can check the spelling, fetch, or refresh your memory on the use of these commands, either in straight alphabetical order or beginning with whatever letter you specify.

Play around with this special CATALOG menu. The commands on *this* menu are fairly self-explanatory, so go ahead – try'em out!

## Immediate Execution ("Do-It-Now") Keys

Now that you know where to find commands and menus, the next thing to notice is that menu-related keys work a bit differently than the typewriter keys.

When you pressed the typewriter keys, the command line came on and the characters that you typed were placed there – but the message you were typing wasn't considered by the memo poster as being ready for posting until you pressed **ENTER**

By contrast, when you press a menu key (either an index tab or a command from a menu), the effect is to "do-it-now." Such keys *don't* wait for you to press **ENTER** before they present themselves to the memo poster; in essence, they "press the **ENTER** on themselves," thus saving you a keystroke.

And some of these immediate-execution keys are so frightfully useful that they've been awarded keys of their own. Of course, **ENTER** itself is one such vital "do-it-now" key. But now it's time to introduce **ENTER**'s counterpart, which is also a "do-it-now" key:

**DROP**

As you know, **ENTER** tells the memo poster to put memos on the bulletin board. But what if you want to take memos *off* of that board? After all, what if you simply make a mistake and don't notice it until after that erroneous memo is already posted? How do you discard and replace it?

You press **DROP**. The **DROP** command tells the memo poster to rip down and trash the last memo and move the rest of the memos on the board down a level.

So if you were to press **DROP** right now, what would you expect to see?

---

**Try It:** Use **DROP** (it's on the right-hand keyboard, just above **9**). Press it once and voila:



Memo 1 is trashed and everything else is moved down one level – just as you would have expected, knowing the rules for **DROP**.

---

Notice, by the way, that **'AB'** sitting up there at Level 3. Where did it come from?

It used to be up at Level 4.

Nothing had really "happened" to it; you just couldn't see it while it was on Level 4. The bulletin board is, for all practical purposes, "infinitely tall." But the display isn't (an infinitely tall display wouldn't fit very well in the calculator).

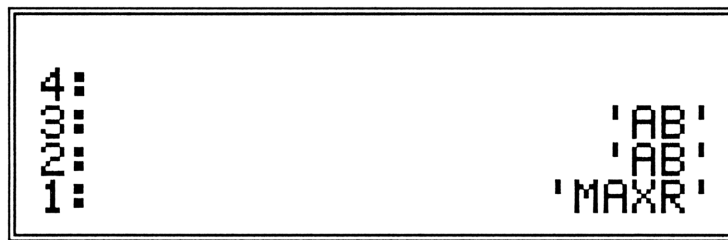
So for practical reasons, the HP-28 shows you, at most, the bottom four levels of the bulletin board. *But*, any items you have posted which have been bumped up above the fourth display line are still on the bulletin board, safe and sound.

All right, so you've seen some immediate-execution ("do-it-now") keys, a couple of which you'll be using quite a bit: **ENTER** and **DROP**.

But there are plenty of other such keys, too. For example, notice that weird-looking one next to the shift key: **↔**.

---

**What Does It Do?** Press it once and see the following:



What happened? The menu went away.

---

**↔** will turn the menu display either on or off, whichever makes sense at the moment (press it again and the menu comes back; once more and the menu goes away again, etc.).

The big advantage of this sleight-of-hand is that when you don't need the menu, you don't have to keep it around cluttering up the bulletin board. That **↔** key is your quick, convenient way to tell your memo poster to set that current menu aside until you ask for it again.

## Messages From the System – Memos From Your Staff

At this point, you've explored some of the ways that you can use to communicate with your staff, but you really haven't seen much about how your staff responds to your commands and messages.

Now everybody knows that one of an employee's most important jobs is to tell the boss when he's messed up, and it's time to see how your staff does this for you.

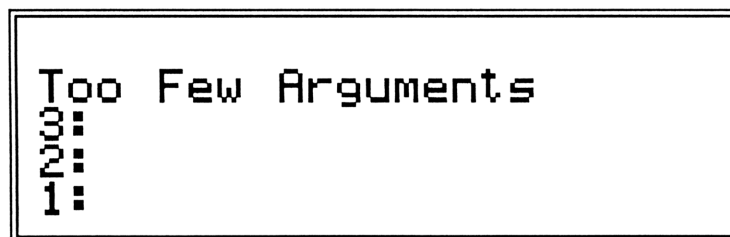
First, of course, you have to make a mistake (this may come as a shock to you personally, not having done such a thing in so long; but of course, if you make a mistake on purpose, then it's not really a mistake, is it?). All right then,

---

**Mess Up:** Press **DROP** three times. All the memos are gone. You've dropped them all.

But now, what happens if you tell the memo poster to drop a memo when there's not one there to drop? ("Let's find out ... 1 ... 2 ... 3 ...") Press **DROP** once more.

You'll hear a beep (to get your attention), and you'll see:



```
Too Few Arguments
3:
2:
1:
```

Is your staff bored because there hasn't been enough bickering in the office lately?

---

Not really. Actually, this mathematical staff of yours is just guilty of using big words. When they say argument, they mean "something to work on."

So your memo poster is simply telling you that when you told him to drop something off the bottom of the bulletin board, he didn't have anything to drop. A reasonable objection, don't you think?

But forgetting for a moment about what this particular message says, you should examine in general what your staff does whenever they notice a mistake of yours.

1. They yell at you (remember the beep? Yep – that was them yelling at you).
2. They post a memo for the whole office to read. The memo says, in effect,

"You Blew It and This Is Why"

3. This message is posted at the top of the bulletin board – as are all error messages. But these messages don't bump others off or push them up the board; they just temporarily cover up what's there.

The next time you do anything to the bulletin-board, the memo poster will remove the error message (and if you just want to be rid of the error message without otherwise changing the board, you can do so by pressing **[ATTN]**, which is the **[ON]** key).

So these error messages are simply one way your employees talk to you.



## Status Messages: The Annunciator Area




But there's another way your staff can tell you things.

There is actually *another* line visible on the display/bulletin board – above the fourth "active memo" line. Up to now, this area has been largely irrelevant as you've been learning your way "around the office." But now take a look at it.

That line is the *annunciator* area, a place where little "wait-a-minute-I'm-busy" and "remember-your-lunch-money" messages are posted by your staff, for your benefit.

For example, you may have noticed – though it wasn't pointed out – that many times during the process of posting a memo (especially after pressing **ENTER**) the symbol ((•)) will appear briefly on the top line.

Simply put, your staff is telling you that they're busy at the moment. As you may have observed, in most cases, they're so fast that this "busy signal" only flashes (but later on you'll know how to issue commands that will keep them occupied for quite some time).

Another symbol you may have noticed is the one that comes on when you press the shift key. The symbol is . It's there to remind you that the next key you press will perform its shifted function (written in red above the key). You can turn the  off by pressing  a second time, thus shifting all keys back to their main functions.

There are some other annunciators that can appear on this top line, but you'll encounter them as you go along; no sense crossing those bridges now.

## A Tricorder Reading

As usual, before going on, it's a good idea to get your bearings in this mental "world" of your HP-28.

This first Monday at the office was all about learning to communicate with your staff through memos and messages on a bulletin board.

You saw how the keyboards are connected to this bulletin board (the display), and how the keys produce either immediate actions (the "do-it-now" keys) or characters for building memos.

You specifically know about 3 immediate-execution keys: **ENTER**, **DROP**, and **↔**.



You know that as you type in characters, your "office boy" will show you your memos-in-progress on the command line. And if these typed-in memos are commands recognized by anyone on your staff, they'll be carried out promptly after you officially give your OK to post them (by pressing **ENTER**). If nobody recognizes them, they'll stack up on the bulletin board, with the oldest memos on top.

You know how the **■** key changes the meaning of keys on both keyboards and how a lot of these red-printed functions bring to the menu keys various sets of related commands for your use. And you know that you can review your entire repertoire of commands by pressing **■CATALOG**.





You know how your calculator staff can give you signals or even temporary error messages when they need to – by using either the annunciator area or the top line of the actual bulletin board.

So here's a set of questions to let you test your understanding before you go on. The answers are on the next page, so check yourself; if you need to go back to review, just look on the pages noted after each answer.

## Quickie Quiz

1. What's the command line for?
2. What's the main purpose of the left-hand keyboard?
3. What's a menu key?
4. What are ((•)) and  and how do you get rid of them?
5. What's a character?
6. When would you expect to see this:  ?
7. How many days hath September?

## Quickie Answers

1. The command line is for typing and editing memos for posting (page 20).
2. The left-hand keyboard is mostly used for typing, especially the alphabetic characters A through Z (page 17).
3. A menu key is one of the six blank keys at the top of the right-hand keyboard which take on the functions of the displayed menu (page 29).
4. ((•)) and  are both annunciators, appearing on the very top of the display. ((•)) is the busy annunciator, which you would get rid of simply by waiting for the machine to finish what it's doing. The  is the shift annunciator, and you would press  to turn it off (page 37).
5. A character is any alphabetic letter, numeral, or special symbol that the HP-28 can generate (page 20).
6. You see  (the cursor) when the command line is active (page 21).
7. September hath thirty days.



**MAKING YOUR HP-28 WORK FOR YOU:**  
**The Command Line**

The command line is where you'll be spending much of your time and energy as you communicate with your HP-28. So now that you've seen most of the various communication channels you have with your office staff, it's time to concentrate on this particular one. This chapter is all about the editing and presentation options you have in the command line.\*

## Typing Characters Into the Command Line

As you know, the command line is where you type in numbers and words – as series of characters – preparing them for posting or for issuing as commands. Indeed, you've seen how directly it can be compared to the output portion of a typewriter. It is, in effect, a very simple text editor.

But have you noticed that there's no command that says "Start the command line"? Rather, certain keys that you often use in spelling out commands and memos *automatically* tell that memo poster" to start the command line.


The most commonly used of these keys are the alphabetic and numeric keys, [A] through [Z] and [0] through [9]. Invariably, if you press one of these keys when you're not yet typing in the command line, the memo poster will start a command line for you and put the character you typed into it as its first character.

And of course, once you've keyed in all the characters you want on the command line, you press [ENTER] to post it.


So if all the command line allowed you to do were to type out commands and other memo postings, life would certainly be fruitful – but it wouldn't be very easy. That is, being not quite perfect, you'll sometimes simply need to correct your typing errors – and mercifully, the command line allows you to do this.


\*Of course, if you're already feeling quite comfortable with all that, then you may skip ahead to page 65.


## Changing a Character in the Command Line




You already know about the most commonly used correction key, backspace () . In the command line, it removes the character immediately to the left of the cursor. In this way, it's quite convenient, especially if you notice your error before you've typed too many more characters.

But if you type something like **CK**ARACTERISTIC, then backspacing over all but the first character is a waste, especially since all but one character are correct. Somehow, you need to be able to move the cursor to the second character and replace the **K** with an **H** – without erasing everything else along the way.

Fortunately, you can: Remember the  key? You've seen how it turns on and off the menu area of the display – but that's not its most important talent. The arrows on its face are the tell-tale signs:

The  key *enables* and *disables* the cursor-movement keys.


Those cursor-movement keys are, non-coincidentally, the same as the menu keys. This is because the  key – much like the shift key – *shifts* the function of the blank menu/cursor keys between the current menu's functions and those printed in white above the menu keys. *The cursor-movement functions of these keys are available only when there is no menu in the display.*

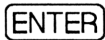
And notice that, *unlike* the shift key, the  key changes the functions of those keys *until the next time the  key is pressed*. In other words, you don't need to repeatedly press  to maintain the menu selection keys' current functions.

So look now at those cursor-movement keys (called cursor keys, for short).


As you might expect, since they affect the cursor (which exists only when the command line is active), these keys work only with an active command line.

---

**Time For Some Practice:** (If at first you see a menu in the display, just press  to get rid of it for now.)


Type in **CKARACTERISTIC**, mistake and all (but don't press  afterwards; you're going to "catch" this "mistake" before actually posting it onto the bulletin board).

You should see:



```
3:
2:
1:
CKARACTERISTIC
```

As you know, characters can be added to the command line only at the current location of the cursor. Thus, typing a character key now would add the character to the end of the word and move the cursor one character to the right.

As you also know, you *could* use  repeatedly to delete all of the characters between the cursor and the first **C**, thus deleting the **K** in the process. But all you really want to do is to move the cursor on top of the **K** and overwrite it with an **H**.

How can you do this?



The key with the white ◀ over it will allow you to do this (but remember: the white cursor symbols are only active when there's no menu in the display. If when you press a menu key there's any menu visible, the function of that menu key – not the cursor control function – will be performed).

So press the ◀ key.

The cursor moves to the left by one character, but it doesn't delete that character. Press it again, and it moves one more character to the left. Press it and hold it down, and the cursor will continue to move to the left until you let up on the key. When the cursor has moved all the way to the left – over the top of the first character – pressing ◀ will no longer move it at all.

Now press ▶. What happens?

No real surprises here, right? ▶ moves the cursor to the right, but notice that if you keep pressing the ▶ key until you reach the last character of the word, the cursor doesn't stop there; it goes one space farther, to exactly where it was when you stopped typing the word in the first place – and for the same reason – so that you can add more characters to the end of the word.



---




Now, go fix that typo.

---


**Playing Editor:** Press  . What happened?

The cursor moved all the way to the left – to the first character.

Press  . What happened? The cursor moved all the way to the right.

These are shortcuts. You could have accomplished the same thing simply by pressing and holding down either the  or  keys, respectively; but pressing  saves you some time.

So press   and then .

The cursor will now be over the **K**. Since characters are added to the command line at the position of the cursor, pressing  now will put an **H** in the command line – right where the **K** used to be. Do it.

As you've come to expect, the cursor then moves one space to the right.

---

## Adding and Removing Characters

Now, what if you had simply omitted a character, rather than accidentally typed the wrong one?

To see how you would deal with this, use ◀ and ▶ to move the cursor so that it's positioned over the **E** in **CHARACTERISTIC**.

Now press ◀. What happened?

The backspace key did what it always does. It deleted the character immediately to its left. In this case, since there were characters to the right of the deleted character, they were all moved one space to the left, to fill up the hole.

Next, press ■▶.

This is what the command line would look like if you had originally forgotten to type the first **T**.



Notice that you have just learned the way to remove a character or characters if you've typed too many:

You use ◀ and ▶ to move to the space immediately *to the right* of the offending character, then press ◀ to delete it.

Now the command line is all set up to look just as it would if you had just keyed in **CHARACTERISTIC**. You want to correct the omission.

Use **◀** to move to the **E**, which is the character that your missing **T** will *precede*.

Now, can you simply type in a **T** to fix things? Nope. Remember that if you type a character now, it will *replace* the **E**. What can you do?

Press **INS**.

What happens? Look closely and you'll see that the cursor – which was a flashing box (**□**) – is now a flashing arrow(**◄**).



**INS** is the **INSert** key. It tells the command line that you want to insert one or more characters before (to the left of) the character that *was* sitting under the flashing box cursor (**□**).

So the arrow is now pointing to the place (between the **C** and **E**) where a character would be added. Make sense? OK, do it: Press **T**. Now what happens?




```
3:
2:
1:
CHARACTER◄ISTIC
```




First, the **T** was added to the command line at the place where the cursor was. Then the cursor moved one space to the right. What's different is that everything to the right of the arrow's point moved with the arrow. You have now corrected the omission!




Press  and  a couple of times. Notice that they work the same way with this arrow cursor as with the box cursor. The only difference is when you press a character key.


When the cursor is a box, the new character will replace the one on which the cursor is sitting.



When the cursor is an arrow, the new character will be inserted *before* the character on which the cursor is sitting.

Finally, press . What happens?

The cursor changes from an arrow back into a box. Repeatedly pressing  will change the cursor back and forth between a box and an arrow. In this way, it's a "toggle key" – like  and  – shifting alternatively between two modes.

Another key that you should find useful when editing the command line is the  (delete) key.  works just like , except that instead of removing the character to the *left* of the cursor, it removes the character *under* the cursor.

And just like , all the characters to the right of the deleted character are moved to the left one space to fill up the hole.

Also, both  and  will repeat their functions if you hold their keys down.

You can see now that you have a number of different ways to correct minor errors you may make while keying in a memo on the command line!

■ **INS**, ■ **DEL**, and **ATTN**

Suppose that your error isn't so minor this time: you need to delete more than one character.

Of course, you could always fix things by moving the cursor with ◀ and ▶ and then using either **DEL** or ◀ – as you just saw.

But what if it's a whole string of characters that you need to remove?

In that case – whenever you need to delete *all* characters to the right or to the left of the cursor – you have yet another option....

Using the word **CHARACTERISTIC** from the previous examples, assume that what you really wanted was the word **CHARACTER**.

Assume also that the cursor is now sitting over the **E** – because you just inserted the **T** (so if you've pressed **INS** to get the ■ cursor, then for the purposes of following along here, press **INS** again. Just bear in mind that the example will work no matter which kind of cursor you use).

So you should see the following:

```
3:
2:
1:
CHARACTORISTIC
```

Obviously, you want to delete **ISTIC**. You could move the cursor to the first **I** and use **[DEL]**. Or you could move the cursor to the right end (with **[▶]**) and use **[◀]**. Or, you *could* move the cursor over the first **I** and press **[DEL]**.

---

**Try It (You'll Like It):** Move the cursor over the **I** and press **[DEL]**. What happens? Everything to the right of – and under – the cursor is deleted, right? *It's exactly as if you had pressed and held down the **[DEL]** key.*

So you're left with **CHARACTER**, and the cursor has been left at the end of the new word, so that you can add more to it if you like.

---

**And Now This:** Type in **MITE** and move the cursor so that it's sitting over the first (left-most) **T**. Press **[INS]**.

See? Everything to the *left* of the cursor is deleted, and the remaining characters are shifted to the left. *It's exactly as if you had pressed and held down the **[◀]** key.*



---



**Last Resort:** If all else fails, you can always press **[ATTN]** (**[ON]**) to clear the whole command line and start with a clean slate.




Try it now. Notice that the **[ON]** key serves two functions: When the HP-28 is off, this turns it on; when it's already on, **[ON]** functions as **ATTN** (attention), interrupting and effectively shutting off the command line, discarding everything that was in it.



---









While you're paused here with such a clean slate, take a minute to review all these options for correcting errors on the command line – just to be sure you have them all straight in your mind.

The  and  keys move your cursor to the left and right, respectively. You can't go any farther left than the first character on the line; you can go exactly one place farther than the last character – to be ready to type another, of course.

Pressing  and  are shortcuts for moving to the very ends of the command line.

The  cursor lets you type a new character right over an existing one (thus replacing it). The  cursor lets you insert a new character between existing ones. You alternate back and forth between these two cursors by pressing the  key.

To delete an unwanted character, you could press , which would delete the character to the *left* of the cursor. Or, you could use the  key, which would delete the character *under* the cursor.

Pressing   and   are shortcuts for deleting all characters from the cursor to the left and right ends of the command line, respectively. The one difference is that   also deletes the character under the cursor, while   doesn't.

OK so far?



■ NEWLINE, ▲ & ▼

If you're at all verbose with your commands, you can certainly overrun the visible 23 characters of the command line. But this is no problem, really, because the command line is effectively infinite; you can type as much as you want.

---

**Test This:** Type in the 26 letters of the alphabet onto a fresh, clean command line (i.e., first press **ATTN** if there's anything on the command line):

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

You'll see:




---

Notice that when you exceeded 23 characters, the command line scrolled to the left and showed an ellipsis ( ...) as the first character to tell you that the command line does indeed continue to the left, but that this beginning part isn't currently visible.


Now press ■◀ to get to that far left end.

The command line will scroll to the right and place the ellipsis at the right end of the display. Makes sense, right?

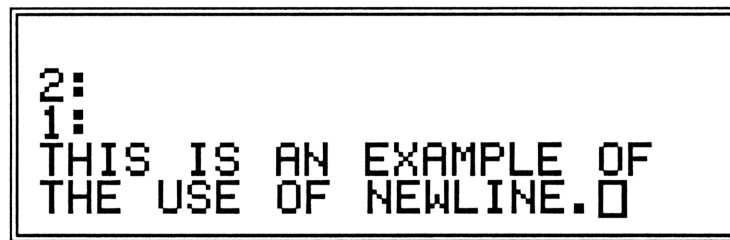
Well, that's all good and fine, but it doesn't take advantage of the other 3 lines in the display that are available to you.

Happily, if you want to see more of the command line, you do have the option of using  **NEWLINE** to separate words....


---

**Try This:** Press **ATTN** and type **T H I S SPACE I S SPACE A N SPACE  
E X A M P L E SPACE O F  N E W L I N E T H E SPACE U S E  
SPACE O F SPACE N E W L I N E .**

This is what you should see:



```
2:
1:
THIS IS AN EXAMPLE OF
THE USE OF NEWLINE.
```

You can use as many  **NEWLINE**'s as you want in order to make things more readable, and you don't need to fill up each line before going on to the next line.

---

Do you see what this implies? Your command line is essentially unbounded, since you can add lines – separated by **■**`NEWLINE`'s – to the point where the text scrolls off the top of the display. And because the bulletin board is unbounded, when these lines do scroll up out of sight, they're faithfully preserved and usable!

OK, but how can you see or edit these lines that disappear off the top?

Simple: You move from line to line and scroll lines back into the display with **▲** and **▼**.

These two vertical cursor-movement keys work in the same way that **◀** and **▶** do – except that they move the cursor up and down rather than from side to side. And, as you might expect, **■▲** and **■▼** also function similarly, sending the cursor to the very top or very bottom line, respectively.

Next question: How can you get rid of these `NEWLINE`'s that you've embedded in your command line?

Next Answer: Use `DEL` to "undo" a **■**`NEWLINE`.

---

**Try It:** Press **■▲** **■▶** to move to the end of the first line, where you pressed **■**`NEWLINE`, and press `DEL`. The two lines are joined into one, with the one that was on the bottom extending off to the right.

---

Note: **■**`DEL` and **■**`INS` affect only the line that the cursor is on. And you can't use **◀** or **▶** to move from line to line; you must use **▲** and **▼**.

## The **[LC]** Key

Up to now, when you've typed something into your HP-28, it has come out in upper case. But that's not the only way to do things. If you need to use Lower Case letters, just press **[LC]** (it's down there on the bottom line of keys on the left-hand keyboard).

Go ahead and do that now.

Nothing obvious happens, but if you now use any of the alphabetic keys you'll find that they all put lower-case letters into the command line.

Notice that **[LC]** is like **[INS]** in one respect. Once you press it, it stays in effect until you press it again (or press **[ENTER]**), much like the upper case ("Caps") lock key of a standard typewriter.

This may not seem like a very important feature, but you must realize that the case of a character in any command is taken quite literally by the HP-28. If you accidentally capitalize some character in a command that's not supposed to be capitalized, the machine won't recognize it.

In order to have your commands recognized, you must spell them exactly the way they appear in the command CATALOG, including all upper and lower-case characters.

**[α] & [CLEAR] [α] LOCK**

So far, you've been concentrating on the keys that function only to put their symbols into the command line. You've ignored most of the immediate-execution ("do-it-now") keys, such as **[+]** or **[-]**. Recall that pressing one of those keys all by itself normally causes the calculator to perform that function.

---

**Watch:** Press **[ATTN]** **[CLEAR]** **[A]** **[B]** **[+]**.

What happened? **'AB'** was posted just as if **[ENTER]** had been pressed, and an error message was displayed. Don't worry right now about why this error occurred. Just realize that immediate-execution keys will normally try to "do their things" *even when the command line is active*.

Sometimes this is convenient; sometimes it's not. After all, what if for some reason you wanted a *symbol* such as **+** or **-** to appear in the command line?

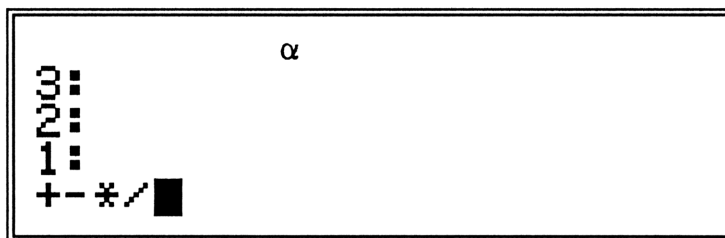
You would press **[α]** first. The **[α]** ("alpha") key tells the system (your memo-obeying staff) to treat the keys pressed as character keys rather than as "do-it-now" keys.

---


As you might suspect, certain vital immediate-execution keys, such as **[ENTER]** and **[ATTN]**, are exempt from the **[α]** key's disabling influence.



---






**So Try It:** Press      .



---

See how convenient  can be when you want to type merely symbols rather than the commands usually associated with those symbols? All of those symbols have been entered as plain old, garden-variety characters into the command line.

And notice that when you pressed , the  $\alpha$  annunciator appeared in the top line of the display and the cursor changed to a solid block () to remind you that you are in this mode where most of the immediate-execution keys are "blocked" from executing immediately.

Alpha mode will stay on until you press  or , but if you don't want it turned off for you like that, you can press  , in which case it will stay on until *you* turn it off again by pressing .

You're adding rapidly to your bag of tricks for controlling the command line. First, you learned how to correct errors. Now you've seen some ways to key in lower-case letters, long strings of characters, or strings involving symbols normally reserved for immediate execution. Review:

The **NEWLINE** key produces an invisible character that lets you break a long command strings into manageable segments, so that you will see these segments in your display (your bulletin board) on adjacent lines. When you have such a multi-line command line, you can move around between lines with the help of **▲** and **▼** and their shortcutting versions, **■▲** and **■▼**.

The **LC** key lets you type in lower-case letters – until you press it again or post the memo.

The **α** key disables most of the immediate-execute keys so that you can use those keys' symbols as characters in your command line. You can cancel this disabling by pressing **α** again or by pressing **ENTER**.

Now go on and look at some convenient variations of skills you already have....

## Item Delimiters and **ENTER**

In past examples you've almost never completed the memo on your command line to the point of pressing **ENTER**.

That is, you seldom actually gave the go-ahead to your faithful office boy, the memo-poster, to officially post a message or otherwise try to make sense of ("evaluate") the command line.

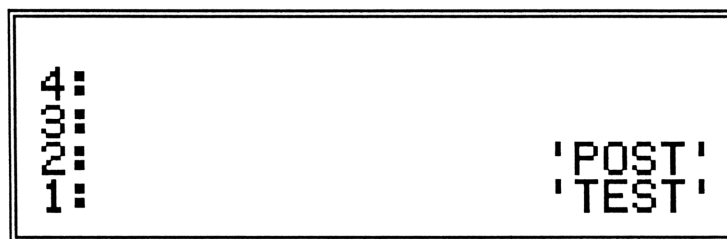
This is because most of what you've keyed in so far just wouldn't make much sense – either to you or to your office staff (the calculator system) when it was evaluated.

And on those occasions when you have pressed **ENTER**, you may have noticed that the command line may not have been posted as a single memo.

---

**For Example:** Type in **ATTN** **P****O****S****T** **SPACE** **T****E****S****T** **ENTER**.

What you'll get is this:



```
4:      'POST'  
3:      'TEST'  
2:  
1:
```

---

What's going on here, anyway?



Three things of interest:

1. The space between the two words in the command line effectively separates them into two postings when **ENTER** is pressed. In this case, then, the space is called a delimiter, because it acts as a marker, denoting the end of one memo and the beginning of another.
2. The memos are posted from left to right; the word on the left was posted before the word on the right (and as a consequence, **POST** now appears farther up on your positionally- "dated" bulletin board).
3. Neither of these words was recognized by your calculator's system, so they were posted as is – with single quotation marks to let you know this.

**Conclusion:** *You can use the command line for posting more than one memo at a time by marking each successive item with a delimiter character!*

So, besides **SPACE**, what other characters will play this role of delimiter?

**NEWLINE** will.

So will the comma: **,** or period: **.** – whichever the HP-28 is *not* currently using as the radix mark (decimal point). In other words, if the current radix mark is the period (i.e. if **1.5** is interpreted to mean one-and-a-half), the comma is a delimiter; but if the current radix mark is the comma (**1,5** = one-and- a-half), then the period is free to be used as a delimiter.

There are many other delimiters too: **<**, **>**, **[**, **]**, **#**, **"**, **'**, **⌘**, and **⌘**. But these all have special meanings to the calculator – meanings you'll see later.

## EDIT and COMMAND

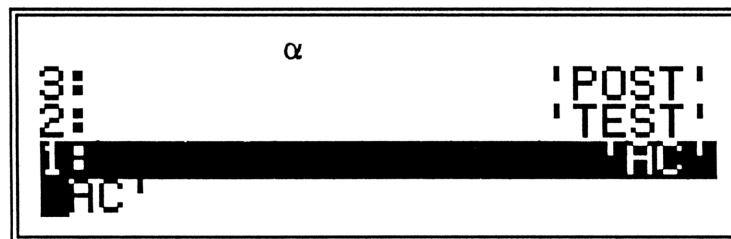
Another variation on something you've already seen: You've seen how to correct errors in the command line – as long as you catch them before you press **ENTER**. But what if you don't catch them that soon? How do you "undo" an error that has gone so far as to be officially posted on your bulletin board?

Of course, you could just unpost the memo (using **DROP**) and totally retype it. But this seems like a colossal waste of time if the error is minor and the memo is major. Wouldn't it be nice if you could just *edit* the posted memo?

---

**Good News:** Press **ACENTER**; but now decide that you really wanted 'ABC'.

So press **EDIT**. You should see this:



---

Notice what has happened: The command line is activated *containing the contents of Level 1*, which is highlit to show that it's being edited. And alpha mode is activated for your convenience, as indicated by the  $\alpha$  annunciator and the solid cursor.

You may now edit the memo in the same way that you would edit anything in the command line. And when you're finished, pressing **ENTER** *replaces* the highlit line with what's in the command line. Or, if you change your mind midway, pressing **ATTN** aborts the edit and does *not* change the highlit line.

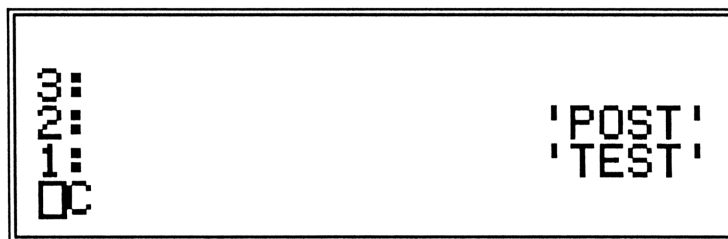
Even if you do DROP an erroneous memo, there is a time-saving shortcut to re-post it:

DROP the bad memo and then press **COMMAND**.

---

**Try It:** Press **ATTN** (to discard whatever command line you may now have in progress).

Then: **DROP** **COMMAND**. You should see the following:\*





```
3:  
2:  
1:  
□□ 'POST'  
      'TEST'
```


---


The last thing you typed and followed with **ENTER** is what will be in the command line after you press **COMMAND**. Then you can edit it in the old familiar way (cursor keys and all that) and re-post it with **ENTER**.


\* This feature is actually an option – one that you can disable ("turn off") if you wish. Only those commands entered while the command memory is enabled will be remembered. It's your choice – and this and other such preference options live in the MODE menu. Thus, you would press **MODE****NEXT** **+CMO** to enable this command memory (it's a command "stack," actually).

By the way,  **COMMAND** has a better memory than you might have first supposed. Not only does it remember the last memo you posted, it remembers the three before that, too!

If you press  **COMMAND** a second, third and fourth time, you'll see the second-to-last, third-to-last, and fourth-to-last commands ("memos") you **ENTER**'ed, respectively. As each one of these comes to the command line, you may edit it or repost it with **ENTER**.

If you press  **COMMAND** a fifth time, it will cycle around and show you the most recently posted command again. And as always, you can get rid of the command line altogether by pressing **ATTN**.



This extra-good, short-term memory may hint to you of another advantage to using  **COMMAND**.

If you're doing a lot of posting and many of the memos are the same, you don't need to retype any that were already posted within the last four postings. You can just use  **COMMAND** to call them up again.















































As you might imagine, this is especially useful if the postings were long or tricky!

Now it's time to put it all together and see how well you know your way around the command line....

## Command Line Quiz

1. How do you turn on the command line?
2. What's the difference between the functions of  and  ?
3. What's a delimiter? Name two.
4. Change 'CONFRONTABLE' to 'COMFORTABLE' (assume that the memo 'CONFRONTABLE' is now sitting at Level 1 of your bulletin board).
5. Change 'centimeter' to 'centipede' (again, assume that what you start with is sitting at Level 1).
6. Change 'Apples' to 'Oranges' (again, at Level 1).
7. What's the capital of Montana?

## Command Line Answers

1. Press any character key (see page 42 to review this).
2.  deletes the character *to the left of* the cursor;  deletes the character *under* the cursor (pages 47-49).
3. A delimiter is a separator. In the case of the HP-28, it is a character separating two memos in the command line.  and  are two examples (page 61).
4.     M    R  or     M F O R  , for example; of course, there are many different ways to accomplish such an editing job (page 52).
5.       LC P E D E   or         LC P E D E  or        LC P E D E , for example (pages 52 and 56).
6.   LC R A N G E S  (page 56).
7. Helena



**REAL NUMBERS, THE STACK,  
AND POSTFIX NOTATION**

How does this imaginary HP-28 world seem to you now? On your first Monday at your new job as president, you familiarized yourself with your office, the bulletin board, your typewriter, your command file, and your faithful office boy, the memo poster. On Tuesday, you spent all day learning all those skills for efficiently posting and editing memos.

It's now Wednesday morning (time flies when you're having fun).

## **Real Numbers – and the Real World**

There are a lot more high-powered brains working back there in the offices beyond the bulletin board. It's time you were introduced.

As you know, many of your HP-28 "staff" system's talents lie in the area of numeric problem-solving. So you can't really relate to them unless you're brushed up on their language: numbers.

It's probably best to begin by using real numbers, since they're probably familiar to you already.\*

Numbers can be broken up into different classes which are useful in different circumstances. *Real numbers* form a collection of most of these classes into the one big group that you normally think of as being numbers: the positive and negative integers (1, 2, -3, -5, etc.), the positive and negative rational numbers (-0.23, 4.56, etc.), the positive and negative irrational numbers ( $-\pi$ ,  $e$ ) and zero (0).

\*If you already know how to key in, format, and otherwise represent real numbers on the HP-28, then now's a good time to skip over to page 81.



Right, then: Your first consideration for dealing effectively with your calculator's math brains is in communicating with it. That is, when it shows you a number, you'd better be able to recognize it.

Usually this isn't too much of a problem, except for extremely large and extremely small real numbers. These are always a bit awkward to deal with (in any tool – from paper and pencil to a high-speed computer), because their representations use a lot of digits.

For example, the number one-half is relatively easy to write. Its representation is 0.5, ("zero-point-five" or "five tenths"). But smaller numbers like one thousandth (0.0001) are more cumbersome and less easy to read.

And *really* small numbers – like one hundred-millionth (0.00000001) – or *really* big numbers – like one billion (1,000,000,000) – are actually unpleasant to deal with, precisely because of all of these zeros you need to carry around.

For this reason, an alternate representation has been developed, called *scientific notation*.<sup>\*</sup> In this notation, you take a number and split it into two parts. The first part consists of all the digits except leading or trailing zeroes. The second part tells you how many of these leading or trailing zeros you also need and whether they're leading or trailing.

Thus, 5,280 is  $5.28 \times 10^3$ , 0.00023 is  $2.3 \times 10^{-4}$ , and 1 is  $1 \times 10^0$ .

Notice the convention here. The first part of the number (called the "mantissa") shows its *precision* and is written with its first digit just to the left of the decimal point, with the rest of the digits, if any, to the right. The mantissa is then multiplied by a power of 10 (called the "exponent") to show the number's *magnitude*.

<sup>\*</sup>It's called scientific notation not because it's in any way more "scientific" than other notations, but because in science one commonly deals with very large or very small numbers. It could as easily have been called "national debt notation," for example.

## Representing Real Numbers On the HP-28

Scientific notation is especially useful for representing numbers on machines. As you would expect, the HP-28 can be used to represent and manipulate real numbers of extraordinary magnitudes. But being just a finite machine, it has some limitations, peculiarities, and rules that you need to understand if you're going to communicate with it well. Fortunately, these are few and reasonable.

### Scientific Notation on Your HP-28

First of all, the HP-28 does not use strict scientific notation. It uses a slightly compacted, computerized version of it.

For example,  $2.5 \times 10^4$  is represented as **2.5E4**

And  $3.9 \times 10^{-6}$  appears as **3.9E-6**

As you can see, the **E** means "...times ten to the...". That is, the number following it is the Exponent of 10.

It's just a convenient way to write scientific notation without resorting to superscripts in the display of the calculator.

## 12-Digit Accuracy: Rounding Error

Secondly, keep in mind that some real numbers have representations that are just plain *infinite*. For example, the decimal representation of  $1/3$  is  $0.333\dots$ , where the 3's continue forever.

Of course, it's unreasonable (and fortunately, unnecessary) to try to use all of those 3's during real-number arithmetic. What you do, naturally, is *round* it, shortening it to a value that is both convenient and accurate enough for your purposes. To be sure, the rounded number is *not* the same as the original, but the difference is negligible in practice.

Well, the HP-28 rounds, too. In dealing with infinite or extremely long representations, it rounds the number, remembering 12 digits of the original number.

The inaccuracy that results is called *rounding error*. And as you would suspect, multiplying together two rounded numbers will multiply this error.

So, just how great an error is this?

"Let's find out."

Say that you're the pilot of a plane flying from Los Angeles to New York – a distance of 3,000 miles. Well, it's a lovely day, and once airborne, your navigator lets it slip that he's been using his HP-28 to do fuel calculations.

Not only that, he freely admits that his computations of the number of miles per pound of fuel are only accurate to 0.000000000001 miles (the 12th digit).

Uh-oh. If his calculations are off by that much per mile, how big an error will this make over a *lot* of miles (3,000)...?

Oh, about one two-hundredth of a millimeter.

Not a lot, really.

If you'd flown clear to the moon and back, instead (roughly 500,000 miles), the accumulated error would be an entire eight-tenths of a millimeter.

And in a round trip to the sun (about 186,000,000 miles) you'd gain or lose about a foot (*now* you're talking *gross* error).

As you can see, digital accuracy to 12 decimal places as given to you by the HP-28 is slightly more than barely adequate. So if an answer isn't exactly what you were expecting, it's very, very close.

## Magnitude: How Big (or Small) Can You Get?

A third limitation of the HP-28 is the *magnitude* of a real number (i.e. the numeric value – not the number of digits) it can represent. And again, it's the finite nature of the HP-28 that imposes this limitation; you simply cannot expect it to be able to represent arbitrarily large or small numbers (everyone has his limit; you do and so does your machine).

The largest real-number value representable on the HP-28 (which you can produce with the command MAXR – "MAXimum Real") is

$$9.99999999999 \times 10^{499}$$

And the smallest representable real-number value (which will result when you use the command MINR – "MINimum Real") is

$$1 \times 10^{-499}$$

These numbers are *extremely* large and small, respectively. It's difficult, if not truly impossible, to convey – or even contemplate – the enormity and tininess of these values.

"It's a tough job...but someone's gotta do it..."

To try to get some idea of the size of these numbers, compare them with some of the largest and smallest things in the known world:

The best approximation for the effective radius of an electron is about  $2.817938 \times 10^{-12}$  m(eters).

Putting this into other units, the electron radius is about  $2.97861963628 \times 10^{-31}$  light years (a light year is the distance which light will travel through free space in one year's time, abbreviated ly).

Therefore, the *volume* of an electron (assuming that it's a sphere) is about  $9.37309265246 \times 10^{-35} \text{ m}^3$ , or about  $1.10696465437 \times 10^{-82} \text{ lyr}^3$  (can you picture a cubic light year?).

Now, consider that the radius of the sphere of the visible (i.e. the "known") universe is only on the order of  $10^{10}$  lyr. That means that the volume of the known universe is about  $10^{30}$  lyr<sup>3</sup>. So if you packed the known universe absolutely solidly with electrons (no wasted space), you would need about  $10^{112}$  electrons.

Now, that's a lot, admittedly – more already than anybody can really envision.

But consider this: The number MAXR is so much larger than this, that if you actually had MAXR electrons, *you would have enough electrons to fill*

[illegible]

On the small end of things, picture in your mind that packed pile of MAXR electrons. Then picture yourself picking out just ten of those electrons. That ten – in relation to the whole – is the fraction you're talking about when you say "MINR."

So you see, the magnitude limits of the HP-28 aren't all that restrictive. Indeed, to further put things in perspective, you may have heard of human societies whose numbering systems went something like:

"1...2...3...more than 3."

And that was all the farther they could describe numerical magnitude.

So it is in every society. In this modern-day, technical world, for example, the numbering goes well beyond 3, but at some point, it runs out of names and meanings, too. "Millions...billions...trillions...quadrillions..." etc, up to about "nonillions" (?), which are on the order of  $10^{30}$ . But what do you call numbers on the order of  $10^{100}$  – or  $10^{400}$ ?\*

Truly, there is a limit to practical needs to describe numbers. Once society's limit may simply be a little higher than another's – but not much.

\*The authors recommend the term "several gadzillion."

## Posting Real Numbers: [CHS], [EEX] and Display Modes

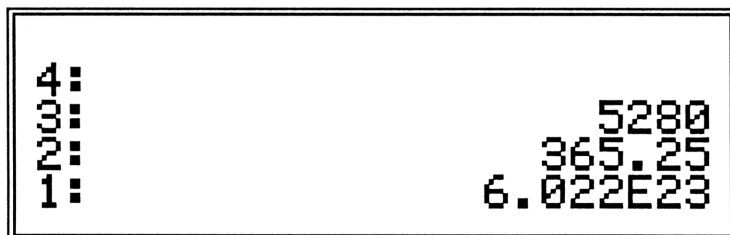
Now then: Those are the details about how the HP-28 can and cannot represent real numbers. Knowing these rules and limitations, it's time you started posting real numbers as memos to your calculations staff. You'll see right away that it isn't much different than posting any other kind of information, except that you use the number keys to key them in.

---

**For Example:** Post 5,280, 365.25 and  $6.022 \times 10^{23}$ .

**Solution:** Press [ATTN] [CLEAR] [STD] [ENTER] [5][2][8][0] [ENTER]  
[3][6][5][.] [2][5] [ENTER] [6][.] [0][2][2][E][2][3] [ENTER].

You should see:



4:	5280
3:	365.25
2:	6.022E23
1:	

---

Notice that when keying in that last number, **6.022E23**, you used the [E] key. You *could* have used [EEX] (Enter EXponent).

[EEX] works the same as [E] except for one case. Press [EEX]. What happens? Since you hadn't specified the number to the left of the E (the *mantissa*), [EEX] supplied you with one: 1. It's just an added nicety of the EEX function.

(Now press [ATTN] to clear the command line).



OK, now how about posting a negative number? You have two ways to do this:

First method: You can post the positive number in the usual way (up to and including pressing **ENTER**) and *then* press **CHS** (CHange Sign).

---

**Try It Now:** Press **CHS**. The number in Level 1 becomes negative.

Press **CHS** again – to make that number positive again.

---

Second method: You can change the sign of either the mantissa or the exponent at any time while you're keying in that part.

---

**Examples:** Post **-1.3**, **4.5E-24**, **-7.8E3**, and **-9E-54**.

**Solutions:** Press **1** **.** **3** **CHS** **ENTER** **4** **.** **5** **EEX** **2** **4** **CHS** **ENTER** **7** **CHS** **.** **8** **E** **3**  
**,** **CHS** **9** **EEX** **CHS** **5** **4** **ENTER**.

You'll see this:

4:	-1.3
3:	4.5E-24
2:	-7800
1:	-9.E-54

---

Notice that pressing **CHS** *before* you start to key in the number will work only if the command line is already active. If not, **CHS** will change the sign of the number in Level 1, as in the first method, above.

## Display Formats

---

**Try this:** Press `4` `,` `F` `I` `X` `ENTER`. Continuing on from the previous example, you should see this:

4:	-1.3000
3:	4.5000E-24
2:	-7800.0000
1:	-9.0000E-54



---

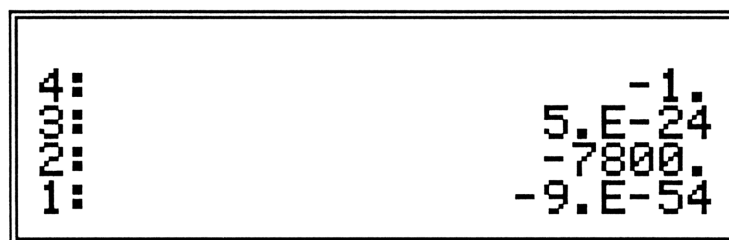
What's-a-goin' on?

Well, you just told the HP-28 to change the *format* of real numbers in the display. That is, the *values* of the numbers haven't changed – just the way you see them.

`4` `,` `F` `I` `X` `ENTER` told the HP-28 to display a FI~~X~~ed number of digits – four in this case – to the right of the decimal point. As you can see, that's just what it did. When there are no more significant digits to be displayed, one or more zeroes are added to the end of the number to fulfill the FI~~X~~ requirement.

---



**Next:** Press  **COMMAND**  **ENTER**. Here's what you'll see:



A calculator display showing a stack of four numbers. On the left, labels 4:, 3:, 2:, and 1: are aligned with the stack entries. The stack contains the following values from top to bottom: 5.E-24, -7800., -9.E-54, and -1.

Label	Value
4:	5.E-24
3:	-7800.
2:	-9.E-54
1:	-1

---

First of all, remember how  **COMMAND** retrieves your most recently posted command (**4, FIX**, in this case). Since the first character, **4**, is the one you wanted to change, you overwrote it with  and pressed **ENTER** to execute the rewritten command.

So there are now zero digits to the right of the decimal point. Again, the numbers haven't changed in value; only their appearances have.

**Remember!** *All these display formats are only the display's "editing" of the numbers for presentation to you. The internal representations of the numbers – for purposes of computation – are always fully 12 digits of precision.*

---

**Once Again:** Press **1****1****,****S****C****I****ENTER**. **Result:**

4:	-1.300000000000E0
3:	4.500000000000E-24
2:	-7.800000000000E3
1:	-9.000000000000E-54

---

This example brings up an important point: In the previous examples there were numbers in the display formatted in scientific notation even though the mode was **FIX**. That's because there are only 12 digits possible to display a real number. Therefore, any number larger than 999999999999 (twelve nines) or smaller than 0.000000000001 ("zero-point-eleven-zeros-and-a-one") will be displayed in scientific notation *by default*, because its magnitude exceeds the ability of the display to show in an explicit, one-part number.

But now, with **SCI** mode, not only have you set the number of digits to be displayed, but you've *forced* the display to use scientific notation for *every* number – regardless of whether or not that number could otherwise be correctly represented in the display. To see this, compare how Level 4 looks now to how it looked after the previous example. Although the exponent is 0, the number is still expressed in scientific notation here.

---

**Finally:** Press **S****T****D****ENTER**.

This is **ST**andar**D** display format, where you started on page 76. All significant digits of all numbers are displayed and scientific notation is used only when the number overruns the display's magnitude limits.

---

## The Stack and Postfix Notation

"OK, ok: Scientific notation...real-number representation limits...display formatting...when are we going to get to the part where I start *doing* things – like arithmetic – with real numbers?"

Right now. Begin by noticing that what you see in the display is, quite literally, a *stack* of numbers. It's true. Everything you've posted so far has been "stacked up" on the bulletin-board.

This particular stack may look upside down to you, compared to other stacks of things you've seen, because you put the latest additions on the *bottom* here. Aside from that, it works in much the same way as any stack of "stuff."

If you think about it for a moment, you'll realize that a stack is merely a Last-In-First-Out type of arrangement, where the last thing you put on the stack is the first thing you take off.

But – as you've so eloquently put it – putting things on and taking things off the stack doesn't accomplish a lot. You want to be able to do other things with the stuff on the stack. With real numbers, for example, it would be nice to do some math. Of course, you can.

But here's the idea to hang onto as you begin: An HP-28 command that uses this stack – any math operation, for example – assumes that *something to operate on will already be in the stack when you invoke the operation itself*.

In effect, you must first put onto the stack any number(s) you want to manipulate, and *then* perform the operation. This way of doing things is called postfix (post-affix: "to add after") because the operation is added *after* the operands.

## Real Number Commands: 0-, 1-, and 2- Number Operations\*

Just so that you have some simple arithmetic to follow,

---

**Do This:** Key in these numbers: 100, 64, 4.6, 7, 3 – in that order.

**Solution:** 100 ENTER 64 ENTER 4.6 ENTER 7 ENTER 3 ENTER.

And here's how your stack should look:

4:	64
3:	4.6
2:	7
1:	3

---

This is the reliable general procedure anytime you want to put a series of numbers in your stack, right? You just key in each number and press ENTER to post it.

\*If you already know how to do simple, postfix arithmetic on the HP-28, skip ahead to page 88.

---

**Start Crunching:** With your stack set up like that, press  $\boxed{\times}$ .

You should see this:

4:	100
3:	64
2:	4.6
1:	21

The numbers in Levels 1 and 2 were multiplied together, and the result was left at Level 1.

---

**Try Another:** Press  $\boxed{+}$ .

4:	-9.E-54
3:	100
2:	64
1:	25.6

The same thing happened – except that the result on Level 1 is now the sum of the (previous) bottom two levels.

---

Notice that, because two numbers were combined into one number here, there is one number fewer in the stack now, and the rest of the stack has therefore dropped one level. This is the way each *2-number* math command works. It takes the bottom *two* numbers from the stack, combines them, and puts the result back on the bottom of the stack.

Notice also that both addition and multiplication are *commutative operations*. That is, their results do not depend upon the order of the two numbers involved. Clearly,  $1 + 2 = 2 + 1$ ; and  $2 \times 3 = 3 \times 2$ .

So for this addition, it wouldn't have mattered if the **4.6** had been above **21** or below it in the stack. This is *not* the case with other arithmetic operations, such as division and subtraction.

---

**To Wit:** Press  $\boxed{\div}$ .

4:	-7800
3:	-9.E-54
2:	100
1:	2.5

Notice that the order of evaluation is "Level 2 divided by Level 1."

---

**Then Of Course:** Press  $\boxed{-}$ . Here's the result – and now you know why, right?

4:	4.5E-24
3:	-7800
2:	-9.E-54
1:	97.5

---



Notice that throughout this little set of examples, all those other numbers you had "floating around" above Level 4 have successively made their reappearances. Your stack has been steadily "settling" downward as you perform these arithmetic operations that combine two numbers into one.

This settling is a very important part of the stack's operation. It becomes obvious with any problem that forces you to compute several intermediate results before combining them into a final answer.

**For Example:** Find  $((2.4 \times 6.8) + (5.9 - 2.3) - (17.5 \div 4)) \times 43.2$

**Solution:**

Press

$\boxed{2} \boxed{\cdot} \boxed{4} \boxed{\text{ENTER}} \boxed{6} \boxed{\cdot} \boxed{8} \boxed{\times}$	$(2.4 \times 6.8)$
$\boxed{5} \boxed{\cdot} \boxed{9} \boxed{\text{ENTER}} \boxed{2} \boxed{\cdot} \boxed{3} \boxed{-}$	$(5.9 - 2.3)$
$\boxed{1} \boxed{7} \boxed{\cdot} \boxed{5} \boxed{\text{ENTER}} \boxed{4} \boxed{\div}$	$(17.5 \div 4)$
$\boxed{-}$	$(5.9 - 2.3) - (17.5 \div 4)$
$\boxed{+}$	$(2.4 \times 6.8) + (5.9 - 2.3) - (17.5 \div 4)$
$\boxed{4} \boxed{3} \boxed{\cdot} \boxed{2} \boxed{\times}$	

The result is 671.544. Notice how you worked from the inner parentheses outward, thus eliminating the parentheses as you go. That hearkens back even to your earliest days in arithmetic class, doesn't it?

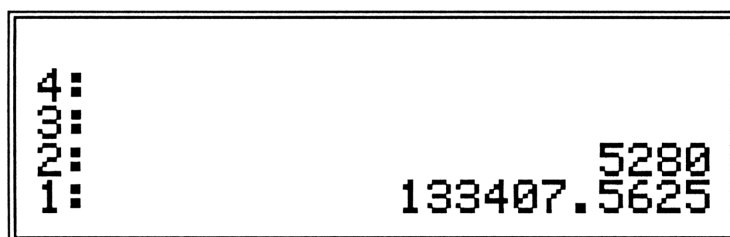
Notice also how you combined the values in each parenthesized portion, "melting" them into intermediate results, which you allowed to "stack up" while you computed the next portion!

OK, that probably gives you some idea of the workings of the 2-number math functions. What about 1-number math functions?

---

**You Asked For It:** First, get rid of some of the extra numbers, by pressing **DROP** **DROP** **DROP** **DROP** **DROP** **DROP** (that's 6 times).

Then press **■****x<sup>2</sup>** (**■****+**). Here's what you should see at this point:

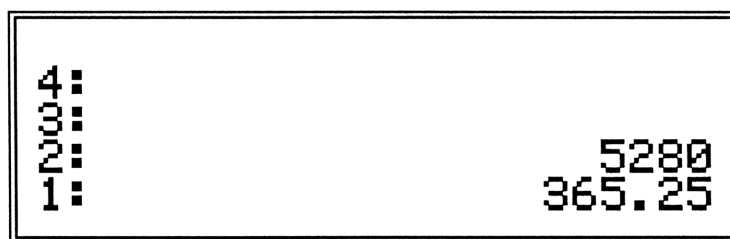


The calculator display shows a stack with four levels. On the left, the levels are labeled 4, 3, 2, and 1. On the right, the values are 5280 and 133407.5625. The value 5280 is aligned with level 4, and 133407.5625 is aligned with level 1.

$x^2$  is a 1-number function, since it takes only one number off the stack. Since it replaces that one number with its result, only Level 1 is affected.

---

**Affect It Again:** Press **■****√** (**■****-**). **■****√** is also a one number function – and no prizes for guessing what operation you just did:



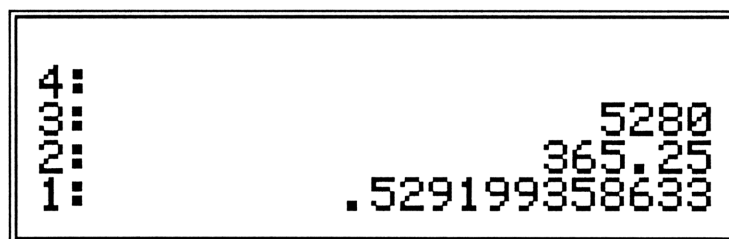
The calculator display shows a stack with four levels. On the left, the levels are labeled 4, 3, 2, and 1. On the right, the values are 5280 and 365.25. The value 5280 is aligned with level 4, and 365.25 is aligned with level 1.

So now you've seen 2- and 1- number operations.

Believe it or don't, there are some *0-number* operations. That is, there are some operations that take nothing from the stack, but leave a value there nevertheless.

---

**Pick A Card – Any Card:** Press `R``A``N``D``ENTER`. You'll see something like this:\*



4:	
3:	5280
2:	365.25
1:	.529199358633

---

RAND is the RANDom number generator. It takes no numbers from the stack, but leaves a random number there at Level 1, thus bumping everything else up by one level.

\*Because the number on Level 1 has been chosen at random by the calculator, the number in your display may not be the same as the one shown above. In fact, if it *is* the same, take a break right now and go buy a lottery ticket.

## Arithmetic Practice

Here are some not-so-trivial problems to let you practice your postfix arithmetic skills and some of those 2- and 1-number functions.

A reminder: You'll notice an abundance of parentheses here, since that's how you're used to seeing such problems expressed on paper. But there's no need for parenthesis keystrokes when solving these on your HP-28 (or any other postfix calculator). The way these numerical "memos" stack up on your "bulletin board" allows your arithmetic "staff members" to work with them without using any parentheses!

And keep in mind the rule of thumb for handling parenthetical expressions:

"Work from the innermost parentheses outward."

1.  $((((1 + 2) \times 3) + 4) \times 5) = ?$

2. Calculate

$$\frac{-12 + \sqrt{12^2 - (4 \times 3 \times (-5))}}{2 \times 3}$$

(You might recognize this as one solution to a quadratic equation  $ax^2 + bx + c = 0$ , where  $a = 3$ ,  $b = 12$  and  $c = -5$ .)

3.  $\sqrt[3]{7} = ?$

(This is the cube root of 7. Notice that there is no cube root command on the HP-28 and that you "raise-to-a power" via  $\blacksquare \wedge$ . Notice also that  $\sqrt[3]{7} = 7^{1/3}$ .)

4.  $173e^{\left[ \frac{-16 + (43 \times 0.004)}{3.2 - 16.3} \right]} = ?$

(e<sup>x</sup> is EXP from the LOGS menu.)

5.  $1 + 0.5 + \frac{0.5^2}{2!} + \frac{0.5^3}{3!} + \frac{0.5^4}{4!} = ?$

(x! is the FACTorial of x. The command FACT lives in the REAL menu. In case you're curious, this problem is asking you to add the first five terms of the Taylor series approximation of e<sup>0.5</sup>. You might want to compare your result here with the result of the HP-28's EXPOnential function.)

## Arithmetic Practice Solutions

1.  $\boxed{1} \boxed{\text{ENTER}} \boxed{2} \boxed{+} \boxed{3} \boxed{\times} \boxed{4} \boxed{+} \boxed{5} \boxed{\times}$

The result = **65**

2.  $\boxed{1} \boxed{2} \boxed{\times^2} \boxed{4} \boxed{\text{ENTER}} \boxed{3} \boxed{\times} \boxed{5} \boxed{\text{CHS}} \boxed{\times} \boxed{-} \boxed{\sqrt{\phantom{x}}} \boxed{1} \boxed{2} \boxed{\text{CHS}} \boxed{+} \boxed{2} \boxed{\text{ENTER}} \boxed{3} \boxed{\times} \boxed{\div}$

The result = **0.38047614285**

3.  $\boxed{7} \boxed{\text{ENTER}} \boxed{3} \boxed{1/\times} \boxed{\wedge}$

The result = **1.91293118277**

4.  $\boxed{4} \boxed{3} \boxed{\text{ENTER}} \boxed{\cdot} \boxed{0} \boxed{0} \boxed{4} \boxed{\times} \boxed{1} \boxed{6} \boxed{\text{CHS}} \boxed{+} \boxed{3} \boxed{\cdot} \boxed{2} \boxed{\text{ENTER}} \boxed{1} \boxed{6} \boxed{\cdot} \boxed{3} \boxed{-} \boxed{\div} \boxed{\text{LOGS}} \boxed{\text{EXP}} \boxed{1} \boxed{7} \boxed{3} \boxed{\times}$

The result = **579.135149585**

5.  $\boxed{1} \boxed{\text{ENTER}} \boxed{\cdot} \boxed{5} \boxed{+} \boxed{\cdot} \boxed{5} \boxed{\times^2} \boxed{2} \boxed{\text{REAL}} \boxed{\text{FACT}} \boxed{\div} \boxed{+} \boxed{\cdot} \boxed{5} \boxed{\text{ENTER}} \boxed{3} \boxed{\wedge} \boxed{3} \boxed{\text{FACT}} \boxed{\div} \boxed{+} \boxed{\cdot} \boxed{5} \boxed{\text{ENTER}} \boxed{4} \boxed{\wedge} \boxed{4} \boxed{\text{FACT}} \boxed{\div} \boxed{+}$

The result = **1.6484375**

$\boxed{\cdot} \boxed{5} \boxed{\text{LOGS}} \boxed{\text{EXP}} = 1.6487212707$

Of course, there's a whole lot more to this machine than just your basic arithmetic. You're surely itching to crack into all that – and who can blame you? It's never any fun to hammer out the fundamentals before getting to the good stuff.

Nevertheless, there's quite a bit more hammering to do before you get into serious number-crunching. In fact, the only reason you're seeing arithmetic with *any* numbers right now is to learn about how the stack works. And there's a whole lot more to see before you're ready to manage your staff. This part of the course, then, is still a filling-in of the details of the everyday operations of your staff and bulletin board. After all, you need to *fully* construct this imaginary world in your mind before you can operate with its help.

## STACK Operations

Here's a quick rundown of all the things you know so far about the stack.

- You've already been introduced to some of the stack's basic math operations.
- You know how `ENTER` and the command line are used to put things on the stack and how `DROP` is used to remove them.
- You've even seen how the operation called DUP can be used to duplicate the first level of the stack (remember way back on page 24?).

There are a lot more stack commands than just these 3, and since the stack is basically your work area, you'd better know your way around it.\* The next few pages, therefore, are a continuation of your introduction to the HP-28's stack commands. Don't expect to inscribe in your mind or on your fingertips after a single pass here; it will take time until you're fluent in using all these different specialized tools. But take a look now, and begin your practice....

\*But if, on an outside chance, you feel at home there already, then by all means, jump now to page 105.

## **ENTER's Second Job**

Before you get to any new commands, take a second look at the hardest-working key of all: **ENTER**. Apart from posting (or evaluating) the command line, it has another use altogether!

This second use of **ENTER** is that, when the command line is not active, **ENTER** functions as a "do-it-now version" of DUP. This is another convenient extra, because most commands "eat" items off the stack; having back-up copies becomes important.

---

**Try It:** First, be sure there's no half-built command line (by pressing **ATTN** if necessary).

Key in a number. Then press **ENTER** several times. See how it duplicated that number?

Now DROP all those duplicate entries out by pressing **DROP** several times.

---



## The **SWAP** Command

Another commonly used stack operation is **SWAP**. It functions to exchange the contents of Levels 1 and 2.

So what good is that? Well, remember that division and subtraction are not commutative operations; their operation depends on the order of the numbers in Levels 1 and 2.

**SWAP** gives you the ability to reverse the order of these two stack levels, and because there are many operations (besides  $-$  and  $\div$ ) that are *not* commutative, **SWAP** becomes very much in demand.

## How to **CLEAR** the Stack

Remember **DROP** – that command that throws away the bottom (most recent) memo and therefore "drops" all the rest of the memos down one level?

Well, **CLEAR**, like **DROP**, is a stack clean-up command. If there are items on the stack that you don't need any more, **CLEAR** removes them. But you'd better be sure about what you're doing: **CLEAR** clears the *whole* stack.

---

**Clean Your Slate:** Press **CLEAR** now – and see this:

4:
3:
2:
1:

Up to now, the stack commands you've been using are all important enough to have keys of their own.

But everyone has to live someplace; the less commonly used stack commands live in the STACK menu (▀G).

---

**Get That Menu:** Press ▀STACK. You should now see this:



---

You see that DUP lives here. But why? If ENTER does the same thing, and it's sitting right there on the keyboard, why put DUP in a menu also?

Well, recalling that ENTER is exempt from the effects of α, if you wanted DUP to appear on your command line, you'd have to type DUP manually (perish the thought) unless it were available on a menu key.

All this planning – just for your convenience!

Continue your perusal of this STACK menu. Reading from left to right in the menu, the next command over is OVER. OVER makes a copy of whatever is in Level 2 and then pushes this copy onto the stack (to "push" something onto the stack means to put it on at Level 1, thus bumping everything else up one level). In effect, then, OVER makes a copy of Level 2, jumps *over* the current Level 1 and pushes the copy on the stack "beneath" it.

---

**Drum Roll, Please:** Press **1** **2** **OVER**. And here's the result:



See how this works? Remember, although you never pressed **ENTER** to put the **1** and **2** on the stack in the first place, OVER is an immediate-execution function, which means that, for all practical purposes, you had a command line that read **1, 2, OVER** before an **ENTER** was "caused" by the **OVER**. Since the HP-28 posts (or obeys) memos from left to right, this explains how the numbers got onto the stack by the time the OVER happened!

Horse around with this some more, but after you've finished, set up your stack so that it looks like this (from the above example, all you would need to do is press **DROP** **3** **ENTER**):



Now then: Just to keep things interesting, skip over to the fifth item in the STACK menu – to the command called ROT.

This command ROTates the bottom three levels of the stack "upward." In effect, Level 3 is *removed* (not copied) and then pushed onto the stack.

---

**Prove It:** Press **ROT**. The result:



See? Just a simple rotation of the three bottom-most "things" in the stack. Press **ROT** twice more to return the stack to its original order.

---

**And Just For Laughs:** Press **SWAP** **ROT**.

What happened? The bottom three levels of the stack have been reversed.



(Now press **CLEAR** to shred all evidence of levity.)

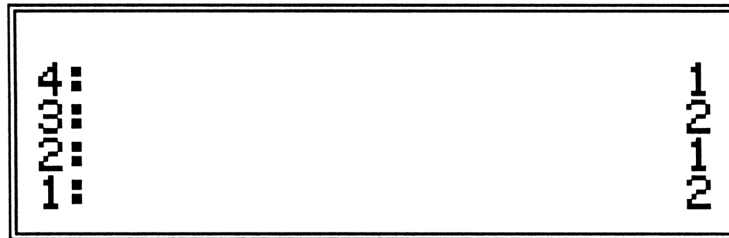
Going back now to pick up those two commands you skipped:

DUP2 and DROP2 are analogous to DUP and DROP but – as their names imply – *they operate on both the first and second levels at the same time.*

DUP2 makes a copy of both the first and the second levels and then pushes them on the stack. That is, it DUPlicates the contents *and ordering* of the bottom 2 levels of the stack.

---

**Watch:** Press `1` `,` `2` `DUP2`. Then press `↔` to get the menu out of the way for a minute so that you can see the bottom four levels of the stack:

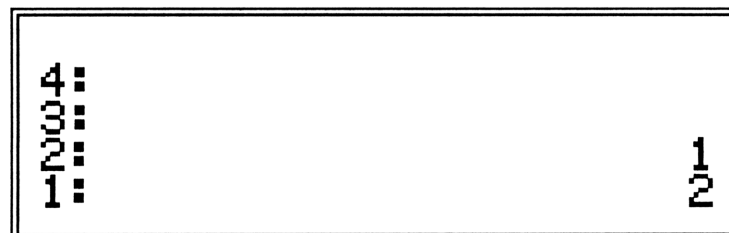


---

And DROP2 drops (discards) the bottom two levels of the stack,

---

**Like So:** Press `↔` `DROP2` `↔`.



---

No sweat, right?

More? All right. Press **↔** to get the menu back and then **NEXT** to see more of it.

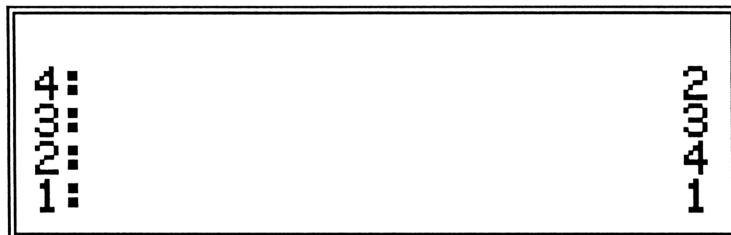
ROLL (which has a key of its own – the shifted **DROP** key) and ROLLD (on the second "page" of the STACK menu (now showing) are a matched set.

ROLL is a generalization of ROT. Its job is to retrieve a number from *any* given level of the stack and push it back on again at Level 1, where it's more directly usable. As you know, ROT does this same kind of retrieval service, but only with Level 3. With ROLL, you can specify whatever level you want.

---

**Easier Done Than Said:** First, press **3** **,** **4** **ENTER** – to load up a total of 4 levels.

Now press **4** **■** **ROLL** **↔**. You see:

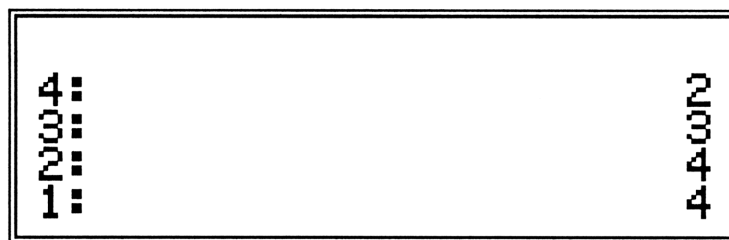


---

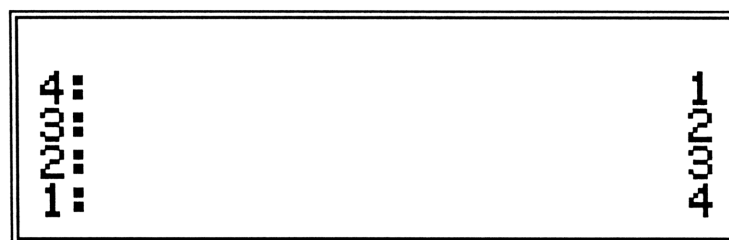
The bottom four stack levels have now been ROLled. You can see that pressing **3** **■** **ROLL** is equivalent to pressing **ROT**. Similarly, pressing **2** **■** **ROLL** is equivalent to pressing **SWAP**.

But while the effect of what you're doing here is really quite simple, this may be one of your first encounters with the use of a command that calls for an "argument." So take a moment for a backstage tour, a behind-the-scenes look at what's going on here....

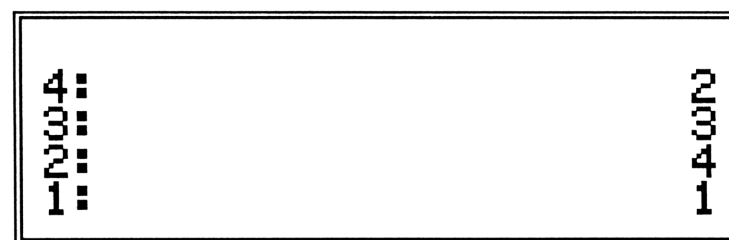
ROLL is a do-it-now function, so it has a built-in **ENTER** after it. Therefore, the argument, 4, that you just keyed in *does* go onto the stack at Level 1 *before* the ROLL gets under way (so keep in mind there's a 1 up on Level 5):



Next, your memo posting "office boy" obeys the do-it-now ROLL, which says, in effect, "DROP (throw away) that bottom **4**, but *note its value* on the way to the dumpster." He obeys, and so the stack (bulletin board) looks like this once again:



He now climbs up the stack to Level 4 (because he just threw away a **4**), removes whatever's there (a **1**) and pushes it back onto the stack at the bottom:



As you get more practice, of course, you won't even need to think about all these intermediate steps. After awhile, it'll feel obvious that if you want the 4th "thing" in the stack to "come on down," you just key in a 4 and use ROLL.

The main point of this backstage tour is that ROLL is a good example of a one-argument, postfix operation.

It's postfix because whatever it operates on (and with) had better be in position by the time it comes along – and this is the case. Everything is sitting on the stack exactly right – so that it does what you want it to.

And it's a one-argument function because it needs one parameter ("argument" – remember your staff's math jargon?) *in addition to the stack's current contents* to tell it the "where's" and "how-many's" of its operation.

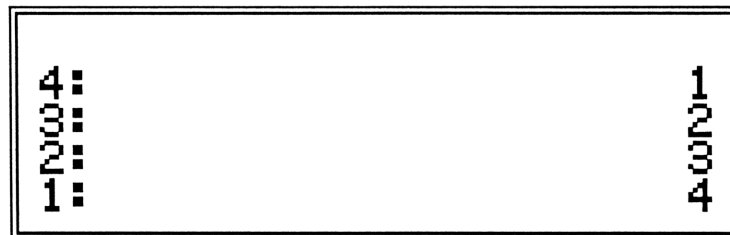
The way in which ROLL simultaneously uses and discards its argument is very typical of the HP-28's treatment of arguments that you load into the stack. It notes them while throwing them away (DROPPing them).

Another good example of this is ROLL's twin sister, ROLLD (in the STACK menu).

ROLLD is the reverse of ROLL. It ROLLs the stack Down in the same way that ROLL rolls it up – sending what's in Level 1 up to the specified level.

---

**Going The Other Way:** Press **4** **↔** **ROLLD** **↔**. Thus, the stack is returned to the order it had before you executed the ROLL:



And you know the reasons for the parameter, 4, and the messing about with the **↔** key, right?

---



Onward and upward to more strange and wonderful stack manipulation stuff (on the second "page" of the STACK menu)...

DUPN and DROPN are generalizations on DUP and DROP in the same way that ROLL is a generalization of ROT – including the treatment of the one parameter. Both commands first DROP a number off the stack and use it to tell the number of levels on which to operate.

---

**For Example:** There are now four items on the stack. Press `4` `↔` `DUPN` `↔`.

Voila! There are now eight levels on the stack. The bottom four levels have been duplicated and then pushed back onto the (bottom of the) stack.

Press `DROP` four times to prove this to yourself...

---

Thus, `1` `DUPN` is the equivalent of `DUP`, and `2` `DUPN` is the equivalent of `DUP2`.

---

**Likewise:** DROPN will *remove* the specified number of levels. Press `4` `↔` `DROPN`. All four of the remaining levels are dropped (nuthin' left)!

---

Thus, `1` `DROPN` is equivalent to `DROP`, and `2` `DROPN` is equivalent to `DROP2`.

"But wait – there's more!"

PICK (same menu) is a generalization of OVER. It drops an argument from the bottom of the stack, using its value to count up the stack. It then makes a *copy* of that level (unlike ROLL, which extracts that entry altogether) and pushes this copy onto the stack.

Thus, **2** **PICK** is equivalent to **OVER**, and **1** **PICK** is equivalent to **DUP**.

And more: DEPTH is a command that takes nothing from the stack. It merely counts the number of levels currently on the stack and then pushes its resulting count onto the stack (as the new bottom number, of course).

---

**Do This:** Be sure the stack is clear, then press **⇄** **DEPTH**, and it returns a **0** – no mystery, right? The stack was empty.

Press it again and it returns a **1** . (Why ?)

---

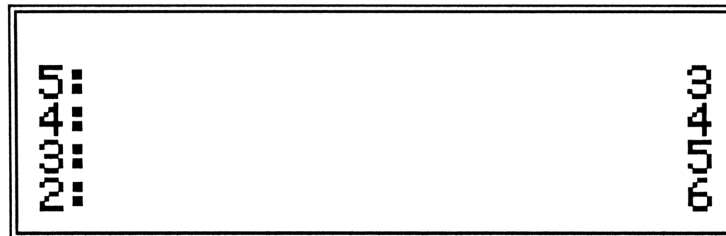
Would you believe...more?

■VIEW↑ (■CHS) and ■VIEW↓ (■EEX) don't actually change the stack in any way. As their names imply, they allow you simply to *view* portions of the stack.

---

**Take A Look:** Press ■CLEAR 1,2,3,4,5,6,7ENTER ⇄.

Now press ■VIEW↑. What happens?



The whole stack was *scrolled* (not rolled) down. You can now see what was beyond the top line of the display.

Pressing ■VIEW↑ again will move the view up one more line. Pressing and holding ■VIEW↑ will scroll until the top line of the stack is visible. ■VIEW↓ has the opposite effect.

---

One more thing:

**■****VISIT** (**■****1**) is just like **EDIT**, except that it uses the number in Level 1 as an argument – DROPping it and noting it in the usual manner – to select the stack level to be edited.

Then, in terms of actual editing, **VISIT** is exactly like **EDIT**, except that when you press **ENTER** the altered contents of the stack level you've been editing are placed back at that level – not at Level 1 (and you would expect this, right?).

Try **VISIT**ing various stack levels, just to get the hang of it.

## Strenuous But Practical Stack Practice Problems\*

Solve the following as efficiently and expertly as you now know how:

1.  $\left(\frac{2}{7} - \frac{14}{15}\right) + \left(\frac{14}{15} - \frac{2}{7}\right) = ?$

2. Assume the bottom three levels of the stack contain the coefficients of a quadratic equation (i.e.,  $ax^2 + bx + c = 0$ ), where Level 3 is  $a$ , Level 2 is  $b$ , and Level 1 is  $c$ . Give a sequence of keystrokes that will produce the equation's lesser root, whose formula is

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

3. You've seen (on page 96) how you might reverse the bottom three stack levels. What are some keystrokes that will reverse the bottom four stack levels? How about reversing Levels 2, 3 and 4?
4. Swap Levels 1 and 2 for Levels 3 and 4. Then swap Levels 1, 2 and 3 for Levels 4, 5 and 6. How about swapping Levels 1 and 4?
5.  $(((((12.45 + 3) \times 12.45 + 3) \times 12.45 + 3) \times 12.45 + 3) \times 12.45 + 3) \times 12.45 + 3 = ?$
6. Who played Elliot Ness in the television series "The Untouchables?"

\*Try saying *that* three times in a row.

## S. B. P. S. P. P. Solutions

1.  $\boxed{2} \boxed{\text{ENTER}} \boxed{7} \boxed{\div}$  (2/7 in Level 1.)  
 $\boxed{1} \boxed{4} \boxed{\text{ENTER}} \boxed{1} \boxed{5} \boxed{\div}$  (2/7 in Level 2; 14/15 in Level 1)  
 $\boxed{\text{STACK}} \boxed{\text{DUP2}}$  (Copy both values)  
 $\boxed{\text{SWAP}} \boxed{-}$  (14/15 - 2/7)  
 $\boxed{\text{ROT}} \boxed{\text{ROT}} \boxed{-} \boxed{+}$

Result =  $\frac{1}{10}$

2. One solution:

$\boxed{\text{STACK}} \boxed{\text{ROT}} \boxed{2} \boxed{\times}$  (Stack = b; c; 2a, in descending order)  
 $\boxed{\text{DUP}} \boxed{\text{ROT}} \boxed{\times}$  (Stack = b; 2a; 2ac)  
 $\boxed{2} \boxed{\times}$  (Stack = b; 2a; 4ac)  
 $\boxed{3} \boxed{\text{NEXT}} \boxed{\text{PICK}}$  (Stack = b; 2a; 4ac; b)  
 $\boxed{\text{X}^2}$  (Stack = b; 2a; 4ac; b<sup>2</sup>)  
 $\boxed{\text{SWAP}} \boxed{-} \boxed{\sqrt{\phantom{x}}}$  (Stack = b; 2a;  $\sqrt{(b^2 - 4ac)}$ )  
 $\boxed{\text{NEXT}} \boxed{\text{ROT}} \boxed{\text{CHS}} \boxed{\text{SWAP}} \boxed{-} \boxed{\text{SWAP}} \boxed{\div}$

3.  $\boxed{\text{STACK}} \boxed{\text{SWAP}} \boxed{\text{ROT}} \boxed{4} \boxed{\text{ROLL}} ;$   
 $\boxed{\text{STACK}} \boxed{4} \boxed{\text{NEXT}} \boxed{\text{ROLLD}} \boxed{\text{SWAP}} \boxed{\text{NEXT}}$   
 $\boxed{\text{ROT}} \boxed{4} \boxed{\text{ROLL}}$

4. 4 **ROLL** 4 **ROLL** ;  
 6 **ROLL** 6 **ROLL** 6 **ROLL** ;  
 4 **ROLL** **SWAP** 4 **STACK** **NEXT** **ROLLO** or 4 **STACK** **NEXT** **ROLLO** **NEXT** **ROT**

5. 1 2 • 4 5 **ENTER** **ENTER** **ENTER** **ENTER** **ENTER** **ENTER** (Six copies.)  
 3 **ENTER** 1 **ENTER**  
**MODE** **NEXT** **+CMD** (Enable **COMMAND**.)  
**STACK** **α** **ROT** **×** **OVER** **+** **ENTER** (Remember these keystrokes.)  
**COMMAND** **ENTER**  
**COMMAND** **ENTER**  
**COMMAND** **ENTER**  
**COMMAND** **ENTER**  
**COMMAND** **ENTER** (Do them five times.)


Result = 4699789.91278

See how easily you can repeat any given set of keystrokes?

6. Robert Stack

**End  
construction.  
Thank you for your  
patience.**

This is it. You've fully constructed your office surroundings in your mind. Now you're ready to learn how to work with all the high-powered brains on your HP-28 staff. Looking back for a moment, you can see how far you've come as "president" of this "collection of mathematicians" called the HP-28.

- You know how the keyboard connects to the "bulletin board" display, and you have a basic understanding of the layout of the keyboard—how the character and command keys are arranged and how the  key serves to change the meanings of most of the keys (changes to the red-printed functions).
- You know how to use the command line to edit and post "memos" or commands on the bulletin board. You know that you have a "card catalog" of reserved words (menu items) that the HP-28 will recognize as commands rather than simple messages. And you know that many of these are "do-it-now" commands, so they won't be posted onto the stack but rather, executed immediately (unless you have switched to  $\alpha$  mode).
- You know how arithmetic works on the HP-28, with its postfix logic, where both operands precede the operator. And you know about 0-, 1-, and 2- number arithmetic operators.
- You know several gadzillion charming and useful stack-manipulation commands – ways to rearrange the numbers in the stack.
- You're familiar enough with the HP-28 that you can now dispense with this "office world" mind game and instead see this machine for what it really is – just a mindless but powerful calculator that will obey your commands.





THE "STUFF" UPON WHICH THE HP-28 WORKS

## An Equal Opportunity Calculator

Unlike most calculators, the HP-28 is not limited to working only with real numbers. Though real numbers are tremendously useful, and much real-world work involves their manipulation, you might also want more flexibility in your problem solving – the ability to manipulate other sorts of information, too.

Unfortunately, with increased flexibility comes increased complexity (you can't get somethin' fer nuthin'). With all these new sorts of information and new ways of manipulating them, there comes a whole slew of new rules – and new exceptions to those rules.

Fortunately, however, there is an underlying, unifying logic to how things work in the HP-28. Once you get a grip on this general operating scheme, you should be able to move from manipulating one sort of information to another without much discomfort.

How can that be? How can you treat characters, for instance, in *any* way similarly to the way you treat real numbers?

Be assured, you can. Though you must remember certain details for certain types of information, *you will find that the machine treats most every "thing" in very much the same way!*

## The HP-28's Philosophy of Information

Despite all evidence to the contrary, there are really only three basic kinds of information that the HP-28 understands. They are:

Real Numbers  
Characters  
Bits (short for Binary digits).

This is because these three basic information types are so very useful, forming the backbone of almost all information processing that goes on in the real world.

But that isn't nearly the whole story.

Though these three information types are used almost universally, *they are almost never used just as they are* (the major exception being real numbers).

Specifically, **characters** are more commonly used as elements of character *strings* or pages of *text*.

**Bits** are more often grouped into *bytes* and *words*, or used as *binary integers*.

Even **real numbers** are often grouped into *vectors*, *arrays*, *tables*, and other more exotic "things."

The good news is this: HP-28 has anticipated your need for such complex groupings of these three simple information types. Built right into this calculator is the capability to allow you to build and manipulate compound objects in familiar, useful and convenient ways!

# Real Numbers

You've already had an introduction to real numbers – what they are and some of the things you can do with them. *Now you must begin to look at them as part of the larger scheme of things.*

Real numbers are "things," *objects* – one specific *kind* of information. Sure, they're somewhat familiar to you because you often use them to solve problems in your daily life – including the problem of how to introduce yourself to the HP-28's stack and arithmetic.

But stop and take a good, hard look at that. Exactly how have you used real numbers in this Course so far?

Two ways:

1. The first way is probably so familiar to you that you didn't even notice it. You used real numbers as *object information*. That is, they were data – information for its own sake – to be manipulated in order to get other numeric information.

This is the way that you look at real numbers when you do things like addition, subtraction, multiplication and division. *You are working on numbers that are meaningful to you in order to get another number that is meaningful to you.*

2. You have also seen numbers used not as data, but as parts of commands. For example, ROLL doesn't use the Level-1 stack value as data. Rather, it uses the number as an indicator of how it is supposed to work. In other words, *real numbers used in this way are meaningful to you only because they help you control the machine.*

3. And here's yet a third way to use real numbers—a way you've not yet encountered in this book: use them as truth values.

That is, use two different real numbers (conventionally 0 and 1) to represent opposing states or responses (e.g., yes or no, on or off, set or clear). In this sense, then, a real number can be used to represent qualitative information.

For example, if you're comparing two real numbers to see if the first is greater than the second, you would use the > command.

---

**Try It:** Press `2``ENTER` `3``ENTER` `█``>``ENTER`. The result is `0`. Then press `3``ENTER` `2``ENTER` `█``>``ENTER`. The result is `1`.

To the HP-28 in this context, `0` means that the comparison is false (i.e. the answer is "no, 2 is not greater than 3"). Conversely, a `1` as a result means that the comparison is true.

All such comparison operations (<, >, ==, SAME, ≤, ≥, ≠) will return either a one or a zero depending upon whether the result is true or false.

---

You can begin to see that a number or any other type of information is actually quite meaningless unto itself. *It gains meaning based on how it is used.* That's a basic concept – an underlying truth – of the HP-28.

You've also seen another underlying truth of the HP-28 – the stack. It may have been a rather new idea to you. It does present some new problems and new ways of doing things, but it also opens up many new possibilities – new and powerful concepts, such as postfix operation.

So it's time to begin exploring these different uses of information, seeing first-hand just how the HP-28 builds upon the three fundamental information types, creates other information "objects" and combines them on its stack....

## REAL NUMBER STRUCTURES

### Complex Numbers

A good starting point: With real numbers as your building blocks, the simplest compound-data-type you can build is the *complex number*.

As an information structure, a complex number is simply an *ordered pairing* of real numbers – a *list* with two real-number *elements*. The first element of the pair is considered the *real* part of the number; the second is called the *imaginary* part.

On the HP-28, complex numbers are represented by bracketing two real numbers between parentheses, separating the two numbers with a delimiter.


---

**Build One:** Press **MODE** **STD** **↔** **CLEAR**.

Then press **(** **2** **SPACE** **3** **.** **4** **ENTER**. This is what you'll see:



Notice these things:

1. You didn't need to press  at the end of the number. If the complex number has other things following it in the command line, then you must use  $\rangle$  as a delimiter. If not, the HP-28 will automatically add the right-hand parenthesis before posting the number.
2. The space you used as a delimiter in this example was replaced by a comma when you posted it. Regardless of the delimiter you use when keying in the complex number, the HP-28 will use a comma (unless you're using the comma as the radix mark, in which case it will use a period) in the posted form.
3. The real and imaginary portions of the number are on the *same stack level*. Both this and the fact that they are grouped together inside parentheses tells you clearly that *this is a single object – one number*. From now on, unless you purposely break it into its components, it will be treated as one number – one object.

That's all there is to it; that's how you key in and post a complex number "object."

Now, what good is it?...

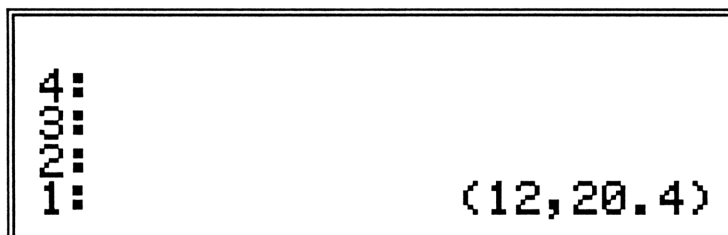
Complex numbers have, by definition, mathematical properties quite similar to those of real numbers. Thus, many HP-28 operations meaningful for real numbers are also meaningful for complex numbers.

You can, for example, perform arithmetic operations (+, -, x, ÷, etc.), trigonometric functions (SIN, ASIN, COS, ACOS, etc.) and logarithmic functions (LOG, ALOG, LN, EXP, etc.) on complex numbers.

And in all these operations, *you can perform arithmetic using a mixture of complex and real numbers*. The real number is converted by the machine into a complex number (with a 0 imaginary part) before the operation commences. The result is always complex.

---

**Watch:** Press `ENTER` `5` `×` `+`.



4:  
3:  
2:  
1: (12,20.4)

---


Pretty slick, right? With those few keystrokes – and the stack logic you now know all about – you multiplied a real number by a complex number, then added the result to a complex number.

So you can see right away that one major advantage of a compound data type such as a complex number is that the components of the object are manipulated together as a unit in new and meaningful ways – and you don't have to expend any energy trying to keep track of all the parts!






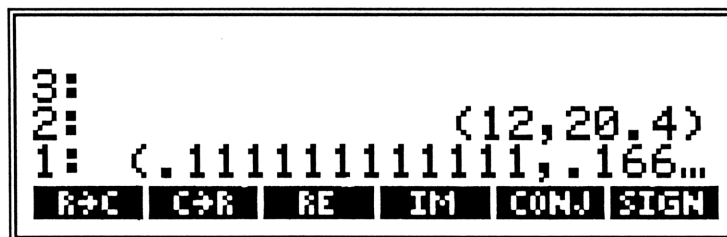
However, another (complementary) advantage is that you can pull apart the complex number into its component pieces, do things to them individually, and then reassemble them into a complex number. In fact, you can build a complex number from pieces generated by other, unrelated operations.

So in addition to the strictly mathematical operations in your complex-number repertoire – operations they share with the real numbers – you have other operations designed *specifically* for complex numbers.

Any guesses as to where these type-specific operations might "live?" In the CMPLX (  ) Menu, of course!

**Try This:** Press         

Then    ("Real to Complex"). Here's what you get:



R→C allows you to construct a complex number from two real numbers that are in stack Levels 1 and 2. The real portion is taken from Level 2; the imaginary portion from Level 1.

As you might expect, the function C→R (also in this menu) allows you to go the other way – decompose a complex number into two real numbers – where the real portion goes to Level 2 and the imaginary portion to Level 1.

Notice that the complex number is longer than the display can hold. When this happens, the HP-28 will use an ellipsis (...) to indicate the over-run – just as you saw with the command line (page 53).

To view the entire complex number, therefore, you must do one of two things:

- (i) Edit Level 1 (with either EDIT or VISIT), using the cursor keys to scroll through the entire object.

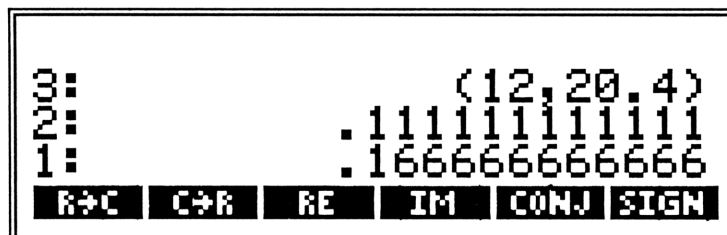
The numbers in the command line will be represented in standard display mode (STD), regardless of the mode the rest of the display is using. Remember to press **ATTN** to leave the command line so that you don't inadvertently change the number;

- (ii) Decompose the object with  $C \rightarrow R$  and examine the elements individually in the stack. Remember to rebuild the complex number with  $R \rightarrow C$ , if necessary, when you're done.

Actually, there is a third option, one that doesn't show you the whole object. You change the display mode (as on page 78) to show you fewer digits of each of the components of the complex number. In this case, **6** **▸** **F** **I** **X** **ENTER** or **3** **'** **S** **C** **I** **ENTER** will show you the whole object with less precision, if you want it.

---

**Now Try This:** Press **C↔R**. You see:



3: (12, 20.4)  
2: .11111111111111  
1: .16666666666666  
R↔C C↔R RE IM CONJ SIGN

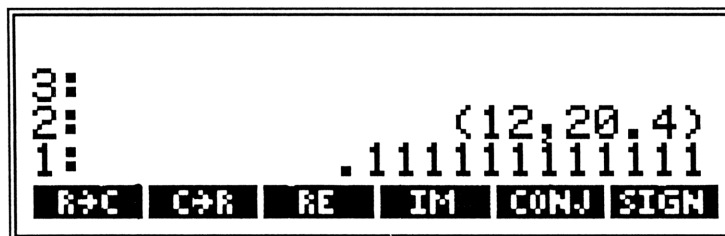
---

See how you decomposed the complex number that was at Level 1 into its two components? The real part is now at Level 2; the imaginary part is at Level 1.

Next, press **C↔R** to recombine those two parts once again. Simple, right?

---

**OK:** Press **RE**.



3: (12, 20.4)  
2: .11111111111111  
1: .16666666666666  
R↔C C↔R RE IM CONJ SIGN

---

Now what have you gone and done?

Nothing too awful: **RE** *replaces* the complex number on Level 1 of the stack with only that number's REal component. **IM** does the same with a complex number's IMaginary component.

Here's a point to help you wrap your mind around the HP-28's idea of data objects:

Although complex numbers are a meaningful form of information and there are plenty of operations that use them as such, you're *not limited to using them just as complex numbers*.

You can also use complex numbers as two-dimensional vectors (which are very similar to complex numbers), or as coordinate pairs in plotting, or just as a convenient way to group two related numbers.

And because you can pull complex numbers apart and put them back together again, you can actually define new operations that give new meanings to the complex number *data type* in the HP-28 but which have no relationship to the mathematical concept of complex numbers in the real world.

As far as you should be concerned, then, complex numbers are just ordered pairs of real numbers – *information objects for you to use in whatever way you see fit*.

And just so you're fully confident of your skills with the mechanics of complex numbers, here's a "little something for the occasion..."

## Pop Quiz: Simple Questions About Complex Numbers

1. How would the HP-28 represent these complex numbers?

$$3 + 4i \quad 2.3 - 1.1i \quad -1 - i \quad \frac{7}{8} + \frac{4}{3}i$$

2. How would you key in the numbers in question 1?

3. Calculate the following:

$$\sqrt{-1} \quad \sqrt[3]{-7} \quad (1,1)^2$$

4. Change (1,2) into (2,1).

5. Calculate the following:

$$\frac{0.34(2 + 3i)(32.4 - 12.2i)}{33.42 - (12.2 + i\sqrt{2})}$$

## Simple Answers to Simple Questions About Complex Numbers

1.  $(3, 4);$   
 $(2.3, -1.1);$   
 $(-1, -1);$   
 $(.875, 1.333333333333).$

2.  $((3, 4) \text{ ENTER})$   
 $((2.3 \text{ SPACE } 1.1 \text{ CHS ENTER})$   
 $((1, 1) \text{ ENTER CHS})$   
 $(7 \text{ ENTER } 8 \div 4 \text{ ENTER } 3 \div \text{ CMPLX } R \leftrightarrow C)$

3.  $(1 \text{ CHS } \sqrt{\phantom{x}} = (0, 1)$   
 $(7 \text{ CHS ENTER } 3 \text{ } 1/x \text{ } ^ = (.956465591387, 1.65664699997)$   
 $((1, 1) \text{ } x^2 = (0, 2)$

4.  $((1, 2) \text{ ENTER } \text{ CMPLX } C \leftrightarrow R \text{ SWAP } R \leftrightarrow C)$

5.  $(.34 \text{ ENTER})$   
 $((2, 3) \text{ X})$   
 $((32.4, 12.2) \text{ CHS X})$   
 $(33.42 \text{ ENTER})$   
 $(12.2 \text{ ENTER } 2 \text{ } \sqrt{\phantom{x}} \text{ CMPLX } R \leftrightarrow C - \div)$

Result =  $(1.54011490435, 1.2690881897)$

# Vectors

Put most succinctly, a vector is an ordered list of numbers. It's ordered such that the left-most element is numbered 1, and the rest follow in ascending order.

You're probably familiar with two- and three-element vectors, but the HP-28 imposes no upper limit on the size of a vector. Vectors are represented by the HP-28 as a list of numbers, separated by delimiters and enclosed within *square* brackets ( [ and ] ).

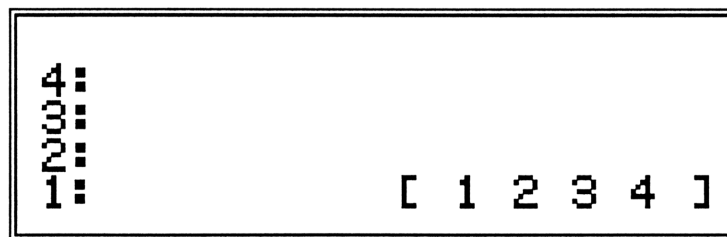
And here's a key point: The number elements of a vector may be real or complex (aha)! You can see already how to build a compound data type from simple data types *and other compound types*.

Better to start simply, though, so

---


**Watch This:** Press  CLEAR  [ 1 , 2 , 3 , 4 ENTER ].

You'll see:



```
4:
3:
2:
1:      [ 1 2 3 4 ]
```

Again, notice certain things:

1. You didn't need to press   $\boxed{1}$  at the end of the number. If the vector needs to be separated from other things following it in the command line, you must use  $\boxed{}$  as a delimiter. If not, the HP-28 will automatically add the right-hand square bracket (and these rules should sound quite familiar, since you just heard similar ones for complex number objects).
2. The commas you used as delimiters were replaced by spaces. Regardless of the delimiters you use to enter the vector, the HP-28 will use spaces to display the vector (also a familiar rule).
3. All elements of the vector are on the same stack level. Both this and the fact that they're grouped together inside square brackets tell you that this is a single object. From now on, unless you purposely break it into its components it will be treated as one object.

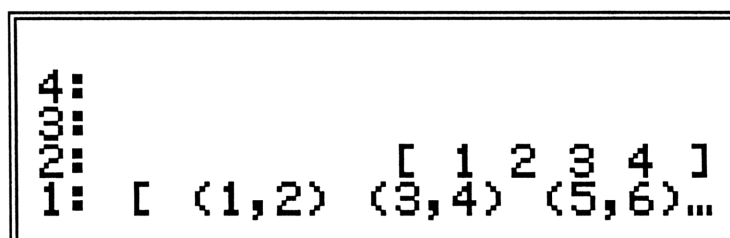


By definition, vectors may be combined via certain mathematical operations, but unlike complex numbers, only a few of these are in any way analogous to corresponding operations on real numbers (operations such as  $+$ ,  $-$ ,  $\times$ ,  $\div$ , SQ, ABS, NEG and INV).

For the most part, then, you have *vector-specific* operations, collected under the ARRAY (■A) menu. Why not a VECTOR menu? Because vectors are really examples of an object type called an *array*.

So an array is an ordered arrangement of objects, and a vector is just one particular arrangement – a one-row arrangement.

**Another Vector:** Press [ ] [ ( ] [ 1 ] [ , ] [ 2 ] [ ■ ] [ ( ] [ 3 ] [ , ] [ 4 ] [ ■ ] [ ( ] [ 5 ] [ , ] [ 6 ] [ ■ ] [ ( ] [ 7 ] [ , ] [ 8 ] [ ENTER ].



```

4:
3:
2:
1: [ (1,2) (3,4) (5,6) ... ]

```

The usual, little things to notice:

1. Since something follows each complex number in this vector (except the last one), you need to use the right-hand parenthesis to delimit each.
2. Neither the last complex number nor the vector needs the final parenthesis or bracket. Since it found both an opening bracket and parenthesis, the HP-28 knows it must close them at [ENTER].

In this case, the vector is longer than the display. As usual when something like this happens, the HP-28 will use ... to indicate the overrun.

To view the whole vector, you must do one of three things:

- (i) Edit Level 1 (with either EDIT or VISIT), using the cursor keys to scroll through the entire object. Then press **ATTN** to discard the command line so that you don't inadvertently change the vector;
- (ii) Decompose the object with **ARRAY→** from the **ARRAY** menu, and if necessary use **VIEW↑** and **VIEW↓** to examine the elements in the stack. Remember to rebuild the vector with **→ARRAY** when you are done;
- (iii) Use **GETI** to step through the vector components.

No sense exploring (i). You already know how to EDIT.

---

**Try Door Number (ii):** Press **ARRAY ARRAY→**. Here's what you get:



The object in Level 1 is something new called a *list*. You can tell that this is a different object type because it's represented within braces (`{` and `}`). A list is a different animal altogether, so you won't see it in all its glory until later. For right now, just learn to recognize it by its braces.

You can see that the numbers up farther in the stack are the former values within the vector you just dissected. But what's that bottom number inside the braces?

It's the number of elements the original vector contained: 4 (complex) numbers, in this case. `ARRY→` will always put this extra value on Level 1, so that you'll know how many stack levels contain vector elements *and* so that the machine will know this too (the array-building command, `→ARRY`, needs to know how many stack levels to use in making the new vector).

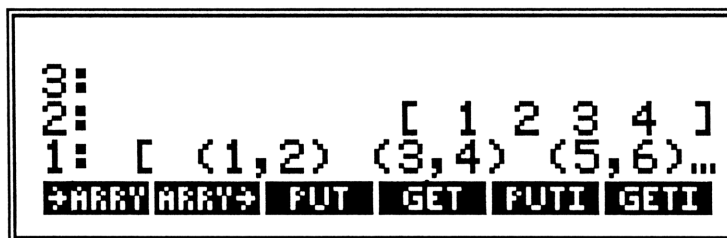
Now, use `VIEW↑` and `VIEW↓` to scan the stack.

The vector has been decomposed from left to right (like the complex number with `C→R`); the first element (the left-most) has been put on the stack first and the last vector element (the right-most) last.

Notice that you can modify any single vector element by using `VISIT` to edit that element right where it is.

---

**Rebuild:** Press **→ARRY**.



The calculator screen displays the following: Level 3 contains the number 3. Level 2 contains a list [ 1 2 3 4 ]. Level 1 contains a list [ (1,2) (3,4) (5,6) ... ]. The bottom of the screen shows the command sequence: →ARRY →ARRY→ PUT GET PUTI GETI.

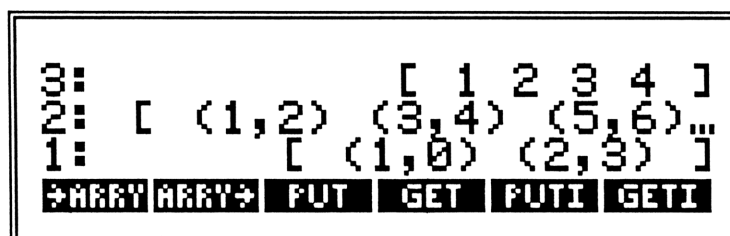
---

The vector has been recomposed from its pieces. The **→ARRY** operation used the **{ 4 }** to know how many stack elements to put into the vector.

Note that you could have filled the stack with real and/or complex numbers of your own, put the vector size in a list in Level 1 and used **→ARRY** to build an entirely different vector.

---

**Like This:** Press **1, ( 2, 3 ENTER { 2 →ARRY**.



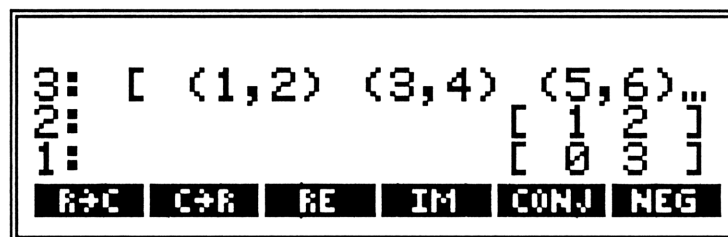
The calculator screen displays the following: Level 3 contains the number 3. Level 2 contains a list [ 1 2 3 4 ]. Level 1 contains a list [ (1,2) (3,4) (5,6) ... ]. The bottom of the screen shows the command sequence: →ARRY →ARRY→ PUT GET PUTI GETI.

What happened? First of all, the real value 1 was changed to the complex value  $\langle 1, 0 \rangle$ , which is its complex equivalent (remember that complex numbers with a zero for the imaginary component are mathematically identical to real numbers). The HP-28 likes to be consistent within a vector, so if any element of the vector is complex, all of the elements are made to be complex.

Secondly, notice that you didn't need to press **ENTER** before pressing **→ARRAY**. Most do-it-now operations will automatically evaluate the command line before they execute, thus saving you a step.

Since there is such a thing as a vector full of complex numbers, it would be reasonable to use the complex number operations on them, too. What happens if you do? To find out, press **◀PREV** to get to the previous page of the ARRAY menu.

Then Press: **↵R**.

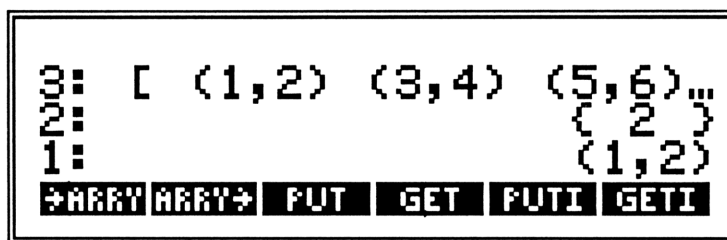


The complex vector has been decomposed into two real vectors, just as a complex number would be decomposed into two real numbers. In the same way also, the real components of the original complex vector are in the Level-2 vector, and the complex components are in the Level-1 vector.

As you might expect, R→C will recompose the original complex vector from the two real vectors in Levels 1 and 2.

---

**Now Try This:** Press **DROP** **DROP** **{** **1** **NEXT** **GETI**. What happened?



The screenshot shows the HP-28C calculator display with the following content:

```
3: [ (1,2) (3,4) (5,6) ...  
2: { 2 }  
1: (1,2)  
→ARRY ARRY→ PUT GET PUTI GETI
```

First, you brought that 4-element complex vector to Level 1.

Second, you keyed in a list containing the number 1. This list was created to be an *index* into the vector.

Third, you pressed **GETI** which told the HP-28 to GET the indexed component from the vector in Level 2, then Increment that index and push the selected component on the stack – in that order. Sure enough, component 1, **(1,2)** was pushed onto the stack and the index was incremented to **{ 2 }**.

---

**Encores:** Press **DROP** **GETI**. Now the second component, **(3,4)**, is pushed onto the stack, and the index is incremented to **{ 3 }**.

Press **DROP** **GETI**. The third component, **(5,6)**, is pushed onto the stack, after the index is incremented to **{ 4 }**.

Press **DROP** **GETI**. Now the fourth component, **(7,8)**, is pushed onto the stack, but since there are only 4 components in this vector, the index is cycled back to **{ 1 }**.

---

At this point, your display will look like this:

```
3: [ (1,2) (3,4) (5,6) ...  
2:                                     { 1 }  
1:                                     (7,8)  
→ARRY ARRY→ PUT GET PUTI GETI
```

You can see how easy it is to gain access to any element within a vector, using the GETI command. Remember that the I in GETI stands for Incrementing the index (not simply the fact that you're using an index) while GETting the indexed element.

---

**A Variation:** Press `DROP` `GET`. Here is the result:

```
3:  
2:  
1:                                     (1,2)  
→ARRY ARRY→ PUT GET PUTI GETI
```

Unlike GETI, GET *consumes* both the index and the vector and leaves only the indexed component on the stack. Therefore, you would probably use this form of GETting when you didn't care about keeping the vector in the stack. When you did want to keep the vector, you would make a copy of it (with DUP) prior to using GET.

By now, you've probably noticed that many commands come as matched sets of complementary functions, such as  $C \rightarrow R$ ,  $R \rightarrow C$ ,  $\rightarrow \text{ARRY}$ , and  $\text{ARRY} \rightarrow$ .

---

**Speaking Of Which:** Press `[DROP]` `[{]` `[1]` `[ENTER]` `[5]` `[ENTER]`. Here's what you'll see:

```
3:          [ 1 2 3 4 ]
2:          { 1 }
1:          5
←ARRY ARRY→ PUT GET PUTI GETI
```

Now try to guess what will happen if you press `PUTI`. Hint: `PUTI` is the complement to `GETI`. Therefore, it should put the number in Level 1 into the vector in Level 3 at the location specified by the index in Level 2. It should also increment the index.

---

**Yes, But Does It Really?** Press `PUTI`.

```
3:          [ 5 2 3 4 ]
2:          { 2 }
1:
←ARRY ARRY→ PUT GET PUTI GETI
```

Shur 'nuff: The machine put the **5** into the vector as element number 1 (the indexed element), and the index was incremented to 2.

---

**And Notice This:** Press `[6]` `PUT`. `PUT` is the complement of `GET`, functioning similarly to `PUTI`, except that it doesn't keep the index around.

---



## A Visit with Vectors

Like complex numbers, vectors are a meaningful form of information, and there are operations that use them as such. But you're not limited to using them solely as mathematical vectors (i.e. representations of "reality"). Keep in mind that because you can "dissect" vectors and reassemble them, you can define new operations that give new meanings to the vector data type – meanings that have no relationship to the concept of vectors in the real world. So before going on, be sure you understand vectors, how to build them, and how to take them apart.

1. Give three ways of putting the vector  $[(1,2) (3,4) (5,6)]$  into Level 1.
2. If  $\mathbf{A} = [1 \ 2 \ 3]$ ,  $\mathbf{B} = [-3 \ 2.5 \ -.2]$  and  $\mathbf{C} = [\frac{1}{3} \ \sqrt{2} \ -6]$  what is  $14.5\mathbf{A} - 0.2\mathbf{B} + (1,1)\mathbf{C}$ ?
3. Sum the real and imaginary components of the result of the second problem. I.e., split that vector into two real-valued vectors and sum them.
4. Using the result of problem 3, find the corresponding unit vector ( $\mathbf{u} = \frac{\mathbf{v}}{|\mathbf{v}|}$ ).
5. For the following table, find the total hours worked per person and overall.

	Abe	Ben	Carl	Dan
Mon	8.00	7.75	4.50	6.40
Tue	7.50	8.25	5.50	7.40
Wed	3.50	6.50	4.75	7.10
Thu	8.00	7.50	4.00	7.50
Fri	8.10	8.00	4.50	7.25

## Results of A Visit With Vectors

1.  $\left[ \left( \begin{matrix} 1 \\ 2 \end{matrix} \right) \left( \begin{matrix} 3 \\ 4 \end{matrix} \right) \left( \begin{matrix} 5 \\ 6 \end{matrix} \right) \right]$  ENTER;  
 $\left( \begin{matrix} 1 \\ 2 \end{matrix} \right)$  ENTER  $\left( \begin{matrix} 3 \\ 4 \end{matrix} \right)$  ENTER  $\left( \begin{matrix} 5 \\ 6 \end{matrix} \right)$  ENTER { 3 **ARRAY** **→ARRY** ;  
 $\left[ \begin{matrix} 1 \\ 3 \\ 5 \end{matrix} \right]$  ENTER  $\left[ \begin{matrix} 2 \\ 4 \\ 6 \end{matrix} \right]$  **ARRAY** **PREV** **R↔C**.

2.  $14 \cdot 5$  ENTER  
 $\left[ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} \right]$  **×**  
 $\left[ \text{CHS} \begin{matrix} 3 \\ 2 \end{matrix} \cdot 5 \cdot 2 \text{CHS} \right]$  ENTER  
 $\cdot 2$  **×** **-**  
 $3$  **1/X**  
 $2$  **√**  
 $6$  **CHS** ENTER  
 $\{ 3$  **ARRAY** **→ARRY**  
 $\left( \begin{matrix} 1 \\ 1 \end{matrix} \right)$  **×** **+**

Result =  $\left[ \begin{matrix} (15.4333333333, .333333333333) \\ (29.9142135624, 1.41421356237) \\ (37.54, -6) \end{matrix} \right]$

3. **ARRAY** **PREV** **C↔R** **+**;

Result =  $\left[ \begin{matrix} 15.7666666666 & 31.3284271248 & 31.54 \end{matrix} \right]$

4. **[ENTER] [A] [B] [S] [ENTER] [1/X] [X];**

Result =

**[ .334265455504 .664186741847 .668672249469 ]**

Note that since the HP-28 won't allow you to divide a vector by a scalar (a real number), you must invert the magnitude (via **[1/X]**) and multiply.

5. **[2] ['] [F] [I] [X] [ENTER]**

(Mon) **[1] [8] ['] [7] [.] [7] [5] ['] [4] [.] [5] ['] [6] [.] [4] [ENTER]**

(Tue) **[1] [7] [.] [5] ['] [8] [.] [2] [5] ['] [5] [.] [5] ['] [7] [.] [4] ['] [ENTER]**

(Wed) **[1] [3] [.] [5] ['] [6] [.] [5] ['] [4] [.] [7] [5] ['] [7] [.] [1] ['] [ENTER]**

(Thu) **[1] [8] ['] [7] [.] [5] ['] [4] ['] [7] [.] [5] ['] [ENTER]**

(Fri) **[1] [8] [.] [1] ['] [8] ['] [4] [.] [5] ['] [7] [.] [2] [5] ['] [ENTER]**

Totals = **[ 35.10 38.00 23.25 35.65 ]**

**[ ] [ARRAY] [NEXT] [NEXT] [CNR] or [ ] [ARRAY] [NEXT] [DROP] ['] ['] [']**

Sum total = **132.00** hrs.

## Arrays

You've already heard the word "array." In fact, you know that a vector is one form of an array. Mathematically, an array is nothing more than an ordered arrangement of numbers in rows and columns, and a vector is merely a one-dimensional array.

So in general, you can think of an array as a list of vectors. The vectors form the rows, and corresponding elements of the vectors form the columns. The lengths of the component vectors must therefore be the same, but the number of rows (vectors) does not need to be the same as the number of columns (elements per vector).

On the HP-28, an array is represented by bracketing a list of vectors with square brackets ( `[` and `]` ). Thus, you'll see a double set of square brackets, since vectors themselves use a set, too.

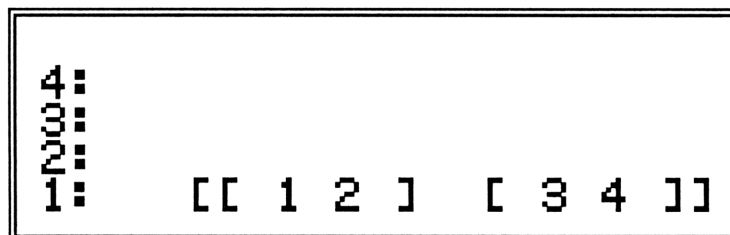
But here's a less obvious point: The HP-28 allows you to deal with arrays only as a two-dimensional list of numbers (i.e. not as a "vector of vectors"), which of course conforms with the normal, mathematical notions of matrices. In other words, it may seem logical to be able to extract entire rows (vectors) from an array and horse around with them, just as you can with the components of complex numbers and vectors. There are no conceptual reasons why you can't do this; there just aren't any built-in commands to let you do it. You can decompose arrays only on an element basis, not on a row (vector) basis.

---

**An Example:** Press **CLEAR** **MODE** **PREV** **-ML** **↔**.

Then press **[** **[** **1** **,** **2** **[** **3** **,** **4** **ENTER**.

Here's what you'll see:



The calculator display shows a 2x2 matrix. The first column contains the row indices 4, 3, 2, and 1 from top to bottom. The second column contains the matrix elements. The first row has a single element 1. The second row has two elements, 1 and 2. The third row has two elements, 3 and 4. The fourth row is empty. The display is as follows:

4:	
3:	
2:	
1:	[ [ 1 2 ] [ 3 4 ] ]

---

Things To Notice That Shouldn't Be Very Surprising Any More:

Notice that you didn't need to delimit the first vector with **]** before starting the second vector with **[**.

Since the HP-28 is expecting the second vector (you told it that there were to be more than one when you typed **[****[**), it doesn't need the closing bracket. The new starting bracket simply tells it that you're finished with the first row/vector (and therefore tells it how many elements are in each row).

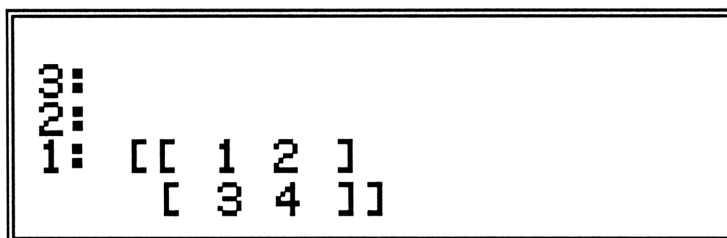
Nothing very new, right? The HP-28 has dealt with the array in much the same fashion in which it deals with vectors. This is in fact the case with almost all of its other operations as well. You really wouldn't want it any other way.

One thing you did that was new and purely optional was to enter the MODE menu and select -ML. What you did was to turn Multi-Line mode off.

The best way to see what this mode does is to look at what the opposite mode does.

---

**Compare:** Press **MODE** **PREV** **+ML** **↔**.



The calculator display shows a 2x2 array of real numbers in multi-line mode. The display is divided into four rows. The first row shows '3:'. The second row shows '2:'. The third row shows '1:'. The fourth row shows '[[ 1 2 ]' followed by '[ 3 4 ]]' on the next line. The array is displayed as follows:

```
3:
2:
1:  [[ 1 2 ]
    [ 3 4 ]]
```

---

See the difference? Multi-line mode reformats Level 1 so that as many lines of the object can be seen as possible.

Notice that the object is still on the same level; it has not been decomposed. In the case of an array, each display line will contain one row (vector) and may trail off the right-hand side of the display. If there are more rows than can be seen in the display at once, you can use **VIEW↑** and **VIEW↓** to scroll the display and see all of the rows.

Now press **↔** **-ML** to get back to single-line mode. You still have a two-by-two array of real numbers here. It's just shown as being all on one line.

Now, the only major mechanical difference between working with vectors and arrays on the HP-28 is in the index – that list denoted with braces – associated with →ARRY, ARRY→, GET, GETI, PUT and PUTI.

Because of its two-dimensionality, the size of an array is represented as a *pair* of numbers within braces. The first number of the pair is the number of rows (vectors) while the second is the number of columns (elements per vector).

---

**Take It Apart:** Press **▢** **ARRAY** **▢** **ARRY→**.



---

Use **▢** **VIEW↑** and **▢** **VIEW↑** to look over the stack. The array has been decomposed into its component numbers, with the elements having been pushed onto the stack in "row-major order." That is, they were pushed on starting with the first row, proceeding from left to right until that row was exhausted. Then the second row was taken left-to-right, and so forth.

The list on Level 1 contains 2 numbers specifying two rows and two columns, reminding you (and the HP-28) where those numbers above actually originated. As with all decomposing commands, the stack is left in a state that allows you to immediately recompose.

---

**Do It:** Press **▢** **ARRY→**. The array is recomposed, using that index list on Level 1 as the blueprint for rebuilding the array.

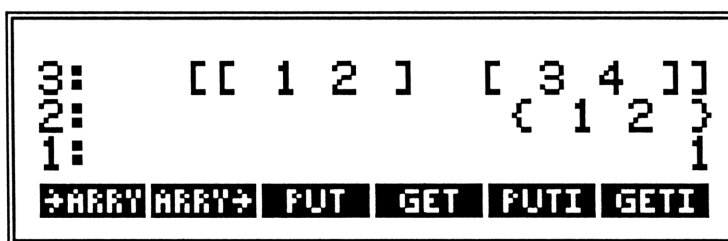
---

Again, remind yourself that you're working in the same **ARRAY** menu as you did with vectors. How, then, do these same commands work on two-dimensional objects?

---

**Find Out:** Press `{ 1 , 1 ENTER`. This is the array's index, pointing to the first row, first column.

Now press `GETI`. Here's the result:



```
3:      [[ 1 2 ] [ 3 4 ]]  
2:      { 1 2 }  
1:      1  
->ARRY >ARRY+ PUT GET PUTI GETI
```

---

This works exactly analogously to the way it does with vectors, with an allowance for the second dimension. Indeed, the only new thing to notice is how the index is incremented: `PUTI` and `GETI` increment the index in row-major order.

Do it a few more times to make sure you see the pattern. Press `DROP GETI`. The index is now `{ 2 1 }`.

Press `DROP GETI`. The index is now `{ 2 2 }`.

Press `DROP GETI`. The index cycles back to `{ 1 1 }`.

Care to speculate on how `GET` and `PUT` work? Go ahead – experiment with them.



## Array Aptitude Test

1. Convert the vector  $[1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$  (a  $1 \times 9$  array) into a  $3 \times 3$  array.
2. Convert the array result from problem 1 into a complex array such that all of the imaginary components are 0. Are there different ways to do this?
3. Given  $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 9 \end{bmatrix}$ , double  $\mathbf{A}$  and subtract 1 from every element (i.e., find  $2\mathbf{A}-1$ ).
4. Given  $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  and  $\mathbf{B} = \begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix}$  show that  $\mathbf{A} \times \mathbf{B} \neq \mathbf{B} \times \mathbf{A}$  (thus proving that matrix multiplication is not commutative).
5. How could you extract individual rows (single vectors) from a  $3 \times 3$  array?

## A.A.T. Results

1. **ARRAY**

[1,2,3,4,5,6,7,8,9]ENTER

**ARRY** DROP {3,3} **ARRY**.

2. Method 1: ENTER **ARRAY** NEXT **SIZE** 0 **CON** NEXT NEXT **R÷C** ;

Find the **SIZE** of the array and use it to create a **CON**stant array of the same size that is filled with 0's.

Method 2: ([1,0]X)

3. [[1,2,3][4,5,9]ENTER

2X ENTER

**SIZE**ENTER (Find the size of the array.)

1,C**ON**ENTER (Create a **CON**stant array of the same size filled with 1's.)

—

Result = [[ 1 3 5 ] [ 7 9 17 ]]

4. [[1,2][3,4]ENTER

[4,3][2,1]ENTER

**DUP**2ENTER X

**SWAP** **ROT**ENTER X (SWAP ROT reverses Levels 1, 2 and 3. See page 96.)

**=****=**ENTER (== tests Levels 1 and 2 for equality. If they are equal, the test will produce a 1. If not, it produces a 0.)

Result = 0. The values at Levels 1 and 2 were not the same.

5. Take the array  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$  as an example. The strategy is to create the  $3 \times 3$  array  $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$  and multiply this by the first array.

Thus:

[ARRAY] [NEXT] [ { [3] ['] [3] [ENTER] [0] ] [CON] (a  $3 \times 3$  array of 0)  
 [ { [1] ['] [1] [ENTER] [1] ] [PREV] [FUT] (put 1 at element {1,1})  
 [ [ [1] ['] [2] ['] [3] [ [4] ['] [5] ['] [6] [ [7] ['] [8] ['] [9] ] ] [X]  
 [ { [3] ] [NEXT] [DIM] (redimension the array to be a vector)

Using  $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$  will give the second row, and  
 $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$  will give the third.

Be sure to keep in mind that the *order* of the two arrays is important in multiplication!

You've now seen three different objects you can construct with real numbers as your basic building blocks. Before building anything else, see if you can put all these different real-number constructions into some perspective. How do they all relate to one another?

- Each of these information objects is built from real numbers, which are one of the three fundamental information types (characters and binary digits are the other two).
- Throughout these last 30 pages, you've seen that innocent-looking little word, "list." The ordered, indexed list is really the key when it comes to thinking about how the HP-28 associates information. In order of increasingly sophisticated lists, you can think of the real-number-based objects like this:
  - A complex number is a 2-element list of real numbers;
  - A vector is an "n"-element list of either real or complex numbers. Therefore, it can actually be a list of other lists (complex numbers).
  - An array is an "n"-element list of vectors. Therefore (in the case of a complex array), it can actually be a list of lists of lists.

Lists of "things." That's the compound-information object "concept" in its pure form – the truly consistent, generalized way to think about these objects.

However, as you've already seen, you don't have exactly the same set of "list arithmetic" and "list decomposing tools" available for each of these objects. Clearly, the HP-28's command set has been tailored toward the real-life math "meanings" of each of the objects. For example, it's true that an array is nothing more than a list of vectors, but the HP-28 won't decompose it into its component vectors for you – probably because this isn't a commonly needed application.

## Characters

Characters are another sort of simple information that you use every day without thinking much about it; you're using them right now as you read this book. They are so simple that they convey very little information by themselves. But in mathematics, if you associate a certain character, say  $X$ , with a value or operation, it gains information value. Notice that this value is not intrinsic; you have given it this value *by association*.

Characters also gain in information value when they are used to make *words*. The characters on this page are only meaningful because of their association with other characters to make words. Thus they attain a higher level of information.

The words on the page in turn gain meaning by being associated with other words in *sentences*. The process goes on through all of the sentences in each *paragraph*, all the paragraphs on a *page*, to all the pages in this *book*. And it doesn't stop there. The book is only meaningful in the context of your HP-28, and your HP-28 is only meaningful in the context of what you want to do with it.

Although its ability to gain meaning from higher and higher levels of associated characters is far more limited than yours, the HP-28 can indeed go a few steps up the ladder. However, it actually has *no* facility to deal with characters simply as characters – only with characters as members of larger information types.

For example, you cannot place a single character on the stack. That is, there is no data object type called a "character." This is different than with a real number, which may, of course, appear on the stack as itself – in its elementary building-block form. Not so with characters. They must always appear within a compound data object.

## Character Strings

A *string* is simply a *list* of characters, displayed within quotation marks ("). And although you can't have a single character on the stack, you *can* have a string of one character(!).

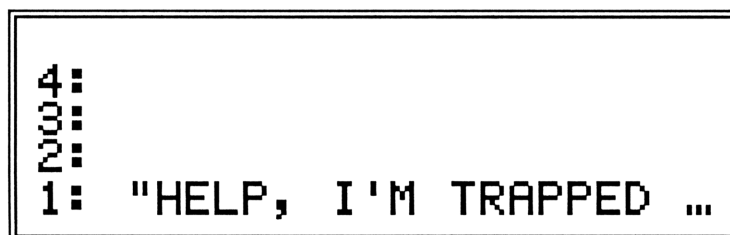
If this distinction sounds a little strange, don't worry. In practice, you'll find it to be irrelevant (i.e. you can "mess with" a 1-character string just as if you're "messaging with" a single character), but you'll see it to be logically true. In fact, the logic is consistent even to the point of allowing strings which contain *no* characters ("empty" strings).

Strings may be arbitrarily long and may contain *any* character.

---

**Build One:** Press    H E L P ,  I ' M   
T R A P P E D  I N  H E R E .

Here's your string at Level 1 on the stack:



```
4:  
3:  
2:  
1: "HELP, I'M TRAPPED ...
```

## The Notice-These-Things-Drill:

1. As always, you don't need to key in the final delimiter. The HP-28 closes the expression for you.
2. All characters except " – including those that usually act as delimiters in other objects – are included in the string.
3. All elements of the string are on the same stack level. Both this and the fact that they are grouped together inside double quotation marks tells you that this is a single object.
4. If a string is too long, it will run off the right hand side of the display. As always, the HP-28 indicates this with an ellipsis. The only (convenient) way to view the entire string is to EDIT it and scroll from end to end using the cursor keys.

Strings are, of course, information objects and can therefore be placed on the stack and manipulated with stack commands. But you can't use them to do math since math isn't defined for such objects.

You can, however, use one command that is normally associated with math. Since the concept of adding two strings together (appending one to another) is similar to the concept of adding two numbers, the + command effectively adds two strings.

Other than +, though, you'll need to rely mainly on the string-specific commands that you'll find in the STRING (F/D) menu.

---

**Explore:** Press F STRING NEXT 1 ' 1 7 SUB 3 3 NEXT CHR +.



The image shows a calculator screen with a stack of three items. The top item is '3:', the middle is '2:', and the bottom is '1: "HELP, I'M TRAPPED!"'. Below the stack, a menu of functions is displayed: →STR, STR→, CHR, NUM, POS, and DISP.



Here's what you did:

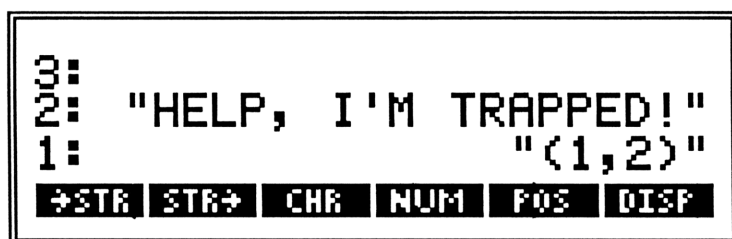
1. You selected the second level of the STRING menu.
2. You entered two parameters for the SUBstring operation. The first number indicates the first and the second number the last character of the original string that you want to keep.
3. You invoked the SUBstring command. The result was the string **"HELP, I'M TRAPPED"** in Level 1 of the stack.
4. You keyed-in the argument for the CHaRacter operation. You may have noticed that not all of the characters you might expect to be available are on the HP-28 keyboards (though there are some you probably didn't expect). In the HP-28 Reference Manual in the section on strings there is a table of characters and character codes. The CHR operation allows you to convert a real number (dang, those things just keep popping up) into a one-character string. For example, 33 is the character code for **!**.
5. You moved back to the first level of the STRING menu (there are only two) and performed the character code conversion. **"!"** was left in stack Level 1.
6. Finally, you added (appended) the string in Level 1 onto the end of the string in Level 2 using **[+]**. The result was left in Level 1.

Since you can't have individual characters on the stack (only one-character strings), there's no convenient way to change a string into characters (sure, you *could* do some tricks with SUB, using several copies of the original string, but total string decomposition isn't generally very useful).

There is, on the other hand, a very powerful method of changing a string into other, very useful things. Strings may be converted to and from *any* information object by using →STR (convert to STRing) and STR→ (convert from STRing).

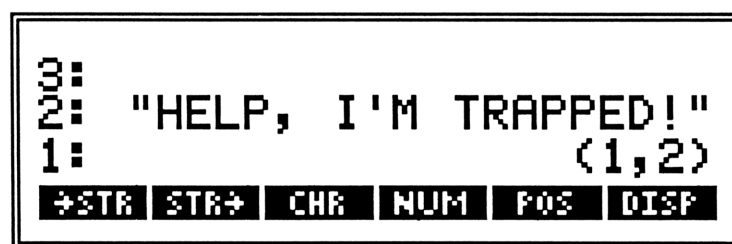
---

**For Example:** Press (1,2)ENTER →STR.



The calculator display shows a stack with three levels. Level 3 contains the number 3, level 2 contains the string "HELP, I'M TRAPPED!", and level 1 contains the string "(1,2)". Below the stack, a menu of functions is displayed: →STR, STR→, CHR, NUM, POS, and DISP.

Now press STR→.



The calculator display shows the same stack as before, but the function menu has changed. The menu now displays: →STR, STR→, CHR, NUM, POS, and DISP. The string "(1,2)" on level 1 of the stack remains unchanged.

---

These two objects on Level 1 don't look much different, but they are. "(1,2)" is a *string*, not a complex number. The HP-28 (and you) can tell this by those quotation marks. (1,2), on the other hand, *is* a complex number and the two objects are used in radically different ways.

---

**Try This:** Press **⏏** (Notice that the  $\alpha$  annunciator comes on. Remember what that means?) **1****SPACE****2****SPACE****+** **ENTER**.

```
3: "HELP, I'M TRAPPED!"
2:                               (1,2)
1:                               "1 2 +"
+STR STR+ CHR NUM POS DISP
```

Press **STR+**.

```
3: "HELP, I'M TRAPPED!"
2:                               (1,2)
1:                               3
+STR STR+ CHR NUM POS DISP
```

---

What happened? **STR→** recognized that the string contained three different objects, and in converting the string into those objects, it posted them.

It read the string from left to right and first found the **1** (delimited by a space). It converted that character into a real number and pushed it onto the stack. It then kept reading and found the **2**. It converted that to a real number and pushed it onto the stack. Finally, it found the **+**, recognized it as the name of a command, and performed it. The first two stack items were therefore added together, with the result landing at Level 1, as usual. **STR→** then reached the end of the string and stopped reading.

The important point: This is *exactly* how the command line would have responded if it had contained those characters when **ENTER** was pressed.

So you see that strings may actually form "pretyped" command lines that you can then post in their command form by converting out of string notation.

However, you probably won't use strings for this purpose nearly as much as you will for other information. In fact, strings may quite possibly be the most information-packed data objects available to you, because they allow the HP-28 to communicate with you in English (or whatever language you prefer). This type of information is probably how you'll encounter and use strings most often.

Test your understanding of them now....

## Character String Query

1. Given that Level 1 contains the number **100.01**, use it to build the string **"Vol.= 100.01 gal."**
2. How would you go about pulling the number back out of this string? Assume that you don't actually know what that number is.
3. Taking the number **6.022E23** from Level 1, format it within a string so that it looks like this: **"6.022 \* 10^(23)"**.  
(Hint: use MANT and XPON from the REAL number menu.)
4. Starting the result of problem 3, what would you expect the result to be if you invoked STR→? Why? Rewrite the string so that STR→ gives you back the original real number.
5. Change the string **"You understand?"** to **"You understand!"**

## C.S.Q. Answers

1. S T D ENTER " V LC O L • = SPACE ENTER SWAP STRING +STR  
" SPACE LC G A L • ENTER ++

Recall why you use  $\oplus$  here – to concatenate (join) strings.

2. ENTER ENTER " SPACE ENTER POS 1 + SWAP NEXT SIZE SUB  
ENTER " SPACE ENTER NEXT POS 1 SWAP NEXT SUB NEXT STR+

Notice how you find the spaces on either side of the embedded number by using the POS function. You then use the position of the space as one of two indices you need to extract a SUBstring.

3. ENTER REAL NEXT MANT SWAP XPON SWAP STRING +STR  
" SPACE X SPACE 1 0 ^ ( ENTER + SWAP +STR + " ) ENTER +

Same idea as in problem 1, really – except for the use of MANT and XPON.

4. The result is a **Syntax Error** because the expression is not in postfix form. If you want a string that breaks the original number up into mantissa and exponent but will still evaluate back to the original number, then use "6.022 10 23 ^ \*"

5. ENTER STRING NEXT SIZE 1 - 1 SWAP SUB 3 3 NEXT CHR +

CHR takes an integer and returns the corresponding character (as a string).

## Names

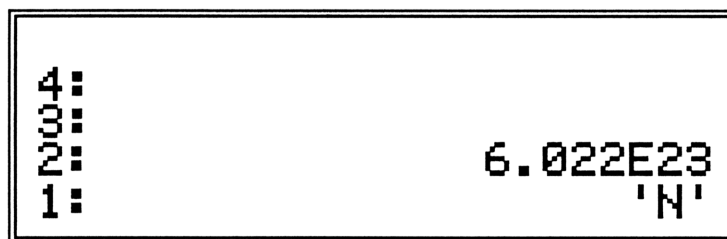
Names are also character strings, but they have special restrictions and a special purpose in life. They're represented by bracketing lists of characters with *single* quotation marks ( ' ).

A name is a descriptive word used to describe an object in the HP-28. In other words, if you have an object – any object – you can give it a name and thereafter refer to the object by that name. This name will be associated with the object until you change the association, which you may do at any time.

You can, of course, elect not to name objects (no object you've seen so far in this book has had a name associated with it). And as a matter of fact, you may also have a name that has no object associated with it.

---

**For Example:** Clear the stack, clear the menu line, and load  $6.022 \times 10^{23}$  and 'N' onto the stack. After doing so, here's how things look:



---

And what have you done? First, of course, you cleared the stack and the menu line. Then you put the real number  $6.022 \times 10^{23}$  onto the stack.

Then you put the *name* 'N' on the stack: 'N'ENTER. You can tell the HP-28 regards this as a name because of the single quotation marks.

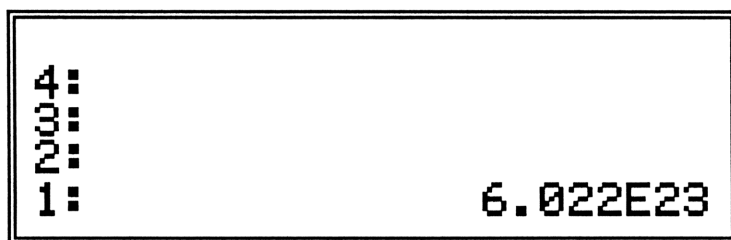
That 'N' on the stack isn't yet associated with any object. Although it need never be so linked to an object, it's often useful to do so, and you can link it to any other object (even another name) by using the STO (STOre) command.

---

**Go For It:** Press **[STO]**.

Both the name and the number are removed from the stack, right?  
What happened to them?

To find out, press **[N][ENTER]**. Here's what happens:



4:	
3:	
2:	
1:	6.022E23

---

You can see that when you put a name associated with an object onto the stack *without* single quotation marks, this tells the HP-28 to "evaluate" the name, thus replacing it with the object itself (and you're going to appreciate this more and more as time goes on).



---

**Do It Differently:**

Press **[']** **[N]** **[ENTER]**.

4:	
3:	
2:	6.022E23
1:	'N'

When you use **[']**, you're telling the HP-28 that you *do* want just the name on the stack. So there you have it.

---

**Now Change Your Mind:** You've decided you wanted the object after all – not just its associated name? No problem. Just press **[EVAL]**.

4:	
3:	
2:	6.022E23
1:	6.022E23

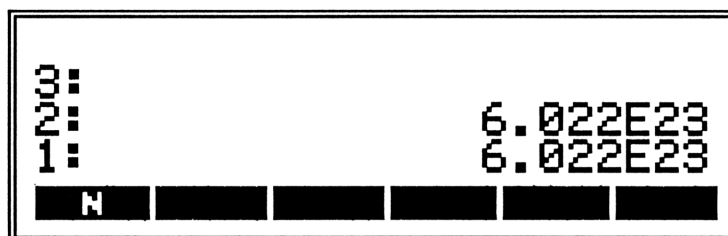
Can you make an educated guess as to what **[EVAL]** does?

It EVALuates the name in Level 1, thereby replacing it with its object.

But typing in an object's name isn't the only way to put the object on the stack.

---

**Watch:** Press **USER**. Assuming that you don't have any other named objects in your calculator yet, you'll see:



---

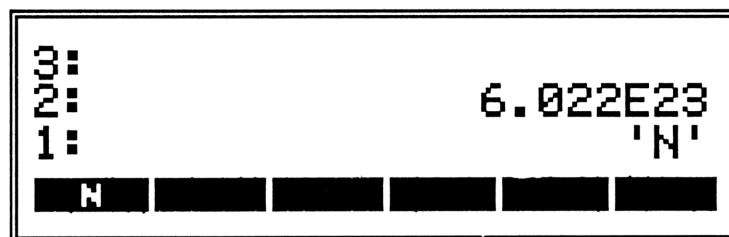
The USER menu is your own personal menu. *You* are the user. Whenever you **STO**re an object in a name (i.e., associate a name with an object) that name will appear in your user menu.

---

**Use Your USER:** Press **CLEAR** **N**. Pressing the menu key is a quick way to get the named object to be put onto the stack.

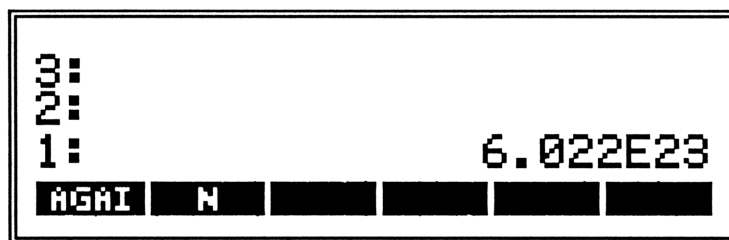
Now press **'** **N** **ENTER** (preceding the menu key with **'** allows you to use the name rather than the object it represents).

Your display should now look like this:



---

Next: Press **[ ' ]** **A** **G** **A** **I** **N** **[ ]** **S** **T** **O** **[ ]** .



---

You have now stored the name 'N' in the name 'AGAIN'! This is an example of using one name to refer to another, which is quite "legal," of course, since a name is just an object like all the others.

And notice that **AGAIN** has been added to the USER menu. The first four or five characters in any name are all the menu has room for (not all of the characters are of the same width).

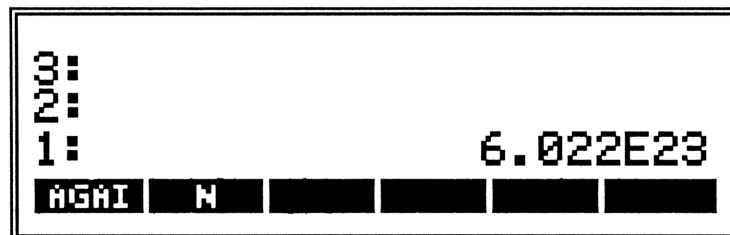
But the name itself has not been shortened.

Prove this by pressing **[ ' ]** **AGAIN**. The complete name is loaded into the command line. Now press **[ ]** **A** **T** **T** **N** to clear it.

But what is **AGAIN**'s "value?" Is it **'N'** or is it **6.022E23**?

---

**Here Goes Nothing:** Press **CLEAR** **AGAIN**.







There you have it. Whenever you evaluate a "name of a name of a name...etc.," the HP-28 continues to follow its nose, evaluating each named object until it encounters one that's not a name.

In this case, **AGAIN** points to **N**, and **N** points to **6.022E23**. Evaluating **AGAIN** causes **N** to be evaluated. And since the object that **N** points to is not a name, the HP-28 stops evaluating and places that object on the stack.

This chain of evaluations has a potential hazard. If you were to store, say, **'A'** in **'B'** and **'B'** in **'A'** and then try to evaluate either name, the HP-28 would go into an infinite loop. **A** would point to **B**, and **B** would point back to **A**, and so on, forever. In fact, the only way to stop this would be to press **ATTN** and **▲** at the same time (a special command just for such emergencies).

Another caution: Not all characters can be used in names. For obvious reasons, you can't use delimiters (**#**, **[**, **]**, **"**, **'**, **{**, **}**, **(**, **)**, **&**, **\***, **,**, **SPACE**, **NEWLINE**). You are allowed to use numerals, but not as first characters. And it's a definite no-no to use symbols or names which are already commands in the HP-28 (**+**, **-**, **\***, **/**, **^**, **√**, **=**, **<**, **>**, **≤**, **≥**, **≠**, **÷**, **DROP**, **CHS**, etc.). If you were allowed to do this, then you would really confuse your calculator!

## Name Games

1.  **PURGE** will dissociate a name and a value. Use  **PURGE** to rename 'N'.
2. The HP-28 has no *data registers*, unlike many calculators. Data registers are usually a numbered series of slots for storing real numbers. Create a named data object that looks like data registers. How would you store a number into it? How would you recall a number?
3. Store 4 in "register" 1 of the object you created in problem 2. Evaluate the named object so that it's in Level 1 of the stack. Store 3 in "register" 1 of the named object. Evaluate the named object so that it's in Level 1 of the stack. Why are Levels 1 and 2 different?
4. Purge the name N. Press  **"N** **ENTER**. Before trying it to find out, try to guess what the result of STR→ would be on this string. Type **S** **T** **R**  **→** **ENTER**. What will →STR do to this object?

## Name Game Winners

1. `N` `ENTER` `'N` `PURGE` `'Q` `STO`

The first entry of N is its value; the second is its name. You dissociate the two and then, with the value remaining on the stack, you associate it with 'Q'.

2. `{` `1` `0` `ENTER` `0` `ENTER` `C` `O` `N` `ENTER` `'M` `E` `M` `O` `R` `Y` `STO`  
 Store : `'M` `E` `M` `O` `R` `Y` `ENTER` `{` `1` `ENTER` `1` `'` `P` `U` `T` `ENTER`  
 Recall : `'M` `E` `M` `O` `R` `Y` `ENTER` `{` `1` `ENTER` `G` `E` `T` `ENTER`

You have created a list of ten "registers" which you may then store into and recall out of by using the list commands PUT and GET.

3. `'M` `E` `M` `O` `R` `Y` `ENTER` `ENTER` `ENTER` `{` `1` `ENTER` `4` `ENTER` `P` `U` `T` `ENTER` `EVAL`  
`P` `U` `R` `G` `E` `ENTER` `{` `1` `ENTER` `3` `ENTER` `P` `U` `T` `ENTER` `EVAL`

The point here is that the value on the stack is *not* what the name is referring to. Once you have evaluated the name, the object left on the stack is effectively a *copy* of what was referred to by the name. Changing the value on the stack will not change the named object and vice versa.

4. Since performing STR→ on a string is identical to keying in the contents of the string into the command line, 'N' is left on the stack as a name that has no object to point to. This will happen to any valid name the HP-28 doesn't already recognize as the identifier of another object. →STR will make the following string out of the name: "'N'"

So that's what you can build from characters. Now summarize for yourself the differences and relationships between these objects:

Although a character is a fundamental information type, it is *not* recognized as an object type on the HP-28; therefore, you can't place a character on the stack or manipulate it in any way. You need to build a compound object from one or more characters.

The main object to build is the character *string*, which is just a list of characters within quotation marks ("). Such a string may be broken down into smaller strings – even a string with one (or zero) characters. But the object is still a string. You can, however, convert a string into another object (and back again).

A specialized form of string is the *name*, which is denoted by single quotation marks ('). The main purpose of a name is to associate itself with another information object (even another name), thus giving you an easy way to refer to large or complicated objects as you manipulate them.

Your repertoire is growing:

- You can use **characters** to build **strings** and **names**;
- You can use **real numbers** to build **complex numbers**, **vectors**, and **arrays**.

It's time now to look at what you can do with the third fundamental information type – **bits**.

## Bits

You can think of bits (Binary digits) as being 1's and 0's, true and false, on and off, or any other pair of mutually exclusive states.

As such, they're used to indicate that some thing or state is either there (valid) or not there (invalid). As with characters and real numbers, bits gain meaning only within the context of their use. And since they are the simplest possible kind of information, they have almost no useful meaning unto themselves.

The HP-28 can use bits individually as *flags*. The word flag is computerese for a value that indicates the current state of something else. When that something only has two possible states, the flag can be a bit.

Many of the HP-28 flags signal certain system states of the machine. For example, there's a flag (28) that it examines whenever it needs to remember whether to use the ■ or the ▀ as the radix (as you'll recall from page 61, you do have this choice). But there's also a generous supply of flags that have no intrinsic meaning to the system – flags that you can therefore define for your own purposes.

As with characters, the HP-28 has no facility to deal with bits as bits on the stack – only with bits as members of larger information types. You can't place a single bit on the stack (i.e. the command TYPE, which tells you the type of object currently at Level 1 has no provision for a bit type). But – also as with characters – this is no serious limitation, since a larger data type may contain a single bit as its only member, and bit oriented operations will deal with this as if it were an elemental bit.



# Binary Integers

A binary integer is a *list* of bits. On the HP-28, the list may be from 1 to 64 bits long. The length of the list is called its *word size*.

You have several choices for the display and entry of a binary integer. You can use either binary (0 or 1), octal (0 - 7), decimal (0 - 9) or hexadecimal (0 - F) digits. A binary integer is entered and displayed preceded by **#**.

---

**Like So:** Press **CLEAR** **BINARY** (**B**) **BIN** **6** **4** **STWS** **#** **1** **1** **0** **1** **ENTER**.



---

You cleared the stack, selected the BINARY menu (because that's where most of the binary operations can be found), then selected the BINARY representation of binary integers (1's and 0's). Then you used STWS (SeT Word Size) to set the size of binary integers to 64 bits and put the binary integer **# 1101** onto the stack.

Notice that you keyed in the binary integer using only zeros and ones (binary digits). Had you tried to use any other digits when keying in the number, the command line would have caught your error, at which point you would be obliged to either re-enter the number using only ones and zeros or select the digit entry mode (DEC, OCT, or HEX) compatible with the digits you keyed in.

Notice also that, although the word size is 64 bits, you only see four bits. All leading zeros are omitted.

---

Now Press: **DEC** **#** **2** **2** **9** **ENTER**. What you see:




Upon pressing **DEC**, the HP-28 understands that DECimal digits are required when keying-in binary integers. Not only that, it also changes the representation of *all* binary integers to use decimal digits (notice Level 2).

Now press **OCT** and **HEX** and notice how the display changes.

---

Like characters, literal bits cannot live on the stack, so there's no convenient method for breaking a binary integer into its component bits. But binary integers *are* information objects; they can be placed on the stack and manipulated with stack commands. You can use them to do a smattering of math (limited to +, -, x and ÷).

In fact, since a binary integer and a real number are both fundamentally numbers, the HP-28 will allow you to mix them within this same restricted set of math operations. The conversion is performed "on the fly," with the result always being a binary integer (any fractional portion of the real number is lost).

There are also several binary-number specific commands in the BINARY and PROGRAM TEST (  ) menus. If you're interested, by all means explore those menus. For now, however, this is enough of an introduction to binary numbers.

## Binary Integer Test

1. What are the binary (base 2), octal (base 8), and hexadecimal (base 16) representations of the decimal (base 10) number 1000 (a.k.a.  $1000_{10}$ )?
2. What is  $2_{10} \times FF_{16}$ ?
3. Calculate  $2 \times (FFF_{16} \div 2)$ . Why is the result not  $FFF_{16}$ ?
4. Set decimal mode, key in **# 100**, duplicate it, and convert the Level 1 copy to a character string. Now set binary mode. Why didn't the "number" in Level 1 change like the number in Level 2?

## B.I.T. Answers

1. **1000** **BINARY** **DEC** **NEXT** **R→B** (Result = # 1000);  
**PREV** **HEX** (Result = # 3E8);  
**OCT** (Result = # 1750);  
**FIN** (Result = # 111101000).

R→B converts a real number in Level 1 to a binary integer in the current base and word size.

2. **HEX** **#FF** **ENTER** **2X**; (Result = # 1FE).

3. **HEX** **#FFF** **ENTER** **2÷** (Result = # 7FF);  
**2X** (Result = # FFE).

The point here is that the result of the division of a binary integer is also a binary integer; any fractional portion in the result is lost. Therefore, the result of the division is accurate only to the next lowest whole digit, and doubling this result gives you a number 1 smaller than your original.

4. **BINARY** **DEC** **#100** **ENTER** **ENTER** **→STR** **ENTER** **FIN**

Level 2 has # 1100100, a binary integer. Level 1 is "# 100", a character string. A character string, even though it may look like another type of object, is simply a string of characters and as such has no binary-integer meaning. If you were to convert this string back into a binary integer at this point, it would be converted to  $100_2$  ( $4_{10}$ ) and not the original  $100_{10}$ .

## A Pause For The Cause

Take another compass reading here. You have now rounded out your repertoire of compound objects that can be built purely from one of the three fundamental information types:

**Real Numbers** may form *complex numbers, vectors* and *arrays*;

**Characters** may form *strings* and *names*;

**Bits** may form *binary integers*.

Now what? Where do you go from here? Is this the sum total of the objects you can build and use on your HP-28?

Not quite. You've seen most of the possible objects, but the few remaining are the most powerful of all. And they're different – because they aren't constructions built from only a single information type. Every object you've built so far has been a list of simpler, *similar* objects (i.e. based upon the same fundamental information type).

This again corroborates what you read on page 144, that the HP-28 deals simply with "lists of things." Up to now, the main concern has been those "things" and what they can mean to you.

But what exactly is a list itself? What good is it? Can you have lists that combine any objects you want?

It's time to answer these questions....

## Lists

The actual description of a list as an object ought to sound quite familiar by now:

A list is a one-dimensional ordering of objects – *any* objects. It is ordered such that the left-most element is numbered 1, with the rest of the elements numbered in ascending order. It may be arbitrarily large or small; in fact, it may even be completely empty.

As you've also seen already, a list is represented by bracketing a collection of objects within braces ( { and } ).

---

**Build One:** Press **CLEAR** **MODE** **PREV** **-ML**  
**{ { 7 } "A" [ (1,3) 2 CHS } ] A ENTER**  
**LIST**.

Here's what you'll see:



The calculator display shows a list structure with three elements. The first element is a list containing the number 7. The second element is the string "A". The third element is a list containing the numbers 1 and 3, followed by an ellipsis. Below the list, the command menu is visible, showing options like LIST, PUT, GET, PUTI, and GETI.

```
3:
2:
1: { { 7 } "A" [ (1,3) ...
⇨LIST⇨LIST⇨ PUT GET PUTI GETI
```

Notice these things:

1. As always, you don't need to press **■****]** at the end of the list unless another object follows it in the command line.
2. All elements of the list are on the same stack level. Both this and the fact that they are grouped together inside braces tell you that this is a single object. From now on, unless you purposely break it into its components, it will be treated as one object.
3. This list contains a list (**{ 7 }**). This is the first object you've seen that can contain an object of the same type as itself.
4. This list is longer than the display can hold. To view the whole list, you must do one of several things:
  - (i) Edit Level 1 (with either EDIT or VISIT). Keep in mind that EDIT mode will always display the object using STD and +ML modes. If the object is still too large, you may use the cursor keys to scroll the display.
  - (ii) Enable +ML and use **■****VIEW↑** and **■****VIEW↓** to scroll through the display (if the objects within the list are quite long, this method is not too helpful).
  - (iii) Decompose the object, in this case with LIST→, and if necessary, use **■****VIEW↑** and **■****VIEW↓** to then examine the individual elements in the stack. Remember to rebuild the list with →LIST when you're done;
  - (iv) Use GETI to step through the list's components.

---

Try That Last Choice:

Press **[1]** **GETI**. Result:



```
3: { { 7 } "A" [ (1,3)
2:
1:
+LIST LIST+ PUT GET PUTI GETI
```

---

The index on Level 2 has been incremented to a value of **2**, and Level 1 now contains **{ 7 }**.

Notice that, unlike arrays, the index for a list is a real number. The index for an array must be a list because arrays may have *either* one or two dimensions (a vector is the one-dimensional version) and thus one or two indices. But a list has only one dimension, so a single real number is used for an index.

---

**While The GETting Is Good:** Press **[DROP]** **GETI**. The index is incremented and Level 1 contains the character string **"A"**, which is the second element in the list.

Press **[DROP]** **GETI**. The index is incremented and Level 1 contains the complex vector (element number 3 in the list).

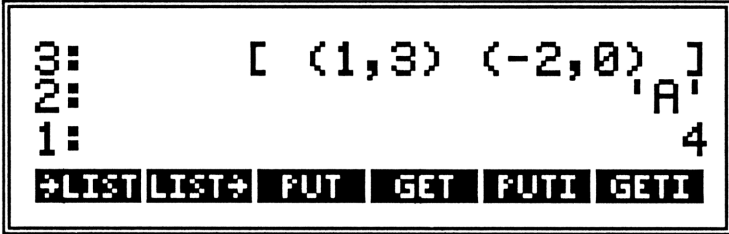
Press **[DROP]** **GETI**. The index is incremented and Level 1 contains the name **'A'**. Notice that the name has its single quotation marks when it's on the stack by itself – but not when it's a member of a list.

---



---

**Secession From The Union:** Press **DROP** **DROP** **LIST→** to decompose the list.



The image shows a calculator screen with a list being decomposed. The list elements are (1,3) and (-2,0). The screen shows the list structure with indices 3, 2, and 1. The number 4 is shown in the bottom right corner. The bottom of the screen shows the command menu with options: →LIST, LIST→, PUT, GET, PUTI, and GETI.

---

The number of list elements is in Level 1, all ready in case you want to rebuild the list by pressing **→LIST**.

Notice that, because the index/list-length is a real number, the list elements now "stacked up" become easily accessible to several other commands, particularly the stack commands ROLL, ROLLD, DUPN, and DROPN (you may recall that **→LIST** and **LIST→** are also available in the STACK menu).

It's quite feasible, therefore, to do some substantial list operations by decomposing the list, manipulating the elements in the stack, and then recomposing the list.

---

**Reconstruction:** Press **→LIST**. Very convenient, no?

---

As with character strings, there's a certain analogy between numerical addition and the addition of an element to a list. If you have two lists at Levels 1 and 2 you can "add" them together and get one list with all the elements of the original two.

---

**Like So:** Stack up two lists by pressing **MODE** **PREV** **+M/L** **↔** **{** **1** **2** **3**.

```

1: { { 7 } "A"
    [ { 1,3 } (-2,0) ]
    A
{123}

```

Now press **+** to combine them:

```

2:
1: { { 7 } "A"
    [ { 1,3 } (-2,0) ]
    A 123 }

```

The contents (**123**) of the list formerly at Level 1 have been added to the *end* of the list that was in Level 2 – the pattern for "list addition."

---

Remember that a list may contain *any* number of *any* data object. This makes the list the most general purpose data object available to you. In keeping with this idea of generality, the HP-28 doesn't restrict you by imposing too many list-specific commands (you'll note that the LIST menu isn't all that whopping huge).

But the convenience of the ability to decompose lists onto the stack, manipulate the various elements, and then restore the lists allows you to dream up new commands to manipulate them however you like!

## List Lessons

1. What's the difference between `{ 1 2 3 4 }` and `[ 1 2 3 4 ]`?  
How would you convert between one and the other?
2. Since a vector's components are limited to either complex or real number objects, how might you "represent" a "vector" whose elements are the *name objects* I, J and K?
3. You can add elements to a list using `⊕`, but how might you *delete* the last element? The first element?
4. Say that you work with lumber. You therefore work with "lumber numbers" in terms of feet, inches, and fractions of inches. What are some ways in which you might use a list on the HP-28 to meaningfully represent six feet, five and three-quarters inches?
5. You want to record the height and weight of a number of people so that you can later do some statistical analyses on them. How might you use lists to store/organize this information?

## List Lessons Learned

1. `{ 1 2 3 4 }` is a four-element list containing the real numbers 1 through 4. `[ 1 2 3 4 ]` is a four-element real vector containing the numbers 1 through 4.

Convert from the list to the vector: `LIST LIST+ 1 +LIST ARRAY +ARRAY`

Convert from the vector to the list: `ARRAY +ARRAY+ LIST LIST+ DROP +LIST`

Each of these decomposes the original object, alters the index so that it matches the new object, and forms the new object out of the items and the index on the stack.

2. `{ I J K }`

3. `LIST LIST+ SWAP DROP 1- +LIST` (This deletes the last element);  
`LIST+ 1- +LIST SWAP DROP` (This deletes the first element).

4. `{ 6 5.75 }` or `{ 6 "feet" 5 "inch" 3 "quarters inch" }` or `{ { 6 "ft" } { 5 "in" } { 3 "/4" } }`, et cetera.

5. `{ { "HEIGHT" "WEIGHT" } { 6.2 210 } { 5.9 170 } { 5.7 135 } }`, for example.

## Procedures: (a) Postfix Programs

Programs are objects just like any other object discussed up to now. They can be put on the stack, associated with a name, and put into a list. In fact, programs are merely one specialized version of a generalized one-dimensional list of objects. But to signify their special differences, programs are represented and treated as an object type in its own right.

A program is indeed entered and displayed as a one-dimensional list of objects, separated by delimiters in the usual manner, but it is bracketed between French quotation marks (⌘ and ⌘) to distinguish it from a generic list.

As data objects used to *store* information, programs are relatively useless. Except in the crudest manual sense – through the command line – you can't add elements to a program, nor can you break it into its components nor build it from its components. You can't perform math on it nor can you convert it to another object type.

Well then, what good is it?









A program is a *dynamic* object, not in the sense that it changes itself or can be changed by any other object, but rather that it does things; *it causes changes to other objects*.

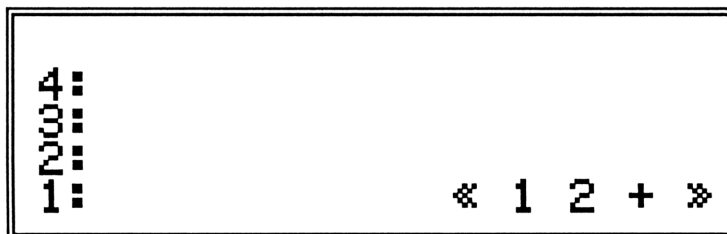
You've already been introduced to the idea of evaluation with names. Remember when a name is evaluated, how the HP-28 actually produces the value of the object associated with that name?

Well, programs can also be evaluated (indeed, that's their purpose in life). And when evaluated, *a program sequentially evaluates its elements*.

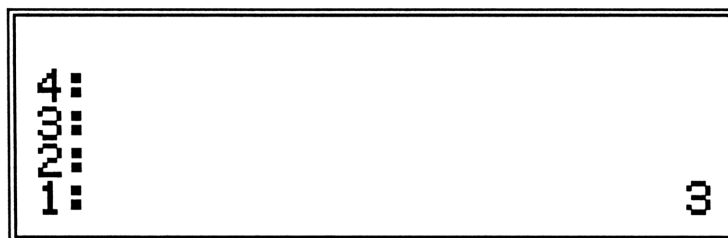
The best way to see this is with an example.

---

**Watch The Birdie:** Press    (notice that alpha mode is automatically activated)     . Result:




Now press .



---

This should look vaguely familiar. In learning about strings, you saw how you could put a similar sequence of objects into a string and then evaluate them by using STR→.

This might also look familiar from your earlier work with the command line. Remember when you lined up several items on the command line (separating each with a legal delimiter) and then pressed , how this "posted" all of them at once, one after another? If they were numbers or other data, they went onto the stack. If they were commands, they were executed immediately, right?

So there are actually three roughly identical methods of doing this same thing:

<b>Command Line:</b>	<code>1</code> <code>,</code> <code>2</code> <code>,</code> <code>+</code>
<b>String:</b>	<code>"1 2 +"</code> <code>STR→</code>
<b>Program:</b>	<code>⌘ 1 2 + ⌘</code> <code>EVAL</code>

Any operation that can be performed through the evaluation of any one of these expressions can also be performed by the others. So how are they different? Funny you should ask....

### The Command Line:

The command line is interactive and immediate. Once you've keyed in the string of characters representing objects, `ENTER` evaluates them. Thus, you have immediate evaluation *and* immediate error detection. The HP-28 tells if you've made detectable errors in your typing, and then you get immediate feedback on execution errors by pressing `ENTER`.

### Strings:

In sharp contrast to the command line, "stringed" collections of evaluable objects are non-interactive and non-immediate – but portable. They are non-interactive because a string may contain any character. The HP-28 won't look at a string for syntax errors. Therefore, you won't know until you convert the collection into non-string form whether or not the string contained errors.

It's also not as easy to evaluate a string as it is a program or a command line. For a string, you must explicitly use `STR→`. And if you have a name associated with it, you must first evaluate the name. But strings do have advantages over programs: they can take up less memory and can be modified by other commands.

## Programs:

So how do program objects stack up beside those other two ways to collect evaluable sequences of objects? A little of this and a little of that: Programs are somewhat interactive, non-immediate, and portable.

They are somewhat interactive because entry errors are detected just as when you use the command line. This happens because a program, though not immediately evaluated, is immediately scanned and turned into objects for storage. And during this scan, certain input errors can be detected.

A program is portable because it's an information object; you can put it onto the stack and "store" it in a name. Its association with a name makes it very convenient to use, because, as you remember, typing an unquoted variable name evaluates the name and all other objects the name points to.

*Therefore, named programs are virtually identical to HP-28 system commands.*



## Program Problems

1. Rewrite the solutions to problem 1 from page 176 as postfix programs. Name the first  $L \rightarrow V$  and the second  $V \rightarrow L$ .
2. Use  $L \rightarrow V$  to convert  $\{ \emptyset \}$ ,  $\{ 1 \ 2 \ 3 \}$  and  $\{ 1 \ \emptyset \ (1, \emptyset) \}$  to vectors.
3. Try to convert  $\{ \}$  to a vector using  $L \rightarrow V$ . What happens? Why?
4. Using  $L \rightarrow V$  and  $V \rightarrow L$ , write a program named LADD ("List ADD") that will add two lists together such that the resultant list's elements are the sums of the corresponding elements of the original two lists.
5. Use the program from problem 4 to add the following:
  - a.  $\{ 1 \ 2 \ 3 \ 4 \} \{ 5 \ 6 \ 7 \ 8 \}$
  - b.  $\{ (1,1) \ (-3,4) \} \{ -5.4 \ (4.3, -8.1) \}$
  - c.  $\{ 9 \ 6 \ 8 \} \{ 1 \ 1 \}$
  - d.  $\{ [1 \ 2] \ [3 \ 4] \} \{ [-3 \ 1] \ [6 \ 9] \}$

Why do c and d fail?

## Program Problem Solutions

1.  $\ll$   $\blacksquare$  LIST  $\blacksquare$  LIST  $\rightarrow$  1  $\rightarrow$  LIST  $\blacksquare$  ARRAY  $\rightarrow$  ARRAY  $\blacksquare$  ENTER  
 'L  $\blacksquare$   $\rightarrow$  V STO ;

$\ll$   $\blacksquare$  ARRAY  $\rightarrow$  ARRAY  $\blacksquare$  LIST  $\blacksquare$  LIST  $\rightarrow$  DROP  $\rightarrow$  LIST  $\blacksquare$  ENTER  
 'V  $\blacksquare$   $\rightarrow$  L STO .

It's certainly not very hard to translate postfix keystroke sequences into postfix program objects, is it?

2. USER { 0  $\blacksquare$  L  $\rightarrow$  V Result = [ 0 ];  
 { 1 , 2 , 3  $\blacksquare$  L  $\rightarrow$  V Result = [ 1 2 3 ];  
 { 1 , 0 ( 1 , 0  $\blacksquare$  L  $\rightarrow$  V Result = [ (1,0) (0,0) (1,0) ].

Notice – as you know – that a *list* will tolerate components of differing types, but a *vector* will not. Therefore, you get a vector with either all real or all complex components.

3. Here's what you do: **{ L→V** And here's what you get:

```
Bad Argument Value
3:
2:
1:                { 0 }
```

Why? To find out, mentally "walk through" the program  $L \rightarrow V$ :

$LIST \rightarrow$  puts the contents of a list onto the stack, followed by the number of elements in Level 1. **{ }** has no elements; its size is 0. Therefore, **{ }** is replaced with **0**.

1  $LIST \rightarrow$  then makes a 1-element list from the **0**, thus preparing the stack for the use of  $\rightarrow ARRAY$ . So at this point, **{ 0 }** is left on the stack at Level 1.

Then  $\rightarrow ARRAY$  tries to use this index to build a vector, but there's no such thing as a zero-length vector on the HP-28, so the error is generated and the stack is left as it was when the error occurred.

4. **\* L→V SWAP L→V + V→L \*** would be one reasonable approach. After all, you can't sum lists directly – but you can sum vectors!

Here are the keystrokes to create the program and name it LADD:

**<< USER L→V SWAP L→V + V→L ENTER 'LADD STO USER**

5. a. { 1 , 2 , 3 , 4 ENTER { 5 , 6 , 7 , 8 LADD

Result = { 6 8 10 12 }

b. { ( 1 , 1 ) ( 3 CHS , 4 ENTER { 5 . 4 CHS ( 4 . 3 , 8 . 1 CHS LADD

Result = { (-4.4,1) (1.3,-4.1) }

c. { 9 , 6 , 8 ENTER { 1 , 1 LADD

Result: The two lists are of different lengths and are therefore converted to two vectors of different length. The **Invalid Dimension** error occurs because the HP-28 can't add two vectors that aren't the same length. Note that the vectors are left on the stack after the error is reported.

d. { [ 1 , 2 [ 3 , 4 ENTER 3 CHS , 1 [ 6 , 9 LADD

Result: **Bad Argument Type** occurs when LADD attempts to make a vector out of vectors (i.e., when  $L \rightarrow V$  tries to perform  $\rightarrow \text{ARRAY}$  on a stack full of vectors.)

## Procedures: (b) Algebraic Expressions

Algebraic expressions are exactly like programs, only different.

They are programs whose syntax is algebraic (i.e. operand-operator-operand) rather than postfix.

Algebraic expressions are represented by bracketing a *syntactically correct* list of *algebraically meaningful* objects within single quotation marks ( ' ). Since those single quotation marks also apply to names, this explains why you can't have a name that looks like a syntactically correct (and therefore executable) expression (see page 160).

For example, compare `1 2 +` and `'1+2'`.

Both evaluate to `3`.

The major difference between them is the *order of the objects within them*. The ordering within a program is postfix (i.e. just like the stack – with the operands first and the operator last); a program can therefore contain stack commands.

By contrast, the ordering of an algebraic expression is, not surprisingly, algebraic, and thus it cannot contain stack commands.

This ability to contain stack commands is very important, since most commands in the HP-28 are stack commands and therefore postfix in nature. Consequently, programs are general purpose and may do virtually anything *because* they may contain any command. Algebraic objects may only contain mathematically meaningful expressions.

Notice that they're called *algebraic* objects, not simply *mathematical* objects. The reason is that algebraic objects may contain name objects (remember them?), thus giving the expressions the classical form of algebra with one or more *variables*.

("Aha!") Say that you want to solve some quadratic equations. The form of one solution is:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

You want to create an algebraic object which, when evaluated, will give you x.





---

### No Problem:

Here are the keystrokes to do this, along with a play-by-play analysis.

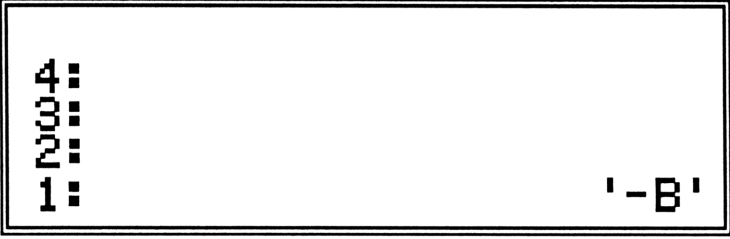
First, press      

Here, you're clearing the stack, selecting multi-line mode, and disabling the menu. This is just "clearing the decks for action."

Next, press    

Why? Well, you want to use the names A, B, and C (you may prefer a, b, and c, but lower-case is somewhat cumbersome). So you PURGE these names, dissociating them from any objects that they might otherwise belong to. This allows you to key in names without single quotation marks – not necessary, but convenient.


Now you start to build your algebraic expression. Press **B****ENTER****CHS**. Here's what you have so far:



The calculator display shows a stack of four numbers on the left: 4, 3, 2, and 1, each followed by a colon. On the right side of the display, the text '-B' is shown, enclosed in single quotation marks.

You've keyed in "negative B." The CHS placed a negative sign in front of the **B**. Notice that before you did the **CHS**, **B** was a *name* sitting on the stack, not an algebraic expression. But since the single quotation marks can mean either object type, whenever you perform any allowable mathematical operation – such as CHS – on a name, *the operation's effect will be to build an algebraic object*.

Next, press **B****ENTER****2****^**



The calculator display now shows the stack with five items: 4, 3, 2, 1, and '-B'. The new item '-B' is at the top of the stack. Below it, the text 'B^2' is shown, also enclosed in single quotation marks. The 'B' is in a bold font.

Here you key in **B** again and square it by raising it to the second power (you could have used **□****X<sup>2</sup>**) and the result would have been **'SQ(B)'**. The two versions of  $B^2$  evaluate to be the same thing; they just look different). Notice that circumflex, **^**. Because the HP-28 can't display superscripts, it uses the circumflex to indicate "raising to a power."

Now press **4** **ENTER** **A** **X** **C** **X**

4:	
3:	
2:	'-B'
1:	'B^2'
	'4*A*C'

You key in **4** and multiply it by **A** and **C**. Notice that the result is **'4\*A\*C'** and not **'4AC'**. If the HP-28 didn't use **\*** to indicate multiplication, neither it nor you could distinguish  $A \times B$  (**'A\*B'**) from the name  $AB$  (**'AB'**). In written algebra, you can omit the multiplication sign (it's implied) because you typically use only single character variables, such as  $x$ ,  $y$ , and  $z$ .

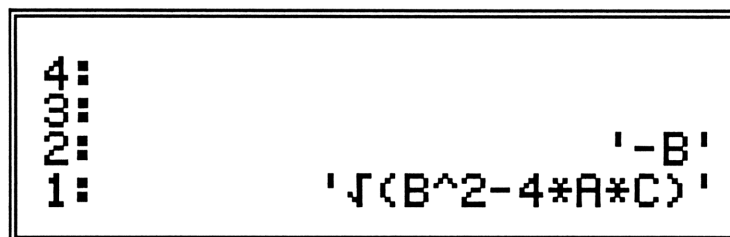
Then: **[-]**

4:	
3:	
2:	'-B'
1:	'B^2-4*A*C'

You subtract **'4\*A\*C'** from **'B^2'**. Notice again that when objects – even algebraic objects – are on the stack, you use *postfix* logic commands. Therefore you pressed **[-]** *after* the two arguments were on the stack. The HP-28 *then* interprets that arithmetic command in terms of the objects on which it must act. When acting upon two algebraic expressions, it just so happens that **[-]** means to combine them into one, with a minus sign embedded in the resulting expression.



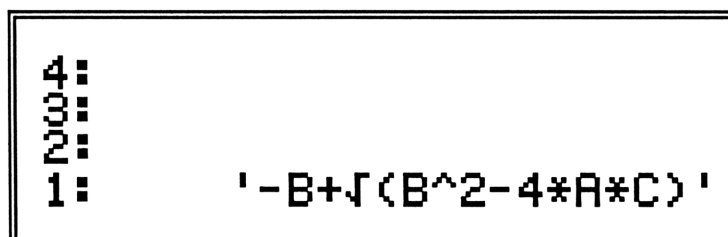
Next step: 



The display shows a four-line stack with labels 4, 3, 2, and 1 on the left. Line 1 contains the expression  $\sqrt{B^2 - 4AC}$ . Line 2 contains  $-B$ . Lines 3 and 4 are empty.

You take the square root of ' $B^2 - 4AC$ '. Notice the parentheses. Again, because the HP-28's display is limited, it cannot draw the radical so that it includes the entire expression under it. Instead, the radical sign is represented as a mathematical function, like  $f(x)$  (read "f of x"), and in the same way, parentheses are used to enclose the argument.  $\sqrt{(x)}$  is therefore "square root of x."

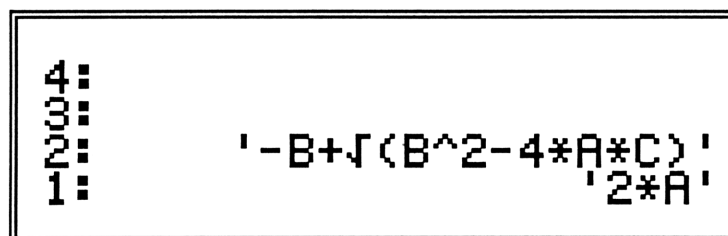
Then press  $\boxed{+}$



The display shows a four-line stack with labels 4, 3, 2, and 1 on the left. Line 1 contains the expression  $-B + \sqrt{B^2 - 4AC}$ . Line 2 is empty. Lines 3 and 4 are empty.

You add ' $-B$ ' to ' $\sqrt{B^2 - 4AC}$ '. No surprises, right?

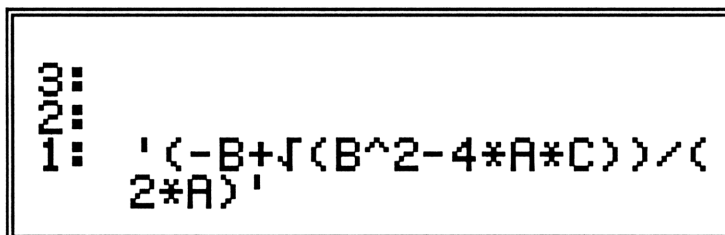
Keep going:  $\boxed{2}\boxed{\text{ENTER}}\boxed{A}\boxed{\times}$



The display shows a four-line stack with labels 4, 3, 2, and 1 on the left. Line 1 contains the expression  $2A \times (-B + \sqrt{B^2 - 4AC})$ . Line 2 is empty. Lines 3 and 4 are empty.

You multiply 2 by A. Again, no surprises.

At last:  $\boxed{\div}$



3:  
2:  
1: '(-B+√(B^2-4\*A\*C))/(2\*A)'

You have divided ' $-B+\sqrt{B^2-4AC}$ ' by ' $2A$ '.

Notice the extra parentheses. Since the display's limited capacity forces the division sign onto the same line with the rest of the expression, it needs a way to indicate what is divided by what. That's where the parentheses come in. They group the things that are in the numerator, ' $-B+\sqrt{B^2-4AC}$ ' and the things in the denominator, ' $2A$ '.

If these extra parentheses weren't there, the order of evaluation of the expression would be different, because in algebraic notation, the convention is that multiplication and division are performed before addition and subtraction.

---

Notice also that the final object doesn't fit on one line. In multi-line mode, an algebraic object in Level 1 will be broken between internal objects and displayed on several lines.

Of course, you could have simply *typed in* the expression above; the result would be the same. It's up to you which you find more convenient.

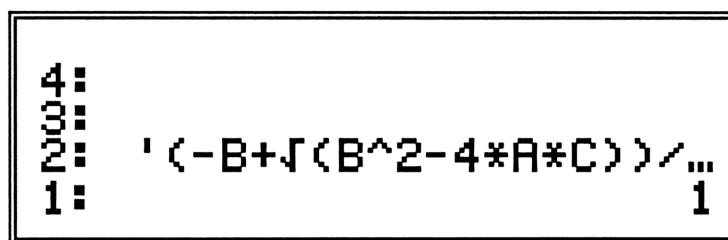
("Now they tell us.")

Now that you have an algebraic object, what do you do with it?

You use it as a mathematical procedure to solve problems.

---

**Like This:** Press `[ENTER] 1, [A] [STO] [2] [CHS], [B] [STO] [1], [C] [STO] [EVAL]`.



A calculator display showing the result of a quadratic formula calculation. The display is divided into four lines, each with a label on the left: 4:, 3:, 2:, and 1:. The expression  $\frac{-B + \sqrt{B^2 - 4AC}}{2A}$  is shown across the lines. The '1' at the bottom right of the expression is a small number, likely representing the denominator 2.

---

What have you done? You've associated a real number 1 with the name A, a 2 with the name B and a 1 with the name C (and you could verify this in your USER menu right?).

Then you've evaluated your algebraic expression, at which time all of the names that have associated objects were replaced by the objects themselves, and all mathematical expressions were performed.

Thus, you get the mathematical result of this expression – for the case where  $A=1$ ,  $B=-2$ , and  $C=1$ .

In this case, the result was a real-number object, 1. With other coefficients, it would, of course, be another result, possibly a complex-number object.

*But at no time will any algebraic expression ever leave more than one resultant object on the stack.*

This is quite different than a program, which may make lots of changes to the stack – say, decompose an array or something equally hairy.

So you see the limitations of an algebraic object. It's merely a way to list certain mathematical objects and operations in a collection that is evaluable in algebraic ("left-to-right") notation – rather than in postfix. And even this collecting process can be done with postfix operations on the stack, unless you choose to type in the entire expression manually.

Happily, this alternative notation *can* refer to named objects as its variables and thus it resembles written algebraic logic quite closely. And since so much of symbolic math is represented in algebraic form, even these few rudimentary capabilities on the HP-28 open up vast horizons for you.

## Algebraic Aptitude Test

1. Build an algebraic object for the expression  $x^2 - 2x + 1$  and evaluate it for:

- a.  $x = 1$
- b.  $x = -2$
- c.  $x = (2,3)$
- d.  $x = \sqrt{2}$ .

2. Evaluate the expression ' $A+B*C-D$ ' for:

- a.  $A = 1, B = -2, C = 3, D = 4$
- b.  $A = (1,2), B = (-2,-2), C = (.5,1.3), D = (104,.2)$
- c.  $A = [1\ 2], B = -2, C = [.5\ 1.3], D = [104\ .2]$
- d.  $A = 14_{10}, B = 20_8, C = 34_{16}, D = 101_2$

3. Evaluate the expression ' $2*X+Y$ ' for:

- a.  $X = -2Y$
- b.  $Y = -2X$
- c.  $X = T, Y = T - 1$
- d.  $X = Z - 3Y, Y = Y - 3Z$

4. Evaluate the expression ' $A+B+C+D$ ' for:

$A = \text{"THIS "}, B = \text{"IS "}, C = \text{"ODD"}, \text{ and } D = \text{"."}$

## A.A.T. Scores

1. `'X` `ENTER` `2` `■` `^` `'X` `ENTER` `2` `X` `-` `1` `+` `'EQ` `STO` `USER`

a. `1` `'X` `STO` `■` `EQ` `EVAL`; Result = `0`

Notice that invoking the (unquoted) name of an algebraic expression does not immediately evaluate it numerically; rather, it is left on the stack in its symbolic form.

This is an exception to the normal immediate-EVALuation rule for other named objects, but there's a good reason for this: Often the forms of the expression itself – and the possibilities for modifying those forms – are as much of interest to you as the numeric results of "plugging in" variables. Therefore, the HP-28 asks you to confirm that you are indeed interested in a numeric answer; you must explicitly use EVAL to tell it so.

b. `2` `CHS` `'X` `STO` `■` `EQ` `EVAL`; Result = `9`

c. `(` `2` `,` `3` `■` `)` `'X` `STO` `■` `EQ` `EVAL`; Result = `(-8,6)`

d. `2` `■` `√` `'X` `STO` `■` `EQ` `EVAL`; Result = `.17157287525`

2. **A+B×C-D** **ENTER** **E** **Q** **STO**

A reminder: According to conventional algebraic notation, you don't necessarily evaluate an expression strictly from left to right. Instead, your evaluation order is based upon the priority of various operators.

Thus, since multiplication and division have a higher priority than addition and subtraction, the expression  $A + B \times C - D$  is taken to mean:  $A + (B \times C) - D$ . Only *after* performing the multiplication are all remaining operations of the same priority; *then* the evaluation proceeds from left to right.

So the fact that you use no parentheses in keying in this expression means that you, too, are reading it and understanding it not just from left to right but also according to this hierarchy of priorities (which is called infix notation, by the way).

a. **1** **A** **STO** **2** **CHS** **B** **STO** **3** **C** **STO** **4** **D** **STO** **EE** **EVAL**;  
Result = **-9**

b. **(** **1** **,** **2** **ENTER** **A** **STO** **(** **2** **,** **2** **ENTER** **CHS** **B** **STO** **(** **.** **5** **,** **1** **.** **3** **ENTER** **C** **STO** **(** **1** **0** **4** **,** **.** **2** **ENTER** **D** **STO** **EE** **EVAL**;  
Result = **(-101.4, -1.8)**

c. **[** **1** **,** **2** **ENTER** **A** **STO** **2** **CHS** **B** **STO** **[** **.** **5** **,** **1** **.** **3** **ENTER** **C** **STO** **[** **1** **0** **4** **,** **.** **2** **ENTER** **D** **STO** **EE** **EVAL**;  
Result = **[ -104 - .8 ]**

d. **BINARY** **DEC** **#** **1** **4** **A** **STO** **OCT** **#** **2** **0** **B** **STO** **HEX** **#** **3** **4** **C** **STO** **BIN** **#** **1** **0** **1** **D** **STO** **USER** **EE** **EVAL**;  
Result = **# 1101001001** (because you're still in BIN mode).

3. `'2XX+Y` `ENTER` `'EQ` `STO`

a. `'-2XY` `ENTER` `'X` `STO` `'Y` `PURGE` `EQ` `EVAL`;

Result = `'2*(-2*Y)+Y'`;

`COLCT` `ENTER`; Result = `'-(3*Y)'`

Notice how the COLCT (COLleCT) command affects an algebraic result. It collects like terms and attempts to reduce the expression to lower terms.

b. `'-2XX` `ENTER` `'Y` `STO` `'X` `PURGE` `EQ` `EVAL`;

Result = `'2*X+-2*X'`;

`COLCT` `ENTER`; Result = `0`

c. `'T` `ENTER` `'X` `STO` `'T-1` `ENTER` `'Y` `STO` `EQ` `EVAL`;

Result = `'2*T+(T-1)'`;

`COLCT` `ENTER`; `'-1+3*T'`

d. `'Z-3XY` `ENTER` `'X` `STO` `'Y-3XZ` `ENTER` `'Y` `STO` `EQ` `EVAL`;

Result = `'2*(Z-3*Y)+(Y-3*Z)'`;

Now use EVAL and COLCT a couple of times on this result. Each time you use EVAL, the expression becomes more complex, because Y is replaced with a more complex expression containing Y.

4. `'A+B+C+D` `ENTER` `"THIS` `SPACE` `ENTER` `'A` `STO` `"IS` `SPACE` `ENTER` `'B` `STO` `"ODD` `ENTER` `'C` `STO` `".` `ENTER` `'D` `STO` `EVAL`

Result= `"THIS IS ODD."`

This works only because addition is defined on character strings.



## Procedures: (c) User-Defined Functions

You've seen how algebraic objects can be used to solve problems. You simply create an algebraic object of the proper form and assign values to the names in it. Then, when you evaluate this algebraic object, it combines the values represented as the expression specifies, and you get a result. Fine and dandy.

But if the algebraic object contains a lot of named objects, then associating (STOring) the data objects with their names can be a lengthy and therefore error-prone process.

One way around this – a method that stream-lines the use of algebraic objects – is the *User-Defined Function*.

Take, for example, the old standby: one of the two roots of a quadratic equation (remember that an algebraic object or expression can generate only one result, and therefore you can't get both roots at once).

You've already generated an algebraic expression (pages 186-190) to do this. And you've seen that in order to use this algebraic object to solve for numerical roots, you must assign values to the names – the *variables* – in the algebraic expression.

So you did that (e.g., 

1	'	A	STO
---	---	---	-----

2	CHS	'	B	STO
---	-----	---	---	-----

1	'	C	STO
---	---	---	-----

) and then EVALuated the expression.

---

**But Try This:** Create the following object:

⌘ → a b c '(-b-√(b^2-4\*a\*c))/(2\*a)' ⌘

And give a name to this odd-looking hybrid (it's a sort of cross between a postfix program and an algebraic object).

For this example, use the name 'RUTE' (there's an HP-28 system command already called ROOT, so you can't use that); type:

RUTESTO.

Now type USER 1 , 2 CHS , 1 RUTE.

What happened?

RUTE took the three objects *off the stack* and used them in its algebraic expression(!). The result, **1**, was left on the stack.

---

That's a real step saver, eh? But how did RUTE do it? Look at the object itself, to see if you can surmise some things from what you already know....

First, the French quotation marks (⌘) make it look similar to a postfix program. And postfix programs have the feature that when evaluated, they evaluate their contents *element by element, from left to right*. This thing should do the same.

So, going from left to right, the first few symbols,  $\rightarrow$  **a** **b** **c** , seem to be something new. But see if you can guess what they mean after looking at the rest of the object.

Skipping therefore to the next (and last) object, you find it to be an algebraic object. This is an algebraic object inside *another kind of program*, which is perfectly "legal" and often very useful.

So, the question is: Where does this user-defined function get the values for its variables, a, b, and c? After all, you certainly haven't created such names nor STORed any values into them. "Aha! The answer must have something to do with the  $\rightarrow$  preceding the list of variable names."

That's right: *This  $\rightarrow$  symbol inside a postfix program associates objects on the stack to whatever names follow the  $\rightarrow$ .*

In other words, the bottom three stack entries will be associated with (stored into) **a**, **b**, and **c**. And pay close attention to the order: **a** will be associated with Level 3, **b** with Level 2 and **c** with Level 1 (because you would naturally load the stack in the order a [ENTER] b [ENTER] c [ENTER]). So this is the way your listing of the variable names will be interpreted also. Makes sense, right?)

Of course, once these *stack* values have been associated with the names, the algebraic object is evaluated in the normal algebraic manner.

So that's how the U.D.F. works. It's truly a function, because it's an algebraic expression (which, as you'll recall, can produce exactly one result), but instead of looking into your USER menu collection of named objects to find its variables, it instead pulls objects off the stack – in the quantity and order you specify with the  $\rightarrow$  in the function definition.

Once again, you can see the hybrid nature of the U.D.F. It evaluates like an algebraic object, but it uses values from off the stack like a postfix program.

And here's an added bonus: U.D.F. names that are created and associated with  $\rightarrow$  are *temporary*. In other words, they are created at the beginning of the program and PURGE'd at the end.

Not only that, they are created in such a way that they don't conflict with other names that might already exist and have the same spellings. So if you had a list or some other object already STORed under the name of  $\mathbf{a}$ , and you nevertheless evaluated RUTE as written (with its *temporary*  $\rightarrow \mathbf{a}$ ), the contents of your named object would *not* be changed!

## User-Defined Function Fun

1. Build a user-defined function for the expression  $x^2 - 2x + 1$  and evaluate it at:

- a.  $x = 1$
- b.  $x = -2$
- c.  $x = (2,3)$
- d.  $x = \sqrt{2}$ .

2. Build and evaluate a U.D.F. for the expression ' $A+B*C-D$ ' when:

- a.  $A = 1, B = -2, C = 3, D = 4$
- b.  $A = (1,2), B = (-2,-2), C = (.5,1.3), D = (104,.2)$
- c.  $A = [1\ 2], B = -2, C = [.5\ 1.3], D = [104\ .2]$

3. Evaluate the expression ' $2*X+Y$ ' using a U.D.F. for:

- a.  $X = -2Y$
- b.  $Y = -2X$
- c.  $X = T, Y = T - 1$
- d.  $X = Z - 3Y, Y = Y - 3Z$

4. Define the U.D.F.  $\rightarrow X \rightarrow 'X+1'$  and name it INCR (increment). Then create both a postfix program and an algebraic object that use INCR to sum a number and its increment (i.e.,  $X + \text{INCR}(X)$ ).

## U. D. F. F. Consequences

1.  $\leftarrow$  SPACE  $\rightarrow$  SPACE X SPACE 'X^2-2XX+1' ENTER 'Q STO USER
- a. 1  $\square$ ; Result = 0
- b. 2 CHS  $\square$ ; Result = 9
- c. ( 2 , 3  $\square$ ; Result = (-8,6)
- d. 2  $\sqrt{\phantom{x}}$   $\square$ ; Result = .17157287525

Note that, unlike a plain algebraic expression, a U.D.F. does follow the rule for immediate evaluation – just like a postfix program and the other objects; if you invoke its (unquoted) name, it will produce its ultimate result right away, without stopping to let you see its algebraic form.

2.  $\leftarrow$  SPACE  $\rightarrow$  SPACE A , B , C , D SPACE ' A + B  $\times$  C - D ENTER  
' Q STO USER
- a. 1 , 2 CHS , 3 , 4 **Q**; Result = -9
- b. ( 1 , 2 ENTER ( 2 , 2 ENTER CHS ( . 5 , 1 . 3 ENTER ( 1 0 4 , . 2 **Q**; Result = (-101.4, -1.8)
- c. [ 1 , 2 ENTER 2 CHS ENTER [ . 5 , 1 . 3 ENTER [ 1 0 4 , . 2 **Q**; Result = [ -104 - .8 ]

3.  $\ll$  [SPACE]  $\rightarrow$  [SPACE] X [SPACE] Y [SPACE] '2X X+Y' [ENTER] 'Q' [STO]

a. 'Y' [ENTER] 2 X [CHS] 'Y' [ENTER]  $\blacksquare$   $\blacksquare$ ; Result = '2\*(-(Y\*2))+Y'

b. 'X' [ENTER] [ENTER] 2 X [CHS]  $\blacksquare$   $\blacksquare$ ; Result = '2\*X-X\*2'

c. 'T' [ENTER] [ENTER] 1 -  $\blacksquare$   $\blacksquare$ ; Result = '2\*T+(T-1)'

d. 'Z' - 3 X Y [ENTER] 'Y' - 3 X Z [ENTER]  $\blacksquare$   $\blacksquare$ ;

Result = '2\*(Z-3\*Y)+(Y-3\*Z)'

4.  $\ll$  [SPACE]  $\blacksquare$   $\rightarrow$  [SPACE] X [SPACE] 'X+1' [ENTER] 'I' [N] [C] [R] [STO]

$\ll$  [D] [U] [P] , [I] [N] [C] [R] , [ + ] [ENTER] which is:  $\ll$  DUP INCR +  $\gg$

'X' + [I] [N] [C] [R] ( X [ENTER] which is: 'X+INCR(X)'

Note the different form that INCR takes in a postfix program and an algebraic object. This is another special advantage of User-Defined Functions – the fact that you may use it in an algebraic in the form you normally expect to see for mathematical functions: f(x) or f(x,y), etc.

Naturally, you also use this conventional form when you invoke standard HP-28 functions – like LOG and SIN – in an algebraic expression.

**"The book stops here."**

It's the end of the beginning. And look how far you've come:

For starters, you know how the stack and keyboard work, how to move around within and between menus, how to edit and use the command line, and how to alter the display to your preference.

And here are all the objects you can now use and combine with skill and cunning:

- From *Real Numbers*, you can build **complex numbers**, **vectors**, and **arrays**;
- From *Characters*, you can build **strings** and **names**;
- From *Bits*, you can build **binary integers**;
- **Lists** are ordered collections of *any* combination of any object;
- The three **procedure objects** are just specialized lists of objects and operations that may be evaluated according to certain rules of sequence and syntax.

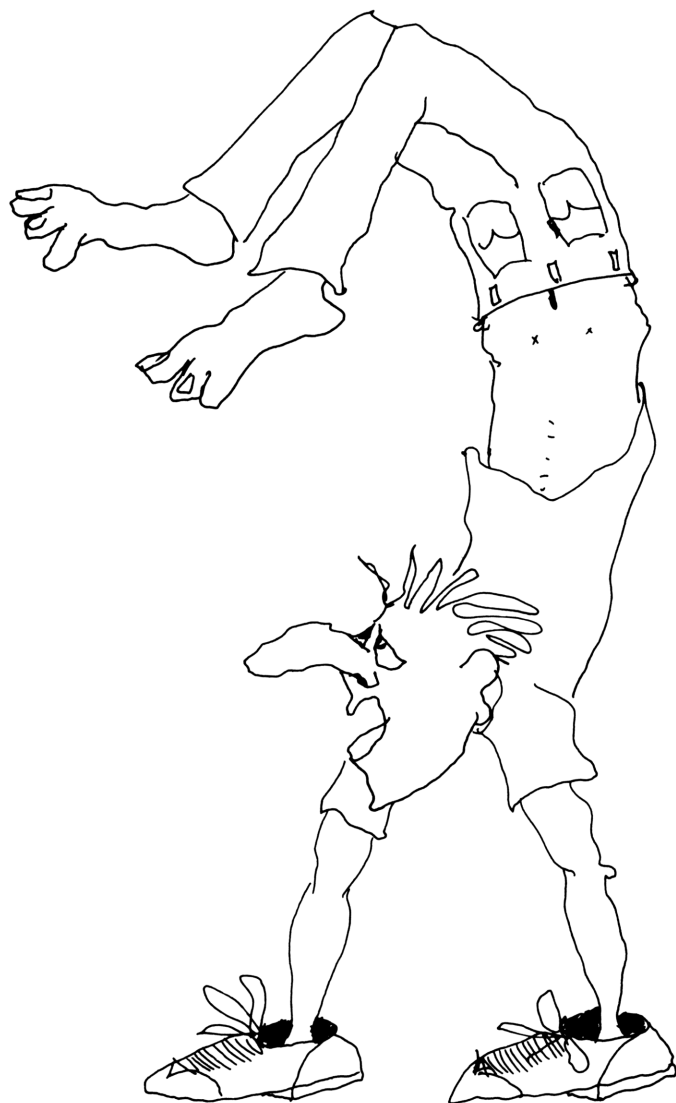
— — — — —

So that's about it, eh? That's all there is to learn about the HP-28, right?

Not quite. These are only the building blocks, which you can now combine into meaningful calculations for your own purposes. As you read at the start of this book, it's only an introduction into the world of the HP-28.

You're ready now to navigate on your own. Nobody but you knows exactly what calculations you need to perform, so...full speed ahead!





## APPENDICES

## Introduction to the Appendices

What we have here is a problem.

This book has been your introduction to the basics of HP-28 operation, but there is much more available to you. These basic lessons gave you a feel for the different kinds of information objects and their use. But there are programs and groups of operations living in the machine that do more sophisticated problem-solving for you. These operations carry far-ranging implications for your use and application – so much so, in fact, that a complete description of any one of them would require a book in itself.

The problem is, a book cannot be 10 books, no matter how hard it tries. So although there's a lot more to say about the following topics, there just isn't the room here to tell the whole story for each one. But they are definitely worth mentioning – if for no other reason than to round out your introduction to the machine. Even with these very cursory sketches of these topics, maybe you'll get some feel for their uses, potentials, and limitations. Above all, these "surface-scratching" pages are meant to give you a little courage and curiosity about all these other capabilities. So take these first impressions and run with them.

## Appendix A : Algebra

For the sake of brevity – and so as not to overload your circuits – the earlier discussion of algebraic objects (in Chapter 4) was somewhat incomplete. While you can now recognize an algebraic object on sight and should even be able to build one of your own, as always, there's more to the story.

Algebraic objects are, as you know, very similar to programs. In a sense, they are a type of program with a restricted area of use: They are limited to evaluating mathematically-valid, algebraic expressions.

At first blush, this may seem to be a severe limitation. It isn't.

When you stop and realize that the manipulation of algebraic expressions (i.e. the symbols themselves) is almost unheard of on a hand-held calculator (or even on larger computers), you'll realize that this is a tremendous enhancement. And whenever you need more general-purpose programming, the postfix program object is always ready and willing to fill that bill.

### Why use an algebraic object?

Algebra is a universal language. Almost every technical field has its own set of algebraic equations and expressions that are meaningful to it. These equations describe and define the relationships that exist within the discipline. It's only reasonable, therefore, that the HP-28 provide you with the ability to manipulate and solve algebraic problems.

Much of your work can also be simplified by writing ad hoc algebraic expressions for the problem at hand. In this way, HP-28 algebraic objects are programming without a programming language; the programming language is algebra!

To understand the full utility of an algebraic object, you should understand exactly what it is to the HP-28.

For example:

1. An algebraic object, when evaluated, yields one and only one result. This result is *typically* a real or complex number, but it need not be. The only objects that *cannot* be the result of the evaluation of an algebraic object are postfix programs and user-defined functions.
2. An algebraic object may contain only names, algebraic operations, algebraic functions, user-defined functions, real numbers and complex numbers.

If this is the case, then how do you get other types of objects to result from the evaluation of an algebraic object? By storing other types of objects under the variable names used by the algebraic! Every object type, except postfix programs, has at least one algebraic operation (namely, + ; addition) that can be used on it. Just keep in mind that the mathematically-defined objects have a greater wealth of operations available to them, but their rules of operation vary. For example, multiplication is *not* a commutative operation on arrays (matrices).

3. An algebraic object may or may not contain one equal sign (=). If it has no equal sign, it is called an algebraic *expression*. If it has an equal sign, it is called an algebraic *equation*.

That's basically it. There are no more restrictions. There are only ramifications and applications. We'll spend the rest of this discussion in looking at these.

## How do you use an algebraic object?

You've seen already that an algebraic object can be used to solve numeric problems. You simply create an object that defines a relationship between names (variables), assign values to the names, and evaluate the expression. The result is (often) a single numeric value.

It isn't always a numeric value, though. If you have certain *unknowns*, i.e. certain names with no associated values, the expression can be *reduced* to a form containing only numeric values, algebraic functions and operators, and the undefined names.

Such reductions or other rearrangements of an expression can often give you insights into the nature and the behavior of an expression – things that the raw form just doesn't give you. That's the whole beauty of algebra, isn't it?

So what ways does the HP-28 give you to reduce, summarize, or otherwise rearrange an algebraic object?

First of all, it provides you with an automated method of investigating *numeric* or *symbolic* solutions to algebraic objects. This method uses the SOLV menu, which is discussed in Appendix B : Using the Equation solver.

You can also investigate the nature of algebraic expressions *visually* on the HP-28, by using the PLOT menu. Plotting the numeric value of an algebraic object as the function of one of its names can really help you to understand an algebraic expression or equation. You'll get a chance to explore this further in the upcoming Appendix C : Plotting.

But here's a run-through of some other ways to use algebraic objects....

## Symbolic Calculus

Symbolic calculus (i.e., the calculus of algebraic expressions that yields other algebraic expressions rather than numeric approximations of differential and integral expressions) has merits and capabilities so vast and varied that it makes no sense to try to describe them here .

If you're unfamiliar with calculus, relax; you needn't be familiar with it in order to use the HP-28 effectively and well in a host of other areas.

If you *are* familiar with calculus, rejoice, because the HP-28 has brought many important uses of this marvelous tool to the realm of hand-held computation.

Differential calculus is well represented and easy to use. You need only place an algebraic object in stack Level 2, the name of the variable of differentiation in Level 1, and press  $\blacksquare \boxed{d/dx}$ . The resultant derivative is left in Level 1.

For example,  $\boxed{1}\boxed{X}\boxed{+}\boxed{L}\boxed{N}\boxed{(}\boxed{X}\boxed{)}\boxed{ENTER}\boxed{1}\boxed{X}\boxed{ENTER}\blacksquare \boxed{d/dx}$  yields : '1+INV(X)' .

Integral calculus' only representatives on the HP-28 are *symbolic* integration of polynomials and *numeric* integration (i.e., definite integration with a numeric result) of any expression.

To symbolically integrate a polynomial, you must put the polynomial in Level 3, the name of the variable of integration (this is also the name of the variable for which the expression is a polynomial) in Level 2, the degree of the polynomial in Level 1 and press  $\blacksquare \boxed{\int}$  .

For example,  $\boxed{1}\boxed{3}\boxed{X}\boxed{X}\blacksquare \boxed{\wedge}\boxed{2}\boxed{-}\boxed{2}\boxed{X}\boxed{X}\boxed{+}\boxed{.}\boxed{3}\boxed{ENTER}\boxed{1}\boxed{X}\boxed{ENTER}\boxed{2}\blacksquare \boxed{\int}$ , yields ' .3\*X-X^2+X^3' .

## Reduction to Lowest Terms

Unfortunately, the flexibility of algebra gives it one drawback: Two expressions that appear radically different may in fact be equivalent.

As you know, if you've used algebra to any extent, perhaps the most time-consuming, laborious and important job of all is the reduction of an algebraic expression to its lowest terms. Fortunately, the HP-28 has provisions for just such problems.

In the ALGEBRA menu, there are operations that allow you to expand (EXPAN) terms, collect (COLCT) terms, and manually reorganize (FORM) algebraic expressions so that they can be made into visually meaningful expressions *without* the possibility of dropping terms or performing illegal or erroneous operations.

You've already seen COLCT. But be forewarned: FORM is not for the casual user! It's *very* literal-minded and expects *very* explicit use of very mundane algebraic laws.

In fact, if you're just a casual user of algebraic manipulations on your HP-28, you will almost certainly find that either using slightly ill-formed expressions or re-entering an offensive object is less time-consuming than using FORM.

## Peculiarities of Representation

Inherent to manipulating algebraic objects on the HP-28 are its representation limits, which can at times be both annoying and confusing. The main limitation for algebraic objects is that all expressions must be represented as lines in the display.

In other words, there's not enough display to be able to present an algebraic expression in the pleasant and easy-to-read formats that you have seen in textbooks and have often written yourself. The HP-28 (and you too, when you're using it) must organize all algebraic expressions and equations so that they can be written and read from left to right, one character at a time.

This can (and does) often result in extremely long and complex-looking expressions, with a prodigious use of parentheses to keep things grouped properly. But there's no alternative.

Here's a word of comfort, however: The presentation of algebraic information is one of the most difficult aspects of computer algebra even on large computers with large displays.

Your HP-28 is in good company.



## Symbolic Constants and Symbolic Function Evaluation

Symbolic calculation is great, but of course, it's not the only way to calculate. Users of most other calculators and computers will recognize this readily. Most calculation tools are intended to help you generate numeric results. After all, you usually want a numeric value when you add 1 and 1, not a symbolic expression. Most calculators and computers do this as a matter of course – both because it's commonly what you want and because they just can't do it any other way.

The HP-28 gives you the *option* of exclusively evaluating objects down to numeric results – if it's most convenient.

You simply clear flag 36 (press `36,CF` `ENTER`).

All evaluations after this point will attempt to generate non-name objects (real numbers, vectors, et cetera). If it cannot find such an associated object for the name it's trying to operate on, it generates an error: **Undefined Name**.

In this mode, the HP-28 acts most like calculators and computers that you have used before. It assumes that you want the name to refer to something, or you wouldn't be trying to calculate with it. In this mode the HP-28 is most like a *calculator* and least like a *symbolic manipulator*.

Of course, even when in symbolic evaluation mode (you can choose this by setting flag 36: `36,SF` `ENTER`), you can always *force* numeric evaluation for individual cases by using `→NUM`. But it's good to know you can choose.

Finally, the HP-28 sports a set of data objects called *symbolic constants*. This is a set of three commonly used mathematical constants and two machine-specific constants. They are:

<b>e</b>	Euler's constant, the base of the natural logarithm;
<b><math>\pi</math></b>	the ratio of the circumference of a circle to its diameter;
<b>i</b>	the square root of $-1$ ;
<b>MINR</b>	the HP-28's smallest representable number; and
<b>MAXR</b>	the HP-28's largest representable number.

Each of these can be considered a numeric function because each can be made to yield a number. But they are in fact constants because they never change.

They're *symbolic* constants, partly because they have names and partly because in symbolic constant mode (flags 35 and 36 set), they resist conversion to their respective numeric values. In that mode, they will remain symbols unless forced to become numbers by  $\rightarrow$ NUM.

This stubborn resistance to change can be extremely useful, especially in the simplification of expressions, because you're often interested in a result that is a function of  $e$ ,  $\pi$ , or  $i$ . If the result were strictly a number, you would potentially lose some information.

For example, which is more meaningful:  $2\pi$  or 6.28318530718 radians?

And which is more exact? The HP-28 (in symbolic constant mode) has certain functions that recognize these constants – noting the fact that any numeric representations of  $e$  and  $\pi$  are only approximations.

Thus,  $\text{SIN}(\pi)$  under symbolic evaluation mode (and radians mode) is  $\emptyset$ , but under numeric evaluation mode, it's  $-2.06761537357\text{E}-13$ .

## Appendix B : Using the Equation Solver

### Numerical Solutions to Algebraic Expressions and Equations

You've seen that you can consider algebraic objects to be programs written in algebraic syntax, and as such, you can use them to solve for numerical results of algebraic expressions. In such cases, the algebraic object's internal names ("variables") refer to numeric objects.

But you've also seen that assigning values to these names can be tedious. SOLV, the equation-solver menu, exists to aid you with just such problems. The SOLV menu allows you both to conveniently load an algebraic object (via the command called STEQ, or STore EQUation) and then use SOLVR fill in the values of its variables!

Try this: press  $\boxed{1}\boxed{3}\boxed{\times}\boxed{X}=\boxed{2}\boxed{5}\boxed{-}\boxed{4}\boxed{\times}\boxed{Y}$  **SOLV** **STEQ** **SOLVR** .

SOLVR locates and identifies all of the names within the object and generates a menu of these names, including **LEFT=** and **RT=** for the expressions on the left and right sides of the equal sign, respectively. From this point, all you need to do is put a value onto the stack and select the menu key of the name you want to give this value. SOLVR takes the value from the stack and associates it with the name.

Press  $\boxed{5}$  **⌘** **▢** **Y**; This stores 5 in X and solves for Y.

Press  $\boxed{2}\boxed{\cdot}\boxed{5}$  **Y** **⌘** **⌘**; This stores 2.5 in Y and solves for X.

Press **LEFT=**; This solves for the value of the left side of the equation when X=5.

Press **RT=**; This solves for the right side of the equation when Y=2.5.

Although this in itself is terribly convenient – especially if you want to play with different values to see how the expression acts – there are some important ramifications that radically increase the usefulness of SOLVR.

First, *it doesn't matter which value is the unknown*. Normally, you would need to rewrite an algebraic expression so that the name of the unknown is on one side of the  $=$  and an expression of known values is on the other. SOLVR will do this rewriting for you, automatically!

Second, there are expressions and equations for which it is impossible or virtually impossible to isolate a particular unknown in this way. For such problems, the only solution may be an approximation to the value of the unknown. But SOLVR knows this, and will find this solution automatically!

Unfortunately, in this latter case, there may be no unique solution; that is, there may be more than one answer that will satisfy the expression. You must be able to recognize expressions that are likely to behave this way, because although SOLVR can find a result, it may not be the only one – or the best one.

So you may supply a guess or guesses as to the value of the unknown before solving for it. SOLVR will start with these guesses as it looks for a solution. Therefore, it will find a result (if any) that's relatively close to one of your guesses.

SOLV and SOLVR are actually friendly ways to use the actual numeric root-finder program, ROOT. They are excellent for interactive problem solving and algebraic object manipulation, but it's quite possible that you'll have a more sophisticated problem, one where a solution to an algebraic object is only half the battle. In that case, you may opt to use ROOT itself in a postfix program.

ROOT takes either a name (most commonly the name of an algebraic function, expression, equation or postfix program) or a postfix program from Level 3, the name of the unknown from Level 2, and one or more guesses from Level 1 of the stack. It returns a numeric result to Level 1.

For example:

Press `'X` `■` `^` `2` `+` `X` `-` `2` `ENTER` `ENTER` `'X` `ENTER` `5` `ENTER` `R` `O` `O` `T` `ENTER` .

Press `DROP` `'X` `ENTER` `5` `CHS` `ENTER` `R` `O` `O` `T` `ENTER` .

The result is 1 when you guess 5, and it's -2 when you guess -5.

Unfortunately, a good discussion of how and why ROOT comes to its results would require a good discussion of the nature of algebraic expressions, equations and functions, as well as the nature of numerical approximations. These are clearly beyond the scope of this book, but there are some such discussions in the HP-28 Reference Manual.

## Symbolic Solutions to Algebraic Expressions and Equations

As you read, there are ways to "solve" algebraic expressions and equations for specific names. In other words, you can *isolate* (using ISOL) a specific name from an algebraic object. The result is an expression which is equivalent to the specified name.

For example, if you had the equation ' $A=B*C$ ' in Level 1 and typed  $\boxed{\boxed{C}}\boxed{\boxed{I}}\boxed{\boxed{S}}\boxed{\boxed{O}}\boxed{\boxed{L}}\boxed{\boxed{ENTER}}$ , the result would be ' $A/B$ '.

You asked the HP-28 to isolate  $C$  for you, and it did so. You can see by inspection that if you were to divide the original equation by  $B$ , then  $C$  would indeed be equal to ' $A/B$ '. This is what is meant by isolation – the rearrangement of the original expression so that the result is equal to the isolated name.

But be careful: While isolation of a name in this way can be very useful when a name occurs only once in an expression or equation, if there's more than one occurrence of the name, the resulting expression will also contain the name. In other words, you will not have achieved very much by using ISOL.

There's another method of "solving" an algebraic object if that object is specifically in the form of a quadratic expression or equation. QUAD will take an algebraic object from Level 2 and the name of the variable for which the expression is quadratic from Level 1. The result will be an expression for one or both roots of the quadratic expression.

But if QUAD returns only one object, how does it show both roots? The answer is an unequivocal "It depends." It depends on whether flag 34, the *principal value flag*, is set or clear.

Both ISOL and QUAD are capable of solving for multiple roots. In other words, if an expression (like a quadratic) has multiple roots, all roots can be found and represented.

To see how, first set flag 34 ( $\boxed{3}\boxed{4}\boxed{,}\boxed{S}\boxed{F}\boxed{\text{ENTER}}$ ) and purge **A**, **B**, **C** and **X** (by pressing  $\boxed{\{}\boxed{A}\boxed{,}\boxed{B}\boxed{,}\boxed{C}\boxed{,}\boxed{X}\boxed{\blacksquare}\boxed{\text{PURGE}}$ ). Next, key in the expression ' $A*X^2+B*X+C$ ' and put a copy in both Levels 1 and 2. Now key in  $\boxed{X}\boxed{\text{ENTER}}\boxed{Q}\boxed{U}\boxed{A}\boxed{D}\boxed{\text{ENTER}}$ .

The result is ' $(-B+\sqrt{B^2-4*(A*2/2)*C})/(2*(A*2/2))$ '.

This is clearly *one* of the roots, but because you specified that you only wanted one of the roots (by setting the principal value flag), you were only given one.

On the other hand, type  $\boxed{3}\boxed{4}\boxed{,}\boxed{C}\boxed{F}\boxed{\text{ENTER}}\boxed{\text{DROP}}\boxed{X}\boxed{\text{ENTER}}\boxed{Q}\boxed{U}\boxed{A}\boxed{D}\boxed{\text{ENTER}}$ . The result is ' $(-B+\pm 1*\sqrt{B^2-4*(A*2/2)*C})/(2*(A*2/2))$ '.

This object is different from the first by the inclusion of the name **±1**. You will recall that the general solution is  $-b \pm \sqrt{\dots}$  et cetera. Well, **±1** functions as the  $\pm$  here. Since there are two solutions and an algebraic object can only return one result, QUAD gives you the option of choosing one, the other or both of the possible solutions.

Here's how:

If you were to associate the value 1 with the name **s1** (in the usual manner – with **STO**) and then evaluate the expression, the result would be  $-b+\sqrt{\dots}$

On the other hand, if you were to associate  $-1$  with **s1** and evaluate the expression, the result would be  $-b-\sqrt{\dots}$

**s1**, therefore, stands for *sign1* and should be interpreted when reading the object as  $\pm$ . (If there were more than one  $\pm$  possible in the object, you would see **s2**, **s3**, **s4**, et cetera.)

This convenience is fine for solutions to a quadratic equation, but what about objects that have more than two roots?

Let's find out. Take a periodic function, like sine, for which there are infinitely many roots.

Do this: Press **DEG** **ENTER** **X** **ENTER** **SIN** **ENTER** **X** **ENTER** **I SOL** **ENTER**.

The result is '**180\*n1**'. Here, **n1** stands for any integer, indicating that the sine of any integral multiple of  $180^\circ$  is zero. (Again, if there were more integral multiples in the expression, you would see **n2**, **n3**, **n4** ....) Realize that if flag 34 were set, the result would be simply **180** (the principle value).

One final thing to note is that if you want strictly symbolic results, take care to **PURGE** any names that the expression uses. If you don't, the names will be evaluated and replaced by their referent objects.



## Appendix C : Plotting

Information comes in many forms. You've seen numbers, letters, bits, and various and sundry compound information types built from them. Each of these forms has advantages based on how it's used and what you want to know.

Graphs are, in a sense, pictures of numbers. Such pictures give you easy access to (1) trends in collected data; (2) peaks and valleys in the output of functions; (3) comparison between different functions; (4) function zeros; et cetera, etc.

In short, graphs give you information about information.

And the HP-28 gives you the ability to generate graphical pictures of numeric information.

The DRAW function in the PLOT menu is basically a program written to automate the process of graphing real-valued *functions*.

A function, in this case, is anything that maps one real value onto another. As far as the HP-28 is concerned, therefore, the function can be an algebraic object, a postfix program, and even a constant or a name.

In the case of the postfix program, the function must be written so that it takes no values from the stack (i.e., it refers to values via names) and so that it leaves only one.

The first and second pages of the PLOT menu contain the operations you will need to set up and plot the function(s) of your choice.

Some examples of a postfix program and an algebraic object used to plot  $X^2+X-2$ :

`'X^2+X-2'`

`« X DUP SQ + 2 - »`

The DRAW command assumes that the function contains only one undefined name. That is, since the plot will be two-dimensional, it must have one and only one name (the independent variable – the "x-value").

You may explicitly select the independent variable using INDEP. If you do not explicitly choose the independent variable and the function contains more than one name, DRAW will scan the object and use the first name it finds as the independent variable.

The y-axis is used to indicate the value of the function given the current value of the independent variable. This does not mean that the algebraic object cannot have more than one name in it. It does mean that DRAW will only vary one of them as it successively evaluates the function. Thus, every other name had better have a value attached to it, otherwise DRAW will generate an error.

## Scaling

Plotting is not always as simple as storing a function (i.e., via STEQ) and invoking DRAW.

Sometimes, in order to get the clearest picture of the function, you must have some idea of the scale of the plot and therefore the domain over which you want to plot (the x-values) and the range of the function (the y-values).

If you don't, it's possible that what you'll see may be so little of the curve that you can't get much information from it, or so much of the curve that you don't see important detail.

Try this: Press  $\boxed{1}\boxed{X}\boxed{\blacksquare}\boxed{\wedge}\boxed{2}\boxed{+}\boxed{X}\boxed{-}\boxed{2}\boxed{\blacksquare}\boxed{\text{PLOT}}\boxed{\text{DRAW}}$ . The resulting plot has its "bottom" cut off. You know that this expression "bottoms out" when X is  $-0.5$ , but you can't see it.

Scale is established primarily by setting the minimum and maximum values of the domain and the range. You do this by making a complex number (x,y) out of the maximum x- and y-coordinates and then using the command PMAX. Then you do the same for setting the minimum values of the domain and range, except that you would use PMIN.

To correct the plot you just generated, press  $\boxed{\text{ATTN}}\boxed{(}\boxed{6}\boxed{\cdot}\boxed{8}\boxed{\text{CHS}}\boxed{,}\boxed{2}\boxed{\cdot}\boxed{2}\boxed{5}\boxed{\text{CHS}}\boxed{\text{PMIN}}\boxed{\text{DRAW}}$ .

Looking in the second level of the menu (via  $\boxed{\text{NEXT}}$ ), you can increase or decrease the width of the plot with  $\boxed{\text{SH}}$  (which multiplies the plot width by a constant). Numbers less than 1 will decrease the width (reduce the domain), while numbers greater than one will increase the width (enlarge the domain).

$\boxed{\text{SH}}$  works similarly with the plot height (the range, i.e. the y-values).

## Digitizing

Once you have stored the function, established an initial scale, possibly selected the independent variable, and invoked DRAW, the HP-28 will plot the function. Depending on the function, this process may take some time. You'll know when it's done, because the busy annunciator will be turned off (but if you become impatient, ATTN will interrupt the plot).

When it's done, the plot is left for your inspection. Then you also have the option of digitizing some points. That is, you may move a special cursor (it resembles a +) around the plot to points of interest by using the cursor keys. Once you've found a likely spot, you can record its coordinates (put them on the stack) by pressing INS. Don't worry that you can't see the recorded value. It will be in the stack when you leave the plot display.

One reason to digitize points is to zero in on an interesting bit of the curve. You may pick two new points on either side of the interesting portion to be your new PMIN and PMAX. By digitizing those points, you'll have them on the stack and thus available for **FMIN** and/or **FMAX** and then replotting with a new scale.

Another good idea: You can use the digitized points as guesses for use in SOLVR.

## Plotting an Equation

One feature of DRAW that's not obvious is what it does with algebraic equations. Since an algebraic equation is essentially a pair of algebraic expressions separated by **=**, the DRAW command *plots both at the same time*.

The advantage to this is that the point(s) at which the two curves cross (if they cross) is the point at which the two expressions are equal (go after it with that digitizer!).

## Appendix D: Postfix Programming

Much can be said about programming. One might wax eloquent about such topics as structured and unstructured programming, top-down versus bottom-up design, control structures, memory management and the like, but these topics won't be covered in these pages.

There are two reasons for this, really:

First, as is readily apparent, much – very very much – can be written about each and every topic listed above. One day, they may be fully and sufficiently discussed as applied to the HP-28, but not here and not now. As always, lack of space is the antagonist.

Second, and consider this well, 95 to 100% of the utility that you will derive from the HP-28 will be achieved *without using a single line of program code* – that is, without postfix programming as opposed to algebraic objects, user-defined functions and already existing system commands.

For those programming enthusiasts among you, this may be hard to believe (and the rest of you may breathe a sigh of relief), but consider this: The vast majority of the software written for hand-held calculators today attacks the problems that the standard functions of the HP-28 solve without user programming.

You have problem-solving capability available to you in a few keystrokes that most other calculator – and even desk-top computer – users can only dream of, unless they write or purchase elaborate programs to do so.

And consider this also: The HP-28 was not designed to be host to large and comprehensive programs. For while it's true that it sports significant programmability – just in case you have an unusually complex problem to solve – it also has some important limitations.

The HP-28 is, and was designed to be, a mathematical problem-solving toolbox. In it resides a host of very powerful commands. It has some memory – not a lot.

But when it comes to input/output options (i.e., methods of loading data and programs into the machine and methods of saving and presenting information), it has almost none. Mass storage (storage of information on external media) is non-existent; programs must be loaded by hand from the small keyboards, and other than residing in the machine itself, programs can be stored only on thermal printer paper.

*Limitations, et cetera, et cetera....*

Users of other, less powerful calculators may find these overly harsh words. Users of desk-top computers will not.

Understand, the intention here is not to knock the HP-28 but rather to describe its intended utility. The things it was designed to do it does exceedingly well – indeed, one would be hard pressed to find the like in popular computation.

But, those things for which it was not designed will prove difficult if not impossible to implement. No doubt, capable and powerful software will be developed by those users with much time, patience and zeal, but ordinary mortals will find such endeavors no trivial tasks.

How then can you best make use of the programmability of the HP-28? Think of it as the ability to develop larger and more specific tools from smaller, more general tools. To borrow an analogy from biochemistry, you would design small functional molecules from the functional atoms provided and then perhaps build even larger functional molecules (proteins, say) from other, previously constructed molecules.

But you wouldn't expect to develop viable organisms from such components. More likely, you would construct particular routines (enzymes?) that make a task easier.

For example: The HP-28 has several functions and operations that deal with vectors. It does not, however, have a unit vector function. This is probably because, given a real vector on stack Level 1, the keystrokes needed to obtain the unit vector aren't all that complicated. They are: `ENTER` `ABS` `ENTER` `1/X` `↓`.

But if you have a frequent need to find unit vectors, the repetition of these keystrokes would become a bit tedious. The corresponding postfix program would be `« DUP ABS INV * »` which you can then name, say, `'UNIT'`, and thus be able to invoke it with one keystroke from the USER menu. This is the archetypical HP-28 program – short, sweet, and effectively a new command.

Notice, though, that you don't need a postfix program to find unit vectors. You could have used an algebraic object (`'INV(ABS(v))*v'`), but you would have had to give the original vector a name.

A user-defined function (U.D.F) would be ideal, since it's effectively an algebraic (more understandable) object that can nevertheless use objects on the stack.

Such a U.D.F. might be `« → v 'INV(ABS(v))*v' »`.

Of course, not all problems can be solved with algebraic objects and UDF's. Sometimes postfix programs are indeed the only solutions – but not as often as you might think.

So how do you judge which is the most appropriate of the 3 procedures to use?

In general, if the problem you have to solve is one that involves stack and/or object manipulation, or if you're looking for the solution to a non-algebraic problem, a postfix program is called for. Decision making problems (IF this is true THEN do this) are common examples of such circumstances.

However, if the problem to be solved is algebraic or can be used in an algebraic context, then you should probably try User-Defined Functions and/or algebraic objects. And since algebraic solutions are more common and more understandable than postfix programs, most of your solutions will and should be in the form of algebraic objects of one type or another.

Here are a few examples of typical problems – and the optimum form of solution for each:

**Problem:** Convert a three-element vector from rectangular to spherical coordinates.

**Solution:** Since this is the manipulation of a non-algebraic object and requires the use of the stack, a postfix program is called for. Thus:

```
« R→P ARRY→ DROP ROT ROT { 3 } →ARRY R→P »*
```

\* This example was taken from Hewlett-Packard's HP-28C Reference Manual, TRIG.



**Problem:** Find the pressure of 1 mole of an ideal gas whose volume is 1 liter at 273.16K.

**Solution:** Since  $PV=nRT$ , and  $R$  is a constant ( $= 8.314 \text{ J/mol}\cdot\text{K}$ ), if you used the algebraic equation

$$'P*V=n*R*T'$$

in conjunction with the SOLV commands, you could solve this and any variations on this in short order. Videlicet,  $P \times V = N \times R \times T$

ENTER SOLV STEQ SOLVR 1 N 1 V 8.314 R 273.15 T F.

**Problem:** In the manufacture of ball bearings, you would like to determine the physical volume of the spheres and use that formula for volume in other algebraic manipulations (such as the calculation of amounts of materials).

**Solution:** Your choice would be a user-defined function:

$$\< \rightarrow r '4*\pi*r^3/3' \>$$





Press  $\leftarrow$  SPACE  $\rightarrow$  SPACE LC R SPACE  $4 \times \pi \times R^3 \div 3$  ENTER  
 $\text{'S'VOL}$   $\text{STO}$ . Remember to set or clear the symbolic constant flag (see page 214) based on whether you want symbolic or numeric  $\pi$ .


## Appendix E : Keyboard Error Recovery


Anyone who's attempted to do any time consuming thing is grateful for a method of recovering from false starts. No matter how careful you are, there will be times that you'll want to redo, undo or throw away and start over whatever it is you're working on. You'll be happy to know, then, that the HP-28 has a set of "ways out" from false starts and blunders.

### **COMMAND**: The Command Stack

You've already been introduced to the command stack (page 63), and after reading this far you've probably gained some appreciation for its utility.


To cover this ground again, the command stack contains copies of the last four command lines that you  **ENTER**'ed. Repeatedly pressing  **COMMAND** recalls successively older command line copies to the active command line for you to edit and/or re **ENTER**. Pressing  **COMMAND** a fifth time cycles back to the most recent command line copy.


The advantage here is two-fold. First, if you've made a keystroke error in a particularly long or involved command line,  **COMMAND** will allow you to recall that command line if it's not too old, then correct it and re-enter it (note that re-entering simple command lines is often easier than using the command stack).

Second (as in the quiz solution on page 107), you can use the command stack to repeat lengthy and redundant commands. It's also a convenient way to enter a series of slightly different commands (see pages 78-79). Notice that, since immediate-execution commands are not normally recorded in the command line, you may need to do some planning ahead and use  **Q** for this kind of command repetition.

## **UNDO** : UNDOing a Command

Sometimes you will find that you will need to UNDO whatever it is you just did. It may be that you did something that you didn't intend to do, or perhaps that last command ate your only copy of some important datum. Never fear, you've a way out.

Any time you press **ENTER** , or any time you press an immediate-execution key since they effectively "press **ENTER** on themselves" (see page 32), the HP-28 makes a secret copy of any stack Levels that are changed by the invoked commands. The advantage of this is that if you find that you need to undo something, you can press  **UNDO** and the following things will happen:


1. Any and all results of the last command will be dropped from the stack.
2. The stack contents eaten by the last command will be replaced; pushed back onto the stack.
3. UNDO will have amnesia. I.e., pressing  **UNDO** a second time will not undo the UNDO, nor will it repeat its action. It will become active again only after another **ENTER**-pressing command has been invoked.


Things will then be as if you had never invoked that last command.

As you can see, this is tremendously convenient. In fact, UNDO will probably be the most commonly used error-recovery mechanism in your arsenal.

## **LAST**: Recalling the Stack as It Was Before the LAST Command

LAST is a slightly different flavor of UNDO. Here's a list of its actions, so you can compare it with UNDO:

1. Any and all results of the last command will be left on the stack.
2. The stack contents eaten by the last command will be replaced; pushed back onto the stack.
3. LAST won't have amnesia. I.e., pressing  **LAST** a several times will repeat its action. You'll get several copies of the remembered stack levels.

The main use for LAST is when you have an especially gnarly object on the stack and you need to do several operations on it. You don't need to re-enter it. Rather, you could press  **LAST** between operations, to recall the original gnarly object for re-use. Meanwhile, the results of the different operations would be pushed onto higher levels of the stack.

## Enabling and Disabling Error Recovery

A final point of interest about error recovery: you can turn it off. In the second level of the MODE menu are commands for turning on and off each of these error recovery schemes.

But why would you want to turn them off? You never know when you might need them.

The answer is: to conserve memory.

Since each of these mechanisms works by remembering (storing) something in case you want it again, there will be times when it's just plain wasteful to take up memory with something that's only *potentially* useful.

Consider also the case of especially large objects: there may simply not be enough memory to remember the last large object *and* put the next one on the stack, too.

## Editorial

Well now...having dug deep into the nitties and the gritties of the HP-28, you should probably poke your head back out into the fresh air and catch a new perspective (or maybe re-catch an old one). It's too easy to get lost in the details and how-to's and forget the big picture. It would be a shame to lose sight of exciting potentials while mired in the mundanity of getting the basics under your belt.

So in case you've forgotten, it's time you were reminded how much you have to be excited about. If you're a serious problem-solver, by now you should be feeling like a kid in a candy store, or maybe like an auto connoisseur at a new car show. There's so much here, so much you can do, and so many new ways to do it that the mind delights – and maybe boggles a little bit too. That's okay. It's all part of the excitement.

Just remember to be excited.

What you have in the HP-28 is more than the Cadillac of calculators. It's more like being on the freeway at rush hour and finding out that your vehicle can fly. You're no longer bound to the pavement. You don't need to go where everyone else is going before going where you want to go. You have a whole new way to travel. Not only can you get there (wherever there is) faster, easier and more directly than anyone else, you can also go places that they can't.

If you haven't caught the drift by now, here it is: This machine is "radibolical."

You just ain't never saw nothin' like it nohow nowhere before, but you can bet you'll be seeing more of it. It's just too useful and too good an idea not to catch on, and if it doesn't, it's because we aren't ready for it – like di Vinci's helicopter.

Don't be cowed by its power and flexibility. Take it slow and get to know the most capable problem-solver you've ever met. He's a little short on small talk, but in his element, he's "dynobitchin'."

Can you tell we like this machine? And actually, we've realized its flexibility even more during the writing of this book. Back on page 8, we called the HP-28 a problem-solving tool, but you can see now that it's really a *collection* of different tools and attachments – more like a full toolbox, actually.

You can also see how hopeless it would have been to try to cover everything in this book, so, true to our early warnings, we didn't try to tell you how to build a house. The design of your house is your job (but once you decide where and how the boards ought to go together, do we have a toolbox for you)!

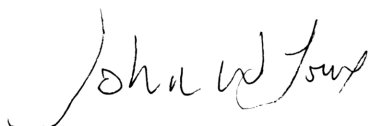
Just remember – with all the real satisfactions you should get from such great tools – you'll be wasting them if you build more house than you need.

This seems to be true of a lot of modern inventions. Two related questions come up over and over again: What peaks of performance can you squeeze out of them? *Should* you push them that far?

Like most machines, the HP-28 answers these two questions very differently, and being so representative of the age, it gives you a chance to begin asking better questions about technology. Instead of idly wondering "How many neat-o-nifty-awesome-but-useless things can I make this little box do?" we hope you'll ask "What better things can I do with the time and energy these tools save me?"

Without sorting out the advisable from the possible, you'll be no better off than before you ever had the tools. Your time and talents will have gone merely to "gee-whiz" tinkering, and you – and the world – will be the poorer for it. A sophisticated machine may be the subject of a course, but it's not the object of the game.

Thanks for listening...and happy hammering!



August, 1987

We hope you've enjoyed this Easy Course book--and that you'll let us hear any comments you may have. Remember: Your response is our only way to know whether or not we have succeeded in what we work hard to do--provide books that are both informative and enjoyable. So please--your opinions (and proof-readings) are welcome--and we always read our mail!

And by the way, if you liked this book, here are some others that you or someone you know might enjoy also:

- \* An Easy Course in Programming the HP-41
- \* An Easy Course in Using the HP-12C
- \* An Easy Course in Programming the HP-11C and HP-15C
- \* The HP-12C Pocket Guide
- \* The HP Business Consultant (HP-18C) Training Guide
- \* The HP Business Consultant (HP-18C) Pocket Companion
- \* Computer Science on Your HP-41 (Using the HP Advantage ROM)
- \* An Easy Course in Using the HP-16C
- \* An Easy Course in Using the HP-28C

You can use this handy set of order forms here --->

Or, you can contact us for further information on the books and where you can buy them locally:

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

**Call: 1-800-338-4331** (in Oregon, 1-754-0583)



# ORDER FORM *(Impress a Friend!)*

**Yes!** Please send:

___ copies	<i>An Easy Course in Programming the HP-41</i>	\$20 ea. = \$ _____
___ copies	<i>An Easy Course in Using the HP-12C</i>	\$20 ea. = \$ _____
___ copies	<i>An Easy Course in Programming the HP-11C and HP-15C</i>	\$20 ea. = \$ _____
___ copies	<i>The HP-12C Pocket Guide</i>	\$ 5 ea. = \$ _____
___ copies	<i>The HP Business Consultant (HP-18C) Training Guide</i>	\$22 ea. = \$ _____
___ copies	<i>The HP Business Consultant (HP-18C) Pocket Companion</i>	\$ 8 ea. = \$ _____
___ copies	<i>Computer Science on Your HP-41 (Using the Advantage ROM)</i>	\$15 ea. = \$ _____
___ copies	<i>An Easy Course in Using the HP-16C</i>	\$20 ea. = \$ _____
___ copies	<i>An Easy Course in Using the HP-28C</i>	\$22 ea. = \$ _____

Note: These prices may change without notice.

## Shipping Information:

For orders consisting of POCKET GUIDES *only* -.....FREE SHIPPING

For orders **less than \$16.00**.....ADD \$1.00 \$ \_\_\_\_\_

For all other orders- **Choose one:**

**Post Office shipping and handling - ADD \$2.00** \$ \_\_\_\_\_  
(allow 3 weeks for delivery)

**UPS shipping and handling - ADD \$3.50** \$ \_\_\_\_\_  
(allow 7-10 days for delivery)

**Note:** UPS will not deliver to a P.O. Box; please give street address.

**TOTAL AMOUNT:** -----> = \$ \_\_\_\_\_

## PAYMENT:

Your personal check is welcome. Please make it out to Grapevine Publications, Inc. *or:*

Your VISA or MasterCard #: \_\_\_\_\_ Exp. date: \_\_\_\_\_

Your signature: \_\_\_\_\_

**Thank You!**

"Please send these books to:

---

Name

---

In Care Of (a company, maybe--or some other person)

---

Street Address (**Note:** UPS will **not** deliver to a Post Office Box!)

---

City

State

Zip

---

( ) -

*Your* Daytime Telephone Number

(Depending upon how it will be shipped, please allow 2-3 weeks for  
delivery of your order.)

# ORDER FORM *(Impress a Friend!)*

**Yes!** Please send:

____ copies	<i>An Easy Course in Programming the HP-41</i>	\$20 ea. = \$ _____
____ copies	<i>An Easy Course in Using the HP-12C</i>	\$20 ea. = \$ _____
____ copies	<i>An Easy Course in Programming the HP-11C and HP-15C</i>	\$20 ea. = \$ _____
____ copies	<i>The HP-12C Pocket Guide</i>	\$ 5 ea. = \$ _____
____ copies	<i>The HP Business Consultant (HP-18C) Training Guide</i>	\$22 ea. = \$ _____
____ copies	<i>The HP Business Consultant (HP-18C) Pocket Companion</i>	\$ 8 ea. = \$ _____
____ copies	<i>Computer Science on Your HP-41 (Using the Advantage ROM)</i>	\$15 ea. = \$ _____
____ copies	<i>An Easy Course in Using the HP-16C</i>	\$20 ea. = \$ _____
____ copies	<i>An Easy Course in Using the HP-28C</i>	\$22 ea. = \$ _____

Note: These prices may change without notice.

## Shipping Information:

For orders consisting of POCKET GUIDES *only* -.....FREE SHIPPING

For orders **less than \$16.00**.....ADD \$1.00 \$ \_\_\_\_\_

For all other orders- **Choose one:**

**Post Office shipping and handling - ADD \$2.00** \$ \_\_\_\_\_  
(allow 3 weeks for delivery)

**UPS shipping and handling - ADD \$3.50** \$ \_\_\_\_\_  
(allow 7-10 days for delivery)

**Note:** UPS will not deliver to a P.O. Box; please give street address.

**TOTAL AMOUNT:** -----> = \$ \_\_\_\_\_

## PAYMENT:

Your personal check is welcome. Please make it out to Grapevine Publications, Inc. *or:*

Your VISA or MasterCard #: \_\_\_\_\_ Exp. date: \_\_\_\_\_

Your signature: \_\_\_\_\_

**Thank You!**

"Please send these books to:

---

Name

---

In Care Of (a company, maybe--or some other person)

---

Street Address (**Note:** UPS will **not** deliver to a Post Office Box!)

---

City

State

Zip

---

( ) -

*Your* Daytime Telephone Number

(Depending upon how it will be shipped, please allow 2-3 weeks for  
delivery of your order.)

# An Easy Course in Using The HP-28C

This cover flap is handy for several different things:

- Tuck it just inside the front cover when you store this book on a shelf. That way, you can see the title on the spine.
- Fold it inside the back cover--out of your way--when you're using the book.
- Use it as a bookmark when you take a break from your reading!



# **An Easy Course in Using the HP-28C**

If you're looking for a clear, straightforward explanation of the powerful HP-28C, then this is your book! Authors Loux and Coffin sort through the myriad features of this machine, giving you the pictures and the practice you need to make the HP-28C your favorite calculating tool.

The first several chapters bring you up to speed on the mechanics of operation – what keys you need to press to control and command the display, the stack, and the menus. You'll get lots of practice problems and explanations designed to get your fingers trained for action.

Then you go straight to the heart of the machine, exploring all the different information "objects" and how you can manipulate them, combine them, name them and (best of all) think about them. You'll see how HP's well-known stack-oriented (postfix) arithmetic becomes the engine behind all this math power, and soon you'll be harnessing it for yourself!

Then in brief separate discussions, this Easy Course touches upon specialized topics, such as symbolic algebra and calculus, plotting, and programming.

It's all in Grapevine's familiar Easy Course format, with Robert Bloch's illustrations – a book filled with examples, review questions, and quizzes, designed to let you work at your own speed (and your own speed will soon amaze you)! It's always a pleasant surprise that learning about a calculator can be this satisfying – but it can be, when the right explanation transforms a mysterious machine into a friendly and powerful tool.



**FROM THE PRESS AT  
GRAPEVINE PUBLICATIONS, INC.**

P.O. Box 118 • Corvallis, Oregon 97339-0118 • U.S.A. • (503) 754-0583



ISBN 0-931011-17-5