

An Easy Course In

Using The HP-42S



A GRAPEVINE PUBLICATION

By Dan Coffin and Chris Coffin
Illustrated by Robert L. Bloch

An Easy Course In Using The HP-42S

by Dan Coffin
and Chris Coffin

Illustrated by Robert L. Bloch

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

Acknowledgements

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

© 1989, 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, electronic or mechanical, without written permission from Grapevine Publications, Inc.

Printed in the United States of America

ISBN 0-931011-26-4

First Printing – August, 1989

Notice Of Disclaimer: Grapevine Publications, Inc. makes no express or implied warranty with regard to the keystroke procedures and program materials herein offered, nor to their merchantability nor fitness for any particular purpose. These keystroke procedures and program materials 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 and program materials prove defective, the user (and not Grapevine Publications, Inc., nor any other party) shall bear the entire cost of all necessary correction and all incidental or consequential damages. Grapevine Publications, Inc. shall not be liable for any incidental or consequential damages in connection with, or arising out of, the furnishing, use, or performance of these keystroke procedures or program materials.

CONTENTS

Start Here.....	10
------------------------	-----------

Before You Get Your Hands Dirty	12
--	-----------

What's In This Machine?	13
What's <i>Not</i> In This Machine?	13
What's In This Book?	14
What's <i>Not</i> In This Book?	15

A First Tour Of Your Workshop	16
--	-----------

The Big Picture	17
The <input type="checkbox"/> ON Button	20
The <input type="checkbox"/> OFF Button.....	20
Starting With A Clean Shop	21
The Display: Your View Of The Workshop.....	22
The Viewing Angle	22
Your Display's Two Lines	22
The Message Display	23
The Stack Display	23
The Menu Display	24
The Program Display	24
Where Are You?	25
The Annunciators	26
How Many Digits Do You Have?	27

[DISP] Your First Menu: How Many Digits Do You See? ...	28
The Keyboard	30
The Arithmetic Keys: Your "Nuts 'n' Bolts"	30
Menus And Their Pages	31
Data In The HP-42S	34
The Stack Registers	35
The Alpha Register	35
Data Types In The HP-42S	36
Containers And Labels For Your Data: Variables And Their	
Names	37
Real Variables	38
ALPHA Variables	38
Complex Variables	39
Matrix Variables	40
Storage Registers: The REGS Matrix	41
Programs	42
Reviewing Your Storage Area	43
The [CATALOG] Menu	43
The [CLEAR] Menu	44
Pop Quiz	45
Pop Answers	46

Tools! Glorious Tools!48

Your Favorite Tools: Arithmetic And The Stack	49
RPN	50
The Stack	52
What Is Stack-Lift?	53
Putting Two and Two Together	56

Chain Calculations	58
Changing The Sign Of A Number	62
That [E] Key.....	63
The [←] Key And [CLW]	64
Moving Numbers Around In The Stack	68
The [R+] Key	69
The [X↔Y] Key	70
Notes, Doodles, And Stack Practice	71
The L-Register	72
The [LAST X] Key	73
Reality Check.....	75
Real Answers	76
 Bigger Hammers: Higher Math And Number-Crunching	78
Exponentiation: Roots, Logs And Powers.....	79
Common Trigonometry	82
The [MODES] Menu	85
The [CONVERT] Menu	88
The [PROB] Menu	90
A Deep Breath.....	92
 A Complete Inventory Of Tool Menus	93
Special Use Menus.....	95
The [ALPHA] Menu	96
The [ASSIGN] And [CUSTOM] Menus	98
Your Tools Can Work Together.....	102
The [TOPFCN] Menu	103
Tool School Reviewal	105
Tool School Reviewal Solutions	106

YOUR RAW MATERIALS:

Data And Variables110

Handling Data.....	111
How Do You Name It?.....	111
All About [STO]	114
All About [RCL]	115
Real Variables	120
ALPHA Data And The ALPHA-Register.....	121
Complex Numbers	126
Seeing Vs. Doing: The Difference Between The [MODES] And [CONVERT] Menus.....	131
Entering Complex Numbers	132
Using Complex Numbers	134
What Are Complex Numbers Good For?.....	136
Matrices	140
What Are Matrices Good For?.....	141
Entering Matrices	142
Storing Matrices	145
Editing Matrices	146
Complex Matrices	148
What You Can Do To A Matrix	149
Special Matrix Functions.....	151
What You Can Do With Two Matrices	155
Notes (Yours).....	159
Variable Vexation.....	160
Victory Versus Variable Vexation	162

BUILDING NEW TOOLS: Programming....166

Programs: A Different Kind Of Tool	167
A Little Memory Reminder	168
■ PRGM And Program Mode	170
The Program Display Revisited.....	171
The "No-Frills, Straight-Arrow" Program	173
How To Start A Program	175
How To END A Program	178
Using Your First Program.....	180
Editing A Program	181
Moving Around In Your Program.....	183
What Do You Know?.....	185
Variables And How To Use Them	186
Using the INPUT Function.....	188
Getting Your Program To Talk Back	192
PROMPTing For ALPHA Data: MON and MOFF	196
Variable Menus.....	199
Creating Two-Line Messages	204
Review	209
Preliminary Programming Probe.....	210
P.P.P. Answers	212
 Using Labels For All They're Worth.....	 218
Global Vs. Local Labels.....	219
Which Type Of Label To Use	222
Branching Out With GTO and XEQ	223
Routine Subroutines	226
Selective Branching: The Programmable Menu.....	229
Multi-Page Programmable Menus	235

Letting The Calculator Decide What To Do	241
Conditional Tests	242
Flags	246
MODE And Flag 21	250
Testing The Data Type	251
Loop Counters	252
Indirect Addressing.....	260
Using Indirect Addressing To Make Decisions	262
Reviewing Your Programming Tools	266
Prove Your Programming Proficiency	268
Programming Proficiency Proofs	270
 Programming Big: A Case Study	 276
The Challenge	278
Stage One: Planning.....	279
Stage Two: Writing The Program Section by Section	284
Stage Three: Debugging.....	302

SOLVING PROBLEMS:

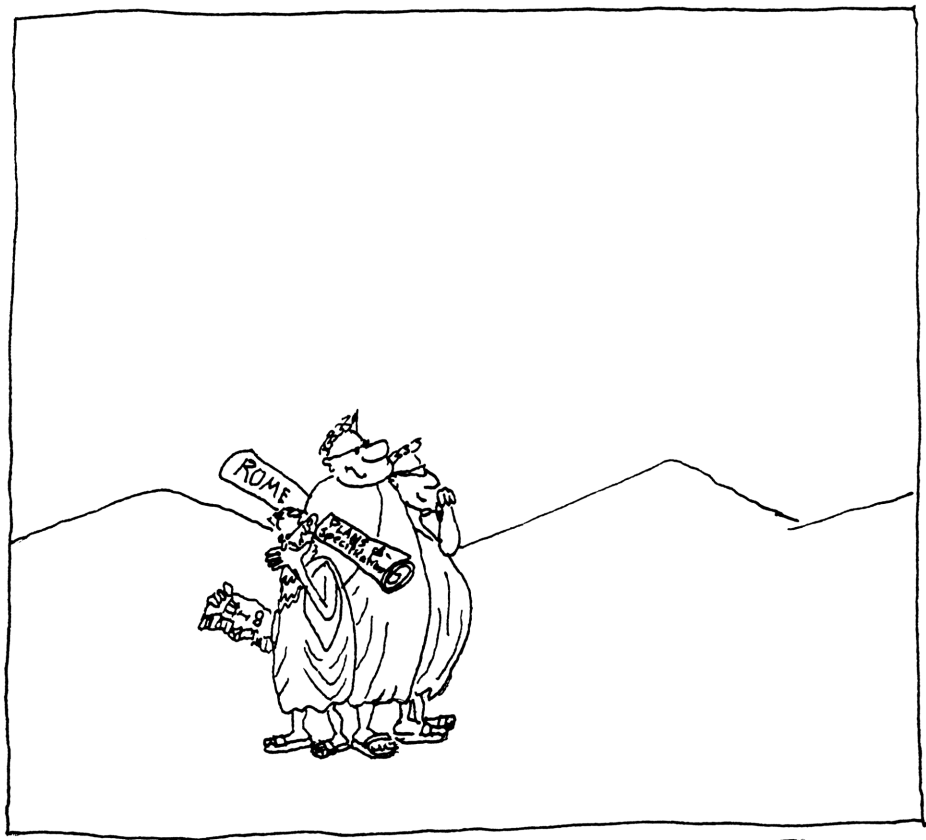
Using Your Power Tools304

The MODE Menu.....	306
Negative Numbers In The MODE Menu.....	310
Fractions And Integers And Bases	313
The UNIT Menu	314
Solving Problems in MATRIX	317
Matrix Multiplication	317
Simultaneous Equations	319

The STAT Menu	322
What Are Statistics, Anyway?	323
Entering Some Statistical Data.....	325
Checking and Editing Your Data	328
Just Plain Mean	330
A Moving Average	331
A Weighted Mean	332
Deviations From The Average	335
Two-Variable Statistics	339
Getting Trendy: Regression and Estimation	340
Solver And Integrate: Power Tools That Use Programs	350
The f(x) Menu	351
Spikes, Asymptotes, And Other Beasties.....	356
The SOLVER Tool	359
What SOLVER Is Really Doing.....	362
Guess the Answer: Telling SOLVER Where To Look ..	363
Don't Be Misled By False Solutions	364
The Good News	365
Powerful Problems	373
Powerful Solutions	374

Foundation Completed380

"The Book Stops Here"	381
Comments, Books And Order Forms	382



START HERE

...at the very beginning...with enthusiasm, energy and a dream or two....

Yes, there's a lot to be done – and a lot to learn – but remember that the hardest part of learning anything new is simply

Beginning

This means you've just finished the hardest part of this Course!

But you're not quite *completely* finished yet. You've begun with a powerful problem-solving workshop (your HP-42S), and now, with this Easy Course book, you're going to master that workshop and its tools.

So follow through on the Course you've started. It is an "*Easy Course*," but it's still a Course, and it's been carefully planned – so follow it faithfully and let it be your guide and friendly mentor.

It's not a long apprenticeship – just a few good hours. But *do* it! The rewards will definitely be worth your while; when you finish, you'll know enough to solve problems that haven't even been invented yet!

How's that?

It's like this: The goal here is *not* just to leave you knowing how the buttons work or remembering a handful of keystrokes and formulas. You're going to get a *fundamental understanding* of problem-solving, so that whenever you encounter *new* problems, you'll be able to analyze and solve them confidently. No more the apprentice, you'll be the master of your own workshop!

Now then....



Before You Get Your Hands Dirty

What's In This Machine?

Inside the HP-42S you'll find a wonderland of tools that are useful in solving real-world mathematical problems of nearly every kind. And you can use these tools to build even more wondrous tools – to get you to those "hard-to-reach" places. Congratulations on a wise choice!

Now you simply need to learn how to use and master the tools and possibilities your HP-42S offers.

What's *Not* In This Machine?

There's no English-speaking person inside your calculator (one of these days, though...). You still need to "speak" your machine's "language" in order to *translate your everyday calculation problems into shapes and forms it can understand*. These are the skills you've chosen to learn.

Your part of the work is in *defining* the problem correctly for yourself and then *restating* it for your calculator. After all, only a human being can really understand our human world (and even then it's tough sometimes)!

This translation skill isn't really very hard to learn. It's a lot like learning to use any sophisticated tool – like a typewriter, a car, a lathe or a forklift. Everything seems strange at first, but after some training and practice, you'll be doing things quite proficiently – without even thinking much about them! And that's where this book comes in....

What's In This Book?

This book is a sort of a workshop apprenticeship.

As your mentor, this Course will introduce you to the tools available in the shop, watch over you as you use the power tools for the first time, demonstrate some of the more useful operations that a Craftsman has to know, and make sure that you know how to keep your working environment clean, orderly and inviting.

During your apprenticeship with the HP-42S, you'll encounter lots of explanations, diagrams, pop quizzes and answers, as well as useful advice about visualizing your work. Ultimately – as any good mentor does for his apprentice – this book should inspire you to build powerful *new* tools that move you into your own "brave new world" of problem-solving.

So, what makes this Course so Easy?

It's because you can control how fast you learn things. You needn't worry about showing off or competing with anybody. You can repeat your lessons whenever and as often as you wish – without making your teacher the least bit grumpy. After all, it's *your* Course!

What's *Not* In This Book?

You won't find discussions of *everything*, because some parts of your HP-42S just don't need much explanation at all. Included here, therefore, are just those things that most "craftsmen" tend to need some practice on – *most* of the time.

For example, you'll get an overview of the higher math tools available on your machine, but not an exhaustive and detailed survey of them; some of those tools will be discussed more fully than others. This isn't because those tools are any more important in the Great Scheme Of Things; rather, it's because they're the most complicated to learn how to use correctly, so they deserve more space in the book.

By contrast, some other menus tend to explain themselves after a few uses of them, so that's what you'll do here – just use them a few times – as a reminder that they're available if you want them.

Anyway, if you want a *complete* description of *everything*, you already have it right there in your Owner's Manual. That's what it's there for – as a reference manual to let you conveniently look up keys, functions, examples, and generally browse through the Great Unknown.

But this book you're now reading is *not* a reference manual; it's an entirely different approach, with a careful selection of topics that are meant to be taken in order. So start here at the beginning and stay on Course!

Now, roll up your sleeves, flex your fingers, oil your thinking cap and...go....



A First Tour Of Your Workshop

The Big Picture

The best place to begin your first tour is to look at the Big Picture – the "floorplan" of your HP-42S workshop.

At first glance, such an opening view of "everything-in-the-HP-42S-all-at-once" may overwhelm you with its size and variety. Don't worry: you'll come to know and love this floorplan as you familiarize yourself with the workshop. You'll be introduced to each nook and cranny in the workshop – one piece at a time. And as you accumulate more and more understanding of these details, you'll easily keep track of where you are by glancing back at the Big Picture from time to time.

Now, *before you turn the page*, consider what you're likely to see: What would you expect to find in a craftsman's workshop?

- There would be a *work area* – an area designated for actually building all your projects;
- There would also be *tool storage areas* – say, one area for hand tools and one area for power tools;
- There would be an *area to store the work and materials themselves* – both finished and unfinished projects.

OK...turn the page and see your HP-42S workshop....

The HP-42S Workshop

Work Area→

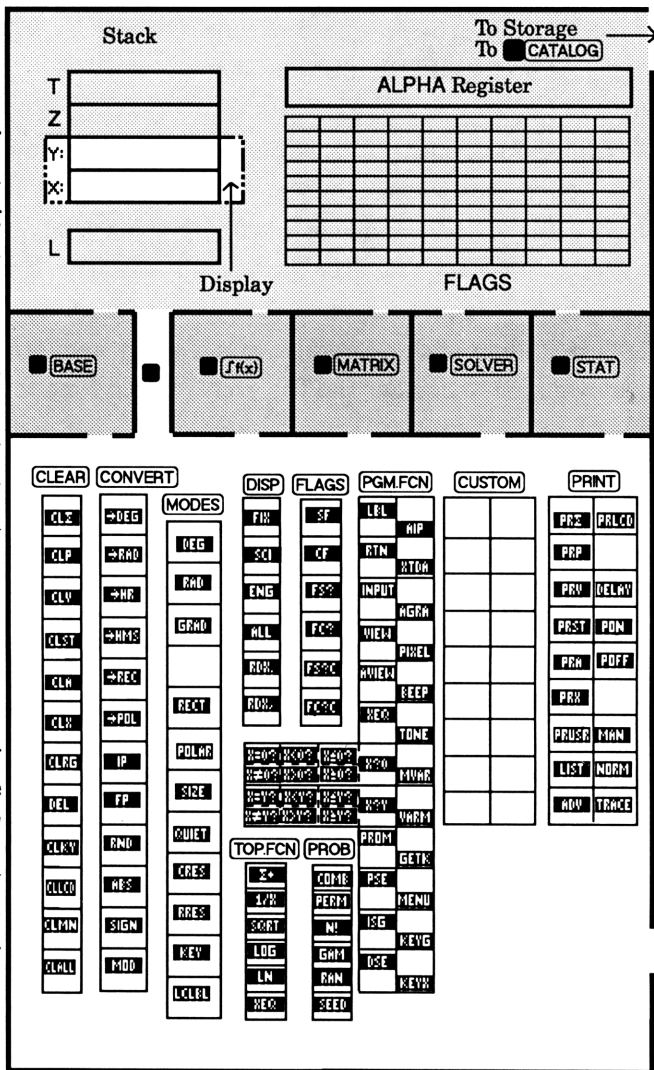
You'll be spending a lot of your time here, with the Stack acting as your "workbench."

PowerTools→

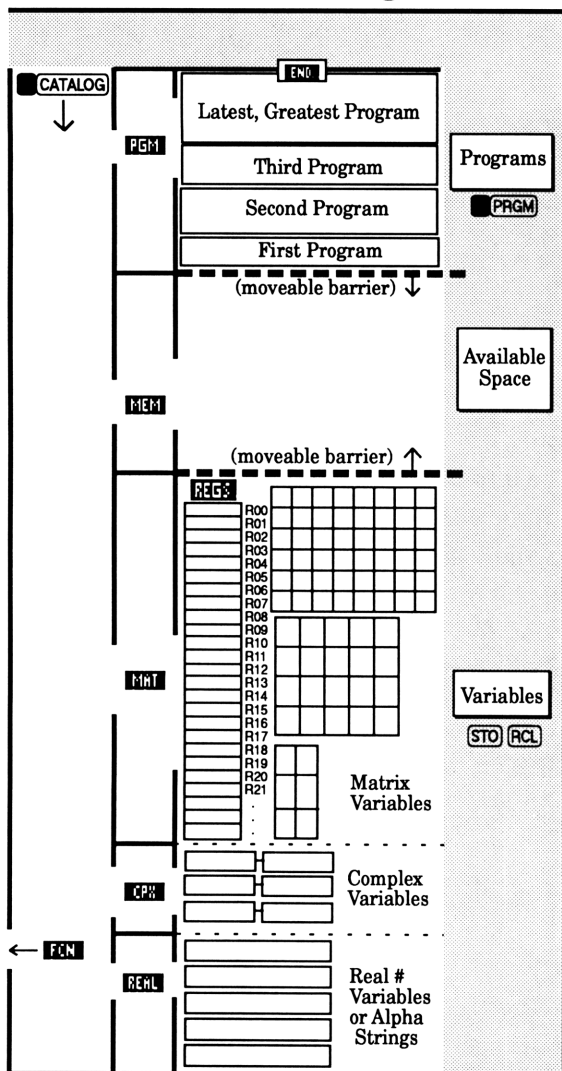
These are complex and powerful tools, each in its own "room."

Hand Tools→

These tools perform most of the "simple chores" and are stored on "shelves" (i.e. in menu collections) here.



The HP-42S Storage Area



- The memory available in your HP-42S Storage Area will depend on the numbers and sizes of *programs* you've built and stored.
- The **CATALOG** allows you to "see" what you have in each storage area, and it also shows how much memory space is available.
- The data *variables* you store (of matrix, complex, real and ALPHA data types), also affect the available storage space:
- The **CATALOG** key also lets you view all the available hand tools.

The **ON** Button

You may have noticed from studying the Big Picture that your workshop apparently has no outside doors.

You'd better find the entrance: To get into your workshop, press the button in the lower left corner of your calculator – the button marked **EXIT**, with ON written in silver below it and OFF written in orange above it.

"How does it know which of the three instructions I mean?"

Well, if you push the button when the calculator is turned off, it acts like the **ON** button. You "go into the workshop." If you push the button when the calculator is turned on already, the button acts as the **EXIT** key (details about the **EXIT** key later.)

The **OFF** Button

Is your calculator on? All right, now turn it off again, by pressing the **OFF** key...

Hmm...the machine doesn't turn off simply by pushing that same **EXIT** key again, does it?

Since the word OFF is written in orange above the **EXIT** key, you need to press the blank, solid orange "shift" key (■) before pressing **EXIT**. Now it works, right?

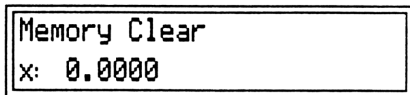
Starting With A Clean Shop

OK, press **[ON]** again. Now, the best way to start making sense of that Big Picture is to look at your HP-42S display....what do you see?

That's hard to say, actually. Since your HP-42S has continuous memory, it will look now just as it looked when you last used it. So, to be sure that you're starting with a clean slate, here's how to reset the machine.

Warning: You're only doing this now to be sure that you can follow along on this Course. This procedure completely wipes out all meaningful information in the machine. This is *not* something you want to do very often. There are plenty of other, safer ways to selectively erase parts of its memory. *This* reset is an Equal Opportunity Erasure! It erases everything without regard to its importance or value!

Drum Roll, Please: Press *and hold down* the **[EXIT]** key. While holding it down, press *and hold down* the **[Σ+]** key (upper left corner). While holding *both* those keys down, *press and release* the **[XEQ]** key (upper right corner). Now release the **[Σ+]** key, then the **[EXIT]** key. You should see this:

A rectangular display box with a double border. Inside, the text "Memory Clear" is on the top line, and "x: 0.0000" is on the bottom line. The text is in a monospaced font.

(If this doesn't work the first time, just try again.)

The Display: Your View Of The Workshop

To begin to understand and use your HP-42S workshop, you need to understand the display and what it's showing you. The HP-42S display is indeed your view of your tools, projects and work areas.

The Viewing Angle

Can you see and read the display comfortably? The LCD (Liquid Crystal Display) is hard to read if you're not looking at it from a certain angle – but you can adjust that angle to whatever's best for you!

Try It: With the machine still on, press and hold down the **[EXIT]** key. Then press and hold the **[+]** or the **[-]** key and watch how the display's best viewing angle varies....

The Display's Two Lines

There you see it, a machine with utter amnesia. But notice that even now, in its emptiest state, your HP-42S uses its two-line display to tell you what's happening – so when in doubt, look at the display for clues!


With two lines of display, the HP-42S offers more power and flexibility (and possible confusion) than do calculators with one-line displays. So it's time to learn what the HP-42S is telling you with its display.

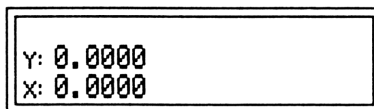
The Message Display

This kind of display is what should be showing right now: the top line has a message (Memory Clear) and the bottom line shows X: 0.0000

A message itself is a temporary note that simply covers over part of some other display – notifying you or instructing you to do something. After you follow its instructions or acknowledge it, you'll once again see what's underneath. For example, that 0.0000 on the bottom line is part of "what's underneath" right now – the part that isn't being covered by the message.

The Stack Display



Now, press the  key to clear the message line. You should see:




This is the second type of display, the Stack Display. By clearing the message, you've now uncovered the top line of this display.

These two numbers are the ones that your calculator happens to be ready to work on at the moment. You can think of them as "sitting on the workbench" right in front of you in a sort of stack, waiting for you to grab a hand tool and start pounding or cutting on them or something (and of course, you'll learn all about the relationships between the X: and Y: numbers and the rest of the Stack later in this chapter).


The Menu Display

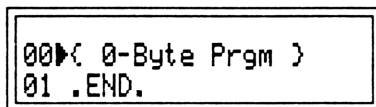
You'll find a third type of display by pressing  CATALOG (i.e. the solid orange key, then the  key with the orange CATALOG written above it). You should see the following Menu Display:



A Menu Display is similar to a Message Display in that it usually covers up or displaces a part of another (underlying) display. But a Menu Display uses the *bottom line* of the display; and instead of a message, it shows a set of items from which you select to help you in your work on the top line. For example, the Menu Display above shows part of a Stack Display on the top line; the selections offered on the menu (bottom) line are meant to help you view and use certain tools (to get back to the full Stack Display now, press the  key).

The Program Display

The fourth type of display shows the program you're currently building or editing. You see a Program Display by pressing  PRGM (do it now):

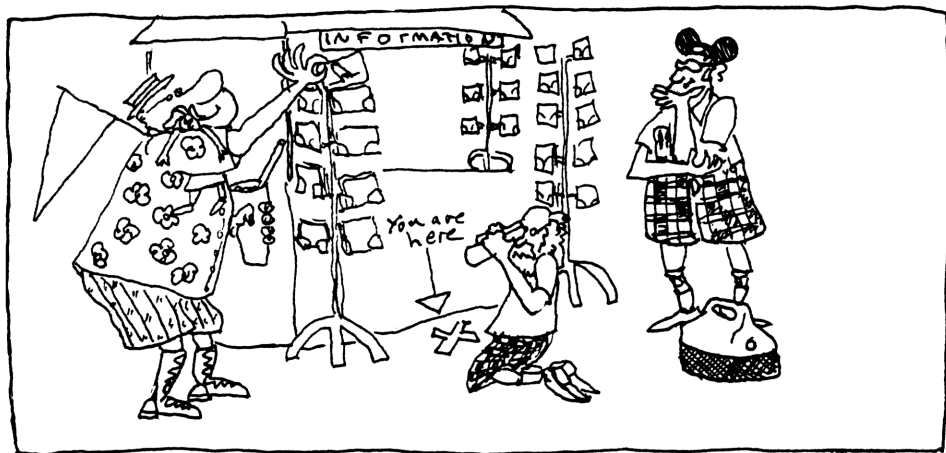


Again, don't worry yet about what the information in this display is. This is just to get you familiar with the *types* of displays you'll be seeing.

Where Are You?

The whole point of these first lessons on the display is: *Always let the display orient and remind you "where you are" in the workshop:*



- If you see the Stack Display, you're at your "workbench," preparing to do some calculations.
- If you see the Program Display, you're building or editing a program over in the program storage area. To go back to the Stack Display, just press the **[EXIT]** key (do so now if you still see a Program Display).
- If you see a Menu Display, it means that, besides working in the Stack or in a program, you're also using one set of tools. To get rid of the Menu Display (i.e. to return that set to its storage bin and go back to your "workbench" – just the Stack Display), use the **[EXIT]** key.
- If you see a message display, this is only a *temporary note* covering part of another display. The **[X]** key will clear the message.




The Annunciators

Besides the two full lines of information in your HP-42S display, you also have some space above them. This space is for the *annunciators*.

"Fine. ...What's an Annunciator ?"

Press the  key and look at the display above the top line. You should see a small upward pointing arrow: .

Now press the  key again while you watch the display. The arrow went bye-bye. This arrow is a good example of an annunciator.

As you might guess from its name, an annunciator "announces" something. In this case, the arrow is announcing that you're "shifting" the meanings of the keys on the keyboard – shifting to the meanings listed in orange lettering above the keys. Because this is similar to a "shift key" on a typewriter, this annunciator is called the "shift" annunciator.

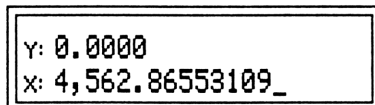
Of course, there are lots of other annunciators in the display, and it wouldn't pay to try to learn them all now. Since you'll run across most of them during this Course, you might as well wait until you see each of them in action. For now, just realize that you may occasionally see something in the display besides the stuff on the two lines.

How Many Digits Do You Have?

You may have wondered why your display shows exactly four decimal places in every number. This is usually enough for most technical calculations – and HP had to decide on *some* such display setting for the HP-42S to "wake up" in after a total memory reset (recall page 21).

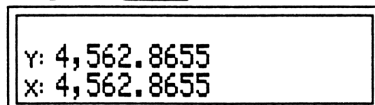
"But is this the best precision you can get?" Not at all. In fact, *every* number carries 12 digits of precision, *no matter how many you can see at the moment*.

For Example: Key in this number: 4562.86553109. You'll see:



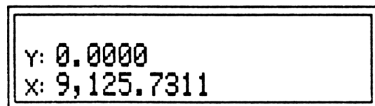
Y: 0.0000
X: 4,562.86553109_

Now press **(ENTER)**:



Y: 4,562.8655
X: 4,562.8655

Was the 0.00003109 lopped off by the calculator?
Find out by pressing **(+)** to add these two numbers:



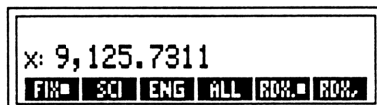
Y: 0.0000
X: 9,125.7311

The machine kept all 12 digits, even when you couldn't see them (if it hadn't, you'd now see 9,125.7310).

"OK, so how do you tell the machine to display more (or fewer) digits?"

DISP Your First Menu: How Many Digits Do You See?

Since the display shows you only *requested portions* of the full 12-digit numbers, you have certain tools you can use to "change your request." To bring these DISPlay control tools to your "workbench," press the **DISP** key. You should see this *menu*:

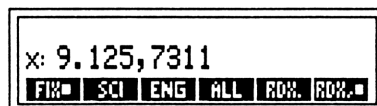


Notice a few things here: Notice the little boxes in the menu items that look like this: **FIX** and **RDX.**. Those boxes remind you which "choices" you've currently selected. Right now, the **FIX** tells you that you have a "FIXed" number of decimal places (and you can see how many digits this is simply by looking in the number above).

The **RDX.** stands for "RaDiX," which is the technical term for the decimal point. At the moment, it is a point. In Europe, they would choose the other option, a comma.

So how do you choose which options you want? How do you use a menu?

Like This: Press the **DISP** keys, then the **XEQ** key. What happens?



The decimal point and the comma exchanged places, and the little box is now next to the **RD%**, indicating that the radix is now a comma. So, press **[LN]** to change it back....

"Why the **[LN]** key?" Because that's how the menus work: *Whenever you see menu items in the display, they are aligned and matched with the keys in the top row of the keyboard* (directly beneath them). You choose menu items by pressing the corresponding top-row keys!

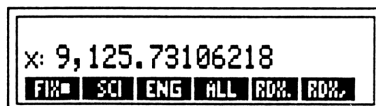
Try Some: Change the number of fixed decimal places from 4 (where it is currently) to 5. Choose **FIX** (by pressing the **[Σ+]** key). What happens? You get a Message Display, this one prompting you to *do* something.

Such prompting messages are usually associated with Menu Displays – as in this case. Here it's asking you for how many decimal places you want to show to the right of your decimal point. Press **[0][5]** (fill up both of the provided spaces) and voilá...you have five decimal places.

Now set your display's notation to **FIX** eight decimal places.

Solution: Select **FIX**, then **[0][8]**.

Result:



The Keyboard

Of course, there's a lot more to using this calculator than just knowing what the display is telling you. It's now time to look at all the keys (but if you already know these keyboard basics, then jump now to page 34).

The Arithmetic Keys: Your "Nuts 'n' Bolts"

Look first at the white writing on the actual key-faces themselves. See how the digit keys (0-9) and the four arithmetic functions (+, -, ×, and ÷) are all conveniently placed together on the lower four rows? After all, you shouldn't need to search around just to add two and two!

But (as you'll soon see), there are other keys that you could consider as being arithmetic/math keys also: There are the ENTER, +/−, and E keys, and the ⇐ key is for correcting mistakes (just in case you should ever make one). Then the *top two rows* contain other mathematical functions common to most scientific calculators (e.g. SIN, COS, √x, LN).

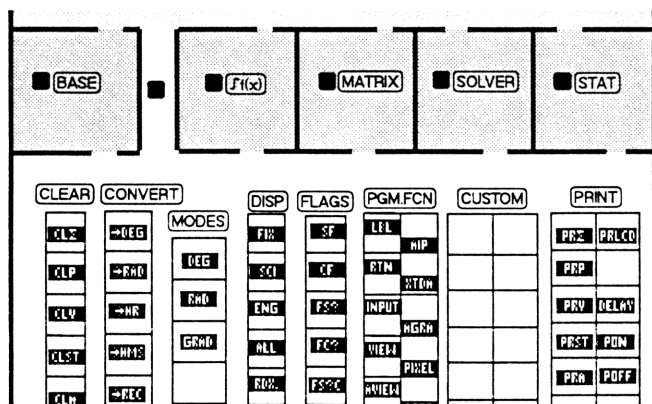
And don't forget the shift key. You've already been introduced to this plain orange key, (which will appear in this book as ■). It activates all the orange-printed functions on the keyboard – similar to the shift key on a typewriter. But *unlike* a typewriter, you don't hold the ■ key down while pressing the orange-printed function you're after. Just press *and release* ■, and then press the key you want. The ⇑ will always tell you when the "shift" is in effect. (Notice also that the shift is cancelled after every use; you need to re-press the ■ for every orange-printed function you want, because ■ is not a "Caps Lock").

Menus And Their Pages

All right, so far, you've looked at the display and keyboard of your HP-42S, connecting what you see on your machine with the Big Picture. Next up: menus and how they work....

As you can tell by looking at the Big Picture (on pages 18-19), most of the orange functions in the lower part of the keyboard are names of "storage shelves" in the your workshop. Here's where you'll find many useful *collections* of "hand tools."

Each such collection is a set of *related* tools (such as the DISPlay tools, for example), stored on its own shelf and marked with an appropriate name. That name appears (usually printed in orange on the keyboard) as a *menu* key. Therefore, in the Big Picture, each "shelf" in the Hand Tool Storage Area is headed by the name of a menu key:



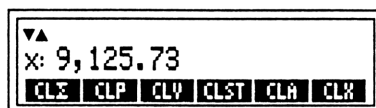
In the next few pages, get acquainted with each of these menu keys on your HP-42S (and though you've already seen the **DISP** Menu, you'll review it now, too, because there are some other points to learn)....

F'rinstance: Make sure you see the Stack Display (press **EXIT** until you do). Now go to the **DISP** Menu and **FIX** the number of decimal places at 2: **DISP** **FIX** **0** **2**.

Now, notice that you're automatically returned to the Stack Display after you finish with that tool in the menu. *But*, if you press **DISP** **DISP** **FIX** **0** **2**, you'll need to press **EXIT** yourself in order to return to your work area. Try it! Anytime you don't want to return automatically to the Stack Display after a single use of a tool in a menu, press the menu key *twice*.

All menus work similarly to the **DISP** Menu, but as you can see from the Big Picture, some menus are much larger – so there's one more thing to...

Notice: Look at the **CLEAR** Menu (press **CLEAR**):



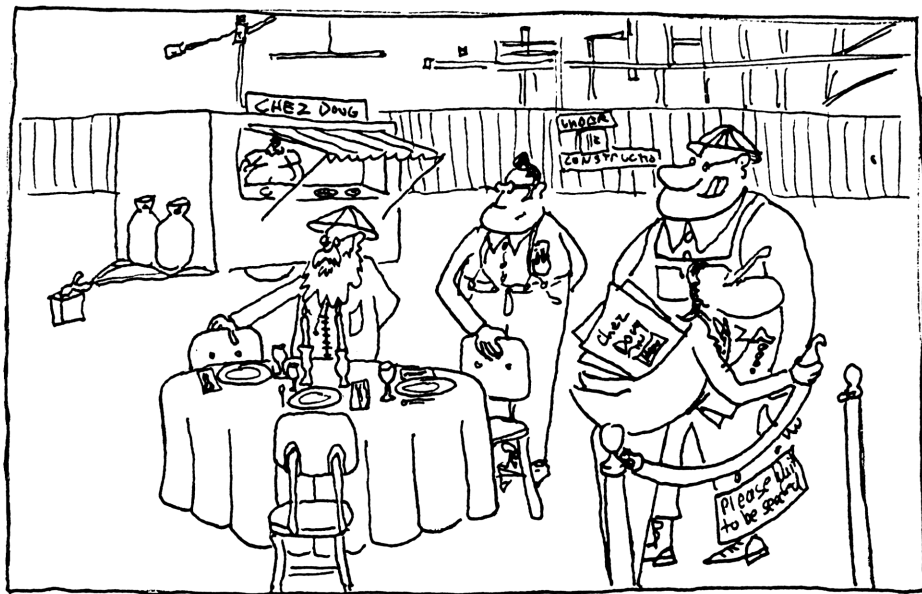
Only six menu choices are visible, but there's also an annunciator (▼▲). The ▼▲ means there are more choices available but not visible. To find these other choices, use the ▲ and ▼ keys (just above the **key**)....

Get the idea? Try a few more...

Examples: Press **CLEAR** again to keep you from automatically returning to the Stack Display. Now "page" through your menu choices by pressing the **▲** key several times....

Obviously, there are just two "pages" in the Clear Menu, and they just keep alternating in the display as you push the **▲** key. That is, the listing is circular— you just keep going around and around on the list as you push **▲** (and the same is true of the **▼** key). And each page in a menu can show only six choices, but from the Big Picture (page 18), you know there are twelve choices available under **CLEAR**, so it makes sense that this is a two-page menu.

Now look at the **PGM.FCN** "shelf" in the Big Picture. How many pages do you expect to find in that menu? Test your answer by trying it on your calculator.



All right, at this point, you've had your first look at the display, the keyboard and menus. It's now time to consider all those storage bins in your HP-42S Workshop Storage Area (see the Big Picture, page 19).

Data In The HP-42S

Many of the bins in your HP-42S Storage Area are for storing various kinds of *data*. And what's data? "Data" as it's used here is just another name for computer information, namely, "numbers and characters."

You probably have a good feel for what a number is: it's a *quantity* with which you (and your HP-42S) can do numerical calculations. In this sense, it's an abstract idea; how you *represent* this idea to your own eye and mind is irrelevant. The quantity "twelve" can be represented as IIII IIII II or 12 or XII. The *idea* of the quantity you're talking about is always the same, no matter what symbol you use.

A character, by contrast, *is* the visual *symbol* – a letter, punctuation mark, numeral – anything you'd find on a typical typewriter keyboard. And if you're talking about the *numerals* "12", these are simply two symbols that happen to be set beside each other – much like "AB". Characters do *not* mean numbers to the HP-42S; the *numerals* "12" do *not* mean the *number* twelve.

You can also combine or *link* numbers and characters into other data structures or types. For example, if you link two numbers together in an ordered pair, you get a vector. If you link two characters, you get a string. By thus *linking* simpler data types, you make more intricate (and presumably, more useful) data types.

With that in mind, consider the various storage bins in your workshop:

The Stack Registers

The Stack registers are bins named with letters – X, Y, Z, T, and L. These registers are the "workhorses" of your workbench area. Here's where you bring data to work on them. If you look at the Big Picture, you'll see the reason why this set of five registers is called "the Stack:" they're stacked on top of one another and are linked together in a special way that makes "data-crunching" easy and efficient.

The ALPHA Register

With the help of this register, the computer can store and manipulate characters (also called ALPHA characters to remind you that they are indeed only letters and symbols – not numerical quantities).

The ALPHA register can hold up to 44 ALPHA characters at one time – *but it can never hold numbers*. You can't add, subtract, or otherwise use a "12" in any math calculation while it's in the ALPHA register. If you need to do math with the number twelve, you'd need to convert the ALPHA *string* (i.e. the combination of ALPHA characters), "12", into a true number, 12.00, in one of the Stack registers.

Now...what kinds of data types can you build and store, using simple numbers and characters – and using the Stack and the ALPHA register as your "working containers?"

Data Types In The HP-42S

Your HP-42S will let you build and store these four types of data:

1. Real numbers
2. ALPHA strings
3. Complex numbers
4. Matrices

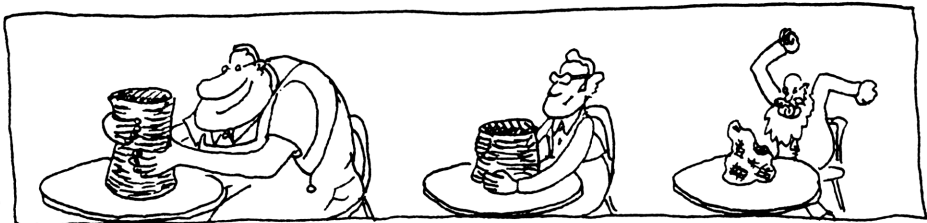
1. A *real number* data type is just a single real number – plain and simple;
2. An *ALPHA string* data type is a set of ALPHA characters – at least one but no more than six such characters;
3. A *complex number* data type is set of two real numbers, linked and ordered to represent the real and imaginary parts of a complex number. You can think of the complex data type as *linking together* simpler data types (real numbers);
4. A *matrix* data type is an ordered set of numbers, grouped in rows and columns. In the HP-42S, you can have matrices filled either with all real numbers or all complex numbers. You can think of the matrix data type as *linking together* simpler data types (real numbers or complex numbers).

These are the *forms* in which you can store and manipulate data. But you might have lots of examples of each form – so you need a way to store and identify each datum....

Containers And Labels For Your Data: Variables And Their Names

Whenever you build a datum, it will have one of those four data types. And if you want to store it somewhere, that "somewhere" will be a *variable*. As you can tell from the Storage Area in the Big Picture (page 19) *variables are erasable, namable storage bins*. They *vary* because you can delete them altogether – or change their names or sizes.

That's is an important distinction between a variable and a register: *A register is a built-in storage bin* – a named space (usually a space for "working in"), which is reserved and labelled with a certain name. For example, you've already seen the Stack registers and the ALPHA-register. They are indeed registers, for although you can clear these bins of their current data (i.e. set them to 0.0000 or blank), you can't delete the bins themselves or change their names or sizes. By contrast, a *variable* has no such built-in existence. Since you create it in the first place, you can also destroy it. Also, you can change its name, and sometimes increase or decrease its size.



Of course, there's an HP-42S *variable type* for each *data type*. So, keeping your thumb marking page 36, look briefly now at each of the four variable types (don't worry yet about how you actually build and store these variables yet; remember that this entire chapter is just your first guided Tour of your HP-42S workshop)....

Real Variables

Not too surprisingly, a real variable is a named storage bin containing one real number. But here's something not quite as obvious: Unlike other kinds of variables, you cannot expand a real variable to hold more than one number. It has only that limited amount of storage space.

ALPHA Variables

As you know, the ALPHA string data type is limited to a maximum of six characters. This may seem odd, since the "work space" where you build these strings – the ALPHA register – can hold up to 44 characters at once. But notice in your Big Picture that *real and ALPHA variables are shown as being stored in the same area*. The idea is to let you go back and forth between these two common data types (reals and strings). So both types are limited to the same non-expandable amount of storage space in a variable: "six ALPHA characters = one real number".

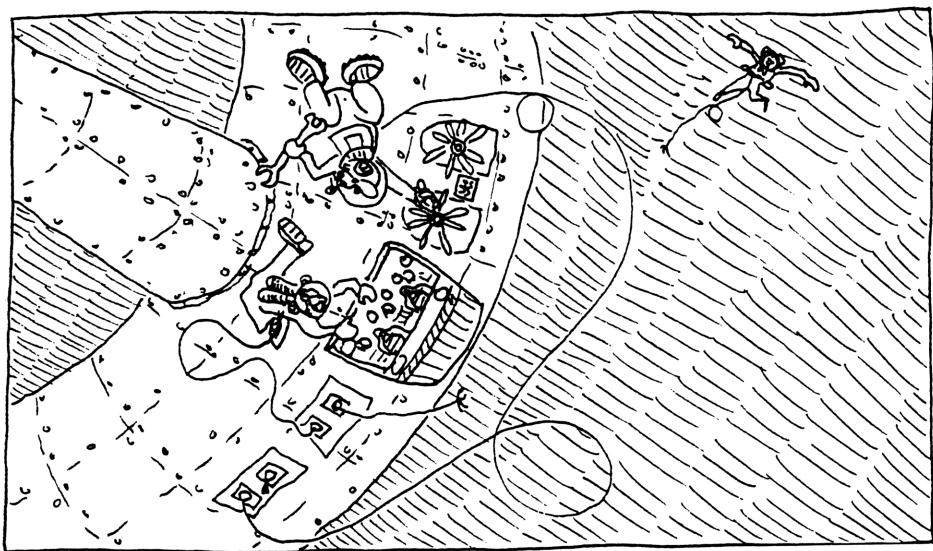
Of course, you can also have an ALPHA string with fewer than six characters, but this won't save you any storage space; it will use the same amount of space as a full six-character string (or one real number).

The interchangeability of real and ALPHA variable types in the HP-42S makes it just as convenient to refer to all such variables as "real." So when you see references to "real" variables – either here in the book or in the menus of your HP-42S – you need to realize that, unless it's stated otherwise, this means ALPHA strings too.

Complex Variables

Another totally shocking revelation: Complex variables are named storage bins containing complex numbers (recall from page 36 that a complex data type is simply a linked, ordered pair of real numbers – the real part and imaginary parts of the composite complex quantity).

As with the simpler real variables, the complex variable type is fixed in its size: You just can't squeeze any more into it than two real numbers – one for each part. But unlike real variables, you cannot substitute ALPHA strings for real numbers in the fixed spaces of a complex variable. Your HP-42S is absolutely stubborn about this: Alpha data is invalid in a complex variable type.*



*"Dave.....Dave?.....What are you doing, Dave?.....Sorry, Dave, I'm afraid I can't do that...."

Matrix Variables

Here's the "big-ey" – the real macho variable type: You can actually enter, store and manipulate *entire matrices*, using named storage bins that are matrix variables.

And here's where you can play around with the size of variables: If you have, say, a two-by-two matrix of values, you can actually re-size it to, say, a three-by-two matrix, then enter the two extra values!

Not only that, you can change the type of values that go into the matrix: If that three-by-two matrix you just filled up is a matrix of real numbers, you can actually convert it to a matrix of complex numbers instead (all the real values in it will be converted)!

This means that matrix variables are re-sizable *two* different ways – the number of elements and the size (type) of the elements themselves. Not bad, eh? Of course, this means that larger dimensions or complex types will eat more available memory, but it sure is nice to know you have this flexibility built right into the machine, no?

One caution, though: You can't mix-and-match. That is, a matrix is either all-real or all-complex in the type of elements it handles. And – consistent with the restrictions on ALPHA data within real and complex variables – ALPHA data *are* allowed in *real-valued* matrices, but *not* in complex-valued matrices.

Storage Registers: The REGS Matrix

Now look again at the Big Picture of the HP-42S Storage Area (page 19). You'll see a long, skinny matrix already stored under the name REGS in the "Matrix Variables" section....What is that beast, anyway – and what's a variable doing in your HP-42S already? *You* didn't put it there, did you? Nope – this one's a *built-in* variable....

"Built-in? Hmmm – then it sounds more like a register to me...."

That's basically what it is – a group of *registers* (hence the name, REGS) that you can't delete.

"So why are they shown as a *variable* – and in *matrix*, of all things?"

It's because you can do certain things to this set of registers that you can normally do only to a matrix *variable* – things such as changing the type from real to complex, and changing the number of elements. And yet you can also do something with this REGS matrix that you cannot normally do with matrix variables: You can call *each element* by its own "register-like" name (e.g. R_{12} , R_{19} , etc.).

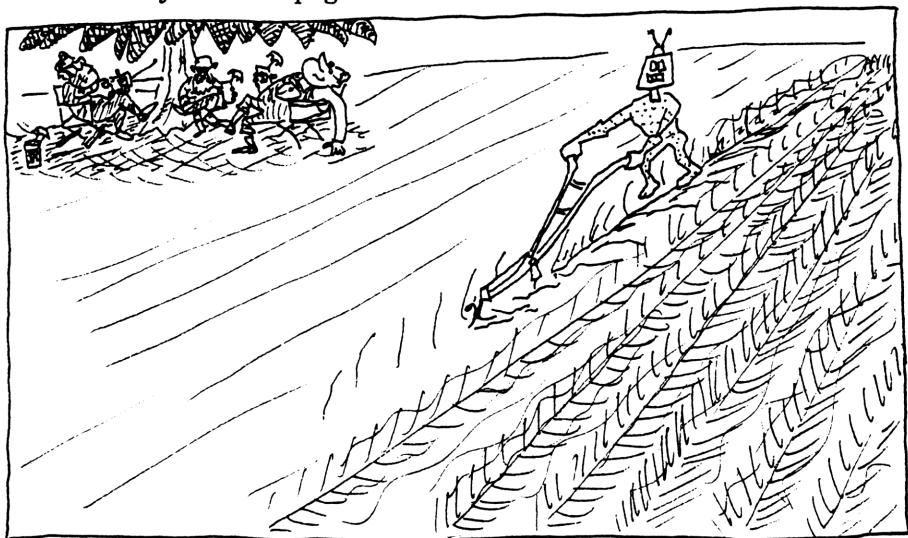
But what good is such a hybrid "variable register-matrix?" Well, often it's a real pain to need to give every stored value its own unique name and variable space. It's often more convenient to keep a set of related numerical values in a neat, ordered set, with each value indexed with a number for easy automatic sorting and access (like Zip+4 or something). So HP has reserved such a set for you, and since – like a register – it is a reserved space and name: they named it REGisterS.

Programs

There is one other kind of container that can be named in your HP-42S – but it isn't a data variable (see page 19 *one more time*). It's called a program, a set of automatically executing instructions by which you can actually add to your collection of tools: A program is a tool you build from other tools – the most powerful thing you can build in your workshop.

Like variables, programs can be varied in their sizes and names – and they take up space in your HP-42S Storage Area, but that's about where the similarities end, since programs are not "raw materials," but customized tools of your own design.




As with all aspects of this First Tour, you'll see plenty of detail on programming later. Right now, it's enough that you remember that programs also must share the Storage Area in your calculator with the variables you create – and that's why the Big Picture shows them stored as you see on page 19.





Reviewing Your Storage Area







Here's the bottom line from this chapter: Your HP-42S Storage Area is divided into areas for programs, variables, and still-available space. And *everything* you store will use some of that available space.

The CATALOG Menu



To make it easy to review what you've stored, your calculator has a special menu, the  CATALOG menu, which acts like a series of windows (notice the windows in the Big Picture) looking into the Storage Area. Press  CATALOG  CATALOG now to look at the menu list:






 FCN  PGM  REAL  CPX  MMT  MEM

Each menu item provides an overview of what's available to use in the various parts of your workshop. Try each selection as you read (and use  EXIT to go back to the  CATALOG menu after each item):








-  **FCN** gives a complete list of all the **functions** (hand tools) available;
-  **PGM** gives a list of any **programs** you've stored (none yet);
-  **REAL** lists all the **real** (and ALPHA) **variables** you've stored (none yet);
-  **CPX** lists all the **complex variables** you've stored (none yet);
-  **MMT** provides a list of all **matrix variables** (you have none of your own as yet, but you always have the REGS matrix – that built-in matrix of "registers" – of either real or complex numbers);
-  **MEM** tells you how much memory you still have available (you must *hold* down this menu key to continue viewing the memory).


The CLEAR Menu

That  CATALOG menu shows you what you've got stored – but it doesn't let you do anything about it. If you want to keep your storage areas well-organized and free of clutter, you need to use the  CLEAR Menu. This menu lets you clean out your data, variables, programs, etc. *selectively* (i.e. you don't clear the entire machine to get rid of one item).

Press  EXIT and then  CLEAR  CLEAR to see this menu (it has two "pages," so you'll need to use the  and  to see all the selections).

Although there are some items on it that won't mean much to you yet, notice these selections (try them, too, if you wish):

-  CLP shows you a list of programs that you can clear;
-  CLV shows you the list of all variables – any type – that you can clear;
-  CLST clears all five Stack registers;
-  CLA clears just the ALPHA-register;
-  CLX clears just the X-register;
-  CLRG clears all the storage registers in the REGS matrix;
-  CLALL deletes all programs and variables, and clears all registers – a fairly drastic move, so if you choose this selection, the calculator asks you to confirm that it's really what you want to do.

Just remember this – in keeping with the idea of **registers vs. variables**: When you use the  CLEAR Menu to clear a register, you remove the data but leave the container; but when you clear a variable, you delete it entirely – data *and* container.

Pop Quiz

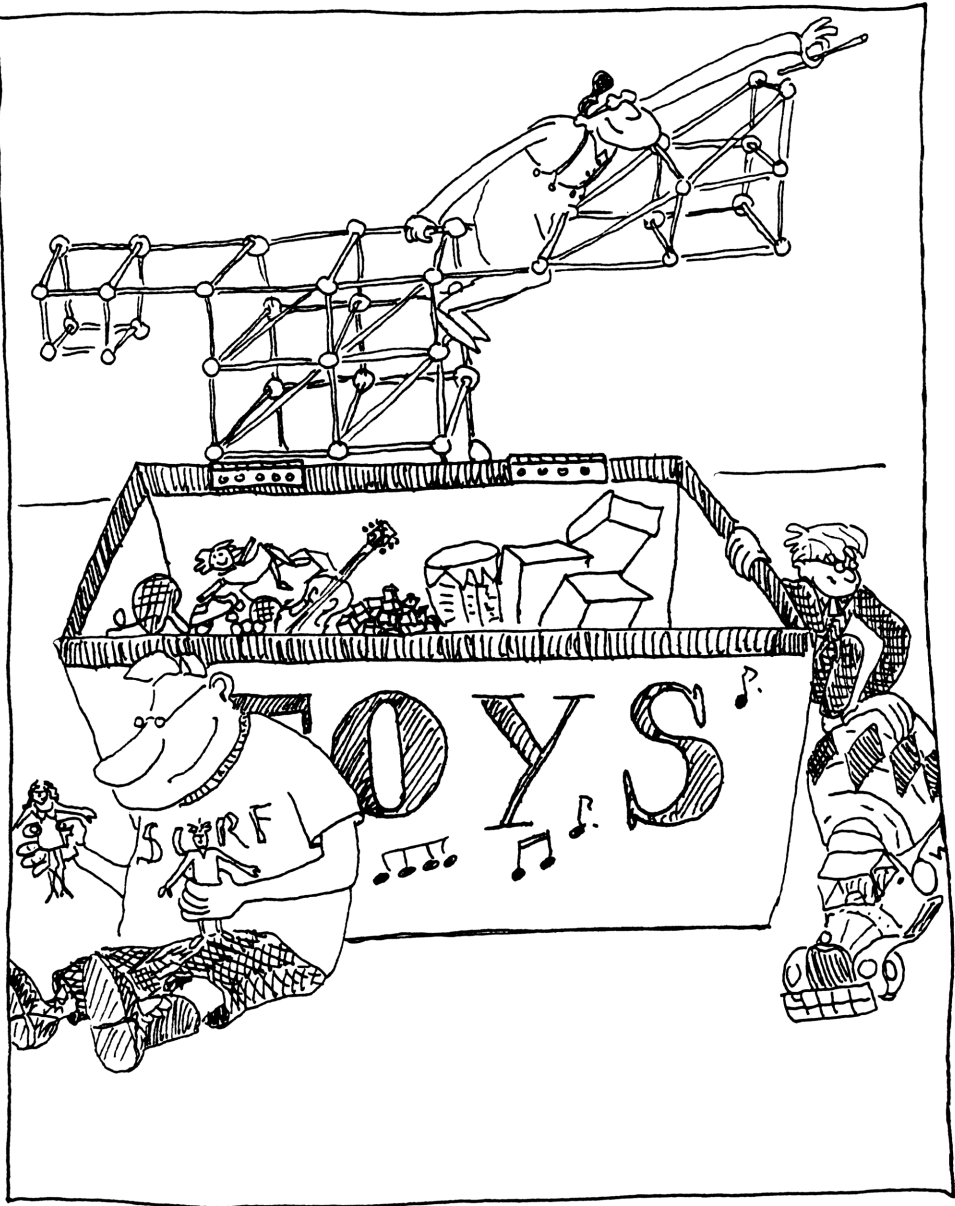
No sweat – just make sure you can answer these general questions:

1. You see in your HP-42S display: `HELLO` What register is this?
2. What kind of register is the Z-register?
3. What's a menu key and how does it work?
4. What are `↵` and `▼▲`?
5. Name the four kinds of data allowed in your HP-42S.
6. What's the difference between a register and a variable?
7. Name the four display types. How do you move between them?
8. What's so special about the ALPHA-register?
9. What if you left for Katmandu without turning off your HP-42S?

Pop Answers

1. This could be a Message Display, or it could be showing you the contents of the ALPHA-register or the X-register. If it's the X-register, then it would display X: "HELLO". If it's the ALPHA-register, then you'll see the alphabet menu on the bottom line and HELLO on the top line. See pages 23-24 to review displays.
2. The Z-register is a Stack register – part of the automatic Stack of registers where you do most of your "data crunching." These registers are named with letters to remind you that they're in this special Stack (see page 35 to review).
3. The meanings of the top-row keys depend on the type of display you see. For example, if you see just the Stack Display and press **LOG**, your HP-42S will simply take the common logarithm of the number in the X-register. But when a Menu Display is showing, this very same key acts as a *menu key*. Pressing **LOG** in that case will select the fourth menu item (see pages 31-33 to review).
4. **▼▲** and **↵** are annunciators. The **▼▲** announces that the current menu has more "pages" – that you'll see more choices if you press the **▲** or **▼** keys. The **↵** announces that you've pressed the orange "shift" key (**■**), so the machine will interpret the next key you press in terms of the orange writing above the key instead of the white writing on the key itself (see pages 26 and 32-33).

5. The four data types are: real numbers, ALPHA strings, complex numbers, and matrices. Complex numbers are linked pairs of real numbers; matrices are linked sets of either complex or real numbers (but not mixed). An ALPHA string data type is allowed anywhere a real data type is allowed. See page 36 to review.
6. *Variables* are storage containers that you can create, name, fill, empty *and* delete entirely. By contrast, *registers* are permanently-created-and-already-named storage containers that you can only fill and empty – but not delete from existence altogether.
7. The four display types are: Message, Stack, Menu, and Program. Much of the movement between displays is automatic, but you have some control over it. To move from any other display to the Stack Display (your "work- bench") you press **EXIT** (sometimes more than once). To move from the Stack Display to a Menu or Program Display, you select a function that requires that type of display (see pages 23-24).
8. The ALPHA-register is the only storage register in the calculator that cannot be directly involved in mathematical calculations. It holds up to 44 ALPHA characters (see pages 35 to review).
9. After about 10 minutes, your HP-42S would turn itself off. Then, if you returned later and turned it on, its display (and memory) would be the same as before. As long as its batteries last, your calculator will continuously remember everything you store in it.



Tools! Glorious Tools!

Your Favorite Tools: Arithmetic And The Stack

All right – so you have this whole HP-42S workshop full of tools to try out. Where do you begin?

Probably right there in front of you – at the workbench itself, no? After all, you can't do much else until you learn the basics of "number-crunching" with your favorite tools and workspace – the Stack.

If this is your first experience with Reverse Polish Notation (RPN, for short), you'll want to spend some time in this section. And even if you're an old hand at HP-42S arithmetic, you're going to need a detailed knowledge of the Stack in order to write good programs – so a review with this section may be just the ticket.*

So turn your machine on (if it's not on already), and make sure you see the Stack display, showing both the X- and Y-registers.

Your tools are ready....

*But, if you really do know your way around the Stack, and you're already quite comfortable with doing arithmetic on your HP-42S, including negative numbers and scientific notation, then feel free to go on to page 74 now.

RPN

So what is RPN? It's a way of doing arithmetic that's a little different and a lot more efficient than the way you might "say it out loud."

For example, if you were "saying" the solution to an addition problem, you'd probably say something like "5.79 plus 34.6 equals 40.39." This seems easy and intuitive, right?

OK, but try "saying" the solution to this: $19 - \frac{11((5.79 \div 25) + 34.6)}{(40 \times 22)}$

"uh...19 minus the quantity 11 times the quantity 5.79 divided by 25, plus 34.6, divided by the quantity 40 times 22 equals ...?"

A person could get a sore throat with all those "quantities," which are meant to denote the parentheses, right? And it's pretty hard to tell where the parentheses should open and close, isn't it? If you get several nested sets, it's a real pain to make sure you've got a closing parenthesis for every opening one.

The point is, *all those troubles are still with you* if you use a calculator that tries to mimic the way you "say it out loud" – a calculator that uses "algebraic notation," rather than RPN.

With RPN, you never use parentheses (no matter how complex the calculations get) nor do you even need an [=] key (notice that your HP-42S has neither [()] keys nor an [=] key).

"How is this possible?"

Glad you asked!....

- The main "strategy" with RPN is:
- A. Key in a number;
 - B. Do something to it.

And most RPN calculations are just repetitions of this simple pattern!

Example: Press **[EXIT]** so that you see the Stack, then add 5.79 and 34.6 on your HP-42S.

Solution: Do the above procedure twice:

- A. Key in **[5][.][7][9]**.
- B. Press **[ENTER]** to "finish" it.

- A. Key in **[3][4][.][6]**;
- B. Press **[+]** to add it to the previous number.

Result: x: 40.39

Putting the action at the end of the sequence of numbers is what makes RPN so different – and so much easier – than the way you're used to "saying" a solution (and it's also what puts the R in RPN*).

For one thing, *it eliminates the need for an [=] key*, since pressing **[+]** signifies that you're ready to have the operation performed. But the biggest advantage is that *you never need parentheses*, because you're always "combining as you go." You don't need to give the machine the entire problem before beginning to solve it!

*See the footnote on page 42 of your Owner's Manual for more details about how RPN got its name.

The Stack

The Stack is what makes RPN arithmetic possible: You don't need to key in an entire problem at once with an RPN machine *because it can accumulate the intermediate results* (partially completed calculations), letting them "stack" up until it's time to use them.

So whenever you do arithmetic on your HP-42S, you're using that set of five registers – named X, Y, Z, T and L –known collectively as the Stack. Four of these registers (X, Y, Z and T) work in a closely linked fashion to do the arithmetic (the L-register is important, but not while you learn the details of arithmetic. Until further notice, when you think about "the Stack," just think about the X, Y, Z, and T-registers).

The first thing to realize about the Stack is that whenever you key a number into your HP-42S, you're keying it into the X-register. *All numbers are keyed into the X-register.*

Challenge: Set the Stack up like this:

T: 56.87
Z: 43.6
Y: 10.02
X: 6_

Solution: 56•87ENTER43•6ENTER10•02ENTER6

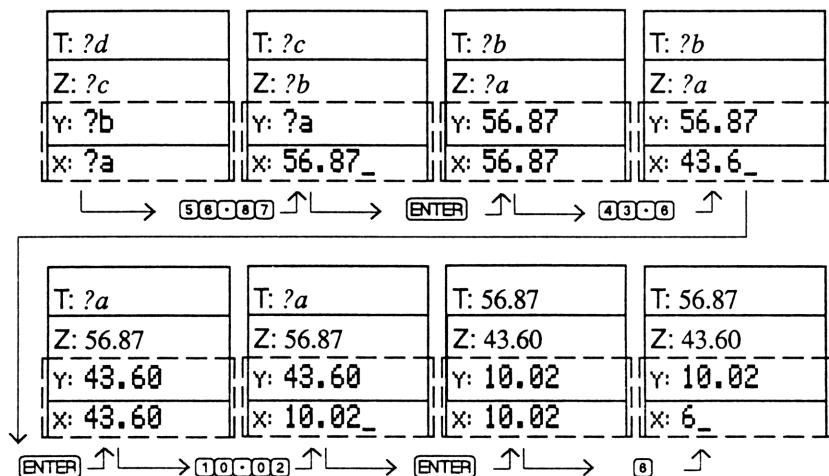
OK, but what's really happening when you press that ENTER key?

What Is Stack-Lift?

Take a closer look at the inner workings of the Stack as you re-solve the previous **Challenge**....

Below is a series of Stack diagrams, a handy way to demonstrate what's happening in the four Stack registers.

In these diagrams, the visible portion of the Stack (what you see in the display), is outlined in the dashed boxes. And the '?'s mean, "some number is there, but for the purposes of the example, it doesn't matter what it is."



Study this diagram and compare it with your HP-42S display as you re-solve the example (press \blacksquare to clear the 6_ if it's still in your display).

Notice a few things that tell you a lot about how the Stack works....

1. Often, when you key a number into the X-register, the numbers in the other registers are "bumped" up one notch (and the value that was in the T-register is bumped off the top – gone for good, never to return). This is how the Stack automatically saves your partial results when you're doing a big arithmetic problem: as you key in the next number, the previous (partial) result bumps up a notch, "floating around" above until you need it.

This "bumping up process is called *"Stack-Lift,"* and whenever the Stack is ready to do this, you say that Stack Lift is *enabled*. When the Stack is *not* ready to lift, you say that Stack Lift is *disabled*. Most of the time, as you do arithmetic, you want the Stack Lift to remain enabled – and, sure enough, that's how it works. But sometimes you specifically want to disable it....

2. Whenever you press **ENTER**, you cause three things to happen:

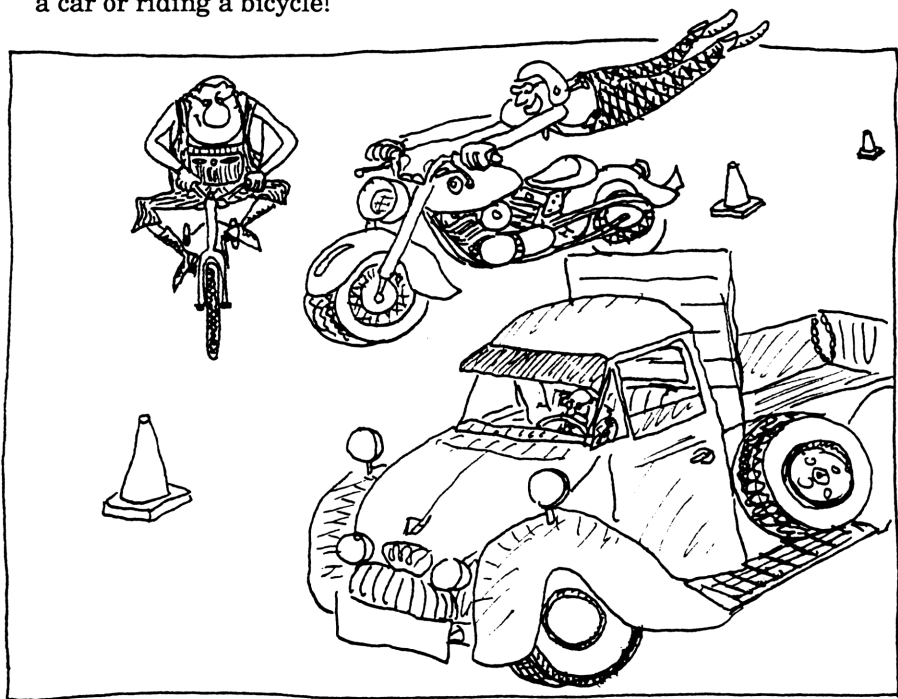
First, you *terminate digit entry*, which is a fancy way of saying that you tell the machine you're finished keying in the number in the X-register. If you think about it for a minute, you'll realize that you must have some way of telling this to the machine, since with RPN arithmetic, when you add 2 and 2, the order of entry is **22+**, so you need *something* in between the two **2**'s to tell the machine that you do indeed mean two 2's, not one 22.

Next, you *cause a Stack Lift* – but without keying in any new value to the X-register. That is, all Stack values are bumped up one notch – just as in any other Stack Lift – but the new value in the X-register is still the same as the previous value.

Finally, you *disable the Stack Lift* – so that if you now key in a new number to the X-register, it will *not* bump everything else up, but simply overwrite the previous contents of the X-register.

Thus, in the example, as you key in every number, you press **ENTER** to (i) terminate the keying-in process; (ii) lift the Stack (thus saving a copy of this latest number in the Y-register); and (iii) disable the Stack to ensure that the next number can slip right into the X-register without further affecting the rest of the Stack!

Right now, you might be thinking that this is an awful lot of detail to have to remember just to do arithmetic – but don't worry. It takes only a little practice and the Stack will become as automatic as, say, driving a car or riding a bicycle!



Putting Two And Two Together

Of course, to do actual arithmetic, you'll need to do more than just "set up the Stack." Here's how you use the Stack to crunch numbers....

Do This: With your Stack properly set up from the previous Challenge, press the \boxtimes key to multiply together the contents of the X- and Y-registers.

Result:

Y: 43.60
X: 60.12

Look at what happens to the stack during this multiplication:

T: 56.87	T: 56.87
Z: 43.60	Z: 56.87
Y: 10.02	Y: 43.60
X: 6_	X: 60.12

└─→ \boxtimes ─┐

When you press the \boxtimes key, several things happen:

The values in the X- and Y-registers combine in the multiplication operation, and the result lands in the X-register.

Next, since two numbers have just been combined into one, the rest of the values above simply drop one notch (the T-register stays the same).

And (best of all) at the end of the operation, the stack is left *enabled*, so that you can now key in another number to the X-register, and the previous result (60.12) will "bump" up to the Y-register – all ready to be combined with your new entry!

So Go For It: Subtract 42 from that last result.

Nuthin' To It: Just key in (4)2), then press [=].* Here's the result:

Y: 43.60
X: 18.12

And here's what happened:

T: 56.87	T: 56.87	T: 56.87
Z: 56.87	Z: 43.60	Z: 56.87
Y: 43.60	Y: 60.12	Y: 43.60
X: 60.12	X: 42_	X: 18.12

|————→ (4)2) ———→
|————→ [=] ———→

See how an enabled Stack Lift allows the upper Stack values to simply "ride" along during the calculation?

And notice how the subtraction operation subtracts the X-value from the Y-value (i.e. it's "Y minus X"), not the other way around. Each operation that combines two numbers has a specific order (although in some operations, order doesn't matter).

*Notice, by the way, the efficiency of RPN here: If this were not an RPN machine, then to do this example, you would need *four* keystrokes [=](4)2)[=], rather than three (4)2)[=]!

Chain Calculations

By now you can begin to see the advantages of the RPN Stack – especially when Stack Lift is enabled for subsequent ("chain") calculations. For that very reason, most operations will leave the Stack Lift enabled (indeed, **ENTER** is one of very few keys that disable it).

Now try this problem as a fuller example of the Stack, its lifting and dropping – and the beauty and simplicity of RPN:

Problem: Calculate $3.57 \times (67.12 + (89.43)^2)$ *without* storing any numbers or writing down any intermediate results (and of course, there ain't no **()** or **=** keys either)!

A Solution: **89.43** **X²** **67.12** **+** **3.57** **X**

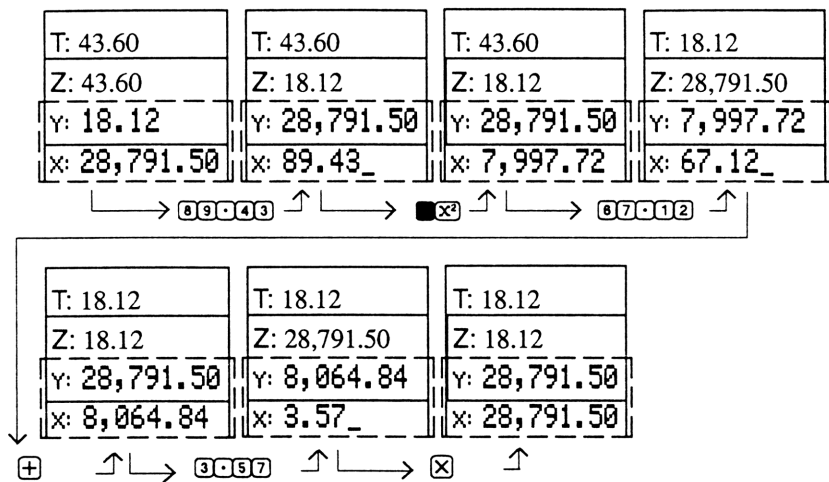
Answer: 28,791.50

You may be wondering how to decide on the "order of doing things."

That's not too hard – remember how you were taught to analyze such involved, chain-calculation problems in school?

"Work from the innermost parentheses outward."

It's still good advice – especially for RPN arithmetic! Study the following diagram as you repeat the keystrokes....



Notice that any operations, such as \times and $+$, that combine two numbers into one result will cause the Stack to drop, but an operation such as $\blacksquare X^2$, will affect only the "X-value," as the name implies.

Of course, there are other solutions, too. For example, you could have chosen to key in the numbers in the order you encounter them in the problem: $\boxed{3}\boxed{5}\boxed{\cdot}\boxed{7}\boxed{\text{ENTER}}\boxed{6}\boxed{7}\boxed{\cdot}\boxed{1}\boxed{2}\boxed{\text{ENTER}}\boxed{8}\boxed{9}\boxed{\cdot}\boxed{4}\boxed{3}\boxed{\blacksquare X^2}\boxed{+}\boxed{\times}$

But this takes more keystrokes than the other way. And besides, if you limit yourself to this left-to-right method, you simply can't solve a problem like this one: $112 - (386,000 \div (35.7 \times (67.12 + (89.43)^2))$

You have only four Stack registers, remember; and the whole idea of RPN is that *you key in only part of the problem at once, reducing it and simplifying it as you go*. So the best way is always to work from the "innermost" parentheses "outward."

Are you starting to get "oriented" to RPN?

Try Another: $\frac{6.7 + \sqrt{(6.7)^2 - 4(3.9)(2.12)}}{2(3.9)}$

Solution: Again, work from the "innermost" expression "outward:"

6 • 7 \times^2 4 ENTER 3 • 9 \times 2 • 1 2 \times - \sqrt{x}
6 • 7 + 2 ENTER 3 • 9 \times \div

Answer: 1.30

Getting easier, isn't it? And of course, with practice, you won't be needing even to think about what's really going on in the Stack as you do this. After all what would be the point of a more efficient form of arithmetic if you spent a lot of time trying to remember how to do it and why it works?

For now, though, check the Stack diagram opposite, just to be absolutely sure you know what you're really doing....

T: 18.12	T: 18.12	T: 18.12	T: 28,791.50
Z: 18.12	Z: 28,791.50	Z: 28,791.50	Z: 28,791.50
Y: 28,791.50	Y: 28,791.50	Y: 28,791.50	Y: 44.89
X: 28,791.50	X: 6.7_	X: 44.89	X: 4_

\rightarrow $\boxed{8 \div 7}$ \uparrow \rightarrow $\boxed{\times^2}$ \uparrow \rightarrow $\boxed{4}$ \uparrow

T: 28,791.50	T: 28,791.50	T: 28,791.50	T: 28,791.50
Z: 44.89	Z: 44.89	Z: 28,791.50	Z: 44.89
Y: 4.00	Y: 4.00	Y: 44.89	Y: 15.60
X: 4.00	X: 3.9_	X: 15.60	X: 2.12_

$\boxed{\text{ENTER}}$ \uparrow \rightarrow $\boxed{3 \div 9}$ \uparrow \rightarrow $\boxed{\times}$ \uparrow \rightarrow $\boxed{2 \div 12}$ \uparrow

T: 28,791.50	T: 28,791.50	T: 28,791.50	T: 28,791.50
Z: 28,791.50	Z: 28,791.50	Z: 28,791.50	Z: 28,791.50
Y: 44.89	Y: 28,791.50	Y: 28,791.50	Y: 3.44
X: 33.07	X: 11.82	X: 3.44	X: 6.7_

$\boxed{\times}$ \uparrow \rightarrow $\boxed{=}$ \uparrow \rightarrow $\boxed{\sqrt{x}}$ \uparrow \rightarrow $\boxed{8 \div 7}$ \uparrow

T: 28,791.50	T: 28,791.50	T: 28,791.50	T: 28,791.50
Z: 28,791.50	Z: 28,791.50	Z: 10.14	Z: 10.14
Y: 28,791.50	Y: 10.14	Y: 2.00	Y: 2.00
X: 10.14	X: 2_	X: 2.00	X: 3.9_

$\boxed{+}$ \uparrow \rightarrow $\boxed{2}$ \uparrow \rightarrow $\boxed{\text{ENTER}}$ \uparrow \rightarrow $\boxed{3 \div 9}$ \uparrow

T: 28,791.50
Z: 28,791.50
Y: 10.14
X: 7.80

T: 28,791.50
Z: 28,791.50
Y: 28,791.50
X: 1.30

$\rightarrow \boxed{\times} \rightarrow \boxed{+} \rightarrow$

Changing The Sign Of A Number

Of course, if you're going to do much arithmetic, sooner or later you'll encounter some negative numbers, so you'd better know how to handle them, right?

Try This One: Find $-4.56 \times (3.4 + 6.79)$

Solution: Press 3.4ENTER6.79+4.56+/-×

Answer: -46.47

See? The +/- key simply changes the sign of the number in the X-register.

Of course, +/- goes both ways: It changes a positive number to a negative – and vice versa. Press it again now to change the -46.47 to 46.47.

That **E** Key

Problem: What's $2,500,000 \times 3,000,000$?

Solution: **25000000ENTER30000000X**

Answer: **7.50E12**

And what's *that*? It's just a shorthand way of writing very large (or very small numbers), a notation called "scientific notation," since scientists often need to use such numbers. You'd read **7.50E12** as "seven-point-five-zero times ten to the twelfth power."

Of course, you can *key in* numbers in scientific notation, too. For instance, you can do the above problem without even using the **0** key: **2.5E6ENTER3E6X**

One More: Calculate $(-3.46 \times 10^9) \div (4.23 \times 10^{-7})$

Solution(s): **3.46+/-E9ENTER4.23E+/-7÷**, or
3.46+/-E9ENTER4.23E7+/-÷, etc.
Answer: **-8.18E15**






You can use the **+/-** key *at any time* while keying in a number. And in scientific notation you'll change the sign of either the mantissa or the exponent – whichever part of the number you're keying in at the time.


The Key And



All right – you now know some basic arithmetic, how to use negative numbers and a little bit about scientific notation.









But still, nobody's perfect. What do you do if you make a mistake while keying in a number?

Like This: Suppose you wanted to multiply 56.87 and 23.4.

So you blithely rattle off  ...(...oops).

Not To Worry: This is the main reason for the  key. When you're keying in digits, it is indeed a simple *backspace* key, so use it as such....

First, back out your erroneous digits: .

Now key them in correctly:  and finish the problem: . Answer: 1,330.76


Not too mysterious, right?





OK, but what if you want to get rid of the entire number (i.e. "start over") altogether? What if it's not just a digit or two – the whole thing's bad?...

Try This: You want to add 1.61803398875 and 3.14159265359.

1 . 6 1 8 0 3 3 9 8 8 7 5 ENTER

3 . 1 4 1 9 5 2 6 5 3 5 9 ...oops! – you botched that second one – and it's going to be easier to start over than  back to fix it. So you'd really like to clear that number in the X-register entirely. But how?

Solution: Remember the  CLEAR menu? Press that key now and choose the selection on the far right: .

OK, you've now cleared away the 3.14159265359_ and replaced it with just 0.00. And here's your display:

Y:	1.62
X:	0.00

So far, so good....


Next Question: When you start to re-key in $\boxed{3} \cdot \boxed{1} \boxed{4} \boxed{1} \boxed{5} \boxed{9} \dots$, will the 0.00 be bumped up? That is, is the Stack Lift *enabled* right now?

Hopefully not, right? After all, you still want to add this number to what's in the Y-register (1.62).


Good News: The **CLX** function leaves Stack Lift *disabled*, so the 0.00 that you've temporarily placed in the X-register (to clear it) does *not* "bump up" and get in the way of your arithmetic—it's overwritten by the corrected entry that you're keying in. HP designed **CLX** that way, because they knew that the main reason you normally use it is to correct an entry in the middle of some calculation – and you don't want the rest of the Stack "messed up."


So although most operations leave the Stack *enabled* (making it easy to do chain-calculation arithmetic), there are two keys that *disable* the Stack.





First, you had the **ENTER** key; now there's **CLX**, too.


Yes, But: Now repeat the previous problem – exactly – except this time do it the long way: "back out" the erroneous number with the  key....



Guess What: It works the same way, doesn't it?

It's a fact: Whenever you use the  key to clear the very last remaining digit in the X-register, you are actually using the **CLX** function.

Not only that: If you're not keying in a number at all, then  is not "backspace" at all – it's **CLX** then, also!

Prove It: Do some arithmetic (or anything else that leaves Stack Lift enabled):    .

Now press . Voilà! That's not backspace – that's **CLX**. It cleared the entire X-register and disabled Stack Lift.

So remember:  and **CLX** are the two common functions you'll use that *disable* Stack Lift. And except when you use it to "back out" part of the number you're keying in, the  key is also a convenient **CLX** key.

Moving Numbers Around In The Stack

Those Stack diagrams were pretty helpful in understanding "where the numbers go" when you're not using them, weren't they? After all, you can see only two Stack registers at a time – the X and Y- registers, so you must simply "imagine" the other two, right?...

Hmmm – come to think of it – *is* there any way you can check the current contents of those other two registers (Z and T) – without messing up the values in the Stack?

You Betcha: Set up your Stack so that it looks like this:

T: 8.00
Z: 3.90
Y: 4.60
X: 9.4

Now, your challenge is to replace that ~~3.90~~ in the Z-register with a ~~1.30~~ – *without changing the values in any of the other registers.*

Solution: First, you must set up the Stack: `8 ENTER 3 . 9 ENTER 4 . 6 ENTER 9 . 4`. No sweat, right? You've done this kind of thing before.

Now for the "surgery:" `R↓R↓↕1 . 3 R↑R↑`.
Not bad, eh? See how this works?

The **R↓** Key

The **R↓** ("roll down") key causes the Stack to do just that – its contents rotate or "roll down" by one register. The value in the T-register is sent down to the Z-register; the Z-value down to the Y-register, and so on, *and the value in the X-register is "rolled" up into the T-register to complete the rotation.*

So in a **R↓**, no numbers are lost; only their locations are changed. The Stack registers are linked in a circular pattern (hence the "roll"), where the bottom register (X) is connected to the top register (T).

*This means that four consecutive **R↓**'s will roll the Stack completely around once – back to its original state. This is how you can quickly look at all four values without changing any of them.*

So in this last solution, the trick was to roll the Stack down twice – to get the doomed 3.90 into the X-register where you could clear it out with the **CLX** key. Then (because the **CLX** key acts as a **CLZ** in this case, as you know), the Stack was left disabled, so you keyed in **103**, then rolled the Stack down twice more – that's now four **R↓**'s in all – to get back to the original positions of the values. Pretty slick!

The $\boxed{\text{X}\leftrightarrow\text{Y}}$ Key

There's one other key that you'll often find useful to manipulate your Stack registers: The $\boxed{\text{X}\leftrightarrow\text{Y}}$ key. Suppose that you wanted to...

Do This: Set up the contents of the Stack as shown here (this should be exactly how they ended up from the previous problem, but if you want to review them quickly to check, you now know what to do – $\boxed{\text{R}\downarrow}\boxed{\text{R}\downarrow}\boxed{\text{R}\downarrow}\boxed{\text{R}\downarrow}$ – right?):

T: 8.00
Z: 1.30
Y: 4.60
X: 9.40

Now, the challenge is to *reverse* the order of these values.

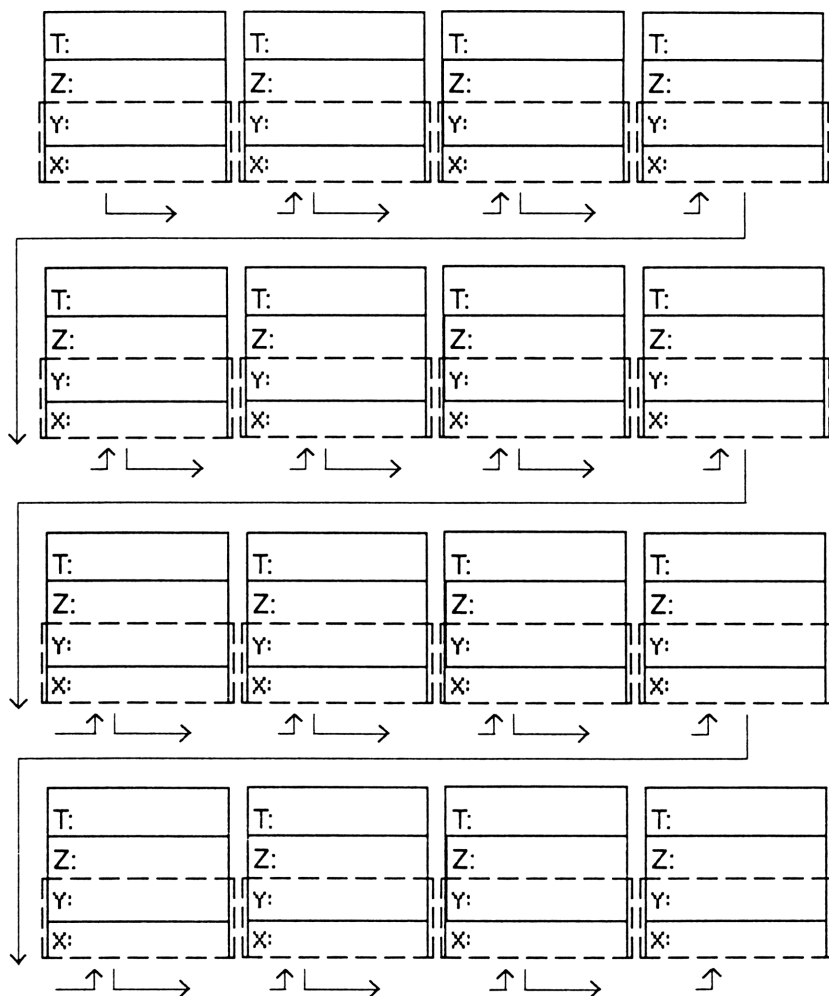
Solution: As with most Stack manipulations, there are many ways to do this, but here's one of the easier ones:

$\boxed{\text{X}\leftrightarrow\text{Y}}\boxed{\text{R}\downarrow}\boxed{\text{R}\downarrow}\boxed{\text{X}\leftrightarrow\text{Y}}$ (Deceptively simple, isn't it?)

The $\boxed{\text{X}\leftrightarrow\text{Y}}$ key ("X exchange Y") simply *exchanges* the values of the X- and Y-registers – which is handy for all sorts of things (not just stupid little exercises like the one above).

For example, the $\boxed{\div}$ and $\boxed{-}$ keys are defined only as "Y \div X" and "Y - X" – not the other way around. So, to divide or subtract in the *other* order, you just use $\boxed{\text{X}\leftrightarrow\text{Y}}$ once to swap the two numbers, *then* use $\boxed{\div}$ or $\boxed{-}$.

Notes, Doodles, And Stack Practice

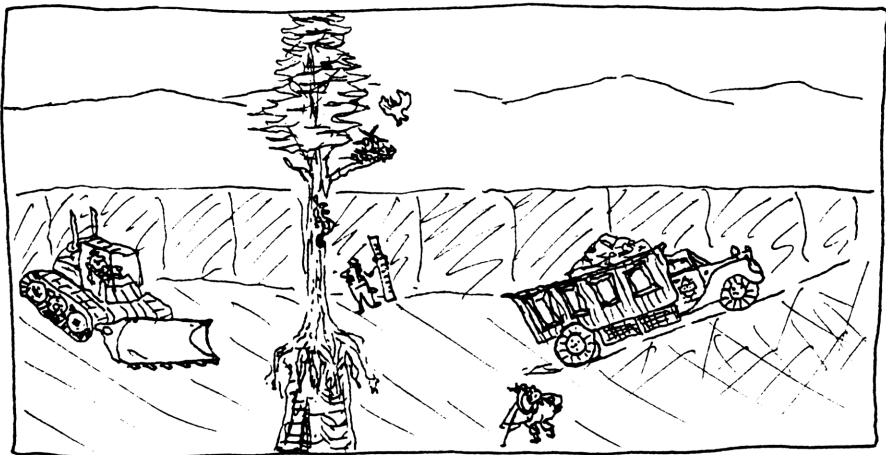


The L-Register

OK, ok, now that you've got a little Stack practice "under your fingers," it's finally time to let you in on the big secret: "What-in-L is the L-register?"

Think of the L-register as an automatic precaution your calculator takes, "just in case" you didn't really want to do whatever you just did to the number in the X-register. That is, any time you do an operation on the number in the X-register – anything that would affect its value, – a "back-up" copy of that number is placed into the L-register.













In other words, operations that *alter* numbers (such as \oplus , \otimes , and \ominus) will indeed save a copy of the X-value in the L-register before changing the original in the X-register. But functions that *clear or move numbers around* (like \leftrightarrow , $\times\div y$, $R\downarrow$, and ENTER) do *not* do affect the L-register.








"Fine – so what if you ever need to make use of this back-up copy? How do you retrieve it?"


The LAST X Key

Here's the key for retrieving that "emergency copy" of the X-register because you've made a little goof-up during your calculation.
















Watch: You're trying to multiply $\sqrt{216.14} \times (789)^2$, but unfortunately you press ...uh-oh! Drat! Now what? You *could* start over, of course, but why not use the L-register?

First, press  LAST X and  to undo what you just did (thus recovering $\sqrt{216.14}$). Now press  LAST X again and proceed correctly:  Bingo! Answer: 9,152,117.35

Keep in mind that when you use the  LAST X key, you're bringing a *copy* of the L-register to the X-register; that same back-up value will stay in the L-register until you next alter the contents of the X-register.

And you can also use  LAST X simply to save yourself keystrokes...

Like This: Calculate $\frac{3.51 - (4.33 \times 10^{-2})}{(4.33 \times 10^{-2})}$

Solution: 

Answer: 80.06

Take a breather here for a moment and reorient yourself a little bit:

All this time (the last 25 pages), you've been learning and practicing with the tools that are most immediate to your workbench – the Stack. And look at what you know already:

- You know something about arithmetic – and about how Stack Lifts (and drops) happen as you're doing a calculation;
- You know that most operations leave Stack Lift *enabled* – so that it's ready to lift when the next number is put into the X-register – except for **ENTER** and **CLX**;
- You know that the **◀** key can act either as a "backspace" (when you're keying in a number) or as a convenient way to **CLX**;
- You know at least a few tricks for manipulating the values in the Stack – moving them around without changing them – with the **↔** and **⇌** keys.
- You know how to use the L-register and the **LAST X** key to recover from an error with a back-up copy of the value previously in the X-register.

So, *in theory*, you're perfectly comfortable with all these things, right?

But that's only theory....

Reality Check*

1. To double the number in the X-register, you can use the keystrokes **ENTER** **2** **×** (three keystrokes). Can you accomplish the same thing with only two keystrokes?
2. Compute $-8.9 \div (4.57 - (3.5 \times \sqrt{7}))$.
3. Without using the **ENTER** key, set up the Stack to look like this:

T: 1.6000
Z: 3.5000
Y: 2.2000
X: 4.7000

4. Using the Stack you just set up in the previous problem – and without keying in any numbers, compute:

$$\frac{(3.5 - 2.2^2 + 4.7)}{1.6}$$

5. What four keystrokes can completely clear all of the Stack registers? Don't use the menus for this procedure.

*A.K.A. a "Pop Quiz"

Real Answers

1. Yep: $\boxed{\text{ENTER}} \boxed{+}$. (Right? ...Right.)

2. Here are two different approaches:

$\boxed{7} \boxed{\sqrt{x}} \boxed{3} \boxed{\cdot} \boxed{5} \boxed{\times} \boxed{4} \boxed{\cdot} \boxed{5} \boxed{7} \boxed{\times y} \boxed{-} \boxed{8} \boxed{\cdot} \boxed{9} \boxed{+/-} \boxed{\times y} \boxed{\div}$ or
 $\boxed{7} \boxed{\sqrt{x}} \boxed{3} \boxed{\cdot} \boxed{5} \boxed{\times} \boxed{+/-} \boxed{4} \boxed{\cdot} \boxed{5} \boxed{7} \boxed{+} \boxed{8} \boxed{\cdot} \boxed{9} \boxed{+/-} \boxed{\div} \boxed{1/x}$

Answer: 1.90.

Do you see how you can use the $\boxed{+/-}$ and $\boxed{1/x}$ keys to effectively reverse the order of $\boxed{-}$ and $\boxed{\div}$, respectively?

3. First, notice that the display should be set to FIX 4 decimal places (press $\boxed{\blacksquare} \boxed{\text{DISP}}$, then $\boxed{\text{FIX}} \boxed{4} \boxed{\text{ENTER}}$). Notice also that the number in the X-register should be terminated (i.e. it must have no trailing cursor like this: 4.7_). Here are some of the possible solutions:

$\boxed{1} \boxed{\cdot} \boxed{6} \boxed{\times y} \boxed{3} \boxed{\cdot} \boxed{5} \boxed{\times y} \boxed{4} \boxed{\cdot} \boxed{7} \boxed{\times y} \boxed{R\downarrow} \boxed{2} \boxed{\cdot} \boxed{2} \boxed{\times y}$ or
 $\boxed{4} \boxed{\cdot} \boxed{7} \boxed{\times y} \boxed{\blacktriangleleft} \boxed{3} \boxed{\cdot} \boxed{5} \boxed{\times y} \boxed{2} \boxed{\cdot} \boxed{2} \boxed{\times y} \boxed{1} \boxed{\cdot} \boxed{6} \boxed{R\downarrow}$ or
 $\boxed{4} \boxed{\cdot} \boxed{7} \boxed{\times y} \boxed{\times y} \boxed{1} \boxed{\cdot} \boxed{6} \boxed{\times y} \boxed{3} \boxed{\cdot} \boxed{5} \boxed{\times y} \boxed{2} \boxed{\cdot} \boxed{2} \boxed{\times y}$, etc.

The point is, almost any non-digit keystroke except $\boxed{\text{ENTER}}$ or $\boxed{\blacktriangleleft}$ ($\boxed{\text{CLY}}$) leaves Stack Lift *enabled* and terminates your digit entry (whereas $\boxed{\text{ENTER}}$ *disables* Stack Lift and terminates digit entry).

4. Try $\boxed{R}\boxed{-}\boxed{X^2}\boxed{R}\boxed{X^2Y}\boxed{R}\boxed{R}\boxed{R}\boxed{+}\boxed{X^2Y}\boxed{-}$ (one of many possibilities)

Answer: 3.9938

Wouldn't it be lovely if there were a "roll up" function to go along with "roll down?" It sure would be a lot more convenient in this case, wouldn't it?

Well, there *is* such an animal. It's a "hand tool" – like all the Stack-manipulation tools you've been using up to now – but it isn't used often enough to merit a key of its own. You'll find it, therefore (along with all the "hand tools"), listed in the $\boxed{\text{CATALOG}}$, under $\boxed{\text{FCN}}$ ("FunCtioNs").

Press $\boxed{\text{CATALOG}}$ $\boxed{\text{FCN}}$ $\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}\boxed{\uparrow}$ (that's 14 $\boxed{\uparrow}$'s – rolling backwards through the many pages of this huge menu – which is the fastest way) to find the menu choice, $\boxed{R}\boxed{\uparrow}$.

5. $\boxed{\leftarrow}\boxed{\text{ENTER}}\boxed{\text{ENTER}}\boxed{\text{ENTER}}$ or $\boxed{0}\boxed{\text{ENTER}}\boxed{\text{ENTER}}\boxed{\text{ENTER}}$ will both work. Of course, if you're allowed to use the menus, then $\boxed{\text{CLEAR}}\boxed{\text{CLST}}$ (Clear STack) would be the best choice.

Bigger Hammers: Higher Math And Other Number-Crunchers

All right, now that you've been introduced to the Stack and its tools, you're going to start using them as you look at the other tools in your workshop.

You'll notice that most of the common Stack functions have keys of their very own (e.g. the arithmetic operations, $\boxed{\times \div \sqrt{}}$, $\boxed{R\downarrow}$, $\boxed{\text{ENTER}}$, $\boxed{\blacktriangleup}$, etc.) – because you use them so often. You might say that the keyboard is like a shelf built right over your "workbench," where you keep your most-used tools "within easy reach" (i.e. you don't have to use menu keys to go to the hand-tool storage area to "get" these functions).

Well, there are some other commonly-used "number-crunching" tools on the keyboard – on that "easy reach" shelf, too – and now it's time to get acquainted with them.



Exponentiation: Roots, Logs, And Powers

The keys you need for any root, logarithm or power problems, are located in the top row on your HP-42S keyboard.

Try This: Find $\left(34.19 + \frac{1}{11}\right)^2$

Solution: Press $\boxed{3}\boxed{4}\boxed{\cdot}\boxed{1}\boxed{9}\boxed{\text{ENTER}}\boxed{1}\boxed{1}\boxed{1/x}\boxed{+}\boxed{\blacksquare}\boxed{x^2}$
or $\boxed{3}\boxed{4}\boxed{\cdot}\boxed{1}\boxed{9}\boxed{\text{ENTER}}\boxed{1}\boxed{1}\boxed{1/x}\boxed{+}\boxed{\text{ENTER}}\boxed{\times}$
or $\boxed{3}\boxed{4}\boxed{\cdot}\boxed{1}\boxed{9}\boxed{\text{ENTER}}\boxed{1}\boxed{1}\boxed{1/x}\boxed{+}\boxed{2}\boxed{\blacksquare}\boxed{y^x}$

Answer: 1,175.1807

When it comes to squaring a number, you have all sorts of options – including the $\blacksquare y^x$ key.

Try Another: Find $(4.56)^{3.6}$

Solution: Press $\boxed{4}\boxed{\cdot}\boxed{5}\boxed{6}\boxed{\text{ENTER}}\boxed{3}\boxed{\cdot}\boxed{6}\boxed{\blacksquare}\boxed{y^x}$
Answer: 235.6532

Note that, like subtraction and division, the $\boxed{y^x}$ (exponentiation) key is a function that requires you to give the two numbers *in a particular order*; for exponentiation, you need to enter the Y-value (the base) first and then the X-value (the exponent).

When you exponentiate, you're raising the base number to the power indicated by the exponent, right?

Now go the other way: If you have a number that is the result of raising a known base (say, 10) to some power, how do you find out what that power was? You take a *logarithm*.

Like This: First, find $10^{2.5}$: $\boxed{1}\boxed{0}\boxed{\text{ENTER}}\boxed{2}\boxed{\cdot}\boxed{5}\boxed{\text{y}^x}$ or $\boxed{2}\boxed{\cdot}\boxed{5}\boxed{\text{10}^x}$

Answer: 316.2278 Now, starting with this answer, work back to find the power that 10 was "raised to."

Solution: $\boxed{\text{LOG}}\boxed{\text{Answer: 2.5000}}$ Shur 'nuff!

Hmmm....: The LN function is the Natural Logarithm (base e); the LOG function is the common LOGarithm (base 10).

However, there's a formula you can use to convert between the logarithm of *any* base, b, and the natural (LN) base, e: $\ln(x) = \log_b(x)\ln(b)$

Verify this for the common LOG base (i.e. use $x = 100$ and $b = 10$):

Solution: First, the easy way: $\boxed{1}\boxed{0}\boxed{0}\boxed{\text{LN}}$ Answer: 4.6052
Then the indirect route, using the above formula:
 $\boxed{1}\boxed{0}\boxed{0}\boxed{\text{LOG}}\boxed{1}\boxed{0}\boxed{\text{LN}}\boxed{\times}$ Answer: 4.6052

Good Questions: What's the easiest way to key in the natural LN base, e (to 12 digits, of course) on the HP-42S?

How about the same approximation for π ?

How much is $\frac{e^{10\pi}}{\pi e^{10}}$?

Decent Answers: To get e , just compute e^1 : $\boxed{1} \boxed{e^x}$ (to temporarily see the 12 digits, you can press $\boxed{\text{SHOW}}$ and *hold* it down): 2.71828182846

To get π , just press $\boxed{\pi}$ and $\boxed{\text{SHOW}}$ to peek:
3.14159265359

As for that bizarre and hairy fraction, do this:

$\boxed{1} \boxed{e^x} \boxed{\pi} \boxed{10^x} \boxed{\times} \boxed{\pi} \boxed{10} \boxed{e^x} \boxed{\times} \boxed{+}$

Answer: 0.0544

Perhaps that last one threw you a bit.

Remember to treat a number base and its exponent together – as if they were enclosed together in parentheses. When looking at the top half (the numerator) of the fraction, you should see $e(10^\pi)$ – two numbers, not three. The same goes for the bottom (the denominator) of the fraction: $\pi(e^{10})$. If you do this, then knowing about RPN arithmetic and how to work from the insides of parentheses outward, the problem is much easier, right?

Common Trigonometry

You're slowly adding to your repertoire of workbench tools, aren't you?
You now know about the exponentiation tools within your easy reach.

Next, move down a row, and look at those functions on the three right-hand keys. It's time you got comfortable with your basic trig tools....

What would you do, for example, if you encountered a problem like this?

$$\text{Atan}(\sin(22^\circ) + \cos(\pi/6) \div \tan(-125^\circ))$$

You'd whip out this book to review trigonometry, wouldn't you?*

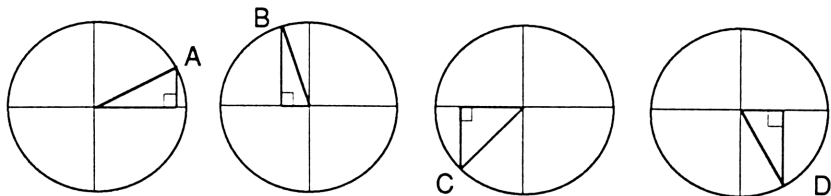
The word trigonometry itself means "the measure of the trigon." And if a pentagon is a five-sided figure, then a trigon must be...yep—a three-sided figure (also known as a triangle).

So *trigonometry* is "*the measure of the triangle*" – the *right* triangle, actually. And long ago, somebody noticed the quasi-mystic relationship between right triangles and circles.

Take a moment here and contemplate it....

*Of course you would.

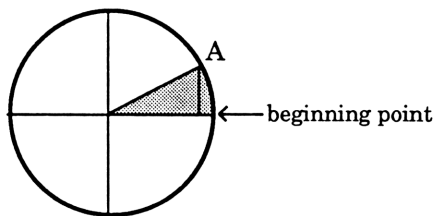
Every point on a circle (centered at the origin of a coordinate system) can define a right triangle:



Each triangle is formed from a radius through the point and a line segment perpendicular from the point to the horizontal (x -) axis.

The important question is how you know exactly *which* point on the circle you're talking about. The points above have been given names A, B, C, and D, but that isn't a very precise system.

Look at point A above. In order to get to A from the positive x -axis, you must "sweep out an arc:"



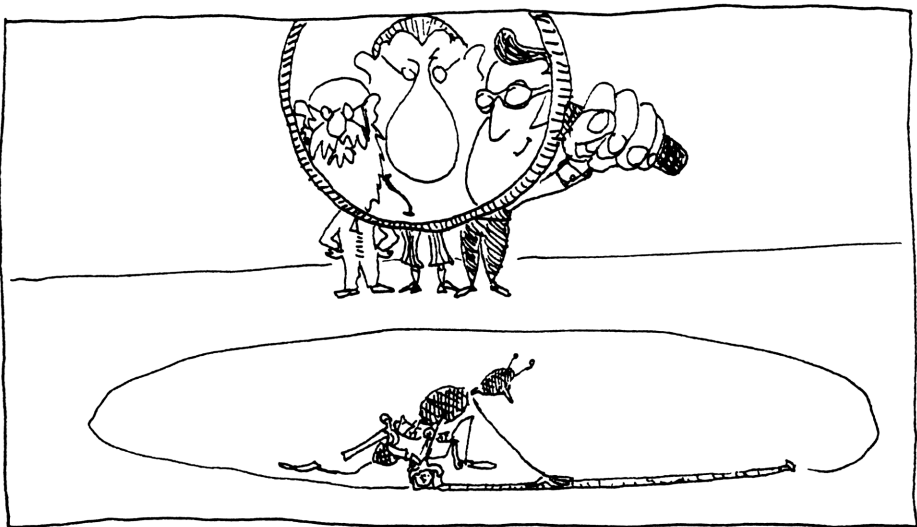
Thus, point A can be "renamed" – identified instead as a *measure* of the arc "swept out" by the hypotenuse as it "sweeps" from the beginning point (on the positive x -axis) counterclockwise to point A.

So, how do you measure this circular arc – in what units? The HP-42S offers you three different ways:

Degrees Radians Grads

One way to measure arc is to split up the circle's total swept angle into 360 equal pieces. There are 360 **degrees of angle** (or arc) in a circle.

Another common method to measure arc has to do with the distance an imaginary ant would cover if it were to start at the x-axis and crawl along the circle until it got to the point in question:



The total distance around the circle (its *circumference*) is always equal to 2π times the radius of the circle. So if the ant walks around the circle for a distance of exactly one radius, it will have walked an arc length of exactly one **radian**. There are 2π *radians of arc length* in a circle.

Thus, you can measure circular arc either by the angle it sweeps out (degrees) or by its length (radians).*

*The third method of measuring circular arc, as offered by your HP-42S, is not as common as the other two, but you should know what it is. It was developed probably because 360 degrees is an awkward, non-decimal number to work with. Thus, there are 400 **grads** of arc in a circle – or exactly 100 in each *quadrant* (quarter part of a circle).

The **MODES** Menu

So how do you tell your HP-42S what units you're using in your trig calculations – whether you're talking in, say, degrees or radians?

It's easy – you just...

Do This: Press **MODE** **MODE** (twice so that you don't EXIT after the first selection). Now look at the first three menu items.

These items are the three possible methods of measuring arc. One of them is selected at all times – the one with the little box next to the menu choice.

Right now, **DEG** is selected, so your HP-42S is "thinking" in degrees. But it doesn't know what *you* are calculating in – only what *it* is calculating in. And if you're calculating in radians and it's calculating in degrees...well, your answers will be garbage. You must keep in mind which MODE your machine is set to!

Actually, your HP-42S will remind you: Press the buttons to choose **RAD** and **GRAD**. Notice the little annunciators, RAD and GRAD, that appear when you do this. Now select **DEG** once again. See? If *neither* RAD nor GRAD is showing, then the machine is in degrees – and the menu confirms this.

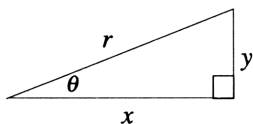
So, what do you know about your workshop's trig tools so far?

- Every point on a circle defines a right triangle, which can also be specified (more conveniently) by its arc measure;
- Your HP-42S lets you choose (with the **MODES** menu) between three different units of arc measure.

But do you remember why you want to measure arcs in the first place? What does all this have to do with SIN, COS, and TAN—you know...trig?

The *ratios of the sides* of these right triangles are what make them so handy for finding distances and angles that you can't easily measure – such as the height of a skyscraper or the distance across a lake.

Here are those ratios for a right triangle defined by an arc measure of angle θ :



$$\text{SIN}(\theta) = \frac{y}{r}$$

$$\text{COS}(\theta) = \frac{x}{r}$$

$$\text{TAN}(\theta) = \frac{y}{x}$$

$$\text{cosecant}(\theta) = \frac{r}{y}$$

$$\text{secant}(\theta) = \frac{r}{x}$$

$$\text{cotangent}(\theta) = \frac{x}{y}$$

Thus, for example, to find SIN(22°), you must:

- find the point on the circle at 22 degrees of arc from the beginning point (on the positive x -axis);
- look at the right triangle defined by that point;
- calculate the ratio y/r for that triangle.

Now Then: No kidding – find $\text{Atan}(\cos(\pi/6) \div \tan(-125^\circ) + \sin(22^\circ))$

Gulp: In this ugly problem, you have *mixed units*. Whenever you see the little $^\circ$ mark, the angle is being expressed in degrees; *otherwise, it's in radians*.

Also, you haven't been told in which angular units the final ATAN should be calculated.

And keep in mind that $\text{ATAN}(x)$ means "the Arc (or Angle) whose TANGent is x ." There happen to be a lot (mathematicians say "an infinite number") of angles that have x as their TANGent (and the same is true for ACOS and ASIN) – because you can continue to sweep out arc by going around the circle repeatedly (after all, 22° is the same point on the circle as $36,000,022^\circ$, right?).

So your HP-42S makes the assumption that you want the *first* angle it encounters that has x for its TANGent. Now, knowing all *that*, how do you solve the problem? Start in radians mode: **[MODES] [RAD]**. Then

**[π] [6] [+][COS] [MODES] [DEG] [1][2][5][+/-][TAN][+]
[2][2][SIN][+][ATAN]**

Answer: 44.4506 (*degrees*. You know this because this is the last MODE you set – and neither the RAD nor the GRAD annunciator is showing).

Yes, But: What if you now want to see this result in radians? Do you need to repeat the entire calculation?

Nope: Press **◀CONVERT▶CONVERT**. This menu allows you to actually alter the number in the X-register from one set of angular units to another (by contrast, when you switch in and out of RAD mode, for example, this affects only *subsequent* results – not the number already sitting in the X-register).

To convert from DEGreEs to RADians, choose **→RAD** and voilà! Answer: 0.7758

To convert back again, just press **→DEG**.

Get This: You can even convert to and from Degrees, Minutes and Seconds! For example, when you solve for an answer in degrees (as you just did), it is given to you in "decimal degrees." That is, this 44.4506 means "44 degrees, plus 45.06/100ths of a degree."

But press **→HMS** ...bingo! You now have 44.2702, in HH.MMSS format (44 degrees, 27 minutes, 02 seconds). It's HMS (not DMS) because H is for Hours; hours and degrees both have 60 minutes of 60 seconds each.

Press **→HR** now to convert back to decimal degrees.

One More: Find $\cos^2(37^\circ) + \csc^2(48^\circ) + \sec^2(156^\circ)$

Solution: There are two new problems here. First of all, in what order do you use $\boxed{\text{COS}}$ and $\boxed{x^2}$? Is it first the $\boxed{\text{COS}}$ and then the $\boxed{x^2}$ – or vice versa?

The correct method is the first one: you need to calculate the COSine and then square the result.

Second, how do you calculate \csc and \sec ? There are no keys for those functions!

Well, if you noticed back on page 86, \csc (cosecant) and \sec (secant) are the exact reciprocals of SIN and COS , respectively. This means you'll need to use the reciprocal key, $\boxed{1/x}$, to calculate these functions. To find $\csc(48^\circ)$, for example, you'd first find $\sin(48^\circ)$, then press $\boxed{1/x}$.

Now, knowing all that, turn to the task at hand:




$\boxed{3}\boxed{7}\boxed{\text{COS}}\boxed{\blacksquare}\boxed{x^2}$
 $\boxed{4}\boxed{8}\boxed{\text{SIN}}\boxed{\blacksquare}\boxed{x^2}\boxed{1/x}\boxed{+}$
 $\boxed{1}\boxed{5}\boxed{6}\boxed{\text{COS}}\boxed{\blacksquare}\boxed{x^2}\boxed{1/x}\boxed{+}$

Answer: 3.6468

Chances are, you're heartily sick of trig by now. But it's nice to know you have all this right there "near your workbench" (on the keyboard).

Speaking of chances, you haven't yet had a chance to explore one of those menus you see sitting on the shelf in the HandTool Storage Area (see page 18). Since you've seen some of what the keyboard has to offer directly to your "workbench," how about a quick excursion into the "tool room" to see what you can "bring" from there?

The PROB Menu

And speaking of chances, how are you at probability? As an example of a set of hand tools stored under a menu key, look now at the  PROB key and its menu (press  PROB  PROB now), and try this






Example: If you have 6 red marbles, 5 orange, 4 yellow, 3 green, 2 blue, and 1 purple – all in a leather pouch – how many possible *combinations* of six marbles (any color and never mind the order) can you draw out of the pouch?

What if you count *permutations* – where the order does matter – but you take them out only 3 at a time?

Solution: You're really just asking how many ways you can combine 21 objects, taking them 6 at a time. The COMBina-tions selection in this PROB menu will tell you; press:

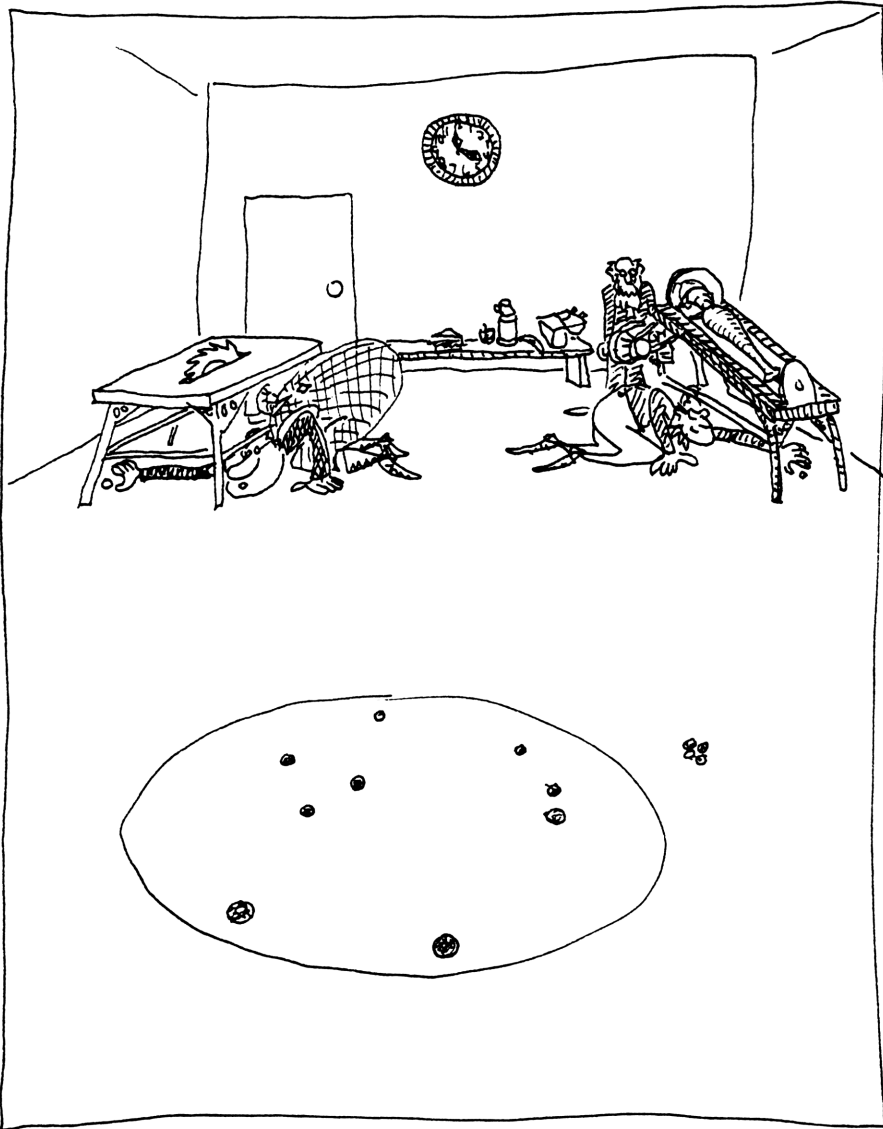
  ENTER   Answer: 54,264.0000

If you want PERMutations in sets of three, then press:

  ENTER   Answer: 7,980.0000. Press .

Aren't menus great? And what a collection of tools you have! Can you imagine, for example, what it would be like trying to do such probabilities calculations on your own – without such easy and powerful tools? You might well end up...

...losing your marbles.



A Deep Breath

To avoid even further marble loss, you'd better stop for a moment now and survey where you've been and where you're going. This machine is so jam-packed with stuff to learn that it's going to be very easy to get lost in the forest while you're trying to learn about each tree:

- In the First Tour of your "workshop," you got an orientation of the shop with the Big Picture, hopefully giving you a general idea of what the machine was all about and how it was organized.
- Now in this chapter on Tools, so far you've learned only about the Stack, RPN arithmetic, and some other some number-crunching tools, including exponentiation, trig and probability.

So, what's next? Is this chapter finished now?

Not yet. Actually there are *many* other tools, but they don't all appear in this Tools chapter. That would be a **HUGE** chapter – way too much. Besides, many of those tools wouldn't even make sense to you yet.

Sure – there are some more you're ready for now – you'll see those next. But others deal with **variables** (they'll be covered in the next chapter) or **programming** (the chapter after that). And the **Power Tools** are even big enough for a chapter of their own.

So don't worry – you'll encounter each tool when it makes the most sense to. Just keep this larger perspective as you continue now....

A Complete Inventory Of Tool Menus









To help keep that larger perspective, go back and look at the Big Picture (pages 18-19) once again.

You know now about the Stack, your arithmetic keys, Stack manipulation keys, and some of the other "keyboard functions" that form your "workbench and tools within easy (keyboard) reach." But now look for a moment at all the tool *menus* in your workshop – menus that are part of the different Storage Areas:

Hand Tools Menus

Menu

Uses Of Tools

 CLEAR	Clear specific data or storage areas of your machine;
 DISP	Control format of displayed data;
 MODES	Set default data units and other machine features;
 CONVERT	Convert data between various units and notations;
 PROB	Collection of probability functions;
 FLAGS	Control flags – useful in writing programs;
 PGM.FCN	Collection of other functions useful in programming;
 PRINT	Control output of machine to infrared printer;

(Up to now, you've seen the first five of these menus, because they're useful in "number-crunching" on the Stack. But as for the others, you'll see them when you learn about programming, later.)

Special Use Menus

Menu

Use

- | | |
|------------------|--|
| ■ CATALOG | Gives you access to and a complete inventory of all functions, programs, and variables available in machine; |
| ■ ALPHA | Gives you access to the ALPHA register and characters; |
| ■ CUSTOM | Calls a special menu that you have "custom-built" as a convenient set of often-used tools and variables; |
| ■ ASSIGN | Allows you to actually build the ■ CUSTOM menu by assigning tools and variables to it. |
| ■ TOP.FCN | Allows the selection of the top-row keys via a menu; |

(Of these, you seen only the ■ **CATALOG** menu so far, and now that you know the basics of RPN "number-crunching" on the Stack, the other Special Use Menus are what you'll see next.)

Power Tools Menus

Menu

Use

- | | |
|---------------------------------|---|
| ■ BASE | Allows you to use of different number bases; |
| ■ $\int f(x)$ | Allows you to calculate definite integrals; |
| ■ MATRIX | Allows you to calculate and manipulate matrices; |
| ■ SOLVER | Allows you to solve an equation for any variable in it; |
| ■ STAT | Gives you access to statistical functions and analyses. |

(Each of these power tools is involved enough to get a section all to itself – but that's later – after you study variables and programming.)

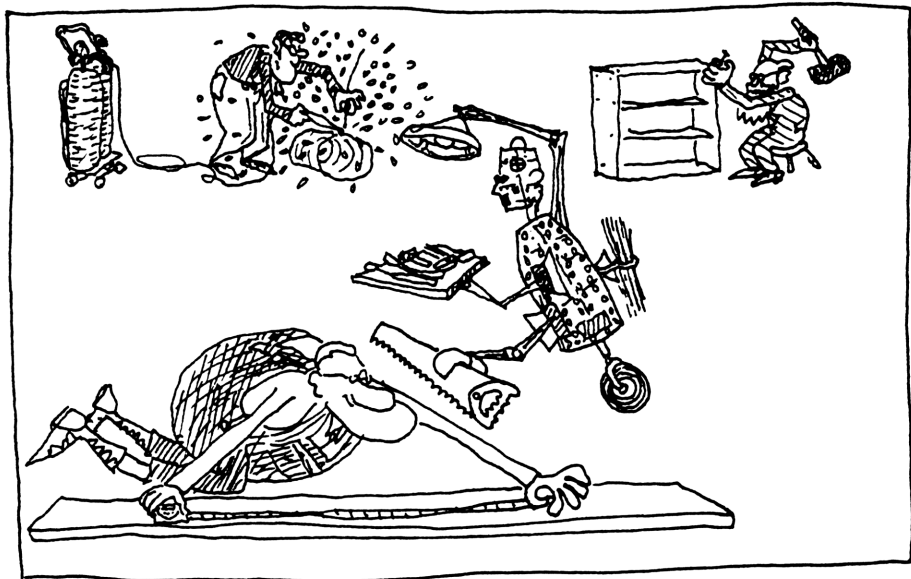
Special Use Menus

Special Use Menus are named such because they just aren't like the other tool menus. Make no mistake: They *are* tools – and important ones. But they're somehow related to other menus in unusual ways, and so they fall into this catch-all category.



For example, you've already seen the **CATALOG** menu (page 43). Remember how it worked? You could use it to find and select any tool or variable you wanted – no matter in what menu you could otherwise find that tool or variable.


In this way, **CATALOG** was a "menu of menus" – a sort of master menu (recall how it's portrayed in the Big Picture – pages 18-19 – as presenting you access to the storage areas through a set of windows).

Well, one way to think about the other Special Use Menus is that they all provide some kind of "special access" to certain tools or menus.







The ALPHA Menu

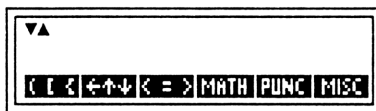
The  ALPHA menu is a Special Use Menu. It's "where to go" whenever you need to "spell out" anything. You may want to spell out a command, a variable name, or build a character string (as data) to store, but no matter what you spell – and no matter where you are in the workshop – the "spelling tools" of the  ALPHA menu are always available to you.

Try This: Press  ALPHA. You should see the ALPHA menu:



In this particular example, the menu is displayed under the current contents of the ALPHA register (it's empty). You "went" to the ALPHA register here because you pressed  ALPHA – no other keystrokes. That told the HP-42S to *save what you type as data in the ALPHA register*, rather than to use what you type as a command or a variable name.


Now This: Notice the ▼▲ annunciator. Press either  or  to see the other page of the  ALPHA menu.







You get quite a wide variety of characters to type, no?


So, how's your typing?




Practice: Put HELLO THERE! into the ALPHA-register



Solution: First, press  to get back to the first page of the menu.




Then:  H  E  L  L  O
  T  H  E  R  E
  !

Now, you can press either  or  to clear the menu and leave the ALPHA register...

Are your characters still saved in the ALPHA register?
Press  again to check....HELLO THERE!

In this particular case, you "called" the  menu right from your "workbench" (the Stack). But keep in mind that you can also "call" the  menu from within *any* other menu – and when you finish with your typing (with ) , the machine will return you right to the menu from which you "called."

Prove It: Enter another menu, say, . Now call the  menu. There's the ALPHA register again: HELLO THERE!

Now press  or ....Sure enough – you're back at the  menu, from where you did the "calling."

The **■** **ASSIGN** And **■** **CUSTOM** Menus

Here are two more Special Use Menus.

The main task of the **■** **ASSIGN** Menu is to help you *create* your own **■** **CUSTOM** Menu – gathering together whatever collection of tools and variables you might find convenient.

Watch: Press **■** **ASSIGN**. Here's what you'll see:



Notice that this **■** **ASSIGN** menu is nearly identical to that of **■** **CATALOG** (recall page 43).

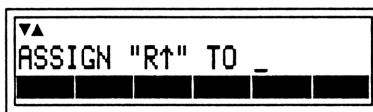
■ **ASSIGN** will let you search out and collect any functions (**FCN**), programs (**PGM**), or variables (**REAL**, **CPX**, or **MAT**) that you use frequently, putting them all together on one handy "shelf" – the **■** **CUSTOM** Menu. This searching procedure works just like that in **■** **CATALOG**; thus the similarity of the menus.

So, what items do you want to gather into your **■** **CUSTOM** Menu?

How About: Back on page 77, you noticed how useful the "roll up" tool could be for Stack manipulations. The problem was, it took *a lot* of "thumbing" through the pages of the **CATALOG** menu to get to it.

Well, here's your chance to make it handier by putting it on your **CUSTOM** Menu!

Here's How: Make sure that your **ASSIGN** Menu is showing, then press: **FCN** **▲▲▲▲▲▲▲▲▲▲▲▲▲▲▲▲** **R↑**. You should see:







Since you signified a completed selection when you pressed **R↑**, the machine is now showing you your **CUSTOM** menu (which is empty as yet).

You need to select the menu position where you'd like **R↑** to be placed in your **CUSTOM** Menu. Put it in the left-most slot for now (press the **Σ+** key).


Not too tough, right?





But did you notice that when you pressed **ASSIGN**, the display showed some quotation marks – as if the machine expected you to spell out the name of what you were going to assign?

Yep – as a matter of fact, you can do it *either way*: You can select what you want from the  **ASSIGN** menu or (if that's too much trouble – and it *was* a bit of a pain here – with all the menu pages to flip through) you can simply spell out the proper name of the tool.



Try Spelling: Assign the  **R↑** tool to your  **CUSTOM** menu again – but this time *spell out* its name rather than selecting it from the  **ASSIGN** menu.

Solution:  **ASSIGN**  **ALPHA**  **RSTUV**  **R**         **ALPHA**

See how the  **ALPHA** key helps you spell out the name (and signal when you're done)?

Now all you need to do is choose where to put this second  **R↑** on the  **CUSTOM** menu (yes, you can put multiple "copies" of the same tool on the  **CUSTOM** menu): put it next to the first one (press the  **1/x** key).

Notice that when you used the  **ALPHA** key here, you didn't need to press  first.

This is usually true whenever you're going to use the **ALPHA** menu to spell out the *name* of a tool, variable or command (rather than put actual data into the **ALPHA** register): you don't need to press  first; the context of your other keystrokes (in this case,  **ASSIGN**) will tell the machine what you mean.

Now Undo It: You don't really want two copies of the **RT** tool on your **CUSTOM** menu, so here's a good opportunity to learn how to "unassign" an item from that menu:

You simply *assign a nameless item* there instead – by spelling out no name at all:

ASSIGN **ALPHA** – but don't spell out anything – now press **ALPHA** again (to end the "spelling"), and select the second **RT** on the **CUSTOM** menu...bye-bye!

One More: Just so you know – besides tools, you can also put variable names on your **CUSTOM** menu.

Recall, back on page 41, that built-in matrix variable, REGS. You can save **REGS** for handy access in your **CUSTOM** menu (put it in, say, the fourth "position").

Here's How: **ASSIGN** **MMT** **REGS** **LOG**

Of course, you could have spelled it out instead:

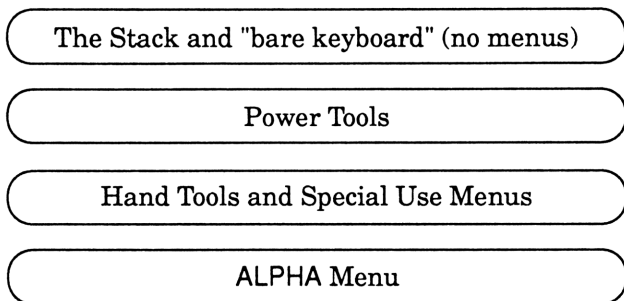
ASSIGN **ALPHA** **RSTUV** **R** **ABCDE** **E** **Fghi** **G** **RSTUV** **S**
ALPHA **LOG**

But that would have been the less convenient method in this case, right?

Your Tools Can Work Together

As you learn about these Special Use Menus, it's important to realize that there's a certain *hierarchy* to all the menus in your HP-42S.

That is, there are some fairly simple rules about which menus you can "call" from which – and whether you will return to the "calling" menu after finishing with the "called" menu.




If you call a menu *lower* than where you are, when you return, it will be back to the menu from which you did the calling.








However, if you call a menu *higher* than your own level, you'll actually be "jumping" to the highest level (the Stack) and "calling" from there; for when you return, you'll return to the Stack.

Of course, whether your returns are automatic or require you to use the **EXIT** key will depend on the circumstances. This hierarchy merely shows your return "destinations." Experiment with it until you get the idea. Where do you return from a call at the same level where you are?

The TOP.FCN Menu



With this hierarchy in mind, it's a lot easier to understand the reason for this last Special Use Menu. You might have wondered why the folks at HP put the six top-row keys onto its own menu as well....


Try This: "Call" the Power Tool  SOLVER. Don't worry about what you might be doing there. Just suppose you're there and now, for a side calculation, suppose you find yourself needing to compute $(1/3)^5$.


Normally, you'd press     . But  isn't  right now. None of those top-row keys are "themselves" – they're menu keys instead!

Solution:  TOP.FCN to the rescue:

Press  TOP.FCN  TOP.FCN      
Answer: 243.0000

Notice that you're using the  key with a menu choice here (because  is an orange function on the keyboard).

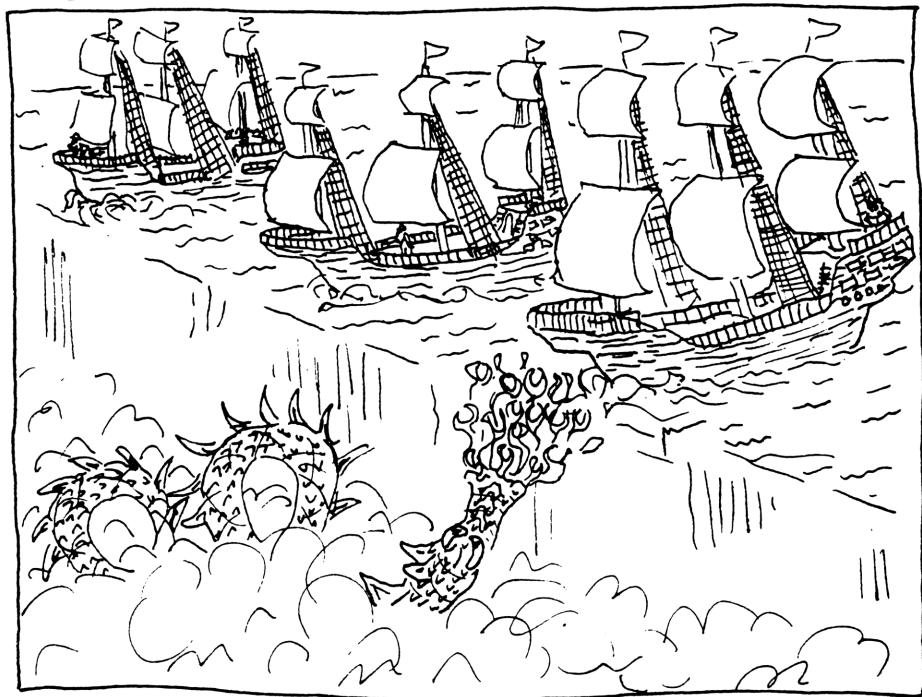
This is the only menu in which you can do that;  TOP.FCN exists solely to preserve convenience in the hierarchy.

Press  a couple of times now to return to the Stack.

Well, by now you must be feeling like a graduate of "Mr. Goodbench Tool School."



Over the last 30 pages or so, you've seen quite an inventory of the various tools in your HP-42S, and its time you "graduated on to the next part of the Course" to build something with all these tools.

But before you get going, maybe you'd better check your understanding so you don't get bogged down later on.



Yes fans – that's right – ready or not, here it comes....

Tool School Reviewal

1. Find $\sqrt[8]{6561}$, using two different methods.
2. Find $\frac{\log\left(12^2 + \frac{2704}{3}\right)}{e^3}$
3. Find $9 + \sqrt{(12 \times 3.3)^{1.4} - 75}$
4. If eighteen U.S. states and three Canadian provinces meet to decide how to stop acid rain, how many ways are there to cast just the minimum majority of "no" votes to prevent any action?
5. An arc measure of 180° is equivalent to π radians. Compute $\sin(180^\circ)$ and $\sin(\pi)$ on your HP-42S.
6. Find $\text{Acot}(3\pi/4)$. Express it in degrees, minutes and seconds.
7. What display do you see when:
 - (a) you enter a Power Tool Room?
 - (b) from that Power Tool Room, you select a hand tool menu?
 - (c) from that hand tool menu, you use a particular hand tool?
 - (d) you then press  **ASSIGN**  **ENTER**?

Tool School Reviewal Solutions

1. $\boxed{6}\boxed{5}\boxed{6}\boxed{1}\boxed{\text{ENTER}}\boxed{8}\boxed{1/x}\boxed{\text{■}}\boxed{y^x}$ or $\boxed{6}\boxed{5}\boxed{6}\boxed{1}\boxed{\sqrt{x}}\boxed{\sqrt{x}}\boxed{\sqrt{x}}$
Answer: 3.0000

Finding the "eighth root" is the same as "taking the 1/8th power."
 And then $(6561)^{1/8}$ is the same as $((((6561)^{1/2})^{1/2})^{1/2})^{1/2})^{1/2}$, which is also:
 $\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{6561}}}}}$.

2. $\boxed{1}\boxed{2}\boxed{\text{■}}\boxed{x^2}\boxed{2}\boxed{7}\boxed{0}\boxed{4}\boxed{\text{ENTER}}\boxed{3}\boxed{+}\boxed{+}\boxed{\text{LOG}}\boxed{3}\boxed{\text{■}}\boxed{e^x}\boxed{+}$
Answer: 0.1503

3. $\boxed{1}\boxed{2}\boxed{\text{ENTER}}\boxed{3}\boxed{\cdot}\boxed{3}\boxed{\times}\boxed{1}\boxed{\cdot}\boxed{4}\boxed{\text{■}}\boxed{y^x}\boxed{7}\boxed{5}\boxed{-}\boxed{\sqrt{x}}\boxed{9}\boxed{+}$
Answer: 18.8739

As always, work from the "innermost" parentheses or expressions "outward."

4. The question is "How many different *combinations* of 11 'no' votes are possible among the 21 voters?" (combinations, not permutations, because the order of voting doesn't matter).

A piece of cake for the $\boxed{\text{■}}\boxed{\text{PROB}}$ menu:

Press $\boxed{2}\boxed{1}\boxed{\text{ENTER}}\boxed{1}\boxed{1}\boxed{\text{■}}\boxed{\text{PROB}}\boxed{\text{COMB}}$
Answer: 352,716.0000

5. $\sin(180^\circ)$:  Answer: 0.0000
 $\sin(\pi)$:  Answer: -2.0676E-13

The difference is due to the fact that you're not really taking the SIne of π radians. You're taking the SIne of 3.14159265359 radians – and that's different. Like any finite machine, the HP-42S is limited to using an *approximation* of π , not π itself. Nobody can ever use the "exact" value of any irrational number (one with an infinite number of non-repeating digits).

Your HP-42S uses an approximation rounded at the 12th digit: 3.14159265359. Since the true value of π is 3.14159265358... the 12-digit rounding represents a slightly *upward* rounding. Therefore, when "sweeping around" to find the corresponding point on the trigonometric circle, your HP-42S goes slightly *past* where it should be – exactly on the negative x -axis – thus extending slightly *below* that axis and producing a negative SIne (which, after all, is the *vertical* leg of the corresponding right triangle).

Interestingly enough, if you take the COSine of $\pi/2$ radians, you *will* get zero, but *only because the error happens to be "further out" than the last digit used in this machine* (i.e. a large computer that carries more digits should *not* return zero).

The point is, *your machine is being exactly faithful to the numbers it is working with*. It just never works with any truly irrational value (e , π , $\sqrt{2}$, etc.). And if you find a calculator (certainly not any HP machine) that will give you *exactly* zero when you ask for SIne(3.14159265359) – or *exactly* two when you ask it to square 1.41421356237 – feel free to be outraged: it's doing bad math.

6. Hmm... there's no ACOT tool, is there?

Nope. That's a problem....Ah, but isn't $\text{Acot}(3\pi/4)$ the same as $\text{Atan}(4/3\pi)$? (think about it – and review page 86 if you need to).

To compute this, first make sure the machine is in RAD mode (by pressing **MODE** **RAD**, if necessary).

Then press: **3** **π** **×** **4** **÷** **ATAN**

Answer: 0.4014 radians







To convert this to degrees, minutes and seconds, first you need to convert from radians to degrees:

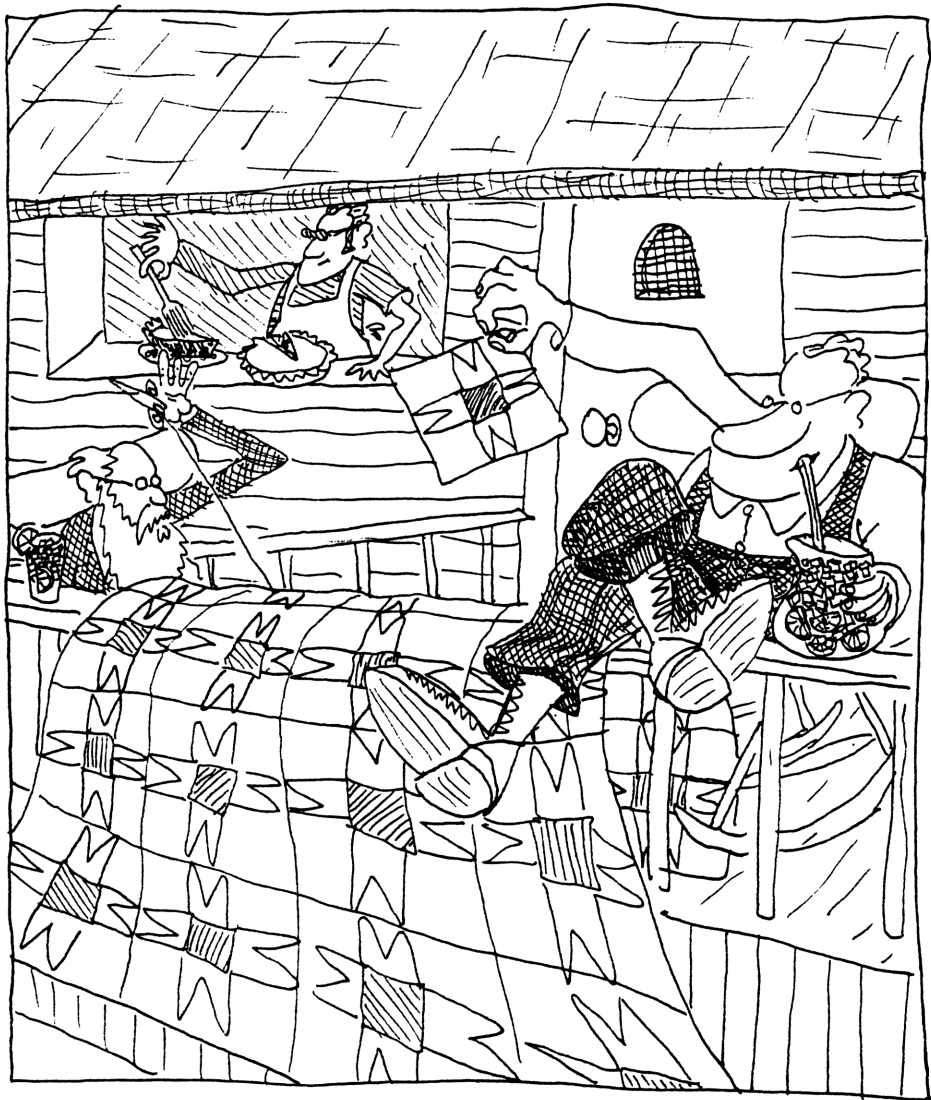
CONVERT **→DEG**

Result: 22.9970 (these are *decimal* degrees, remember)

Now convert to degrees, minutes, seconds: **CONVERT** **→DMS**

Answer: 22.5949 (which means: 22° 59' 49")

7. (a) When you enter any Power Tool Room, you'll see the menu for that Power Tool. For example, press:  **SOLVER**.
- (b) When you then select a hand tool menu, you'll see (the first page of) that menu. For example, press  **MODES**.
- (c) When you select a hand tool from that particular Menu, the display will automatically revert back to the Power Tool (unless you had pressed the hand tool menu key *twice* upon entering that menu – step (b) above – in which case, the display would return to the hand tool menu). For example, press  **DEG**.
- (d) Anytime you press  **ASSIGN**  **ALPHA**, you'll see the  **ALPHA** menu, indicating that the calculator is prepared for you to spell out the name of a variable or a tool.



YOUR RAW MATERIALS:
Data And Variables

Handling Data

If the functions and operations are the "tools" in your HP-42S workshop, then your "raw materials" are the data you "crunch" with them.

Remember (from page 36) that your HP-42S can handle four types of data: real numbers, ALPHA-characters, complex numbers, and matrices. It's time now to learn the details about how to name, store and retrieve these various data types.*

How Do You Name It?

Most of the data you've used so far have been only *temporarily* useful. That is, you used the numbers only once during a quick problem-solving session; it hardly seemed necessary to name or save them.

But suppose instead that you're reusing a particular number frequently in your calculations – and you're tired of keying it in every time. What would you do?

You would save this number in one of those data-bins you can create and name – a *variable*.

*Unless, of course, you already feel comfortable with some of these things, in which case you should skip ahead. So, if you already know about storing and retrieving real and ALPHA variables, skip to page 126; if you also know about complex variables already, go to page 134; and if you also know about matrices already, jump ahead to page 155.

Example: Put the natural logarithm base, e , into a variable named EE.

Solution: First, obtain the value of e (at least its 12-digit approximation): press $1 \blacksquare e^x$.

Now press $\text{STO} \text{ALPHA}$. You'll see the following:



Here's that ever-useful $\blacksquare \text{ALPHA}$ Menu again, waiting for you to spell something out. In this case, it's the name of the variable in which you're going to STO re e .

Notice that with STO you don't need to press the \blacksquare key before pressing ALPHA – just as with the $\blacksquare \text{ASSIGN}$ Menu (recall page 100): the fact you just pressed STO tells the machine that you may want to spell something out now, so if you press ENTER , it assumes you really mean ALPHA .

So, press $\text{ABCDE} \blacksquare \text{E} \text{BCDE} \blacksquare \text{E}$ to spell "EE_", then press ALPHA again to tell your machine you've finished your typing.

That's all there is to it!

Try Another: Store the value 6.92 into a variable named FRED.

Solution: $\boxed{6} \cdot \boxed{9} \boxed{2}$ puts the number in the X-register. $\boxed{\text{STO}} \boxed{\text{ALPHA}}$ then signals that you're going to store it in a variable whose name you're about to type. Then:

$\boxed{\text{F}} \boxed{\text{G}} \boxed{\text{H}} \boxed{\text{I}} \boxed{\text{F}} \boxed{\text{R}} \boxed{\text{E}} \boxed{\text{D}} \boxed{\text{E}} \boxed{\text{F}} \boxed{\text{R}} \boxed{\text{E}} \boxed{\text{D}} \boxed{\text{E}} \boxed{\text{D}}$ types in "FRED_" and $\boxed{\text{ALPHA}}$ signals the end of your typing...Zing! – you've just stored the value 6.92 into FRED!

Yes, but how do you *know* that your calculator has really STORed your numbers? Is there a way to find out for sure – to verify this?

Naturally: Press $\boxed{\text{RCL}}$.

Lo and behold, you now have $\boxed{\text{FRED}}$ and $\boxed{\text{EE}}$ – menu items on the list of things you can ReCaLL!

So press $\boxed{\text{EE}}$ to bring that value, e, back into the X-register.

See how $\boxed{\text{RCL}}$ works in opposition to $\boxed{\text{STO}}$? And as soon as you've named a variable, it will appear on the $\boxed{\text{STO}}$ and $\boxed{\text{RCL}}$ menus – so you don't need to keep spelling it out every time you want to use it – very handy!

All About **STO**

There are a few important things to know about the **STO** key and what you're really doing when you use it:

- First of all (as you probably already know), **STO** stores the value that's currently sitting in the X-register. This means that if you want to store a value now sitting somewhere else, you'll have to bring it to the X-register first (probably by using the **RCL** key), right?
- Secondly, **STO** is a *copying* process. That is, when you **STO** a value into a variable, you are storing a copy of the value that's currently sitting in the X-register. Notice in the previous examples that when you finished **STO**ring, the value was also still in the X-register; only a "clone" was sent into the variable.
- Since it *is* a copying process, **STO** doesn't affect the Stack very much. It certainly doesn't lift the Stack or alter any value in the Stack registers. But keep in mind, that, like most functions on your HP-42S, **STO** will leave Stack Lift *enabled* – so that if the *next* action taken brings a number to the X-register, the Stack will preserve the X-register's previous contents by bumping the whole Stack up.

All About **RCL**

And now notice these details about **RCL** (compare them with **STO**):

- **RCL** recalls the value from wherever you specify and brings it to the X-register – nowhere else.
- Whenever you **RCL** a value from a variable, you are actually making a *copy* of that variable's value and *placing this copy* in the X-register.
- **RCL** does affect the Stack of course, since it's bringing a value to the X-register. Does this new value "bump" the Stack up in a Stack Lift? That depends. ***RCL** behaves just as if it were "keying in" the value:*

If Stack Lift has been left *disabled* by the previous action (say, **ENTER** or **↔**, for example) then this (copied) value that **RCL** is bringing to the Stack will *overwrite* whatever had been in the X-register previously – just as you would do if you were keying in a value under these circumstances.

However, if Stack Lift has been left *enabled*, **RCL** will bump everything up and insert its new value underneath – again, just as it would work for you if you were keying in a value to an enabled Stack.

See how **STO** and **RCL** work together to help you save and retrieve data conveniently? To make sure you understand, try a few more....

Examples: Press **4****ENTER****3****ENTER****2****ENTER****1** to fill up your Stack. Now press **RCL** **FREQ**.... Did the Stack Lift when you performed this **RCL**?

Solution: Yep – press **R↓** four times to check....The **1.0000** that had been in the X-register is now in the Y-register. This is because the keystroke immediately prior to the **RCL** was a digit-entry keystroke (a **1**), which left Stack Lift *enabled*. Remember that most keystrokes – including the digit-entry keys – leave Stack Lift enabled.

Compare: Now press **4****ENTER****3****ENTER****2****ENTER**. Then **RCL** **FREQ**. Did the Stack Lift this time?

Nope – because the previous keystroke was an **ENTER**, which always *disables* Stack Lift, right?

What If: You press **4** **STO** **FREQ** **RCL** **FREQ**. How many copies of **4.0000** will this produce? Where will they be located?

Solution: You'll have *three* separate values of **4.0000**: the original you key in, the copy of this original that you save into **FREQ** by pressing **STO**, and then a copy of this copy that you put back into the X-register when you press **RCL**. That's two **4.0000**'s in the Stack (the original in the Y-register, the second copy in the X-register) and one (the first copy) in **FREQ**!

And now, discover a few more places you can store things:

Try This: Calculate 5.67×7.83^3 and store the result in the data register R_{08} (recall from page 41, these registers are the individual elements that make up that *built-in matrix* called REGS. By special design, remember, you're allowed to refer to these elements individually by name).

Solution: $\boxed{7} \cdot \boxed{8} \boxed{3} \boxed{\text{ENTER}} \boxed{3} \boxed{\text{Y}^\times} \boxed{5} \cdot \boxed{6} \boxed{7} \boxed{\text{X}}$ does the calculation, leaving the answer in the X-register, as always.

Now $\boxed{\text{STO}} \boxed{0} \boxed{8}$ stores a copy of this result into the data register R_{08} . That's how simple it is to use that built-in matrix, REGS: you don't need to refer to REGS itself; you simply name which of its elements you want to use!

Challenge: Put 8.7900 into the Z-register without using $\boxed{\text{MT}}$ or $\boxed{\text{R0}}$.

Solution: Press $\boxed{8} \cdot \boxed{7} \boxed{9} \boxed{\text{STO}} \cdot \boxed{\text{ST} \text{ 2}}$. Surprise! You've just discovered the hidden Stack Menu, which you find by pressing either $\boxed{\text{STO}} \cdot$ or $\boxed{\text{RCL}} \cdot$. The \cdot says you want to $\boxed{\text{STO}}/\boxed{\text{RCL}}$ to/from a Stack register. In this case, you're copying the value in the X-register (8.7900), storing it directly into the Z-register. And what happens to the number that used to be in the Z-register? It's destroyed. Remember that $\boxed{\text{STO}}$ always *overwrites* the previous value in the "destination" variable or register.

There's one other clever thing that **(STO)** can do, too: It can work in combination with the four basic arithmetic operators **(+)**, **(-)**, **(X)**, and **(÷)** to alter the contents of a variable or register.

Example: Multiply the value of FRED by 5 *without bringing that value to the X-register*.

Solution: Press **(5)(STO)(X)(FRED)**. The **(STO)(X)** combination multiplies the value in FRED by the value in the X-register.

And where does the answer end up? It's *not* in the X-register – it's in FRED! Press **(RCL)(FRED)** and see that it no longer contains 6.9200 (its previous value), but now 34.6000 (fives times as much).

It's as if you had temporarily moved the "arithmetic site" to FRED: When you pressed **(STO)(X)** (instead of just **(STO)**), the 5.0000 from the X-register was copied into FRED, but instead of overwriting the *previous* value in FRED (6.9200) this "bumped it up somewhere" for a moment – as if FRED had a "Y-register." Then the **(X)** combined the two values and left the result right there in FRED!

So this is how to use the **(STO)** key as a shortcut, doing two things simultaneously: arithmetic and storage. Instead of having to do **(RCL)(FRED)(5)(X)(STO)(FRED)**, all you need to do is **(5)(STO)(X)(FRED)** ! And you can do similar "on-site" arithmetic with the other operations, too.

OK, so how about **RCL**? Can you do this kind of shortcut with the **RCL** key, too? Hmm... how do you suppose a keystroke sequence like **RCL** **+** would work?

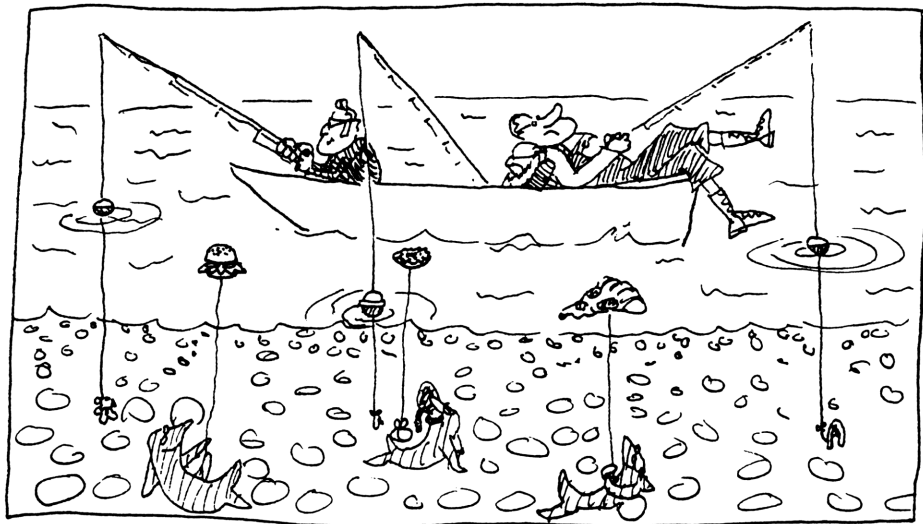
Try This: Divide the value of **EE** into the value of **R₀₈** and take the square root of the result.

Solution: **RCL** **0** **8** **RCL** **+** **EE** **√x**

Answer: 31.6437

In the case of **RCL**, it's pretty clear what's happening – and not that much of a shortcut: Normally, you would say **RCL** **EE** **÷** **√x**; here you're saying **RCL** **+** **EE** **√x**.

Using "recall-arithmetic" is so straightforward that it's a good way to remind yourself how "storage-arithmetic works:" Just picture yourself looking at things from the variable's "point of view" instead of from the Stack – so that your **STO** commands seem to be "recalling" values to a "Stack" located *in* the variable!




Real Variables

All this storing and recalling you've been doing so far has involved just a couple of variables (EE and FRED).



And these are *real* variables because they contain real-number values, right? So you should have a good grasp of how to create real variables – but just to be sure, try one more.

Create: A real variable named LORI, containing the result of this calculation: $56.89 \div 3.56^{2.15}$.

Behold: 
That's the calculation (Answer: 3.7104).

Now for the creation of the variable in which to store that result:



And now, whenever you  or look in the , you'll find the variable LORI along with EE and FRED.



All this should seem pretty familiar by now, no?



ALPHA Data And The ALPHA-Register








All right, so much for real variables. But recall (from page 38) that a real variable is a named "storage bin" that can contain either one real number (as in EE, FRED and LORI) *or* an ALPHA string (up to six ALPHA characters), right? So, try one with a string now....

How 'bout: Put this ALPHA-string into a variable named HI:

HOWDY!

Shucks: First, press  ALPHA to get to the ALPHA-register. As you know from page 96, when you press  ALPHA like this – with no preceding keystrokes – you'll be "sent" to the ALPHA register, ready to key in some ALPHA data (characters). The *only* way to get ALPHA data into the machine is to key it into the ALPHA register (just as the *only* way to get numeric data into the machine is to key it into the X-register).

When you start to type, you'll *overwrite* whatever is now sitting in the ALPHA-register: 
.

Now press  ALPHA and key in the variable name: 
 ALPHA, then  to the Stack and 
, copying the value in  to the X-register: "HOWDY!"

Notice in that last example that your calculator began by showing **ASTO __** (instead of the usual **STO __**) when you pressed **[STO]**.

This is because you were at the ALPHA menu (in "ALPHA mode") at the time, thus **STOring** something directly from the ALPHA- register into the named variable. In other words, you were *Alpha-STOring*.

ASTOring works as a copying process similar to **[STO]**ring – except that you copy from the ALPHA-register instead of the X-register.

Try This: Put the ALPHA-string, **THE ANSWER IS**, into a variable named **YEP**.

Solution: Keying the string **THE ANSWER IS** into the ALPHA-register is no problem:

[ALPHA] [RSTUV] T [FGHI] H [ABCDE] E [▲] [MISC] [▲]
[ABCDE] H [NOPQ] N [RSTUV] S [WXYZ] W [ABCDE] E [RSTUV] R
[▲] [MISC] [▲] [FGHI] I [RSTUV] S .

Now **ASTO**re it into **YEP**: **[STO] [ALPHA] [WXYZ] Y [ABCDE] E**
[NOPQ] P [ALPHA].

Now press **[EXIT]**, to get out of ALPHA mode entirely, and check yourself by recalling a copy of **YEP** to the X-register: **[RCL] [YEP]**

You see **"THE AN"** in the X-register(!)

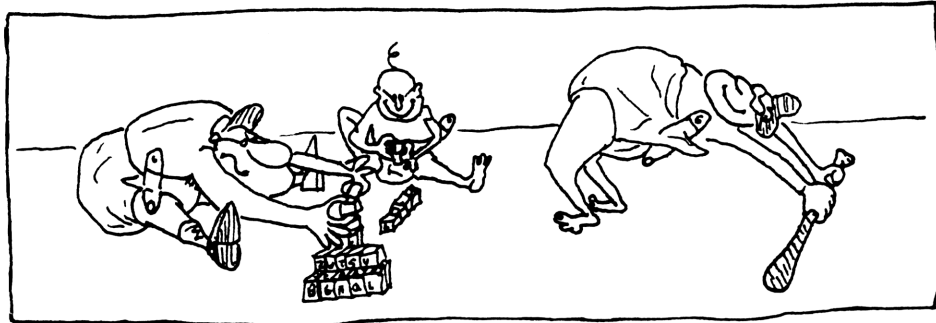
The Obvious Question: What happened to the rest of the string that was supposed to be stored in YEP?

The Logical Answer: It never got there. Remember that a real variable can hold at most only *six* ALPHA characters. So when you use ASTO, *it will copy only the first six characters from the ALPHA-register* into the variable you name.

Of course, the entire phrase, **THE ANSWER IS** is still in the ALPHA-register; nothing has happened to it (since ASTO is a copying command, like **STO**). But YEP has only the first six characters.

So while you're in ALPHA mode (i.e. you're at the ALPHA-register, having pressed **ALPHA**), both the **STO** and **RCL** keys are automatically converted into **A_{STO}** and **A_{RCL}** keys. Or, you can select them, **A_{STO}** and **A_{RCL}**, in a menu – under **CATALOG FCN** – if you'd rather.

Just remember that you're dealing with ALPHA data and the ALPHA-register (and that you can't do any math with it).

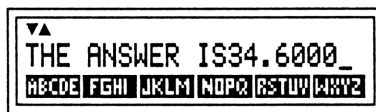


But you haven't used ARCL yet! The **[RCL]** you used in the last example was a normal **[RCL]** that put a copy of YEP's contents into the X-register (remember? you pressed **[RCL]** *after* you had **[EXIT]**ed from ALPHA mode).

Just as ASTO had a few details to learn about, so does ARCL....

Such As: Use **A[RCL]** to recall a copy of the contents of FRED to the ALPHA register.

Solution: Go to ALPHA mode (**[ALPHA]**) and press **[RCL]** (which is actually **A[RCL]**) **[FRED]**. You should see:




A calculator display showing the text "THE ANSWER IS34.6000_". Below the display is a keyboard layout with the following keys: ABCDEF, GHIJ, KLMN, OPQR, STUV, WXYZ.


Do you see what's happening here? ARCL is indeed a copying function (similar to **[RCL]**); it makes a copy of the contents of the named variable. But when this copy is sent to the ALPHA-register, it is *appended to* (tacked onto the end of) any characters that are already there.












Since you already had THE ANSWER IS in the ALPHA-register, ARCL appended to it the value it copied from FRED (34.6000).



So, whereas **[RCL]** may overwrite or "bump up" the contents of the X-register (depending on the status of Stack Lift), ARCL will always *append* to the ALPHA-register.

Yes, But: That phrase in the ALPHA-register isn't quite "pretty" yet. Fix this small glitch and put in the missing space.

Hmm...how *do* you edit the ALPHA-register, anyway? With the X-register, remember, you could use the  key, either as a backspace or a **CLR**. Is there something similar for ALPHA mode?

Naturally: It works the same way: When you're building a string, by keying in (or appending) data to the ALPHA-register, the  key will indeed operate as a backspace – just as it does when you're building a new number in the X-register. And in both cases the cursor _ signals this status.

So, to edit this "string-in-progress," first backspace until you've erased the **FRED** part:        . Then add the space: **WKY2** . Finally, **A** **FRED** once again (but just press  **FRED** – you're already in ALPHA mode)... Aaaaaah – that's better!

OK, **EXIT** from ALPHA mode and press  **CATALOG**  **REAL** to confirm that you've now created five real-type variables: **YEP**, **HI**, **LORI**, **FRED**, and **EE** (remember: variables containing ALPHA characters are real-type variables, too)!

Real data are the simplest data in your HP-42S. And though ALPHA strings are more complicated because they're linked collections of ALPHA characters, they're still fairly simple. Now it's time to start looking at the other ones.

Complex Numbers

The first of these is the complex data type. Complex numbers are mixtures (linked pairs) of real and imaginary numbers. A complex number is the *sum* of a *real* number and an *imaginary* number.

An imaginary number (remember your algebra class?) is a *number whose square is negative*. You represent such a number as a *real number multiplied by $\sqrt{-1}$* . Thus,

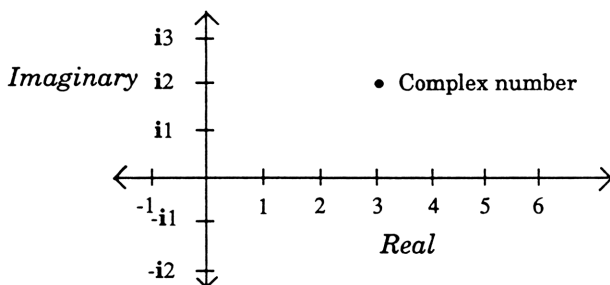
$$\sqrt{-25} = 5\sqrt{-1} = 5i \text{ (or } i5\text{)}$$

where $i = \sqrt{-1}$ (engineers often use j instead of i).

But how can you possibly add real numbers and imaginary numbers together (like adding apples and oranges)? Well, you can do it because it's not addition in the simple sense; it's "directional" or *vector addition*.

Consider this: If you're following a map and it tells you to go 3 miles east and 2 miles north, then that's the most it can do to describe your destination. *You can't "add" directions*; you can't say that your destination is "5 miles east-north" or something. If you did that, you'd have no idea if your destination were really 1 mile east and 4 miles north, or 2 miles east and 3 miles north, etc.

Well, real numbers and imaginary numbers are truly different "directions," mathematically, and so you can't mix them directly. Instead, you represent their mixture (a complex number) as coordinates (i.e. as a *vector*) on a "map" (the complex number plane); the horizontal direction is the real direction and the vertical is the imaginary:



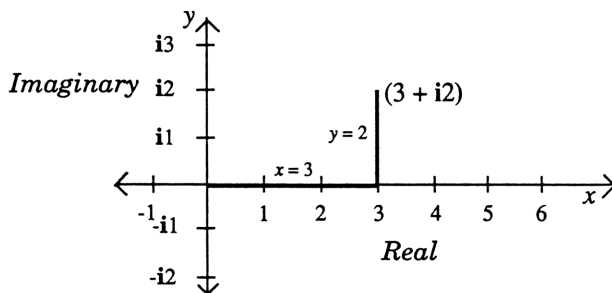
The problem is, just as there are many ways to give directions to any one destination on a map, there are many different ways to represent complex numbers in the complex plane.

For example, these formats all describe the very same complex "destination:"

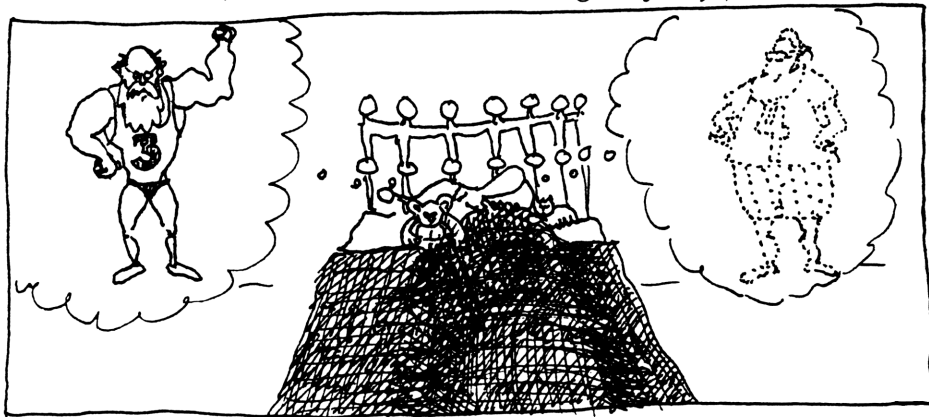
$$3 + j2 \quad 3 + 2i \quad 3.6(\cos 0.6 + i\sin 0.6) \quad 3.6 \operatorname{cis} 33.7^\circ \quad 3.6 \angle 33.7^\circ$$

So how do you know which notation to use with your HP-42S? Well, there are two common strategies or *formats* used to describe a complex number's "destination" – rectangular and polar.

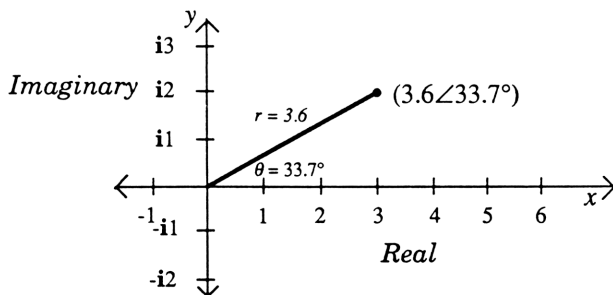
The **rectangular** format (the first two examples above) contains **i** or **j**. Its method is simply to tell you to go "so many units east" (the real direction), then "so many units north" (the imaginary, or **i**- direction):



Your HP-42S will always display the rectangular form as $3 + i2$. The 3 is the *real* (or *x*-) coefficient and the 2 is the *imaginary* (or *y*-) coefficient.



The other way to describe a destination is to give a distance and an angle "north of east" (in either degrees or radians). This is the **polar** format:



Notice that simple trigonometry will tell you what the two rectangular legs of this triangle are – simply from knowing the hypotenuse (the distance) and the angle:

$$3.6\cos(33.7^\circ) = 3 \quad \text{and} \quad 3.6\sin(33.7^\circ)\mathbf{i} = 2\mathbf{i}$$

Therefore, you can get the *rectangular* "instructions" to your destination simply by saying $3.6(\cos 33.7^\circ + \mathbf{i}\sin 33.7^\circ)$, or $3.6 \text{ cis } 33.7^\circ$ for short.

And then remember that you can specify an angle either in degrees or radians: $3.6(\cos 33.7^\circ + \mathbf{i}\sin 33.7^\circ) = 3.6(\cos 0.6 + \mathbf{i}\sin 0.6)$, right?

Finally, you could dispense with *any* reference to the rectangular form (i.e. omit the \mathbf{i} notation altogether) and just describe your destination with the bare minimum – the direct distance and angle: $3.6 \angle 33.7^\circ$

Since this latter form is so short and succinct, it is the polar form used by the HP-42S.

OK, so your HP-42S uses $3 + i2$ for a rectangular form and $3.6\angle 33.7^\circ$ (or $3.6\angle 0.6$ radians) for a polar format of complex number. But how does it decide when to use each format? ...It doesn't – you do!

Watch: Find $\sqrt{-25}$. Express this first in rectangular format, then in polar (degree) format.

Like So: Press $\boxed{2}\boxed{5}\boxed{+/-}\boxed{\sqrt{x}}$ This is the rectangular form.
Answer: $0.0000\ i5.0000$ (which means $0 + i5.0000$)





Now switch to polar form: First, press $\boxed{\text{MODES}}\boxed{\text{MODES}}$ and $\boxed{\text{DEG}}$ to switch to DEG mode (if necessary), then $\boxed{\text{POLAR}}$.
Answer: $5.0000\ \angle 90.0000$


And what does the answer look like in polar *radians* mode?
To find out, press $\boxed{\text{RND}}$. Answer: $5.0000\ \angle 1.5708$


The point is, *it's always the same number*; the $\boxed{\text{MODES}}$ menu simply offers you different *display* formats – just different descriptions of how to get to the same complex destination:




Seeing Vs. Doing: The Difference Between The

MODES and **CONVERT** Menus

Back on page 85, you learned how to change the units of angular measure with the  **MODES** menu. When you went back and forth between, say, DEG and RAD, the number didn't change, did it? In order to actually change the numerical value in the X-register, you had to use the  **→DEG** and  **→RAD** selections in the  **CONVERT** menu, right?

So what's going on here with complex numbers? In that last example, you were *not* at the  **CONVERT** menu and yet you certainly seemed to be changing numbers. When in POLAR mode, you went back and forth between DEG and RAD – and didn't that complex number change?

No – only its *appearance* changed. Your HP-42S is being entirely consistent: The only thing you change at the  **MODES** menu is the machine's *interpretation* of the values it's holding. And it will make every effort to alert you to its interpretation.

In the case of real numbers, your HP-42S was saying, for example, "The 44.4506 in the X-register is 44.4506 *degrees* right now, as far as I'm concerned." Then, when you chose  **RAD**, it said, "OK, now it's 44.4506 *radians*" (and it turned on the annunciator to alert you to this re-interpretation). *But it still had the value 44.4506 in the X-register.* Now, with complex numbers, there are simply more ways to alert you to the current interpretation: The display changes the *format* of the number itself – not just an annunciator. But the underlying value in the machine's memory is still the same. *You cannot actually alter a numerical value with the  **MODES** menu – only with the  **CONVERT** menu.*

Entering Complex Numbers

All right, now that you know how the HP-42S handles and interprets complex numbers, you'd better learn how to enter them, no? So far, you've encountered them only as *results* of your calculations. But what if you want to key them in – build them from scratch?

Do This: Make sure you're in RECTangular Mode and press **5** **ENTER** **4** **■** **COMPLEX**.

See? **■** **COMPLEX** will link together two real numbers in the X- and Y- registers, combining them into a single complex datum in the X-register: **5.0000 i4.0000**

Now This: Press **■** **COMPLEX** one more time. What happens?

Your calculator has now *unlinked* the complex number that was sitting in the X-register, putting the first (real) coefficient into the Y-register and the second (imaginary) coefficient into the X-register.

What's going on here?

It's simple: **■** **COMPLEX** forms a *complex* number whenever you have *real* numbers in the X- and Y-registers. But **■** **COMPLEX** forms *two real* numbers whenever you have a *complex* number in the X-register. OK?

And naturally, you create complex *variables* to store complex numbers:

Try This: Store $5 + i4$ as a complex variable named PLEX.

Solution: You do it the same way as with real variables: `STO ALPHA`
`NOPR P JKLM L ABCDE E WXYZ X ALPHA`.

If you press `EXIT RCL` you'll see that you now have *two* menu pages of variables that you can `RCL` (the \blacktriangledown is on), since you now have more than six such variables. And pressing `PLEX` confirms that you've done it all properly.

Of course, you can also combine already-existing values...

F'rinstance: Create a complex number from LORI and FRED, with FRED the imaginary coefficient. Store the result of this union in LORI.

Like So: `RCL LORI RCL FRED COMPLEX STO LORI`
Result: 3.7104 i34.6000

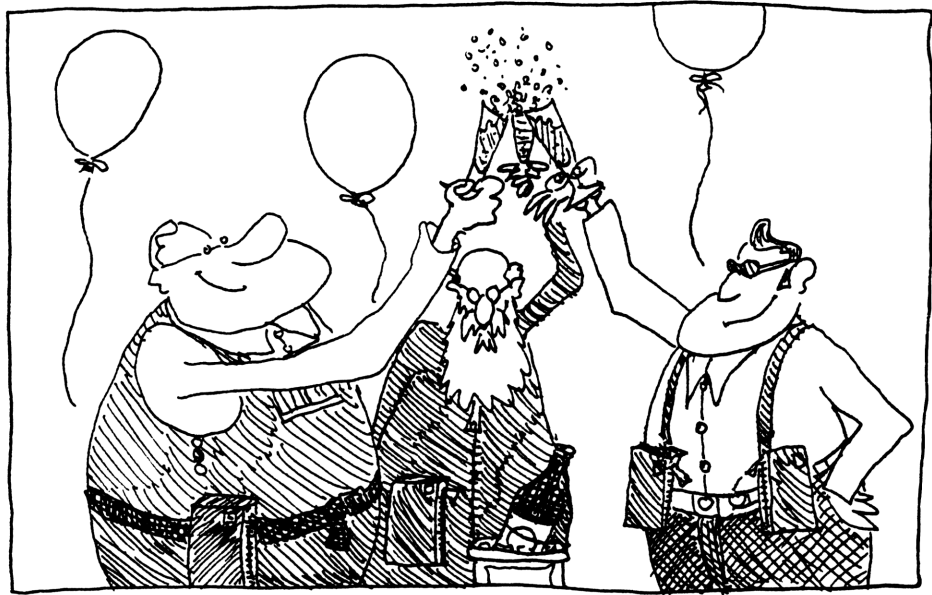
So from now on, you'll find LORI in `CATALOG CFX`, because it's now a complex variable.

So real numbers can become complex numbers (but remember from page 39 that ALPHA strings *cannot* be a part of any complex number).

Using Complex Numbers

The Bad News: It's time for a review of the basics of complex-number arithmetic.

The Good News: You get to use your calculator!



- Try These:**
1. $(6 - i2) + (7.5 + i3.66)$
 2. $(5\angle 166^\circ) - (-4\angle 39^\circ)$
 3. $(5 + i4.6) \times (2.3 - i8.1)$
 4. $(-3\angle \pi/3) \div (4\angle 4\pi/5)$

Solutions: 1. $6 \text{ ENTER } 2 \text{ +/- } \text{COMPLEX} 7 \cdot 5 \text{ ENTER } 3 \cdot 6 6$
 $\text{COMPLEX} +$ Answer: 13.5000 i1.6600

Notice that you need to use a negative number (-2) as your imaginary ("i") coefficient; your calculator always assumes a plus sign between the two parts of a complex number unless you say otherwise.

2. $\text{MODES} \text{ POLAR } 5 \text{ ENTER } 1 6 6 \text{ COMPLEX } 4 \text{ +/-}$
 $\text{ENTER } 3 9 \text{ COMPLEX}$ (notice here that your HP-42S changes the -4 $\angle 39$ (the number you input) into 4 $\angle -141$. It will do this whenever you give a negative distance in polar mode like this. It's just another way to indicate the same value; it changes neither the problem nor the answer*). Now \square .
Answer: 4.1143 $\angle 115.0633$

3. $\text{MODES} \text{ RECT } 5 \text{ ENTER } 4 \cdot 6 \text{ COMPLEX } 2 \cdot 3$
 $\text{ENTER } 8 \cdot 1 \text{ +/- } \text{COMPLEX} \times$
Answer: 48.7600 -i29.9200

4. $\text{MODES} \text{ POLAR } \text{MODES} \text{ RAD } 3 \text{ +/- } \text{ENTER } \pi \text{ ENTER}$
 $3 \div \text{COMPLEX } 4 \text{ ENTER } 4 \text{ ENTER } \pi \times 5 \div$
 $\text{COMPLEX} \div$
Answer: 0.7500 $\angle 1.6755$

*See page 92 of your Owner's Manual if you'd like more details about this conversion.

What Are Complex Numbers Good For?

OK, so you now know how to do complex arithmetic. So what? What's so outrageously useful about complex numbers?

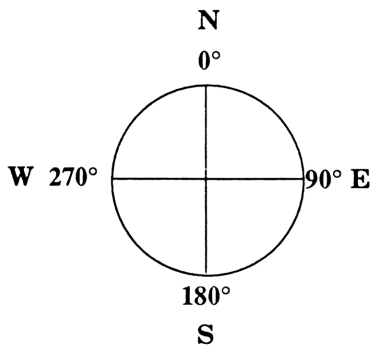
Well, for one thing, remember that a complex number is nothing more than a *two-dimensional vector* – a pairing of two different coordinates or directions (recall page 126). And those directions don't have to be limited to "real" and "imaginary."

Sure – when you want to do certain kinds of arithmetic with a complex number, the machine will naturally assume it's made up of a real and an imaginary part, mathematically speaking. But there are some vector operations – such as addition and subtraction – that are valid for any kind of two-dimensional rectangular or polar coordinates.

Thus, you could use the two parts of a *rectangular* complex number to represent, say, a position (north and east) on a map. Or you could represent a velocity (speed and direction) as a *polar* complex number.

For example, a complete description of a plane's velocity might be "450 mph on a heading of 95." This velocity might be represented as the polar vector $450\angle 95^\circ$.

On a navigator's coordinate system (shown here), a heading of 95 means the plane is flying almost (just south of) due east.



Challenge: Assume the pilot of a plane has its nose pointed on a heading of 95 at an airspeed of 450 mph. But he also has a hefty cross-wind of 56 mph blowing at a heading of 167. What is the groundspeed and true heading of the plane's flight?

Solution: The engine velocity and the wind velocity must be added together to find the total plane velocity. You can add these two vectors by treating them as complex numbers. So the problem becomes: $450\angle 95^\circ + 56\angle 167^\circ$.

To solve: **MODES** **DEF** **4** **5** **0** **ENTER** **9** **5** **COMPLEX**
5 **6** **ENTER** **1** **6** **7** **COMPLEX** **+**

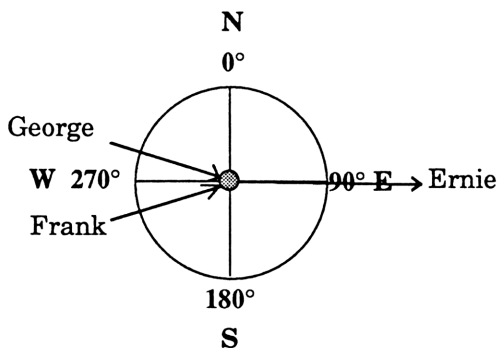
Answer: 470.3302 \angle 101.5020.

The plane is traveling about 470 mph on a heading of 101.5° – which makes sense, because the wind is pushing it south and slightly east, thus increasing its speed.

Notice that this navigator's coordinate system is different than the one you saw on page 128 for complex numbers (and on page 83 for trigonometry). But this vector math still works!

The beauty of vector math is that it doesn't matter where you call "zero," as long as you're working with a rectangular or polar coordinate system; you'll always get answers consistent with – and correct for – the system you're using!

One More: Ernie, Frank and George are moving a boulder. George shoves it with a force of 195 N at a bearing of 105. Frank, pushes 30° more northerly than George, with a force of 175 N. Ernie, on the other side, pulls on a rope with 180 N, due east. What's the net force exerted on the boulder?



Solution: Like velocity, force is a vector quantity, with both magnitude and direction. To solve this problem, you must sum the three force vectors: George's force vector is $195\angle 105^\circ$, of course, and Ernie's is $180\angle 90^\circ$. The bearing at which Frank is pushing is $(105 - 30)$, or 75, so Frank's force vector is $175\angle 75^\circ$.

Thus: DISP FIX 01 195 ENTER 105 COMPLEX
175 ENTER 75 COMPLEX+ 180 ENTER 90
COMPLEX + Answer: 537.4 \angle 90.6




So the boulder rolls away at a heading of 90.6° (almost exactly the direction Ernie is pulling – very efficient of them). And the total force acting on the boulder is 537.4 N.


Again (as these examples illustrate): Although complex numbers are themselves mathematically meaningful (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 them as *two-dimensional vectors*.

In fact, you can even use them for non-vector applications – simply as *a convenient ways to group related number pairs*. Because you can pull complex numbers apart and put them back together again, you can give your own meanings to the complex number *data type* in the HP-42S – meanings unrelated to the mathematical concept of complex numbers or directional vectors of any kind. You can use complex variables simply to store and manipulate ordered pairs of real numbers – *information objects for you to use in whatever way you see fit*.

Try This: The heights and circumferences of a grove of trees are: 18 feet and 24 inches; 25 feet and 23 inches; 28 feet and 29 inches. What are the average height and circumference of these trees?

Solution: Enter the information as three rectangular complex numbers, then add them together:

1 8 ENTER 2 4  COMPLEX 2 5 ENTER 2 3  COMPLEX
2 8 ENTER 2 9  COMPLEX ++.

Now divide by three: 3 . Answer: 23.6 i25.6.
So the averages are about 23.6 feet of height and 25.6 inches of circumference.

Matrices

Real Numbers?...("Check!")...

ALPHA Strings?...("Check!")...

Complex Numbers?...("Check!")...

Matrices?....Ah, yes....Looks like you're now ready to study this fourth and final type of data. Matrices are the most powerful and most involved of all the variables and data types – so much so, in fact, that the HP-42S workshop has an entire work area dedicated to helping you build matrices (see the Big Picture on pages 18-19 for a quick reminder)....Welcome to your first Power Tool!

A complex number was a simple linking of two real numbers, right? Well, a matrix takes this linking idea one step farther. It links together two or more numbers in a particular order, grouping them in *rows and columns*. The *dimensions* of a matrix are the numbers of rows and columns it has. Look at these examples:

2 x 3	3 x 2	1 x 5	3 x 3	3 x 1
$\begin{bmatrix} 4 & 5 & 9 \\ 2 & -2 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 2 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -3 & 9 & 0 & 7 & 2 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 9 & -3 & 5 \\ 2 & 7 & -1 \end{bmatrix}$	$\begin{bmatrix} 7 \\ -9 \\ 6 \end{bmatrix}$

The number of *rows* is always listed *first*, then the number of columns.

Hmm...If you think about it, a complex number (or any two-dimensional vector) is really just a small (**1 x 2**) matrix, right?

What Are Matrices Good For?

Matrices are incredibly useful tools for solving problems. That's why your HP-42S devotes so much power and memory to them. They allow you to solve large (or small) systems of simultaneous equations in ways much faster and easier than any other. You'll find these systems in physics, engineering, economics, business, and most other sciences and social sciences.

Matrices are essential to computer-aided design (CAD) systems, because they make it easy to "rotate" a graphic image through space.* And astronomers and engineers use them to calculate trajectories of heavenly bodies and space shuttles as they hurtle through space.

Matrices are helpful in spreadsheet analysis and many sophisticated types of "what-if" modelling in business and finance, and they're widely used by computers to provide quick, efficient sorting routines, making this information age possible.

And best of all, matrices give you the only feasible way to calculate physical phenomena in higher dimensions (part time! on weekends!). With matrices, you can describe how subatomic particles move through, oh, say, 10 or 11 dimensions – whatever you need to make your calculations come out right (without even building a clay model)!

*Yes, Virginia, there really is a matrix in your video game.

Entering Matrices

So how do you get a matrix into your calculator? You need to start learning to use that Power Tool Room now.

To begin, suppose you wanted to enter the following 2×2 matrix:

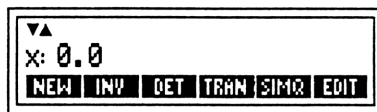
$$\begin{bmatrix} 7 & -5 \\ 3 & 4 \end{bmatrix}$$

What do you do?

Well, *the very first* thing to do is tell your machine the dimensions of the matrix you're about to enter. It needs to prepare the necessary amount of storage space *before* you dump in a bunch of data.

Press **(2)(ENTER)** to place 2.0 in both the X- and Y-registers. When you dimension a new matrix like this, your HP-42S will assume that the number of *rows* in the new matrix is entered in the *Y-register* and the number of *columns* is in the *X-register*.

Press **(MTRX)** to enter the Matrix Power Tool Room. You'll see its Main Menu:



The annunciator, $\blacktriangledown\blacktriangle$, tells you that there are more pages of menu items, – but first things first: Choose **NEW**, to declare this new matrix you're creating, and notice what happens....

See? Your calculator has combined the row and column dimensions you just entered into one single dimension statement (similar to the way **COMPLEX** combines the X- and Y-registers into one complex datum) in the X-register:



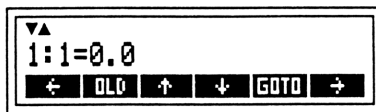
In effect, your HP-42S is now saying, "OK, I've got this **2 x 2** matrix sitting in the X-register, all ready to edit or crunch."

Great.

Only problem is, there's nothing much to edit or crunch yet. This newborn matrix is still full of zeroes and unnamed; in order to put some meaningful data into it, you need to use the Matrix Editor!



You find the Matrix Editor by choosing **EDIT**:



This menu now tells you that you've taken this current matrix (the one that was in the X-register) into the Matrix Editor, where you must go to enter or change any of its data.

While in the Matrix Editor, you'll see only one element at a time. For example, upon first entering the Editor here, you'll see the upper-leftmost element in the matrix: The notation, **1:1=0.0**, means that the matrix element at row 1, column 1 is currently the real value 0.0.

As a reminder, here's the matrix you're trying to build:

$$\begin{bmatrix} 7 & -5 \\ 3 & 4 \end{bmatrix}$$

You need to key in **7** in this window and then move the window one column to the right by pressing **→**.

For element **1:2** (that's *row 1, column 2*) enter: **5****+/−****→**. Since you've no more room to move right one column, pressing **→** moves the window to the leftmost element in the *next row* (**2:1**).

Finish keying in the matrix: **3****→****4** and then **EXIT** the Matrix Editor (going back to the Main Menu of Matrix). This essentially places this newly-edited version of the matrix back into the X-register, replacing the previous version.

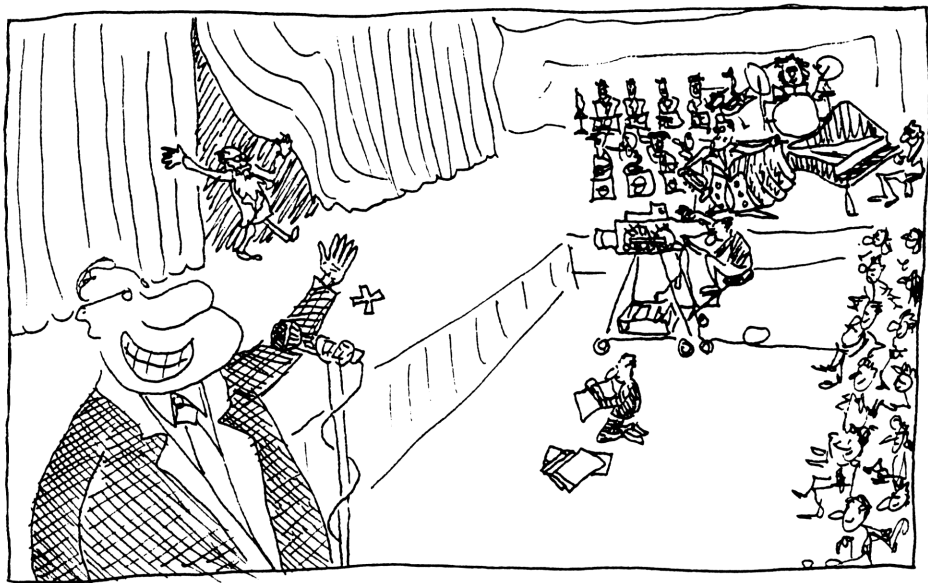
Storing Matrices

Before going any further, you should store your matrix into a named matrix variable. After all, entering the data into matrix is a tedious task, and you don't want to do it more than once. Remember that this matrix can get bumped off the top of the Stack just as easy as any other datum!

So press **STO ALPHA** **MATT** **ALPHA** (by now, you've certainly got the hang of using the ALPHA menu to spell out names and things, so you're not going to see all those hideous "typing" keystrokes anymore – just the letters themselves – you know what to do, right?).

It's just as simple as storing into any other kind of variable, right? You now have a brand-new matrix variable named MATT.

Try recalling it: First press **⬇** to clear the X-register. Then **RCL** **MATT**. There's MATT!



Editing Matrices

Suppose now that you want to edit MATT.

Try This: Replace MATT's element $1:2$ with the value 6.0 .

Solution: To make a change in an existing matrix, you have a choice:

You can **RCL** it to the X-register and then enter the Matrix Editor with the **EDIT** command, which always works on the matrix currently sitting in the X-register.

Or, you can avoid having to bring the matrix to the X-register at all: Press **▼** (to go to the next page of the Matrix Main Menu), then select **EDITN** (try this now)...

You'll see a menu of existing matrix names. Choose **MATT** and there you are, in the Matrix Editor, working on MATT.

So press the arrow keys (you choose which ones) until you come to $1:2=-5.0$. Then key in the new element, **6**, and hit **EXIT** to leave the Matrix Editor.

Done!

But that's only one way to edit a matrix – changing its values. You can also change its *shape*....

Like This: Convert MATT into this 2 x 3 matrix:

$$\begin{bmatrix} 7 & 6 & -3 \\ 3 & 4 & 8 \end{bmatrix}$$

Solution: Put the adjusted dimensions into the Y- and X-registers, **2****ENTER****3**, and select **DIM MATT** to re-dimension MATT.

Now reenter the Matrix Editor with **EDITN MATT** and find the last value in the first row **1:3: 1:3=3.0**

Hmm...if it's "new," why does it already contain the value **3.0**?

It's because when it makes an existing matrix larger (more elements), your HP-42S moves the existing elements as far "up" and "left" as it can (filling the larger matrix in "row-major order"), leaving zeroes in the extra "new" elements:

$$\begin{bmatrix} 7 & 6 \\ 4 & 3 \end{bmatrix} \text{ ----- } \rightarrow \begin{bmatrix} 7 & 6 & 3 \\ 4 & 0 & 0 \end{bmatrix}$$

Keeping this in mind, key in **3****+/-** **→** **3** **→** **4** **→** **8****EXIT** to complete the change to the matrix at the top of the page.

Complex Matrices

Back on page 40, you learned that a matrix can contain *either* all real elements or all complex elements, but not a mixture. It's time to see how complex matrices work.

Example: Create this *complex* 2×2 matrix:
$$\begin{bmatrix} 10 + i4 & i58 \\ -3 + i2 & -17 \end{bmatrix}$$

How To: First, create a *real* 2×2 matrix: **MATRIX** **2** **ENTER** **NEW**.

Next, copy it to the Y-register, **ENTER**, then combine the two real *matrices* into one complex matrix by pressing: **COMPLEX**—just like you did to create a complex number from two real numbers!

Now, in rectangular mode (**MODES** **RECT**, if needed), press **EDIT** and key in the values:

1 **0** **ENTER** **4** **COMPLEX** **→** **0** **ENTER** **5** **8** **COMPLEX**
→ **3** **+/-** **ENTER** **2** **COMPLEX** **→** **1** **7** **+/-** **EXIT**.

That's all there is to it! Now store a copy of this complex matrix for later: **STO** **ALPHA** **PIXY** **ALPHA**

Finally, just for show, *split it into two real matrices*: **COMPLEX** will now *unlink* the two parts of each element of this complex matrix. The real parts form a matrix in the Y-register; the imaginary parts in the X-register!

What You Can Do To A Matrix

Whether it's real or complex, there are a number of things that you can do to a matrix once you get it into the X-register.

The first category of things you can do is called *scalar arithmetic*. This is where you do something – the *same* something – to each element in the matrix. In the world of matrices, a simple number is called a *scalar*.

Example: Multiply the real matrix, MATT, by the real scalar 4.7. That is, multiply each element of MATT by 4.7.

Solution: Press $\boxed{\text{RCL}} \boxed{\text{MATT}} \boxed{4} \boxed{\cdot} \boxed{7} \boxed{\times}$. What do you get?

A scalar arithmetic operation always preserves the dimensions of the matrix involved – so you certainly get another **2 x 3** matrix (as your display tells you). But to see the exact elements of the solution matrix, use the Matrix Editor (via $\boxed{\text{EDIT}}$, since you have a newly created matrix in the X-register, not a named one):

$$\begin{bmatrix} 32.9 & 28.2 & -14.1 \\ 14.1 & 18.8 & 37.6 \end{bmatrix}$$

Of course, you can combine a matrix and a scalar in any of the four basic arithmetic operations.

Press $\boxed{\text{EXIT}}$ now, if you're still in the Matrix Editor.

And you can do *storage arithmetic* (recall page 118) with scalars, too.

F'rinstance: Subtract 1 from each element in MATT.

Solution: $\boxed{1}\boxed{\text{STO}}\boxed{-}\boxed{\text{MATT}}$: That's it. And if you edit MATT now, here's what you'll see:

$$\begin{bmatrix} 6 & 5 & -4 \\ 2 & 3 & 7 \end{bmatrix}$$

How about some of the other one-number functions – such as $\boxed{\sqrt{x}}$ or $\boxed{1/x}$? How do they work on a matrix?

Find Out: Take the square root of MATT: $\boxed{\text{RCL}}\boxed{\text{MATT}}\boxed{\sqrt{x}}$

Ooops: An error message? It's because you tried to use $\boxed{\sqrt{x}}$ while still at the Matrix Main Menu. $\boxed{\sqrt{x}}$ has a totally different meaning whenever a menu is displayed. Either you must $\boxed{\text{EXIT}}$ the Matrix Area altogether, then $\boxed{\text{RCL}}\boxed{\text{MATT}}\boxed{\sqrt{x}}$; or you can recall page 103 and do this: $\boxed{\text{TOP.FCN}}\boxed{\text{SORT}}\dots$

...Now what? The message, "Invalid Data" shows because you're trying to put a complex number into a real matrix. MATT has one negative element ($1:3=-4.0$) and $\sqrt{-4}$ is a complex number – which you can't put into a real matrix.

Special Matrix Functions

In addition to scalar arithmetic, there are other one-matrix functions that are unique to matrices. For example, you have some functions that reshape or re-order a matrix....

Try One: Make sure you're looking at the Matrix Main Menu, then **[RCL]** MATT and choose **TRAN**. What happens?

Result: MATT is transposed – the rows become columns and the columns become rows – and its dimensions change from **2 x 3** to **3 x 2**:

$$\begin{bmatrix} 6 & 2 \\ 5 & 3 \\ -4 & 7 \end{bmatrix}$$

You also have these two functions that let you expand or shrink a matrix – by a whole row at a time:

- ▼ **INER** Adds or "inserts" a whole new *row* of elements *before* (above) the row you're currently viewing. The new row will be filled with zeroes until you change them.
- ▼ **DELR** Removes or "deletes" the whole row of elements that you are currently looking at (unless it's the only row left).

Of course, you may not yet see when and why you'd want to use such reshaping functions, but it's good to know that they're there (you'll see some examples of their uses in upcoming problems).

So, what other one-matrix "stuff" can you do? Well, with **TRAN**, **INSE**, and **DELR**, you can re-shape a matrix in such a way that it doesn't matter what shape (i.e. what dimensions) the matrix has to begin with. But there are two very useful and important matrix functions that work only on a *square* matrix: **INV** and **DET**.

The first of these, **INV**, is the *inverse* or reciprocal function. Recall from your school days that the multiplicative inverse of a number is what you must multiply it by to get 1. So, for example, $1/3$ is the multiplicative inverse of 3, because $3 \times 1/3 = 1$.

But what's the inverse of a matrix? There's nothing you can multiply by a matrix to give you 1. Besides, you don't *want* a 1, anyway; you want the *matrix equivalent* of 1 – the *matrix identity*, which, when multiplied by any matrix, gives you the matrix back again (just as multiplying any number by 1 gives you the number back again).

Identity matrices are all square matrices with 1's on their left-to-right diagonal elements and 0's everywhere else, as in these examples:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The multiplicative inverse of a *matrix*, then, is another matrix, which, when multiplied by the original, gives you an *identity* matrix in the pattern of those above. *And it turns out that the only matrices that have these inverses are square matrices.*

Your calculator makes it easy to find the inverse matrix of a square matrix – if that matrix is sitting in the X-register. So...

Try One: Make MATT into a square matrix by deleting its last column. Then find the inverse matrix of MATT.

Solution: To delete a *column* is a little bit tricky. But deleting a row is easy. In this case, you want to eliminate elements 1:3 and 2:3 (see page 150 for the current picture of MATT).

So, first TRANSpose the matrix, so that the unwanted elements become 3:1 and 3:2 – sharing the same row:

RCL **MATT** **TRAN**.

Now you can delete that entire third row by entering the Matrix Editor (**EDIT**), moving to that row with the arrow keys, and pressing **▼** **DEL** **EXIT**.

Now TRANSpose back again to get your new, pared-down MATT. Store this new matrix as MAT (**STO** **ALPHA** **MAT** **ALPHA**), then go ahead and use **INV** to find the MAT's inverse matrix. Pressing **DISP** **FIX** **0** **3** and then **EDIT** shows you the inverse matrix:

$$\begin{bmatrix} 0.375 & -0.625 \\ -0.250 & 0.750 \end{bmatrix}$$

Now **EXIT** the Matrix Editor and store this inverse matrix as TAM.

The other important one-square-matrix function is the *determinant*. The determinant of a square matrix is a real number – calculated from the matrix elements – that tells you something about the matrix.

Example: Find the determinants of TAM and MAT.

Solution: For TAM, you need only press **DET**, since TAM is still sitting in the X-register from the previous problem....

Answer: 0.125

To find the determinant of MAT: **RCL** **MAT** **DET**:

Answer: 8.000

TAM and MAT are multiplicative inverses of each other, right? What about their determinants (now sitting in the Y- and X- registers, respectively)? Press **↵** to see....

Unity, Identity, Oneness, Reciprocity! O Wow!



Now that you have an idea about what you can do with only one matrix, what types of manipulations can you do with *two* matrices?

What You Can Do With Two Matrices

Of course, what you normally want to do with two matrices is arithmetic – combine them – right?

OK, but before you get started with that, consider this: When you do operations involving two matrices, it's important to pay attention to their *dimensions*. There are strict rules about how matrices can combine with another. And when you violate these rules on your HP-42S, it'll tell you: **Dimension Error**. So remember these rules:

Rule 1: When *adding* or *subtracting* matrices, you must always use two matrices with *exactly the same dimensions*.

You simply cannot add a **2 x 3** matrix to a **3 x 2** matrix, because the definition of matrix addition (or subtraction) is simply to add (or subtract) corresponding elements – so your result would necessarily need to be a matrix with those same dimensions.

Rule 2: When *multiplying* two matrices together, the *order* that you multiply is important! *And the number of columns in the first matrix must match the number of rows in the second matrix.*

This is because the definition of multiplication is row \times column: You multiply the elements in the first *row* of the first matrix by their corresponding elements in the first *column* in the second matrix – then sum those products to get the first *element* of the resulting matrix:

$$\begin{bmatrix} 3 & 2 \\ 4 & 5 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 4 & -3 & -2 & 0 \\ 1 & -2 & 0 & 3 \end{bmatrix} =$$

$$\begin{bmatrix} 3(4) + 2(1) & 3(-3) + 2(-2) & 3(-2) + 2(0) & 3(0) + 2(3) \\ 4(4) + 5(1) & 4(-3) + 5(-2) & 4(-2) + 5(0) & 4(0) + 5(3) \\ -1(4) + 1(1) & -1(-3) + 1(-2) & -1(-2) + 1(0) & -1(0) + 1(3) \end{bmatrix} = \begin{bmatrix} 14 & -13 & -6 & 6 \\ 21 & -22 & -8 & 15 \\ -3 & 1 & 2 & 3 \end{bmatrix}$$

Thus you can multiply a 3×2 by a 2×4 , but you cannot multiply a 2×4 by a 3×2 . And a proper solution matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix. So in this example, you get a 3×4 solution matrix.

Rule 3: When *dividing* matrices, the dividend (the first) matrix can have any dimension, but *the divisor (the second) matrix must always be a square matrix with the same number of rows as the dividend*. Thus, you can divide a 2×3 only by a 2×2 , and a 3×2 only by a 3×3 .

Why? Because dividing is really the same process as *multiplying by the inverse*. And as you just learned on page 152, only square matrices have inverses (and these inverses have the same square dimensions) – so only square matrices can be used as divisors!

Examples: You have six matrices with the following dimensions:

$$\begin{array}{lll} \mathbf{A: 3 \times 4} & \mathbf{B: 2 \times 3} & \mathbf{C: 3 \times 3} \\ \mathbf{D: 4 \times 2} & \mathbf{E: 3 \times 2} & \mathbf{F: 2 \times 2} \end{array}$$

Decide whether the following operations are possible. If so, give the dimensions of the solution matrix. (For these operations on the HP-42S, the first matrix would be in the Y-register and the second one in X.)

$$\begin{array}{llll} \mathbf{1. B + E} & \mathbf{2. D \div F} & \mathbf{3. A \div C} & \mathbf{4. CE} \\ \mathbf{5. AC} & \mathbf{6. CA} & \mathbf{7. B \div F} & \mathbf{8. BF} \end{array}$$

Solutions: 1. No. See Rule 1.

2. No. See Rule 2.

3. Yes. The solution is a 3×4 matrix (see Rule 3).

4. Yes. The solution is a 3×2 matrix (see Rule 2).

5. No. See Rule 3.

6. Yes. The solution is a 3×4 matrix (see Rule 2).

7. Yes. The solution is a 2×3 matrix (see Rule 3).

8. No. See Rule 2.

And there are two other types of common matrix multiplication, also:

The first is called the *dot product* of two matrices: $\mathbf{A} \cdot \mathbf{B}$. To do this, you multiply the *corresponding* elements of two matrices together and then sum up these products (so the result is a *number*, not a matrix). Notice that the formula for a dot product means that you must have two matrices with the *exact same dimensions*.

Try One: Find $\text{MAT} \cdot \text{TAM}$

Solution: $\boxed{\text{RCL}} \boxed{\text{MAT}} \boxed{\text{RCL}} \boxed{\text{TAM}} \boxed{\nabla} \boxed{\text{DOT}}$ Answer: 0.875

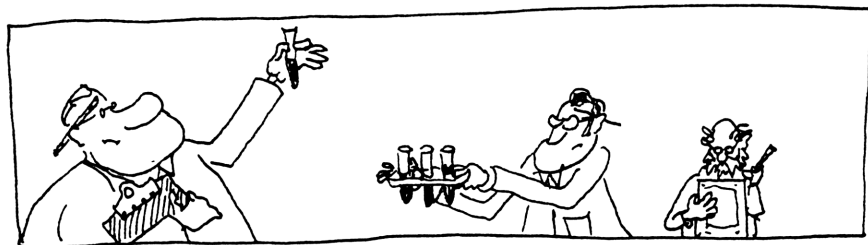
The other type of matrix multiplication is for two- or three-column vectors, which (as you'll recall from page 140), are matrices with only one row. With two of these (same dimensions only), you can compute their *cross product*, symbolized by \times . The solution will be a new vector.

Example: For these vectors: $\mathbf{A}: [2 \ 5 \ -3]$ and $\mathbf{B}: [7 \ -4]$
find $\mathbf{A} \times \mathbf{B}$.

Solution: $\boxed{1} \boxed{\text{ENTER}} \boxed{3} \boxed{\blacktriangle} \boxed{\text{NEW}} \boxed{\text{EDIT}} \boxed{2} \boxed{\rightarrow} \boxed{5} \boxed{\rightarrow} \boxed{3} \boxed{+/-} \boxed{\text{EXIT}}$
 $\boxed{\text{STO}} \boxed{\text{ALPHA}} \boxed{\text{HRCOE}} \boxed{\text{H}} \boxed{\text{ALPHA}}$
 $\boxed{1} \boxed{\text{ENTER}} \boxed{2} \boxed{\text{NEW}} \boxed{\text{EDIT}} \boxed{7} \boxed{\rightarrow} \boxed{4} \boxed{+/-} \boxed{\text{EXIT}} \boxed{\text{RCL}} \boxed{\text{H}} \boxed{\text{X}\nabla\text{Y}} \boxed{\nabla}$
 $\boxed{\text{CROSS}}$

Find the answer via $\boxed{\blacktriangle} \boxed{\text{EDIT}}$: $[-12 \ -21 \ -43]$
Then $\boxed{\text{EXIT}}$ to the Stack.

Notes (Yours)



That's about it for this chapter on data and variables. They are indeed the raw materials used by your calculator to shape solutions for you. The next step is to learn how to fashion your own tools from the workshop's built-in tools.

But first, of course, you need to test yourself – to be sure that you don't ever become exasperated, perspired, and otherwise unduly vexed with data and variables....

Variable Vexation

1. Store 4.56 in data register R_{18} . Then store 9.23 in the T-register. Finally calculate $4.56 \times 9.23^{4.12}$ using the $\boxed{\text{RCL}}\boxed{\times}$ function.
2. What's the difference between $A\boxed{\text{RCL}}\boxed{0}\boxed{4}$ and $\boxed{\text{RCL}}\boxed{0}\boxed{4}$?
3. Pressing $\blacksquare\boxed{\text{COMPLEX}}$ causes different things to happen depending upon what the X- and Y-registers contain. What happens in each of the following circumstances when you press $\blacksquare\boxed{\text{COMPLEX}}$?
 - a. Complex numbers in both X and Y.
 - b. Real numbers in both X and Y.
 - c. A real matrix in Y; a complex number in X.
 - d. A real number in Y; a real matrix in X.
 - e. A 2×3 complex matrix in Y; a 2×3 real matrix in X.
 - f. 3×3 real matrices in both X and Y.
 - g. A real 2×2 matrix in Y; a real 2×3 matrix in X.

4. Sometimes you can take the square root of a matrix and sometimes you can't. Why?
5. Create (and name) the 1×3 matrix, $\mathbf{A}: [3 \ 6 \ -4]$. Then redimension it to a 2×2 matrix:

$$\begin{bmatrix} 3 & 6 \\ 2 & -4 \end{bmatrix}$$

Now find the inverse of this, then the determinant of the inverse.

6. Create and name two vectors, $\mathbf{B}: [2 \ 5]$ and $\mathbf{C}: [-2 \ 9]$. Then show that $\mathbf{B} \times \mathbf{C} \neq \mathbf{C} \times \mathbf{B}$ (X means the *cross* product).
7. During this chapter you created and stored the matrices, PIXY, MAT, and TAM. Calculate the determinant of the solution matrix to the following problem:

$$e^{[\text{PIXY} (1/2(\text{MAT}) + \text{TAM})]}$$

8. If you want to change between Rectangular and Polar formats, what's the difference between using the **MODES** Menu and the **CONVERT** Menu items (**→REC** and **→POL**)? When should you use the **MODES** Menu? When should you use the **CONVERT** Menu?



Victory Versus Variable Vexation!

1. First $\boxed{4} \cdot \boxed{5} \boxed{6} \boxed{\text{STO}} \boxed{1} \boxed{8}$. Then $\boxed{9} \cdot \boxed{2} \boxed{3} \boxed{\text{STO}} \cdot \boxed{\text{ST}} \boxed{\text{T}}$.
Finally, $\boxed{4} \cdot \boxed{1} \boxed{2} \boxed{\text{Y}^\times} \boxed{\text{RCL}} \boxed{\times} \boxed{1} \boxed{8}$. Answer: 43,211.153

2. If you press $\boxed{\text{RCL}}$ while in ALPHA mode, you'll be doing ARCL (hence the notation $\text{A}\boxed{\text{RCL}}$). If you're not in ALPHA mode, $\boxed{\text{RCL}}$ does a normal RCL.

$\text{A}\boxed{\text{RCL}} \boxed{0} \boxed{4}$ makes an ALPHA-character copy of whatever is currently stored in data register 04 (R_{04}) and *appends* those characters to whatever is already stored in the ALPHA-register. If R_{04} currently contains a real number, the numerals (characters) representing that number will be appended to the ALPHA-register in the same format as the display is currently using to format real numbers.

$\boxed{\text{RCL}} \boxed{0} \boxed{4}$, on the other hand, makes an *actual* copy of the contents of R_{04} (a number for a number or a string for a string) and stores this copy in the X-register. Whether this new addition to the X-register bumps the Stack up in a Stack Lift or merely overwrites the contents of the X-register will depend on whether the previous operation left Stack Lift *enabled* or *disabled*.

3. a. Because the number in X is complex, pressing  **COMPLEX** will split this complex number into two real-number parts. The first part (formerly the real part of the complex number) will be stored in Y, the second (formerly the imaginary part of the complex number) in X. The other complex number (that had been in Y) is bumped up into the Z-register.
- b. This time, pressing  **COMPLEX** causes a new complex number to be formed, the real part formed from the real value in Y, the imaginary part formed from the real value in X.
- c. This is the same situation as the first case. Because X contains a complex number, your calculator doesn't care what's in Y. It simply splits apart the complex number in X, sending the two pieces into Y and X.
- d. You'll get the error message, **Invalid Type**. Since you have a *real* matrix in X, your calculator is expecting to find a real matrix in Y, as well. Nothing else will do!
- e. Again, just as in the previous case, because X contains a real matrix, so must Y.
- f. You'll make a **3 x 3** complex matrix.
- g. You'll get the error message, **Dimension Error**, because although you have two real matrices, you cannot combine them to form one complex matrix unless they have exactly the same dimensions.

4. You can always take the square root of a *complex* matrix without worrying about whether any of your elements are negative. But when working with a *real* matrix, you can take the square root only when *all* of the elements are *positive* in sign. Otherwise, you'll be trying to put complex elements into a real matrix.

5. Key in and name A: $\boxed{1} \boxed{\text{ENTER}} \boxed{3} \boxed{\text{MATRIX}} \boxed{\text{NEW}} \boxed{\text{EDIT}} \boxed{3} \boxed{\rightarrow} \boxed{6}$
 $\boxed{\rightarrow} \boxed{4} \boxed{+/-} \boxed{\text{EXIT}} \boxed{\text{STO}} \boxed{\text{A}}.$

Redimension: $\boxed{2} \boxed{\text{ENTER}} \boxed{\nabla} \boxed{\text{DIM}} \boxed{\text{A}} \boxed{\text{EDITN}} \boxed{\text{A}} \boxed{\rightarrow} \boxed{\rightarrow} \boxed{2}$
 $\boxed{\rightarrow} \boxed{4} \boxed{+/-}$

Invert this: $\boxed{\text{EXIT}} \boxed{\text{RCL}} \boxed{\text{A}} \boxed{\Delta} \boxed{\text{INV}} \boxed{(\text{EDIT and } \rightarrow \text{'s to view})}$:

$$\begin{bmatrix} 0.167 & 0.250 \\ 0.083 & -0.125 \end{bmatrix}$$

Find the determinant of this: $\boxed{\text{EXIT}} \boxed{\text{DET}}$ Answer: -0.042

6. First create and name the vectors (each is a 1×2 matrix):

$\boxed{1} \boxed{\text{ENTER}} \boxed{2} \boxed{\text{NEW}} \boxed{\text{EDIT}} \boxed{2} \boxed{\rightarrow} \boxed{5} \boxed{\text{EXIT}} \boxed{\text{STO}} \boxed{\text{ALPHA}} \boxed{\text{MBCOE}} \boxed{\text{B}} \boxed{\text{ALPHA}}$
 $\boxed{1} \boxed{\text{ENTER}} \boxed{2} \boxed{\text{NEW}} \boxed{\text{EDIT}} \boxed{2} \boxed{+/-} \boxed{\rightarrow} \boxed{9} \boxed{\text{EXIT}} \boxed{\text{STO}} \boxed{\text{ALPHA}} \boxed{\text{MBCOE}} \boxed{\text{C}} \boxed{\text{ALPHA}}.$

Now find B X C: $\boxed{\text{RCL}} \boxed{\text{B}} \boxed{\text{RCL}} \boxed{\text{C}} \boxed{\nabla} \boxed{\text{CROSS}}$

Answer (use $\boxed{\text{EDIT}}$ to view): $\begin{bmatrix} 0.000 & 0.000 & 28.000 \end{bmatrix}$

C X B: $\boxed{\text{EXIT}} \boxed{\text{RCL}} \boxed{\text{C}} \boxed{\text{RCL}} \boxed{\text{B}} \boxed{\nabla} \boxed{\text{CROSS}}$

Answer: $\begin{bmatrix} 0.000 & 0.000 & -28.000 \end{bmatrix}$

The answers are opposite in sign!

7. Go to the Matrix Main Menu. Then **RCL** **PIVY** **RCL** **MAT** **2** **÷** **RCL** **TAM** **÷** **X** does everything except for the exponentiation (e^x). To get that, you'll need to press **TOPFCN** **LN** (recall page 103).

Finally, press **DET** Answer: 12.721 -i18.818

8. Complex values in your HP-42S are always kept in rectangular form, regardless of what you see in your display. The **MODES** Menu changes only the way those values are *displayed*. RECT-angular mode will display them as is. POLAR mode will display those values in the corresponding \angle format.

The **CONVERT** Menu, on the other hand, performs a *calculation* (actual math in the Stack) on the (always-rectangular) values it finds in X and Y. **→REC**, for example, assumes that the numbers it finds represent the results of some POLar calculation and that now you want to *convert* these to RECTangular *values*. Press **RCL** **▼** **PLEN**. x: 5.000 i4.000 Now press **CONVERT** **→REC** to *convert* to rectangular...Why the change? Aren't you *already* in rectangular format? ...Well, the *display* is, but when you press **→REC**, your HP-42S reinterprets the *values* in the X-register to be in polar format – no matter whether they actually are or not! The 5.000 and 4.000 are then taken to mean 5.000 \angle 4.000 – regardless whether they appear as 5.000 i4.000 to you – and an actual calculation is done, *putting new values in the Stack*.

So, since the **MODES** Menu changes only the display's representation of the Stack values, always use it rather than the **CONVERT** Menu *unless* you really do want to change the values in the Stack.



BUILDING YOUR OWN TOOLS:
Programming

Programs: A Different Kind Of Tool

Up to this point, you've worked with various hand tools and data types – the raw materials that you manipulate with your tools.

What about programs? Are they a special type of tool or a type of data, or what?

At one level, programs do look a little bit like data. After all, they're stored in memory and are assigned names. But that's about as far as the resemblance goes. Programs are really tools: they *do* things to data you've entered – just like the built-in tools you've already seen.

You can think of programs as little robots that you build to take care of the more repetitive tasks in your workshop. So this chapter is dedicated to the building of little HP-42S "robots" – tools that know how to follow your instructions.



A Little Memory Reminder

Before you get started on building programs, you'd better review how programs affect the amount of memory available in your machine.

As you recall (page 43), you can review the contents of your various storage areas by pressing **■ CATALOG** and selecting one of the menu choices to see the names of the objects stored in each area. So do it:

■ CATALOG REAL shows ALPHA variables YEP and HI and real variables FRED and EE (provided, of course, that you've been following this book in sequence and haven't done a bunch of freelancing in the meantime!).

Then **EXIT CPH** shows that PLEX and LORI are your current complex variables. And **EXIT MMT** shows C, B, A, TAM, MAT, PIXY, MATT, and REGS as your current matrix variables (remember that REGS is always there – it's built-in).

So how much memory do you have left? Press **EXIT** and choose **MEM** from the main **■ CATALOG** Menu and hold down the menu key to view:

Available Memory:
6444 Bytes

You have 6444 bytes of memory with which to create your programs (of course, if you've stored additional variables, you'll have less than 6444.

But how much memory is a byte? To get some feel for this, clear a few variables to see how many bytes you "free up."

Do This: Clear matrix variables A, B and C, and then check to see how many Bytes this clears.

Solution: [EXIT] to the Stack. Then press [CLEAR] [CLEAR] (remember that pressing a Menu heading twice is useful if you need to use a Menu several times in a row) and [CLV] [A] [CLV] [B] [CLV] [C] [EXIT] to clear the matrix variables. Now check available memory with [CATALOG] [MEM]:
Answer: 6570 Bytes

Those three 2 x 2 matrix variables had been using a total of 126 Bytes – an average of 42 Bytes each.

Another: Now clear the complex variable PLEX, and find out how much memory you save.

Solution: [EXIT] [CLEAR] [CLV] [PLEX] [CATALOG] [MEM]:
Answer: 6597 Bytes

This complex variable – including its name– appears to be worth 27 Bytes of memory.




One More: Clear the real variable, FRED, to find out how big it was.

Solution: [EXIT] [CLEAR] [CLV] [▼] [FRED] [CATALOG] [MEM]:
Answer: 6616 Bytes

This real variable uses 19 Bytes, including its name.

PRGM And Program Mode

All right, those are variables. Now how fast do programs "eat up" memory? To find out, you'll need to begin working in Program Mode.

The first thing to notice about the  **PRGM** key (sitting there on the bottom row of your calculator) is that it is your access to a "mode," – not a menu like most of the other "shifted" functions on your keyboard. And  **PRGM** activates such an important mode that it gets its own key instead of being relegated to the  **MODES** Menu.

How does it work?

When you're in "normal" mode (often called Run Mode) and you use a tool or store a variable, you're actually doing something with your data at that moment – directly "crunching" as you press the keys. Data is moved around in the stack, values are calculated or stored, etc.

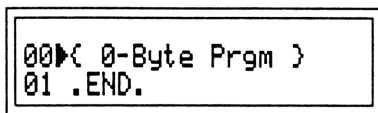
But in Program Mode, most of your actions are treated as a "set of recorded instructions" – instructions for doing something in the *future*. And when you've written these instructions and given them a name, you've built a *program*.

Of course, when you're ready to run your program, you must then get out of Program Mode (back into Run Mode – yep, that's why it's named that).

To *record* instructions, you go to Program Mode; to *execute* those recorded instructions, you go to Run Mode.

The Program Display Revisited

Back on page 24, you briefly glanced at a Program Display – just to see what it looked like. Now it's time to study it for real....Press **EXIT** until you see the Stack Display, then press **PRGM**. You should see this:



```
00▶( 0-Byte Prgm )
01 .END.
```

You're looking at two program lines stored in your calculator (and these are the *only* two program lines in storage right now).

Notice these important features about this display:

- The first two digits of each line are *line numbers* – 00 and 01 here.
- The first line of every program is numbered 00, and it tells you about the size of the program. Here, because there isn't any program yet, it says just that: you have a 0-Byte Program.
- There's a small arrow (▶) following the line number. This is the *program pointer*, which is what the machine uses to "keep its place" in program memory. Since you may occasionally be unsure of the position of the program pointer, this little arrow in the Program Display is your reminder.
- Line 01 contains a special instruction – the Permanent .END. This "END-with-the-periods" says that you're looking at the very end of storage space allotted to programs. This is where you'll start when you begin to key in and build a new program.

OK – now how fast do the bytes add up when you record some program lines? Try some more memory experiments....

Try This: Find out how many bytes a digit takes up when you key it in as an instruction in program memory.

Solution: Press any number key, say, [5].

Result: Your program – which was previously 0 Bytes – is now 2 Bytes

But Now: Add another digit, say, [8]:

Result: Now the total for two digits is 3 Bytes.

Hmm... does the 8 really take up less space than the 5 or is something else going on here?

Answer: It's the ol' "Hidden Byte" trick.

Actually, each digit only uses one Byte of memory. The third byte never shows up as an instruction because its information pertains to the instruction as a whole: This program line is an instruction that tells the machine to "key into itself" (into the X-register) the number 58 (an exact recording of what would have happened immediately if you had been in Run Mode, right)?

The "No-Frills, Straight-Arrow" Program

So, how do you write a program? Well, basically, it boils down to imagining the keystrokes you'd have to perform manually to solve a certain kind of problem.

Imagine, for example, that you've just finished a physics experiment in which you've compiled a list of fifty temperatures as resulting data. But your lab was ill-equipped – you had only Fahrenheit thermometers – so now you need to convert these temperatures to more acceptable units – degrees Celsius (°C).

The formula to do this conversion is: $^{\circ}\text{C} = \frac{(^{\circ}\text{F} - 32) \times 5}{9}$

Try This: Find the keystrokes you need to convert 97°F to °C.

Solution: Press `EXIT`, then `97ENTER32-5X9÷`
Answer: 36.111

And Another: Now do the keystrokes for the conversion of 85°F.

Solution: `85ENTER32-5X9÷` Answer: 29.444

Well, running through these keystrokes fifty times (once for every temperature you took in your experiment) is going to be no fun at all. You'll be repeating the same eight final keystrokes (`ENTER32-5X9÷`) over and over – once for each temperature datum. A big, big drag.

So what do you do about it?...

You write a program that can solve this equation:

$$\frac{(INPUT - 32) \times 5}{9} = OUTPUT$$

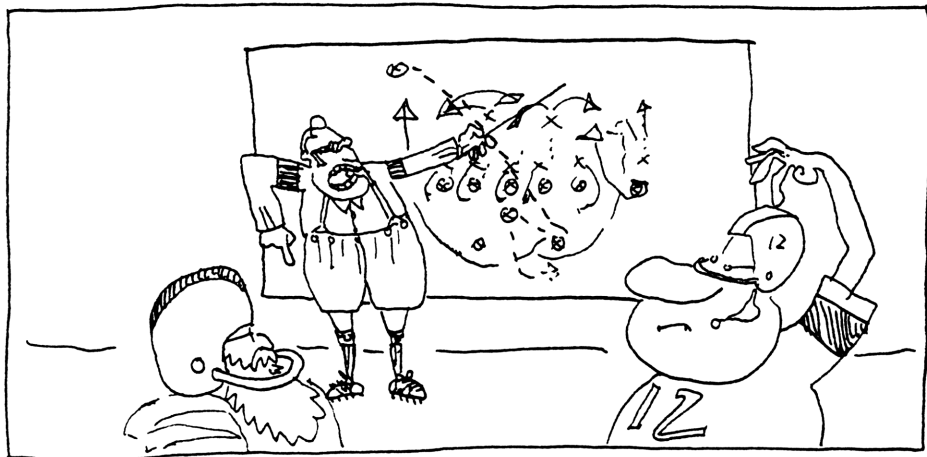
Good news...

You have already written that program!

Did you know you were already programming (at least, on paper)? The keystroke sequence you developed as a solution to this equation is indeed a program.

Right now, the program is not stored in your calculator, but in your mind (and on the previous page). When you worked through the sequence of keystrokes to convert a Fahrenheit temperature to Celsius, you actually visualized ("called up") a logical sequence of instructions in your mind and (possibly with the help of your fingers) worked through these, step-by-step, starting with **ENTER** and ending with **+**.



All you need to do now is to turn this set of mental instructions into ones that your HP-42S can understand.




How To Start A Program




At the beginning of this chapter, you observed that programs and variables are somewhat alike because they both have names. That's true, but the name of a program is more often called a *label*.

The very first thing any program needs is a label.

To label a program you use a function in the  Menu: 


Try This: Begin building a program to convert °F to °C, by labeling it "FIRST".


Solution: Make sure you're in Program Mode. Then use the  key to clear that experimental line 01 58_, so that you see the Permanent .END. on line 01.





Now press   and key in FIRST using the alphabetic keys (and pressing  as usual, when you're finished). You should now see:









```
00 { 9-Byte Prgm }
01 LBL "FIRST"
```

What happened to the .END. in this example? Press  and find out.


Aha! It's now down on line 02. You have *inserted* the label between the 00 line and the .END. Press  to go back to line 01.

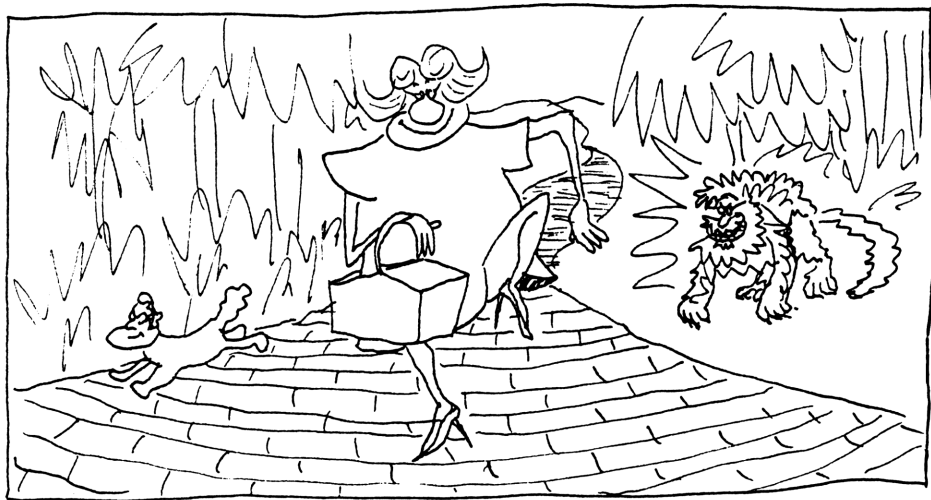
Notice that the  and  keys have a special function when you're in Program Mode. They move the program pointer backwards and forwards through the program lines *without changing anything written on the lines*. The  key does a BackSTep (BST for short) – moving the pointer *backward*, to the *previous* line. The  key – does a Single STep (or SST) – moving the pointer *forward*, to the *next* line.

But, whenever you use a Menu, the  and  keys will revert to their traditional function of cycling you through the menu pages – even in Program Mode. Therefore, to BST and SST when a menu is showing in the display, you'll need to use the   and  keys....




Try This: Choose the  menu and then step backward to line 00 of your program. Now step forward to line 02.








Solution:    

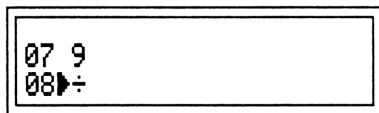
Get the idea? Press .



Challenge: Now finish keying in this program, "FIRST" that converts a Fahrenheit temperature to Celsius.

Solution: Use the  and  keys until the program pointer () is on line 01. Now press the keystroke sequence you developed earlier (on page 173):

       . You should now see this:



Notice how each line you key in will be *inserted following* the line you're viewing. And the pointer then moves to the new line.







































OK – that's it, then. You've got this program all ready to go...right?


Mmmm...not quite....





How To END A Program

How do you tell the calculator that you're finished entering a program? You tell it to END.

END is an instruction like any other. It tells your machine that this is the last instruction in the program. There are **three** ways to put this instruction at the end of your program:

- You can select END from the  CATALOG  Menu:  CATALOG 
                        and then .
- Or, you can directly command your calculator to eXEQute an END, by pressing         .

The  key will become one of your favorite programming keys as you get to know it better. It allows you "spell out" any function from any menu – as an alternative to pressing a key or choosing a menu item.*

- Or, you could press   . This means "Go TO the END with the two 's – the Permanent .END. – and prepare for a new program." In doing this, the calculator *automatically* places an END instruction as the last line of your program (if there's not one there already).

* Of course, this means you need to know how to spell the function correctly. You can find the correct spellings of all the functions in your Owner's Manual's Operation Index, starting on page 310.

So choose one of those three methods and do it now.... When you finish, notice that you're still in Program Mode (END does not change that), but you're now positioned to line 00 of a brand-new program.

Try This: Confirm that you've properly entered your END by looking at the program FIRST again.

Solution: [EXIT] from Program Mode, then press [GTO] [FIRST] [PRGM], which should bring you to the Program Display with your program pointer on: 01 LBL "FIRST". Now, using the [▲] key, BST around to line 09, where you *should* see an END.... ...sure enough!

Question: What's the difference between END and .END.?

Answer: When you see .END. in your display, you know the program pointer is positioned to the very end of program memory (and the last program in memory – the one just "above" this .END. – is always the one you keyed in most recently). There is only one .END..

By contrast, there can be many END's in program memory. Each END acts as a partition, separating one program from the next program. If the program pointer is positioned within one program and you want to move it to another, you need to use [GTO] to "call up" a label in the other program. *Using BST and SST will never move you from one program to another.*

Using Your First Program

OK, here's the moment you've been waiting for....

Coup d'Grace: Using the FIRST program, convert the following Fahrenheit temperatures to Celsius (and change your display setting to FIX 01, to match the precision of these data): 454, 127, 98.6, 212, 72, and -40.

Denouement: [EXIT] to Run Mode, if you haven't already. Then [DISP] [FIX] [0] [1] sets the display. And:



[4] [5] [4] [XEQ] [FIRST] (Answer: 234.4) [1] [2] [7] [R/S]
[52.8] [9] [8] [.] [6] [R/S] (37.0) [2] [1] [2] [R/S] (100.0)
[7] [2] [R/S] (22.2) [4] [0] [+/-] [R/S] (-40.0)


Notice that when you pressed [XEQ], the label "FIRST" appeared as a selection on an XEQ Menu. Your machine knows that when you want to execute (run) something, you're going to be referring to a label.






But why did you use the [XEQ] key for only the first datum, but then just the [R/S] (Run) key after that? Because *after* the first time through your program, you knew that your program pointer is positioned within that program. So when the machine stopped after encountering the END instruction, a mere [R/S] was all that was necessary to get it to cycle around and start at the top again. But on the *first* time through, you didn't know where the pointer is; it could have been in another program entirely – so you used [XEQ] to jump to the correct label.

Editing A Program

Suppose now that you want to include the FIX 01 procedure in the program FIRST – so that the program would automatically change the display precision. How could you go back and add in this feature?

Like This: Re-enter  Mode by pressing . This will always take you to the the program line where the program pointer is currently positioned. Right now, it's at 001(20-Byte Prgm).

This makes sense: A program's instructions "wrap around in a circle," so line 00 is the "next line" after line 09 – the END instruction that halted the program. Again, this is why  works for repeated execution.

Now you're going to *insert* a (recorded) FIX 01 instruction on the program line *immediately following* line 01. To do this, you press  to get to line 01 and then simply use the same keystrokes as if you were doing it immediately from the Stack Display:    .

Notice that when you're inserting new program lines, each new line will be inserted immediately following the current (i.e. the "pointer") line.

Get the idea? Editing is really no big deal. Try....

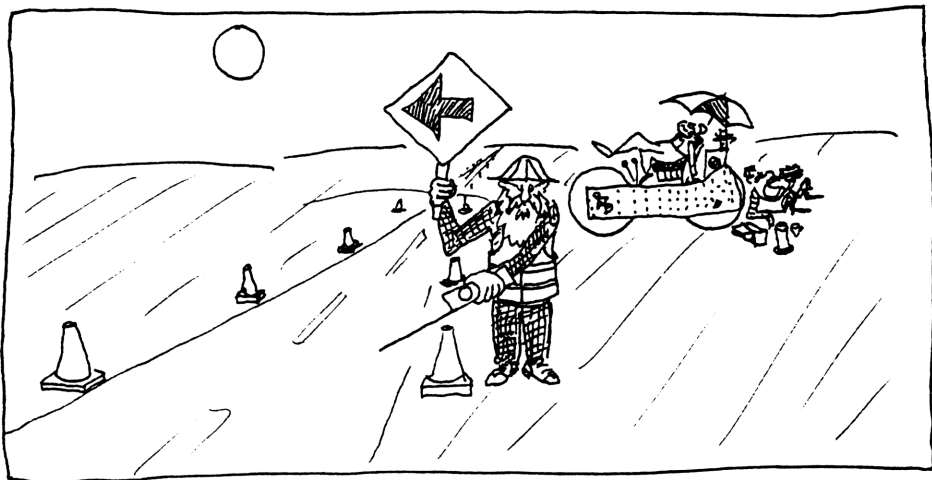
One More: Replace the 5 on line 06 with a 7.

Solution: ▼▼▼▼↵7.

After using the ▼ key to position the program pointer to line 06, you then delete the line with the ↵ key, and finally insert a new line (06).








When deleting a line, you simply place the program pointer on that line and press ↵. This does the deletion and moves the pointer back to the previous line, so that you're all ready to insert a new line following – exactly replacing the deleted line!




















(Now, before you go any farther, you should re-edit your FIRST program back to its correct formula: Replace the 7 on line 06 with a 5, by pressing ↵5.)



Moving Around In Your Program

Hopefully, you already have a good working understanding about how to move around in your program memory:

- To move forward or backward one step in program memory – when there is *no menu* showing – use the  and  keys (page 175); when there *is* a menu showing, use  and  (page 176).
- To move to the permanent **.END.** of program memory, you press . This is what to do whenever you want to start keying in a new program.
- You can also go *immediately and directly* to any line in the current program *without* using SST or BST. For example, to move the pointer to line 07 in your FIRST program, you could...

Do This: . This will work in either Run Mode or Program Mode. In Program Mode, the  sequence (as in  and now ) signifies a command to be done immediately – not to be recorded for future execution.

Voilà! The program pointer is now pointing to line 07. This works only for the specified line 07 *in the current program*. If there were another program, (SECOND for example) your machine wouldn't "see" its line 07 at all – because there is an **END** partition in between.

- And you can use the do-it-now properties of the **■GTO◻** sequence to move to a label in your current program – as well as a line number.
-

Like This: Without using SST or BST – but while in Program Mode – go to (i.e. move the pointer to) the label, "FIRST".

Answer: In Program Mode, press **■GTO◻** **ALPHA** FIRST **ALPHA**.

Nothing to it!

But Notice: Press **▼** a couple of times, then **EXIT** out to Run Mode. Now move the pointer back to label "FIRST" again:

Since you're in Run Mode, there's no need to use the **◻** key to request immediate execution; you always get immediate results in Run Mode – no recording of instruction done here: Just press **■GTO** **FIRST**.

When in Program Mode and jumping to a label with **■GTO◻**, you'll need to *spell out* the label; but in Run Mode, you get to use the menu to select it.

What Do You Know?

Here's a quick summary of what you know about programming and Program Mode up to this point:

- You know how to find the permanent **.END.** and prepare for a new program;
- You know how to begin and name programs with labels;
- You know three ways to put an **END** to a program.
- You know how to write a simple, straight-forward program that solves an equation with one input;
- You know how to change a program by editing it;
- You know how to move around within a program, either step by step, or more rapidly;
- You know how to "run" or "execute" a program using the **[R/S]** and **[XEQ]** keys respectively;
- You know how to find out how much memory you have left.

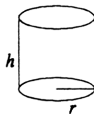
Now, what if you want to write a more complicated (and smarter) program?

"Read on, Macduff!"

Variables And How To Use Them

Suppose you want to write a program to calculate the volume and total surface area of a cylinder, given the radius (r) and height (h). The relevant equations are:

$$\text{Volume} = \pi r^2 h \quad \text{Area} = 2\pi r(r + h)$$



There'll be *two* inputs (radius and height) and *two* outputs (volume and area) in this problem, right? You'll have to be careful to avoid mixing them up. How are you going to do this?

First: Write out *in words* exactly what you want your program to do.

"Hmmm: I want the calculator to tell me when to input the radius and the height of the cylinder, making sure that I don't get them confused.




Then it should calculate the volume and surface area.











Then I want it to show me these answers, making it clear which answer is which."

All right, that's not asking too much of your HP-42S – especially if you take these "wants" one at a time, in the order you listed them here. That's the essence of programming – breaking a large problem down into an orderly series of smaller, simpler problems.

But before you start on these problems, you need to prepare a space and a name for the new program....

That Is: Begin a new program in program memory and name it with the label "C_YL".

Like So:    takes you to the end of program memory and puts you into program mode.

Now begin the new program with a label to name it: Press   (twice because you'll use this menu often while programming), and     
  .

Using The **INPUT** Function

In the **PGM.FCN** Menu, there's a special set of instructions designed to help you manage inputs and outputs more easily: **INPUT**, **VIEW**, **AVIEW**, **PROM**, **MVAR**, and **WARM**.

Now, the first thing you wanted the calculator to do is to prompt you for the "inputs" – making it clear as to which one it wants, so that you don't mix them up.

Well, you can actually tell it when to expect a *particular* input by using the **INPUT** function.

Try It: Tell your calculator to stop, prompt for, and store two inputs. The first one should be stored in a real variable, R (for radius) and the second one in a real variable, H (for height).

Solution: Press **INPUT** **(ALPHA)** **RTN** **R** **(ALPHA)** and **INPUT** **(ALPHA)** **FGH** **H** **(ALPHA)**.

Seems pretty simple doesn't it?

Well, it is simple. The **INPUT** instruction does a lot *automatically* for you. Whenever a running program encounters an **INPUT** instruction, it does several things. Take a look:

- First, it halts execution and looks in variable memory for the variable named in the **INPUT** instruction – and if it doesn't find such a variable, then it *creates* one with that name.
- Next, it prompts you for the value you want to store into that variable – by replacing your normal view of the X-register with a *prompt*, showing the name of the variable and the variable's *current* value (which will be zero if it has just been created).
- Now it's your turn: You key in the new value to store in the variable and then press the **[R/S]** (RUN) key to continue (of course, since you're being shown the *current* value in the prompt, if you decide you don't want to change that, all you do is press **[R/S]**, without keying in *any* value).

All this – an automatic stop, variable search-or-creation, recall, prompt, and storage – with one little **INPUT** command!

And now you've handled all your input problems for your CYL program with just two uses of this command – one for each input. So much for the first of your three problems....

Next up: Do the actual math.

Problem: Write a program that uses the existing values in variables R and H to calculate the volume and surface area of a cylinder according to the formulas given on page 186.

Like This: As with most programming of actual calculations, just use the sequence of keystrokes you'd need to solve this problem manually:

■ π [RCL] [ALPHA] [RSTUW] ■ R [ALPHA]
■ [TOP.FCN] ■ X^2 [X] [RCL] [ALPHA] [FGHI] ■ H [ALPHA] [X]

That's the volume. Then

■ [LAST X] [RCL] [ALPHA] [RSTUW] ■ R [ALPHA] [+]
■ [LAST X] [X] ■ π [X] [2] [X] calculates the surface area.

So the volume will be left in the Y-register and the surface area will be in the X-register when the program CYL is run.

Does the program work – at least the portion you've done so far?

Test It: Use CYL to find the volume and surface area of a cylinder with a radius of 8 cm and a height of 18 cm.

Solution: First **[EXIT]** from Program Mode. Then press **[XEQ] [CYL]** and see the **R?0.0** prompt on the lower display line (where the X-register is usually shown). You're being shown the current value of R and then asked to INPUT its new value. So press **[8]** and then **[R/S]** to continue.

Now respond to the **H?0.0** INPUT prompt similarly, by pressing **[1][8][R/S]**....Lo and behold – the answers!

x: 1,306.9 (area, cm²) and **y: 3,619.1** (volume, cm³)

One More: Find the volume and area of cylinder with a radius of 4 cm and a height of 9 cm.

Solution: Press **[R/S][4][R/S][9][R/S]**.

Since you just finished using the program CYL once, it's the current program and you can just press **[R/S]** to start it running again. Notice the **R?** and **H?** prompts: They now show the values from the previous example.

Answers: **326.7** (cm²) and **452.4** (cm³)

Getting Your Program To Talk Back

Well, the program CYL is fine and dandy as long as you remember which answer is which. But suppose you're not the only one to *use* your program. Sure – *you* understand it because you did the programming, but your friend might easily get confused.

So how can you make your calculator give more helpful instructions and messages? By using **VIEW**, **WVIEW**, and **PROM**.

Challenge: Modify the program CYL so that the volume answer is clearly labeled as such.

Solution: Enter Program Mode with **PRGM**. Then **GTO** **09** **ENTER**, to get to line 09, which is the last calculation for the volume.

Now add a step: **STO** **ALPHA** **VOLUME** **ALPHA** stores the resulting number into the variable, VOLUME.

Now **GTO** **01** **ENTER**, to get to line 19, which is the last step, and add one more step for clearer output: **PGM.FCN** **WVIEW** **ALPHA** **VOLUME** **ALPHA** displays the value of the variable VOLUME, *along with its name*.

Test It: **EXIT** Program Mode and press **XEQ** **CYL** **R/S** **R/S**. Aha! Now the volume is labeled – and so you also know that the other entry (in the X-register) must be the area!

So, you can see how the **VIEW** function is used: to display the *name* of a variable alongside its value. This is very useful in labelling the results of calculations at the end of the program.

But how can you have your program display other types of messages, that can help you (or your friend) figure out what to do?

That's where **AVIEW** and **PROM** come in.... Both **AVIEW** and **PROM** can cause the *contents of the ALPHA-register* to be displayed – and the program to stop running.*



Of course, this means you need to have your message already stored in the ALPHA-register prior to using AVIEW or PROMpt. Thus, you need to do two things for each message you want your program to display:

- You must store the message in the ALPHA-register by entering an ALPHA string.
- You must use AVIEW or PROMpt to then display the contents of the ALPHA-register.



With that understood, you're ready to make your program sit up and speak...

*The PROMpt function always stops the program, whereas this is optional with AVIEW. You'll learn how to exercise your options a bit later in the Course.

Try This: Modify the program CYL to deliver the messages, INPUT RADIUS and INPUT HEIGHT, thus telling the user more explicitly to key in the radius and height.

Solution: The first step is to create the message in the ALPHA-register: Enter  PRGM Mode,  to line 01 and enter the ALPHA string, INPUT RADIUS, as follows:




 ALPHA INPUT RADIUS ALPHA

Then select   PROM to instruct your calculator to display the contents of the ALPHA-register.

But wait – *both* INPUT and PROMpt will stop the program from running – you don't want it to stop *twice* for each input. Hmm...you need to delete one of these two statements. But which one?

Well, you definitely want to see the message you've just created, so you can't delete the PROMpt. But you also need to create and/or store into the variable, R, which the INPUT instruction had been doing; you don't need the INPUT's prompt any more, but you still need to store your data....

Aha! Just replace the INPUT "R" with STO "R"!

Move to 04▶INPUT "R", press  and   to insert the new line.

Now you need to do the same three changes to the other input variable (H).

So ∇ down to: 05 INPUT "H" and



[ALPHA] INPUT HEIGHT [ALPHA]

[PGM.FCN] ∇ PROM

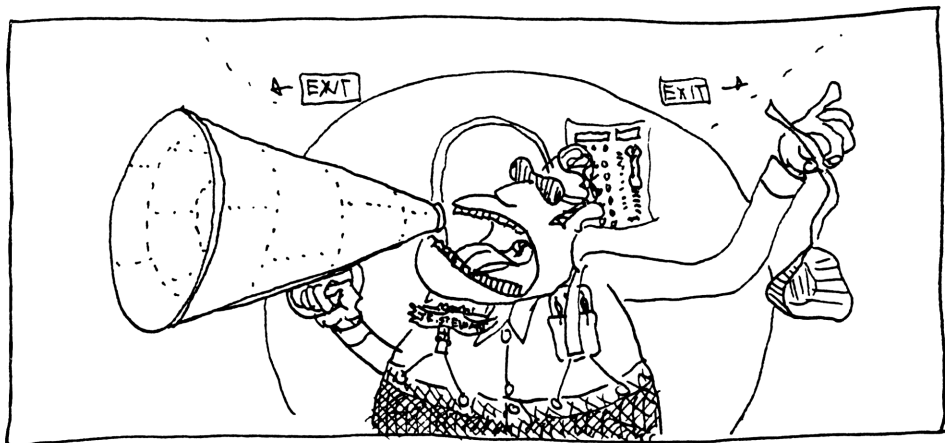
[STO] H

Finally, [EXIT] Program Mode and test your creation by [XEQ] [C/L] using a radius of 5 and a height of 8.

Result:

VOLUME=628.3
x: 408.4

And though you don't get to see what's currently stored in the variables, the prompts that tell the user to INPUT VOLUME and INPUT HEIGHT are much more explicit this way, aren't they?



PROMPTing For ALPHA Data: **AON** and **AOFF**

The program CYL provides a good illustration of how the INPUT or PROMPT instructions can be used to enter numerical data. But how do you enter ALPHA data into a program? Can you use INPUT also?

Generally, INPUT works the same no matter which data type you're using – *but it always assumes that your response will be in the X-register when you press **[R/S]** to continue the program.* That's not true when you key in ALPHA data to the ALPHA-register, is it? And besides, you need to have access to the ALPHA menu, too; using INPUT doesn't automatically give you this access.

Sure, even with all these problems, you could *still* use INPUT, but this would force you to do (or program) a lot of extra steps – to get to the ALPHA register, key in the data, then store this data into the X-register and return to the Stack. What a pain.

In fact, no single command does a very convenient job of prompting for ALPHA data; you need to use a combination of several. But no matter how you do it, the instructions ALPHA ON (AON) and ALPHA OFF (AOFF) are very handy, for they cause the ALPHA Menu to be turned on and off, respectively.

That is, if you want to stop a program to enter some ALPHA data, you should include an AON instruction immediately before a PROMPT statement in your program (and an AOFF immediately after). That way, when the program stops and prompts you for input, the ALPHA Menu is already showing and you can immediately input characters rather than numbers. Walk through an example....

Walk: Write a short program, called "NAME", that prompts you for your first name and then displays a personalized message: HI, (your first name).

Steps: Keystrokes

Display

GTO •• PRGM	00 { 0-Byte Prgm }
PGM.FCN LBL NAME ALPHA	01 LBL "NAME"
ALPHA YOUR 1ST NAME? ALPHA	02 "YOUR 1ST NAME?"
CATALOG FCN ▼ RON	03 RON
PGM.FCN ▼ PROM	04 PROMPT
ALPHA STO 04 EXIT	05 ASTO 04
CATALOG FCN ▼ ROFF	06 ROFF
ALPHA HI, ALPHA	07 "HI, "
ALPHA RCL 04 EXIT	08 ARCL 04
PGM.FCN RVIEW	09 RVIEW
▼	10 .END.

Press **EXIT** and then **XEQ NAME**. Input your first name and press **R/S**.... Not bad, eh?

Notice that if your name has more the six letters, you'll only see the first six letters displayed. This should remind you that you can have a maximum of six ALPHA characters when you store an ALPHA string as a datum in a variable or register.

Look at how the program in this last example actually works:

Line 02 prompts for the input and the AON in line 03 displays the ALPHA Menu (*and* the message, of course, since it's in the ALPHA-register by then). When the PROMPT in line 04 stops the program, you can key in your name, then press **[R/S]** to re-start the program. Notice that you're inputting directly into the ALPHA-register, so that the program must use **ASTORE** to store the first six letters of your name into a register (register 04, in this case).

The ALPHA-register is turned off (you must be tidy!) using **AOFF** in line 07. Then, after putting "HI, " into the ALPHA-register, your name is added onto the end of the "HI, " using **ARCL**. Remember from page 124 that **ARCL** *adds* to the ALPHA-register and doesn't simply replace the contents. Finally, the results are displayed with **AVIEW**.



In sum:



- To prompt for numerical data entry, use the INPUT statement, pressing the **[R/S]** when you're through keying in each datum.
- To prompt for ALPHA data entry, use the AON, PROMPT, AOFF sequence – as you just saw in the program, NAME.

INPUT and PROMPT are very useful tools, especially for short, straight-forward programs. However, the folks at HP have taken great pains to give you another, even slicker, way to key values into a program (and they would be greatly hurt if you didn't come to know and love it)....

Variable Menu

Variable menus can be a lot of fun. Certainly they make it *very* easy for a program user to input variables – and in any order!

When a program uses variable menus, a menu is displayed and the program stops, allowing you to store, recall, and view any of the variables you are using (the Power Tools  **SOLVER** and  **∫f(x)**, which you'll learn about in more detail later, both make extensive use of variable menus).

The two special functions used to make variable menus can both be found in the  **PGM.FCN**  Menu: **MVAR** and **VARM**.

The VARiable Menu (**VARM**) function tells your calculator to build the special menu for a particular program – a menu that may contain any or all of the variables used by that program.

The Menu VARiable (**MVAR**) function tells the VARM function which variables should be put on the Menu it is creating. You use the MVAR function *only for those variables you want to appear on the menu*.

Look at the next page, which compares the program CYL with a new program, NCYL, that does the same thing, but uses variable menus instead of input prompts.

Program listing for CYLProgram listing for NCYL

00 { 79-Byte Prgm }

01 LBL "CYL"

02 "INPUT RADIUS"

03 PROMPT

04 STO "R"

05 "INPUT HEIGHT"

06 PROMPT

07 STO "H"

08 PI

09 RCL "R"

10 X↑2

11 x

12 RCL "H"

13 x

14 STO "VOLUME"

15 LASTX

16 RCL "R"

17 +

18 LASTX

19 x

20 PI

21 x

22 2

23 x

24 VIEW "VOLUME"

25 END

A**B****C****D****E**

00 { 76-Byte Prgm }

01 LBL "NCYL"

02 MVAR "R"

03 MVAR "H"

04 VARMENU "NCYL"

05 "INPUT VARIABLES"

06 PROMPT

07 EXITALL

08 PI

09 RCL "R"

10 X↑2

11 x

12 RCL "H"

13 x

14 STO "VOLUME"

15 LASTX

16 RCL "R"

17 +

18 LASTX

19 x

20 PI

21 x

22 2

23 x

24 VIEW "VOLUME"

25 END

Study the preceding page carefully – and notice these things:

- Each program has five functional sections, **A – E**, which make up the logical progression of many simple programs in your HP-42S:

Section **A** includes the program's **beginning and label**;

Section **B** has all the instructions necessary to **input** the radius and the height;

Section **C** has all the instructions for **calculating** the volume and surface area;



































Section **D** gives the instructions for **output** (just the **VIEW** command here);

Section **E** **ends** the program.

- The programs differ only in Section **B**, in the way in which the values for the radius and the height are **input** into the calculator.
- NCYL stops just once for data input (line 05), whereas CYL stops twice, at line 03 and line 06.
- There are some instructions in NCYL that you've not yet learned about!

Do This: Key in sections **A** and **B** of NCYL (page 200).

Here's How:    prepares a new program space.

Then:    NCYL 
     NCYL
 INPUT VARIABLES    
                  


Here's how these instructions will work:

First of all, the two **MVAR** instructions *must* come right after the label, "NCYL". Why? Because the **VARM** "NCYL" instruction tells the machine to "find the **MVAR** instructions immediately following the label "NCYL" and build a variable menu with the variables named in those **MVAR** instructions." In this case, it will find two **MVAR** instructions (for R and H) and therefore build a menu with just those two choices.

Then, of course, the next two lines provide a **PROM**pting message to help you know what to do. You've seen that kind of thing before, right?

As for line 07 **EXITALL**, its purpose (as its name implies) to **EXIT** ALL Menus, so that both lines of the Stack Display are visible once again. This is important in this NCYL program because both display lines are needed at the end to show the two answers: the volume is displayed in a friendly message covering the Y-register, and the surface area is displayed in the X-register.

Next Job: Finish keying in the rest of NCYL, then use it to calculate the volume and surface area of a cylinder with a radius of 15 cm and a height of 21 cm.

Solution: π RCL R \times^2 RCL H \times STO VOLU LAST \times RCL
R + LAST \times \times π \times 2 \times PGM.FCN VIEW VOLU

Then $\boxed{\text{EXIT}}$ Program Mode and press $\boxed{\text{XEQ}} \boxed{\text{NCYL}}$

Now key in the data... $\boxed{15} \boxed{R} \boxed{21} \boxed{H} \dots$
and let 'er rip: $\boxed{R/S}$.

Answers: 14,844.0 cm³ and 3,392.9 cm²

Now This: Verify that the current radius is 15 cm. Then change the height to 43 cm and recalculate the area and volume.

Like So: Press $\boxed{R/S}$ to run the program again, thus regenerating the variable menu. Then \blacksquare and *hold down* \boxed{R} to view its current value. Result: R=15.0

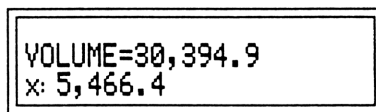
Now change a value and recalculate: $\boxed{43} \boxed{H} \boxed{R/S}$
Answers: 30,394.9 cm³ and 5,466.4 cm²

Some kinda wonderful, eh? By using MVAR and VARM, you can both store and view variable values – in any order you want!

Creating Two-Line Messages

Now that you know all the different ways of getting numbers into a program, how about trying to improve what the answers look like as they come out?

The output of NCYL now looks like this:



```
VOLUME=30,394.9  
x: 5,466.4
```

That is, only one of the two answers is labeled. Now, this probably doesn't cause too much confusion the way it is, but wouldn't it be nice if you could label *both* outputs – putting each output on its own line?

Of course it would. But to accomplish this task you'll need to use two important new characters that live in and around the ALPHA Menu.

Recall from page 35 that the ALPHA-register can hold up to 44 characters. That means you could theoretically create a message up to 44 characters long. And you learned on pages 193-194 that you can also create these messages with "recorded" instructions – program lines, right?

But what you didn't know is that *you can't put more than 15 characters on one program line*. The length of a program line is just more limited than is the ALPHA-register; that's the way it is.

So how are you going to create a message longer than 15 characters?

You're going to *combine more than one program line* into one single ALPHA message using the APPEND character, `␣`.

Example: Create a new program, labelled "HOWDY", that displays the message, HOW'RE Y'ALL DOIN'? (which is a total of 19 characters).

Solution: `■ GTO ■ . ■ PRGM` takes you to new program space.

`■ PGM.FCN ■ L&L HOWDY ALPHA` creates the name.

`■ ALPHA HOW'RE Y'ALL ALPHA` puts the first part of the message into the ALPHA-register.

Then `■ ALPHA ENTER` begins the next instruction with a `␣` character, thus telling the machine to append what follows to the current contents of the ALPHA-register. So you complete that instruction by typing in the rest of the message: DOIN'? `ALPHA`.

Finally, key in an instruction to display the entire message: `■ PGM.FCN ▼ PRGM`.

Now `EXIT` and try it out with `XEQ HOWD`....Voilà!

It is important to remember spaces. The APPEND instruction doesn't automatically put in spaces for you!

If you understand this, how about trying for a message that uses *two lines* when it is displayed?

Challenge: Change the program HOWDY to display a two-line message: HOW'RE Y'ALL DOIN'
THIS FINE DAY?

Solution: Simply APPENDING "THIS FINE DAY?" won't do this, because all you'll get is one long message on a single line – so the message will be half hidden.

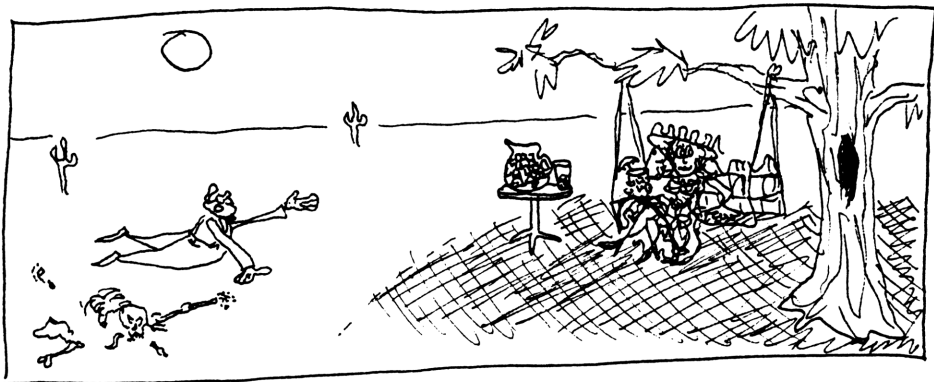
You need to use a special character on the ALPHA Menu:

PUNC ▼ **Lf** – called Line Feed (LF, for short):

PRGM ▲▲ takes you to line 03, where you then need to delete the "?" with a **+**, then APPEND a Line Feed character: **ALPHA** **ENTER** DOIN' **PUNC** ▼ **Lf** **ALPHA**.

Then for the last instruction, just APPEND the rest of the message: **ALPHA** **ENTER** THIS FINE DAY? **ALPHA**.

Now **EXIT** and press **XEQ** **HOWDY** to see a thing of beauty!



To review, then:

APPEND (⌈) allows you to add ALPHA strings to the end of messages already in the ALPHA-register.

Line Feed (⌊) allows you to break the contents of the ALPHA-register into two-lines for display purposes.

ARCL (remember, from page 124?) recalls the value of a particular variable or register and APPENDs *the characters representing that value (not a number)* to the current contents of the ALPHA-register.

So use the APPEND instruction to "tack onto" whatever's already in the ALPHA-register.

And use ARCL to recall (changing any number into a numeral) and APPEND to the ALPHA-register, all in one swell foop.

Now you're ready to put all of these ALPHA tricks to good use as you *really* fix up your NCYL program....

Fix It Good: Change NCYL so that it produces output like this:

VOLUME=30,394.9
SURFACE AREA=5,466.4

The Big Fix: **[GTO][NCYL][PRGM]** takes you to "NCYL".

Step up to line 24, delete it (**[↓]**), and add:

[STO][ALPHA]AREA[ALPHA] stores the second answer as the variable, AREA.

[ALPHA]VOLUME=[ALPHA] puts VOLUME= in the ALPHA-register.

[ALPHA][RCL][VOLUME] recalls the variable, VOLUME, and appends its *numerals* to the ALPHA-register, right after VOLUME=.

[ALPHA][ENTER][↑]SURFACE AREA=[ALPHA] appends the entire ALPHA-string to what's already in the ALPHA-register. The LINE FEED (**[↑]**) causes everything that follows it to be displayed on the second line of the display.

[ALPHA][RCL][ALPHA]AREA[ALPHA] recalls the variable AREA and appends its numerals to the ALPHA-register, right after SURFACE AREA=.

[PGM.FCN][↓][PRGM] causes the program to stop running and the entire message to be displayed.

[EXIT] and **[XEQ][NCYL][R/S]** to test the program.... It works!

Review

Already in this chapter, you've learned a lot of new things about programming your HP-42S. Pause here and look at what you now know:

- You can find the permanent **.END.** and prepare for a new program.
- You know how to begin and name programs using labels.
- You know three ways to END them.
- You know how to change a program.
- You know how to move around within a program, either step by step, or more rapidly.
- You know how to "run" or "execute" a program using the **[R/S]** and **[XEQ]** keys respectively.
- You know how to find out how much memory you have left.
- You know two different ways to input values into a program.
- You know how to use the VIEW instruction to label an output.
- You know the program instructions needed to store messages in the ALPHA-register.
- You know how to use the PROMPT and AVIEW instructions to display those messages.
- You learned that the EXITALL instruction exits all menus, taking you to a Stack Display.
- You know how to use the APPEND and LINE FEED characters to develop longer, more complicated, even two-line messages.

You really know *all that*?

There's one way to find out....

Preliminary Programming Probe

1. What's the difference between Run mode (also called Stack mode) and Program mode? Can you always tell which mode your calculator is switched to by looking at the display?
2. What happens when you're in Program Mode, the program pointer is pointing at **END**, and you press **▼**?
3. If you have three programs, A, B, and C, stored in memory and you're currently looking at line 05 of program B:
 - a. What's the quickest way to move the program pointer to line 07 of program C?
 - b. What's the quickest way to start a brand-new program?
4. Name three instructions that can cause a program to stop running and wait for you to press a certain key.

How are they different from each other?

How do you restart the program after each of these instructions?

5. The formula for the volume of a sphere is

$$V = \frac{4}{3}\pi r^3$$

where r is the radius of the sphere.

Write a program, labelled "SPHERE", that will compute the volume of a sphere, given its radius. That is, write a program to solve this equation:

$$OUTPUT = \frac{4}{3} \times \pi \times (INPUT)^3$$

Use an **INPUT** statement and label the output answer.

6. Write a program, called GREET, that displays the message, "THE RADIBOLICAL HP-42S AT YOUR SERVICE" on *two* lines of the display. Both lines should be centered in the display.
7. The density of a material is defined as its mass divided by its volume. The volume of a rectangular solid is:

$$VOLUME = Length \times Width \times Height$$

Write a program, labelled "DENSITY" to calculate the density of a rectangular solid (to 3 decimal places) from the mass (M), length (L), height (H), and width (W). Use menu variables, but prompt for the input. Label the output answer with **DENSITY =**.

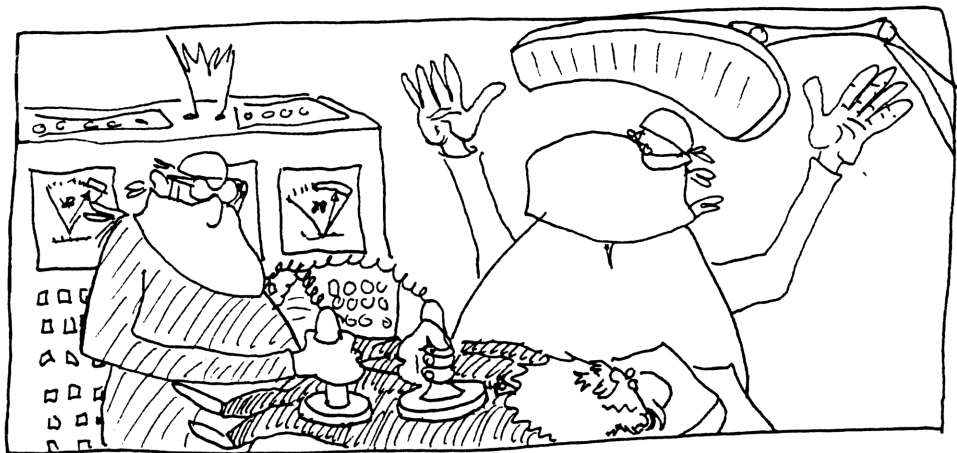
P. P. P. Answers

1. In Run Mode, functions will be executed immediately when you press the keys. In Program Mode, functions are stored as lines of a program (to be executed when the program is run). If the calculator is in Program Mode, you'll see one or more numbered program lines and the program pointer (▸).
2. The program pointer returns to line 00 of the current program. You cannot leave the current program by using the ▲ and ▼ keys (nor the **▢** **BS** and **▢** **SS** keys). The program lines are a kind of continuous loop – the "next" line after the "last" instruction is the "first" line.
3.
 - a. You can *either* **▢** **EXIT** Program Mode and press **▢** **GTO** or stay in program mode and press **▢** **GTO** **▢** **ALPHA** **C** **ALPHA**. Then (in either case), press **▢** **GTO** **▢** and **0007** or **7** **ENTER**.
 - b. Press **▢** **GTO** **▢** **▢** to prepare to key in a fresh, clean program.

4. Three instructions that will always cause a program to stop running are: INPUT, PROMPT, and STOP (which you get by pushing the **R/S** key while in Program Mode):

- INPUT has an additional function for both the display and the program. When the program encounters an INPUT, the user will see the name of a variable and a ?, which means that the user should then key in the desired value for that variable. Then, when the program is restarted, the value in the X-register will be stored in that variable.
- PROMPT affects only the display. When the program stops because of a PROMPT, it displays the current contents of the ALPHA-register until the program is restarted.
- STOP affects neither the display nor the the program. When the program halts because of a STOP, it just stops – nothing more happens.

In all cases, to restart the program after it stops, you need to press the **R/S** key.



5. After moving to a new program space with **▢** GTO **▢** • **▢** PRGM):

You Key In

▢ PGM.FCN **▢** LBL SPHERE **▢** ALPHA
▢ PGM.FCN **▢** INPUT **▢** ALPHA RADIUS **▢** ALPHA
▢ 3
▢ y^x
▢ π
▢ ×
▢ 4
▢ ×
▢ 3
▢ ÷
▢ STO **▢** VOLU
▢ PGM.FCN **▢** VIEW **▢** VOLU
▢ ▼

You See

01 LBL "SPHERE"
02 INPUT "RADIUS"
03 3
04 Y↑X
05 PI
06 ×
07 4
08 ×
09 3
10 ÷
11 STO "VOLUME"
12 VIEW "VOLUME"
13 .END.

When you actually use the program, the INPUT statement makes the display actually say RADIUS?, so that you won't forget what needs to be keyed in.

Then the VIEW statement at the end of the program labels your answer, VOLUME=. Since it's your only output, you don't need to worry about any more complicated ways of labeling it.

Now test the program by pressing **▢** EXIT **▢** XEQ **▢** SPHER, then use a radius of 8: **▢** 8 **▢** R/S Answer: VOLUME=2,144.7

6. This program is similar to the one you did earlier, on page 206. You need to plan your spacing carefully: You can put a maximum of 22 characters on each line, and there are 38 characters in the message, so you'll have six left over on the second line.

The program listed below gives you the following display (including the three spaces before " AT YOUR SERVICE!" and three after):

THE RADIBOLICAL HP-42S
 AT YOUR SERVICE!

You Key In

You See

▀ GTO ◦ ◦ ▀ PRGM

▀ PGM.FCN LBL GREET ALPHA

▀ ALPHA THE RADIBOLICAL ALPHA

▀ ALPHA ENTER HP-42S: AT ALPHA

▀ ALPHA ENTER YOUR SERVICE! ALPHA

▀ PGM.FCN ▼ PROM

▼

00 { 0-Byte Prgm }

01 LBL "GREET"

02 "THE RADIBOLICAL"

03 ␣" HP-42S: AT "

04 ␣"YOUR SERVICE!"

05 PROMPT

06 .END.

Line 03 calls for the careful planning. You need to remember the space before "HP-42S" and the LINE FEED (␣) immediately after it. Then come the *three* spaces before "AT".

GREET shows you an example of how one message can be broken up into *three* program lines! Test it (EXIT) (XEQ) (GREET)!

7. This program has four inputs and one output, making it an ideal candidate for menu variables. Here goes:

You Key In

```

GTO • • PRGM PGM.FCN
PGM.FCN LBL DENSITY ALPHA
MVAR ALPHA M ALPHA
MVAR ALPHA L ALPHA
MVAR ALPHA H ALPHA
MVAR ALPHA W ALPHA
DISP FIX 0 3
VARMENSI
ALPHA INPUT VARIABLES ALPHA
PROMPT
RCL ALPHA M ALPHA
RCL ALPHA L ALPHA
RCL ALPHA H ALPHA
X
RCL ALPHA W ALPHA
X
+
STO ALPHA DEN ALPHA
ALPHA DENSITY = ALPHA
ALPHA RCL ALPHA DEN ALPHA
PGM.FCN PROMPT

```

You See

```

00 { 0-Byte Prgm }
01 LBL "DENSITY"
02 MVAR "M"
03 MVAR "L"
04 MVAR "H"
05 MVAR "W"
06 FIX 03
07 VARMENU "DENSITY"
08 "INPUT VARIABLES"
09 PROMPT
10 RCL "M"
11 RCL "L"
12 RCL "H"
13 x
14 RCL "W"
15 x
16 ÷
17 "DENSITY = "
18 ARCL ST X
19 PROMPT
20 .END.

```

Notice that an `EXITALL` statement was *not* included in this program. This is because there's only one output, so you don't need to clear any menus in order to see both lines of the display.

The position of the `FIX` statement isn't critical, as long as it precedes the `ARCL` (remember that any append – such as that done by the `ARCL` – will use the current display setting for its guide as to how many decimal places to append).

Since all of the `MVAR` statements must come right after the label "DENSITY", the earliest you can `FIX` the display setting is step 06. And it's a good idea to do it there, so that when you're keying in your inputs, those values will be echoed back to you in the same precision as the final answer.

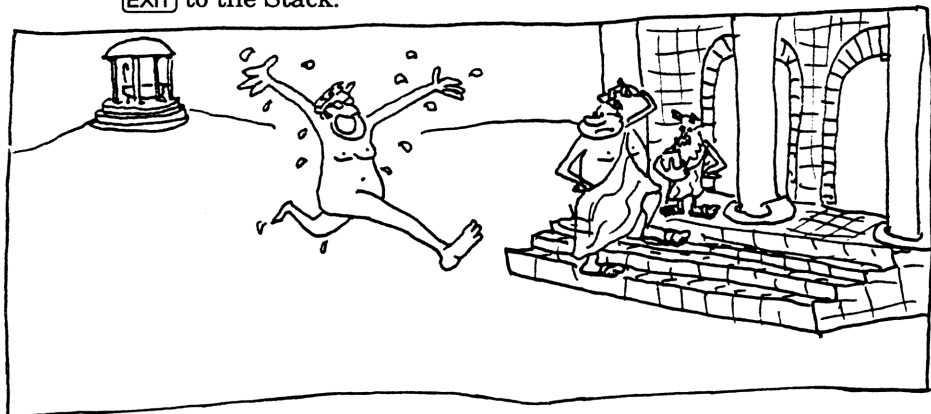
So try the program (`EXIT` `XEQ` `DENS`), using the following data:

mass = 258.9 grams

dimensions: 10.5 cm x 2.4 cm x 5.9 cm

Answer: DENSITY = 1.741 (g/cm³)

`EXIT` to the Stack.



Using Labels For All They're Worth

Up to this point, you've used labels only at the beginnings of programs – in order to name them. But that's not *nearly* all they're good for. Limiting the use of labels to that would be like limiting your sentences to stuff like "See Spot run!"

Besides marking the beginning of a program, *labels can mark important "landing points" for skipping around within a program.* This allows you to write "complex sentences" – programs that have several different paths, instead of just one.

This shouldn't come as too much of a surprise. You already know that the instructions XEQ ("execute") and GoTO are used to make the program pointer jump to particular labels, right?

When you press **GTO** **NCYL**, for example, you're telling the program pointer to search all of program memory for the line containing the instruction **LBL "NCYL"**, and to jump and place itself there. Similarly, when you press **XEQ** **NCYL** you also jump to that instruction, but then the program will begin running right at that point.

Well, by using labels at various different points in a program, you can collect together useful sequences of program steps, called *routines*, thus building very elaborate, yet compact programs.

The next thing you need to learn about, then, is that there are really two kinds of labels – *global* labels and *local* labels.

If you already know all about this, then you may skip to page 223.

Global Labels Vs. Local Labels

The rumors are true: There are two types of labels – global labels and local labels.

Global labels may be used for jumping between any two points in Program Memory. Local labels can be used only for jumping *within* a program. You can always tell a global label, because *it always uses ALPHA characters and is surrounded by quotation marks*.

Try This: Identify the following labels as either global or local:

```
01 LBL "FREE"  
02 LBL 01  
03 LBL B  
04 LBL "Q"  
05 LBL 99  
06 LBL "99"
```

Answer: 01, 04, and 06 are global; 02, 03, and 05 are local labels.

Global labels use up to seven ALPHA characters. Any ALPHA characters, including numerals, may be part of a global label. And global labels are always within quotation marks (and thus easy to identify).

However, all *numeric* labels from 00 to 99 and the *single-letter* labels A through J and a through e are reserved for local labels, and as such they are displayed *without* quotation marks.

Global labels can be found from anywhere in program memory, so they will be listed in the program catalog (CATALOG PGM) and listed on the menus for XEQ and GTO . Local labels will not.

Local labels are...well, local. *The only time the HP-42S can "see" a local label is when the program pointer is positioned within the program containing that label. If there's an END statement between the program pointer and a local label, the HP-42S won't find that label.*

Challenge: Write a program, labeled "COUNT", that starts at zero and counts continuously, pausing at each number.

Solution: Keystrokes

Display

	00 { 0-Byte Prgm }
COUNT	01 LBL "COUNT"
	02 0
	03 LBL 01
	04 PSE
(The PSE function briefly PauSEs a running program.)	
	05 1
	06 +
	07 GTO 01
	08 .END.

"COUNT" is a global label, right? If you press EXIT and XEQ , you'll find it listed (as COUNT) in the menu. But LBL 01 doesn't appear on the list because it's a local label *accessible only from within COUNT*.

Now, notice how COUNT works – press **COUNT**. It definitely works!

Line 02 "keys in" the starting zero to the X-register. Then line 03 serves as the beginning of the counting "loop" (local labels often serve this purpose). Next, line 04 pauses the program to display the contents of the X-register. Then lines 05 and 06 add 1 to the X-register, and line 07 sends the program pointer back up to line 03 to repeat the loop – which will begin with a 1 this time, then a 2 next time, etc.

Now Try This: Stop the program (**R/S**), key in **5**, and restart it by pressing **R/S** once more. The program starts its counting at 5, right? OK, stop it again and key in **34**, but then restart it by pressing **XEQ 01**.

What Gives? The second restart method works also – but only because the program pointer was still somewhere within the program, COUNT, where it could find that *local* label, 01, when you "called" it.

But Try This: Press **R/S** to stop the program, then **GTO 00** to go to new memory space. Now key in **12** and press **XEQ 01** to start it counting up from 12.

What Happens: You get the message, **Label Not Found!** Why? Because you're asking the HP-42S to find a local label when the program pointer is not in the program containing that label. You need to press **GTO COUNT** first, and then **XEQ 01**.

Which Type of Label To Use

OK – so when should you use a global label and when should you use a local label? Why worry about local labels at all? Why not simply use global labels for everything?

First of all, you *must* use a global label at least once in every program – so you can "call up" that program from anywhere else in Program Memory. In fact, you should use a global label at *any point* in a program that you want to be able to "jump" to from another program.

Global labels are more convenient, but it's best to use them conservatively, because they can cost you in two important ways:

- Global labels *use more memory* than do local labels. Why eat up bytes unnecessarily?
- Global labels can *slow down a program* when it's running. You see, when your machine searches for a global label, it must search through all of Program Memory, one global label at a time, to try to find the one you've called. And this search is in one direction only (starting from the **.END.**), so even if the label you want is "nearby," the pointer won't necessarily find it any the quicker, since it doesn't just "look around" to see if it can spot it quickly; it goes through this methodical search, which can take awhile.

So use local labels to make jumps within a program. This will usually make those internal jumps *faster* and take up *less memory* than if you use global labels for the same purpose.

Branching Out With **GTO** And **XEQ**

All right, now that you've got a good idea about labels, it's time to try some programs that use them to branch and "jump about."

Do This: Press **PGM.FCN** **▼▼** **BEEP**. Yep – your HP-42S can make noise! Key in these two short, noisy programs:

```
01 LBL "SONG"  
02 GTO "TN"  
03 BEEP  
04 TONE 0  
05 BEEP  
06 TONE 4  
07 BEEP  
08 TONE 8  
09 END
```

```
01 LBL "TN"  
02 TONE 9  
03 END
```

Solution: For SONG: **PGM** **PGM.FCN** **LBL** SONG
ALPHA **GTO** **ALPHA** TN **ALPHA** **PGM.FCN** **PGM.FCN** **▼▼**
BEEP **TONE** 0 **BEEP** **TONE** 4 **BEEP** **TONE** 8 **EXIT**

For TN: **PGM** **PGM.FCN** **LBL** TN **ALPHA**
PGM.FCN **▼▼** **TONE** 9 **EXIT**

Now, press **XEQ** **SONG** to hear the longer of these two pieces....

Hmmm...something's wrong!

Although you keyed in all those **TONE**s and **BEEP**s, all you get is one note! What's really happening when you run **SONG**?

Well, the calculator starts **eXECuting** at **01 LBL "SONG"**, but since the second line of that program is **GTO "TN"**, it jumps to global **LBL "TN"** and follows those instructions instead. When it reaches the **END** statement in the **TN** program, it just stops.

The **GTO** at line **02** of **SONG** is like a fork in the road. It's no temporary sight-seeing trip – it's a heavy commitment. *Whenever the pointer jumps to a label because of a GTO, it forgets all about where it jumped from* and just forges on, following whatever instructions it encounters in its "brave new world."

Ah, but...

Try This: Go back and change line **02** of **SONG** to **02 XEQ "TN"**

Solution: Press **▢ GTO SONG ▢ PRGM** to move the pointer back to the "SONG" program from the **END** of the **TN** program. Then move to line **02** with **▼**, and press **▢** to delete the line. Now key in the replacement line: **XEQ TN**.

Now **EXIT** and **XEQ SONG**...

That's better, no? So why does the **XEQ** work where the **GTO** didn't?

Well, if you listen closely, you'll hear a **TONE 9** (high pitch) *before* the first **BEEP**. The pointer jumps to the **TN** program, sounds the **TONE 9**, and then *returns* to the point it branched from in the **SONG** program, to continue on from there.

The program pointer knew how to do all that because of the **XEQ** – that's what **XEQ** really tells it: "Start from this point, search for this label, and when you find it, follow your nose through the instructions after it *until* you encounter either a **ReTurN** statement or the **END** of a program. From there you must *return* directly back to this **XEQ** and continue following the instructions that follow on from here."

An **XEQ** is really a side-trip that *temporarily* branches execution to another place until a **RTN** or **END** is encountered. By contrast, a **GTO** is *not* a temporary side trip; once you've "jumped" via a **GTO** command, the only way to jump back is to use another **GTO** command and another label. But **XEQ** carries its return address right along with it!

Notice that even when you manually *pressed* the keys, **[XEQ][SONG]**, you were telling the computer this same "do something and return here" message: Before you pressed the keys, it wasn't doing *anything*. And sure enough, it returns to "doing nothing" after it finally encounters the **END** of that program.

In fact, all the built-in functions and tools in your HP-42S are really little **XEQ** side trips, too – complete with built-in **ReTurN** statements. After using any of these tools, your machine returns to whatever it was doing previously, whether it was running a program or just sitting around.

Routine Subroutines

When you build such side-trips into an HP-42S program with XEQ, they're usually called *subroutines*. And you can even have "sidetrips within side-trips within sidetrips..." etc. – *nested* subroutines up to 8 levels deep. That can be a mess, so before you try anything so intricate, practice building and using some simpler subroutines.

Challenge: Go back and change the program GREET so that it branches to the program HOWDY and displays its message for three seconds before returning and finishing with GREET's original message.

Solution: You need to modify both programs: In GREET, insert a new line after the label that sends the pointer to HOWDY: **GTO GREET PRGM** and **XEQ ▼ HOWDY**.

Then **EXIT**, and press **GTO ▼ HOWDY PRGM** and **▼** to line 05. Now you can replace the PROMPT with two steps: a **CF 21** and an **AVIEW**, then three **Pauses**. The Clear Flag 21 will keep the program from stopping when it encounters the **AVIEW** (you'll more about this shortly). Here are the keystrokes:

↑ **FLAGS** **CF** **21** **PGM.FCN** **PGM.FCN** **AVIEW** **▼** **PSE**
PSE **PSE** **EXIT** **EXIT**.

Try it out: Press **XEQ GREET**.... ...Not bad!

Try another example – this time with some real number-crunching....

Challenge: The quantity of heat absorbed by a solid object when it increases in temperature can be calculated by the following formula:

$$Q = m \times c \times (T_F - T_B)$$

where

Q = quantity of heat (in calories)

m = mass of the object (in grams)

c = specific heat of the object (in calories/gram/°C)

T_F = final temperature of the object (in °C)

T_B = beginning temperature of the object (in °C)

Write a program, HEAT, that calculates Q if you input the mass (m), the specific heat (c), and the beginning and final temperatures in °F.

Hint: That last part is the tricky part: You need to convert °F to °C before you use a temperature in the equation. So, why not use your first program, FIRST (remember pages 175-177?) as a *subroutine* for this HEAT program?

Work on this. See if you can get the whole thing. Then turn the page to see one possible solution....

Solution:	<u>Keystrokes</u>	<u>Display</u>
	▀ GTO ◊◊ ▀ PRGM	00 { 0-Byte Prgm }
	▀ PGM.FCN LBL HEAT (ALPHA)	01 LBL "HEAT"
	▀ PGM.FCN	
	▀ PGM.FCN (Δ) MVAR (ALPHA) M (ALPHA)	02 MVAR "M"
	MVAR (ALPHA) C (ALPHA)	03 MVAR "C"
	MVAR (ALPHA) TF (ALPHA)	04 MVAR "TF"
	MVAR (ALPHA) TB (ALPHA)	05 MVAR "TB"
	VARM HEAT	06 VARMENU "HEAT"
	R/S	07 STOP
	RCL (ALPHA) TF (ALPHA)	08 RCL "TF"
	EXIT XEQ (▽) FIRST	09 XEQ "FIRST"
	RCL (ALPHA) TB (ALPHA)	10 RCL "TB"
	XEQ (▽) FIRST	11 XEQ "FIRST"
	—	12 -
	RCL (ALPHA) C (ALPHA)	13 RCL "C"
	(X)	14 x
	RCL (ALPHA) M (ALPHA)	15 RCL "M"
	(X)	16 x
	STO (ALPHA) Q (ALPHA)	17 STO "Q"
	▀ PGM.FCN VIEW (ALPHA) Q (ALPHA)	18 VIEW "Q"
	(▽)	19 .END.

Now **(EXIT)** and try the following data:

$$m = 200 \text{ g}; c = 1.00 \text{ cal/g/}^{\circ}\text{C}; T_f = 96^{\circ}\text{F and } T_b = 52^{\circ}\text{F}$$

Answer: $Q=4,888.9$ (calories)

Selective Branching: The Programmable Menu

Now, having familiarized yourself with the use of GTO and XEQ jumps to labels within a program, you're sure to appreciate the Programmable Menu.

The Programmable Menu is a menu containing your own special list of jumping instructions that you can select from with the menu keys. Remember VARMENU? That was an easy way to choose which *data* to input. Similarly, the Programmable Menu gives you an easy menu format for choosing which parts of a *program* to run.

In the previous problem, for example, you were asked to write a program to calculate the number of calories of heat absorbed by an object if you input the object's mass, its specific heat, and the beginning and ending temperatures in °F.

But suppose now that sometimes the temperatures were in °F and sometimes in °C. You'd need to have some way of telling your calculator which temperature scale you were using – and you'd want the calculator to treat such entries differently, of course.

This is where the Programmable Menu could be very useful: You could write a little program to give you a menu with choices for which kind of conversion routine you wanted to run. Then you could choose the proper routine for each datum.

Example: Write a program, labeled "QCF", that uses a Programmable Menu containing two choices – °F and °C. When you press °F, the program should XEQ "FIRST"; when you press °C, the program should simply END.

Solution: Keystrokes Display

▢ GTO ▢ ▢ ▢ PRGM	00 { 0-Byte Prgm }
▢ PGM.FCN ▢ LBL QCF ▢ ALPHA	01 LBL "QCF"
▢ ALPHA °F ▢ ALPHA	02 "°F"
▢ PGM.FCN ▢ KEY1 Σ+ ▢ FIRST	03 KEY 1 XEQ "FIRST"
▢ ALPHA °C ▢ ALPHA	04 "°C"
▢ PGM.FCN ▢ KEY2 1/x 0 2	05 KEY 2 GTO 02
▢ PGM.FCN ▢ MENU	06 MENU
▢ PGM.FCN ▢ LBL 0 2	07 LBL 02
▢ ▾	08 .END.

Lines 02 and 03 set up the °F menu item, and lines 04 and 05 set up the °C menu item. Line 06 causes the Programmable Menu to be displayed. Then the program will stop at the .END. – but it's still ready and waiting for you to select one of its pre-defined keys.

▢ EXIT and ▢ XEQ ▢ QCF. Now try 68°F and 34°C to see what happens. Pressing °F converts the temperature, but pressing °C does not, right?




See how the Programmable Menu works? Here's how you use it....

To create a Programmable Menu, you use these three instructions:

- **KEYG** ("on KEY Go to") lets you specify the label you want the program to jump to with a GTO when a particular menu key is pressed.
- **KEYX** ("on KEY eXecute") works just like KEYG function except that the named label will be called with an XEQ instead of a simple GTO when a particular menu key is pressed.
- **MENU** causes the Programmable Menu to be displayed and the program to stop running – analogous to the VARMENU instruction you use when working with variable menus.

The KEYG and KEYX instructions require *two* programming lines for each item you want on the menu.

On the first line you must enter an ALPHA string that will appear as the menu selection itself. *Notice that a selection does not need to be named with the label it calls.*

The second line must begin with the actual command, KEYG or KEYX. Then you tell the calculator on what key to "place" this menu item, by pressing one of the *eligible keys* (the 6 top-row keys and , , and  can be used, and they're numbered 1-9 in that order).

Then you finish the second line by specifying the label to GTO or XEQ when that menu item is chosen.

Another: Create a very short Programmable Menu in a program labelled "HAUNT". The only item on the menu should be **BOO!** (assign it to the **LN** key – that's KEY 5) and it should cause the eXEcution of the program HOWDY.












Solution: Keystrokes

Display

	00 (0-Byte Prgm)
HAUNT	01 LBL "HAUNT"
BOO!	02 "BOO!"
LN	03 KEY 5 XEQ "HOWDY"
	04 MENU
	05 .END.

Try it out to make sure it does what you expect it to do.
Everything OK?

OK, But: Suppose you then decide that you don't really want to use **⌘** in your menu, but **⌘** instead (it's friendlier). And you really wanted to assign it to the **⌘** key, not the **⌘**.

So: You want: 02 "HI!" and 03 KEY 3 XEQ "HOWDY".
To make these changes, step to line 03 while in Program Mode, then press:        
  .

Now try out this modification: Press **EXIT** and **XEQ HAUN**....

Hmmm...What's that **BOO!** item still doing there on the menu? Didn't you "kill" it by replacing it with the **HI!** item? Apparently not. What does it take to get an item off of the Programmable Menu once it gets put there, anyway? Do you need to clear the entire program?

Try It: Clear the program HAUNT from memory.

Solution: Press **CLEAR CLP HAUN**.

Now, just to make double-sure that these items are gone from the Programmable Menu, run another program that creates and uses that menu. **XEQ QCF** once again....

...Uh-oh.....("They're baa-aaack!") Those menu items continue to haunt your calculator. You can't seem to get rid of them. Why not?

You've actually killed ("deleted" is the more civilized term for it) only the *instructions that assigned* "BOO!" and "HI!" (and their associated XEQ and GTO commands) to the **LN** and **⌘** keys, respectively. **BOO!** and **HI!** are still assigned there, even though the instructions that put them there are now gone. Your HP-42S stores the nine programmable key assignments in its built-in system memory, but it doesn't automatically clean this storage area out when you delete the instructions that made those assignments. You need to do that yourself.

So, who ya gonna call?

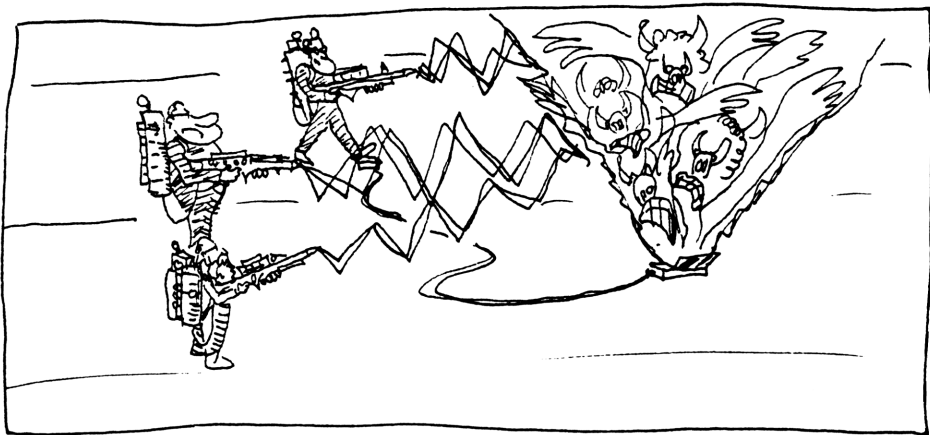
CLEAR **CLMN**, that's who! CLeAr MeNu clears all Programmable Menu key assignments, leaving a fresh slate for a new Menu.

Try It: Press **CLEAR** **CLMN**, and then **XEQ** **QCF**.

Result: You've done it! You've gotten rid of the ghosts! The only Programmable Menu items are those actually being created now by the current program QCF. You're no longer HAUNTEd by past menus or their ancestors.

It's a very good idea to include a CLMN instruction at the END of every program that uses the Programmable Menu – so that you don't go populating the menus of other programs with ghosts! You could also make it a practice to begin each program with a CLMN instruction, too. That way, you *know* you're slate is clean.

One more thing: If the current use of MENU happens to assign items where previous assignments had been, then of course, the new assignments will *overwrite* (replace) the previous ones. So that's one more way you can clear old assignments – overwrite them with new ones.






Multi-Page Programmable Menus

So much for unwanted Programmable Menu items. But how do you create a Programmable Menu with more than one page?

Challenge: Write a program, MUSIC, that uses a Programmable Menu to play the eight notes of a major scale: **Do, Re, Mi, Fa, So, La, Ti, Do** (the TONE's from 1 through 8 produce these scale notes). Use "DO↓" and "DO↑" to distinguish the octaves.









Solution: As always, break a big job down into separate, identifiable, easier-to-handle sub-jobs ("lumps"):

- Lump 1: Begin the program and label it.
- Lump 2: Create the first page of the menu ("DO↓" through "LA")
- Lump 3: Assign *menu-changing* and exiting routines to , , and  (this is how you can do multi-page Programmable Menus).
- Lump 4: Use the MENU command itself.
- Lumps 5-7: Repeat Lumps 2-4 for the other page of the Programmable Menu.
- Lump 8: Use subroutines and TONE statements to actually play the notes.
- Lump 9: Exit the program.

The following solution is organized in these Lumps. Keep your place as you read the discussion following it:

GTO •• PRGM	1	00 (0-Byte Prgm)
PGM.FCN PGM.FCN LBL MUSIC ALPHA		01 LBL "MUSIC"
LBL A ALPHA		02 LBL A
ALPHA DO↓ ALPHA		03 "DO↓"
▲ KEY Σ+ 0 1		04 KEY 1 XEQ 01
ALPHA RE ALPHA		05 "RE"
KEY 1/x 0 2		06 KEY 2 XEQ 02
ALPHA MI ALPHA	2	07 "MI"
KEY √x 0 3		08 KEY 3 XEQ 03
ALPHA FA ALPHA		09 "FA"
KEY LOG 0 4		10 KEY 4 XEQ 04
ALPHA SO ALPHA		11 "SO"
KEY LN 0 5		12 KEY 5 XEQ 05
ALPHA LA ALPHA		13 "LA"
KEY XEQ 0 6		14 KEY 6 XEQ 06
KEY ▲ ALPHA B ALPHA		15 KEY 7 GTO B
KEY ▼ ALPHA B ALPHA	3	16 KEY 8 GTO B
KEY EXIT 9 9		17 KEY 9 GTO 99
MENU		18 MENU
R/S	4	19 STOP
▼ GTO ALPHA A ALPHA		20 GTO A
LBL B ALPHA		21 LBL B
CLEAR ▼ CLMN		22 CLMENU
ALPHA TI ALPHA	5	23 "TI"
PGM.FCN PGM.FCN ▲ KEY Σ+ 0 7		24 KEY 1 XEQ 07
ALPHA DO↑ ALPHA		25 "DO↑"
KEY 1/x 0 8		26 KEY 2 XEQ 08
KEY ▲ ALPHA A ALPHA		27 KEY 7 GTO A
KEY ▼ ALPHA A ALPHA	6	28 KEY 8 GTO A
KEY EXIT 9 9		29 KEY 9 GTO 99

MENU		30 MENU
R/S	7	31 STOP
▼ GTO (ALPHA) B (ALPHA)		32 GTO B
LBL 0 1		33 LBL 01
▼▼ TONE 1		34 TONE 1
▼▼ RTN		35 RTN
LBL 0 2		36 LBL 02
▼▼ TONE 2		37 TONE 2
▼▼ RTN		38 RTN
LBL 0 3		39 LBL 03
▼▼ TONE 3		40 TONE 3
▼▼ RTN		41 RTN
LBL 0 4		42 LBL 04
▼▼ TONE 4		43 TONE 4
▼▼ RTN	8	44 RTN
LBL 0 5		45 LBL 05
▼▼ TONE 5		46 TONE 5
▼▼ RTN		47 RTN
LBL 0 6		48 LBL 06
▼▼ TONE 6		49 TONE 6
▼▼ RTN		50 RTN
LBL 0 7		51 LBL 07
▼▼ TONE 7		52 TONE 7
▼▼ RTN		53 RTN
LBL 0 8		54 LBL 08
▼▼ TONE 8		55 TONE 8
▼▼ RTN		56 RTN
LBL 9 9		57 LBL 99
■ CLEAR ▼ CLMN	9	58 CLMENU
■ CATALOG FCN ▼▼▼▼▼▼▼▼▼▼ EXITA		59 EXITALL

- Lump 1** begins the program and labels it with a meaningful global label ("MUSIC"), so that you can call this program from anywhere else in Program Memory.
- Lump 2** begins with a local label (A), to allow execution to jump back to this point. This section includes all of the instructions needed to make a menu containing the first six items – the first page of the Programmable Menu.
- Lump 3** defines what happens if , , and  are pressed.  and  will send the program pointer to Label B (Lump 5) which is the set-up for page two of the menu. This is exactly what you want the  and  buttons to do – "page" you through the menu.  (KEY 9) causes the program pointer to jump to the exit routine (Lump 9) which "tidies up" a bit before exiting the program.
- Lump 4** includes the MENU command to actually construct a Programmable Menu from the current key assignments. (those in Lines 04, 06, 08, 10, 12, 14, and 15-17).
- Lump 4 also contains a loop of circular reasoning – a *trapping* loop. Its purpose is to keep the menu showing after you select one of the menu items, thereby allowing you to select several in a row. *When the program pointer returns from eXECuting any item on a Programmable Menu, it returns to the line immediately after the MENU instruction itself.* In this case, that line is STOP, which stops the program without changing the Menu, allowing you to choose another "note," etc.

And notice the `GTO A` command after the `STOP`. This forces the user to use the Menu, since `[R/S]` merely restarts that section of the program. Trapping loops are often used like this to prevent the unwitting user from doing something that could "crash" the program.

Lump 5 sets up the second page of the Programmable Menu, *after first clearing the key assignments of the first page*. Without this clearing step, the last four of the *first*-page menu items would still be present on the second page.

Lump 6 sets up the `[▲]`, `[▼]`, and `[EXIT]` keys in a manner similar to Lump 3.

Lump 7 displays the second page of the menu, and then uses another trapping loop to "trap" the display of page two of the Programmable Menu and render `[R/S]` harmless (to see how critical these loops are to the smooth functioning of the program, try temporarily deleting them and then use the program!).

Lump 8 contains all of the small little subroutines that cause the notes to be played. Notice how each one ends with a `RTN` that sends the program pointer back to below the `MENU` statement that sent it.

Lump 9 is the `EXIT` routine that clears the Programmable Menu key assignments and exits all menus. It's always a good idea to try to leave the machine exactly as it was before the program was executed.

[EXIT] out of Program Mode to test this program, MUSIC (after first making sure that you've restored its proper structure after any experimenting you may have done).

Have Fun: See if you can play "Mary Had A Little Lamb."

Not Baa-d: [XEQ] [MUSIC], then play:

MI-RE-DO↓-RE-MI-MI-MI
RE-RE-RE
MI-SO-SO
MI-RE-DO↓-RE-MI-MI-MI
RE-RE-MI-RE-DO↓

Although your HP-42S is a bit limited in what it can do with sound (you can't vary the length of the TONE, for example) it's not completely uncultured, is it?

Experiment some more on your own. Be sure to check out both pages of the menu.

Also, try to mess it up ("crash it") while using it. A well-written program should resist all attempts – intentional and unintentional – to "crash" it (of course, all your "sabotage" attempts should be done in Run mode; you can always cause problems by going into Program Mode and mucking about, right?).

Letting The Calculator Decide What To Do

Menus – whether they're for inputting data or choosing program routines – are there for *you*. They are some of the most convenient ways to let you decide how and when to do something – right up to and during your use of a program.

But up to this point, your calculator hasn't had to make any such decisions. It has merely presented *you* with the options. Well, that's fine for the really important decisions, but if you, the user, are required to make *every* decision in a program of any complexity, it soon becomes so tedious that programming isn't worth the trouble.

So wouldn't it be peachy-keen if your calculator could make certain decisions automatically, without you having to pay attention to it nor worry about it?

Of course it would....

There are three main strategies that you can use to get your calculator to make its own decisions:



- Conditional tests
- Loop counters
- Indirect Addressing

You'll be introduced to each strategy, in turn, beginning with...

Conditional Tests

Conditional tests are just that – functions that test particular conditions in your calculator. *They are always questions which can be answered with "yes" or "no" (or "true" or "false").* That means it's best to use them when you want to do one thing for a "yes" answer – say, XEQ a subroutine – and quite another thing for a "no" answer – say, display a message.

The key is that *you don't know the answer to the question in advance. Only as the program is running will the answer be known.* Using a conditional test is one way of making an automatic decision.

In your HP-42S, the conditional test functions are all those that contain question marks (?) as part of their function names. These functions are spread out in several menus – primarily in the  PGM.FCN and  FLAGS Menus – and all work similarly:

- If the answer to the question is "yes," then the calculator continues right on, performing the next program step, etc.
- If the answer to the question is "no," then the calculator *skips* one step – the step immediately following the conditional test – and then continues executing the program normally.

This general pattern is called the "Do-If-True" rule: The next step will be done only if the answer to the test is "true" or "yes."

Remember that little program called COUNT (on page 220)? You keyed that in and ran it as a demonstration of how a GTO statement and a label can be used to form a continuously looping program, which will just keep going until you stop it. But now try this:

Challenge: Modify COUNT to use a conditional test that allows it to count up to a certain number (say, 10) then stop automatically.

Solution: The new COUNT might look like this (keystrokes are given only for new steps – the rest are easy now, right?):

Keystrokes

10

STO 00

RCL 00

PGM.FCN ▼ X=Y? N=Y?

R/S

R↓

Display

00 { 0-Byte Prgm }

01 LBL "COUNT"

02 10

03 STO 00

04 0

05 LBL 01

06 PSE

07 RCL 00

08 X=Y?

09 STOP

10 R↓

11 1

12 +

13 GTO 01

14 END

Look at this solution step by step: After the beginning (global) label, lines 02 and 03 store the number 10 in register 00. That's your ending count value; when the calculator gets to 10, you want it to stop counting. Then line 04 puts a 0 into the X-register. This is the starting count value.

After pausing to display this current count value (line 06), the program then recalls a copy of the ending value (10) to the X-register, which bumps the current value to the Y-register.

Next, at line 08, the conditional test asks the question, "Is the value in the X-register equal to the value in the Y-register?" As long as the answer to this question is "No" (which will be the case as long as the current count value in the Y-register hasn't reached 10 yet), then the **STOP** statement will be *skipped*. The machine will go on to perform lines 10 through 12, which add 1 to the current counter value. Finally, at line 13, it will go to the top of the loop (back to **LBL 01**), ready to repeat the loop with a new current count value.

When the count value has reached 10, however, the answer to the test will become "Yes," so the **STOP** statement is then performed, and that halts the program!



Notice also that you can easily modify this new COUNT program to count from any number to any other number – up or down, by one's or any other increment – simply by changing the numbers that are placed on lines 02, 04 and 11.*

Such simple changes that accomplish exactly the modifications you want – without rewriting the entire program – are the marks of good program design. Even better is to design the program so that you, the user, don't need to *edit* the program at all. You can key in your preferred counting parameters as you *use* the program.

Bonus Question: Suppose you want the program to count up to the number you key in right before you run it. How do you do this?

Bonus Answer: Delete line 02. Then the STO 00 will store whatever value you've keyed into the X-register before you run the program.

Go ahead and make this change to the program (you know the keystrokes, right?).

*If you were to change the increment value, you'd better be sure that it will eventually give you the ending value *exactly* – not skip over it. That is, the *difference* between your starting value and the ending value should be a whole multiple of the increment value. Otherwise, your current value would eventually become greater than the ending value without ever having satisfied the conditional test (and thus the loop would go on forever), because the test would always be answered as "No." Remember that the question it's asking is whether two values are *equal* (exactly) – *not* equal-to-or-greater-than.

Flags

So, you've used a simple conditional test to make the COUNT program more flexible. In that case, the test compared the value in the X-register with that in the Y-register.

But there's another kind of conditional test that doesn't have anything to do with the X-register. This test is called a *flag* conditional test. A *flag is an indicator with two possible, opposite values: Set or Clear**

A flag is a convenient way to store a certain kind of data without putting it into registers or variables. *Anytime you want to remember something that has just two possible values, you can do it by letting the status of a flag indicate it for you* (and you can choose which flag – they're numbered, similar to the registers).

You can see right away, for example, that a flag is an excellent way to remember the yes-or-no answer from a simple conditional test (such as the $X=Y?$ that you've already seen) long after the question was originally asked. It's not always convenient to retest the condition itself; (maybe those values you need to compare aren't in the Stack anymore or something), *but you can always retest a flag easily.*





This is because flags are stored off by themselves, not in data registers or variable containers (look back at the Big Picture, page 18). There are 100 flags in all (numbered from 00 to 99), and you can test any of them to see if they're set or clear.




*You could also think of these two values as "true or false," "up or down," "yes or no," "1 or 0," "Yankees or Mets" – whatever.

Of these 100 flags, 45 of them (flags 36-80) are called System Flags and are controlled only by the HP-42S (you can only test these; you can't set them or clear them explicitly). The machine uses these flags to help it monitor everything that's happening while you're using it.





Of the other 55 flags, 30 (flags 00-10 and 81-99) mean nothing to the computer. These flags are called User Flags, because they are defined only by you – the user. You can determine their meanings solely by the way you use them in your programs.




The remaining 25 flags (flags 11-35) are called Control Flags because they instruct the calculator to do something (or not to do something) – and you can alter the states of these flags as you wish.*

Try One: Clear flag 26, by pressing  **FLAGS**  **CF**  **2**  **6**

Now try to BEEP by pressing  **PGM.FCN**   **BEEP**

Nothing, right?

OK, now set flag 26, once again (this is its more common state):  **FLAGS**  **SF**  **2**  **6**.

Now try the BEEP again:  **PGM.FCN**   **BEEP**

That's better!

This is how you set and clear flags – and since flag 26 is the sound Control flag, when you "turned it off" (cleared it), you disabled the sound generator in your HP-42S and it couldn't BEEP at you!

*Pages 273-282 of your Owner's Manual give details about each flag in the HP-42S.

Challenge: Suppose you let the user choose to view the count more leisurely: When he/she keys in a *negative number as the upper limit* of the count, *two* PSE's should be used; otherwise, one is enough (the count will still be done in positive numbers). Use a flag to modify COUNT for this.

Solution: Here's one way (keystrokes are given for new steps):

	00 { 26-Byte Prgm }
	01 LBL "COUNT"
■ [FLAGS] ■ CF [0] [0]	02 CF 00
■ [PGM.FCN] ▼ [X?0] [X<0?]	03 X<0?
■ [FLAGS] ■ SF [0] [0]	04 SF 00
■ [CATALOG] FCN [ABS]	05 ABS
	06 STO 00
	07 0
	08 LBL 01
	09 PSE
■ [FLAGS] ■ FS? [0] [0]	10 FS? 00
■ [PGM.FCN] ▼ [PSE]	11 PSE
	12 RCL 00
	13 X=Y?
	14 STOP
	15 R↓
	16 1
	17 +
	18 GTO 01
	19 END

Remember: When you execute this COUNT program, the value then found in the X-register will tell the HP-42S both how long to keep counting (the magnitude of the value) *and* whether to pause once or twice in each loop (the sign of the value).

So, after the label, the first thing to do is to clear the flag you're going to use (line 02). That way, if you find it set later in the program, you know for sure it was set by the program – not prior to running the program. This is called "initializing" the flag.

Then, line 03 tests to see if the value in the X-register is negative, using the conditional test `X<0?` ("Is the value in the X-register less than zero?") If Yes, *do* line 04. If No, *skip* line 04 and continue at line 05. Thus, *only if that input value is negative* (i.e. if two pauses are desired), will flag 00 be set.

Lines 05 and 06 store the absolute value of the input number (the "positive version" of that number) in register 00. This will be the ending count value – the upper limit.

From here, the rest of the program proceeds as usual, except for one minor change: At line 10, following the first PSE, you use the flag you set (or didn't set) back in line 04. The second PSE (line 11) is executed *only* if the answer to the *flag's* conditional test is "Yes" (`FS? 00` means "is flag 00 set?") – because flag 00 was set only if the input was negative, and a negative input meant "two pauses, please!"

Clever, no? Test COUNT now: `[EXIT]` from Program Mode and key in a negative number. Then `[XEQ] [COUNT]`. Now try a positive number. You should see the difference in the length of the pause.

AVIEW And Flag 21

So that's how to use a User flag. But how would you use a Control flag in a program?

Back on page 193, you read that the main difference between PROMPT and AVIEW was that AVIEW *sometimes* caused the program to stop running while PROMPT *always* stopped the program.

Here's the whole story: When an AVIEW instruction is encountered in a program, the status of flag 21 determines whether the program halts or continues. If flag 21 is set, AVIEW will cause the running program to stop and display what's in the ALPHA-register; **[R/S]** must be pressed to continue. But if flag 21 is clear, AVIEW will display the contents of the ALPHA-register *while program execution continues*.

Try This: Key in the following short program. Then run it twice – once with flag 21 set and once with flag 21 clear:

```
01 LBL "FLAG"           04 AVIEW
02 "THIS IS A VERY"     05 XEQ "SONG"
03 L" 4-SHORT ONE"      06 END
```


Solution: Key in the program, **[EXIT]** and **[F] [FLAGS] [3F] [2] [1]**, and then **[XEQ] [FLAG]**. Try it again after you **[F] [FLAGS] [CF] [2] [1]**. Only in the first case will you see the message without hearing the song (until you press **[R/S]**), because flag 21 is *set*, telling the AVIEW instruction to halt the program.

Testing The Data Type

Simple conditional tests, flags,... what other kinds of conditional tests can you use in a program to let it make more of the decisions during execution?

Here's another kind to keep in mind: There are four special conditional tests that test the data type of the number in the X-register:

- REAL? asks, "Does the X-register contain a real number?"
- CPX? asks, "Does the X-register contain a complex number?"
- MAT? asks, "Does the X-register contain a matrix?"
- STR? asks, "Does the X-register contain an ALPHA string?"

You'll find all of these tests in the  CATALOG Menu, listed alphabetically. They're useful for detecting an illegal – or otherwise troublesome – data-type that would cause your program to stop or crash later (like many things in life, you can often avoid a lot of pain later by detecting and avoiding something earlier).

So, you now have *three* types of conditional tests at your disposal:

- Comparison tests, where the contents of the X-register are compared with either 0 or the contents of Y-register.
- Flag tests, where the status of a flag – Set or Clear – is checked.
- Data-type tests, where the data-type of the number in the X-register is checked.

Loop Counters

While Conditional Tests and Flags are clear, direct ways to get your calculator to make decisions without your input, there are two very convenient functions that can do the job even more efficiently whenever you're using repetitive program loops—as in the COUNT program.

These "loop counters" do two useful things at once.

- They provide built-in counting procedures.
- They allow exiting from loops after a given number of cycles.

The two functions are ISG and DSE which mean, respectively:

Increment and Skip if Greater than
and
Decrement and Skip if Equal to or less than

They each require an *argument* – that is, the name or number of a storage container (variable or register) that contains a special *loop-control number*.

For example, ISG "FRED" would retrieve its loop-control number from the variable "FRED." And DSE 18 would retrieve its loop-control number from register 18.

This allows you to store loop-control numbers at one point in your program and then make convenient use of them later. The loop-control number is a more compact, more efficient way to control your jumping in and out of loops than is a series of conditional tests.

Just how do loop-control numbers work? Take a hypothetical case: Suppose the number 27.0571216 is stored in register 02. If your calculator then receives the program instruction `ISG 02`, it does this:

1. It looks at the number in register 02, particularly at the integer portion (27 here) and the first five digits of the fractional portion (.05712 here), ignoring (but not deleting) any other digits.
2. It uses the digits in the fourth and fifth decimal places (at the far right) to form a separate integer (12 – "twelve" – here).
3. It adds (Increments) this new number to the integer portion of the original number. So the new number in register 02 is now 27.0571216 + 12 or 39.0571216.
4. Now the HP-42S makes a little conditional test: It compares the Integer portion (now 39) to the number that appears in the first three decimal places (057 – fifty-seven here). Then, only if the integer portion is Greater, the line following `ISG 02` in the program would Skipped (so it wouldn't skip in this case).

Thus the name: "Increment and Skip if Greater than." *Notice* that in these two conditional tests, *the Do-If-True rule is reversed*: Only if the answer to the test ("Is the Integer portion Greater than the Fractional Portion?") is "Yes," the next step is *skipped*.

Wondering why HP would make such a complicated and "backwards" conditional test for loop control? Try some loop control numbers.

Question: What value should start in register 00 so that repeated executions of `ISG 00` will count from 10 to 27 by three's?

Answer: 10.02703 as a loop-control number is seen as: 10.02703. The **integer** portion is the **current counting value**. The *first three decimal places* form the *ending counting value*. The last two decimal places form the increment – the amount the current value is increased on each loop.

Try Another: What loop-control number should be stored to count from 0 to 9 by one's?

Answer: 0.00901 or .00901 or .00900 or .009.

Because the most common way of counting is by one's, the increment is *assumed* to be *one* if the fourth and fifth place decimal places are zero (or unspecified).

Now, lest you be misled, the ISG and DSE functions *don't* form program loops all by themselves. They still need labels and GTO's, etc.

Try This: Write (but don't key in) a program loop to count from zero to nine, by ones.

Solution:

```
01 LBL "TRYIT"
02 0.00901
03 STO 00
04 LBL 01
05 ISG 00
06 GTO 01
07 END
```

Another: Write a program that counts from -44 to 100 by four's.

Solution:

```
01 LBL "WORKS"
02 -44.10004
03 STO 02
04 LBL 01
05 ISG 02 (The "argument," 02, here refers to a storage register.)
06 GTO 01 (The "argument," 01, here refers to a label)
07 END
```

Get the idea? Of course, these loops don't do anything constructive (not even displaying the count), but you can see how much easier these loops are to build than the loop you "built from scratch" in COUNT.

Now, what about DSE?

Example: Write a program loop to count *down* from 200 to 29 by 7's.

Solution:

```
01 LBL "OKIDOKI"  
02 200.02907  
03 STO "LOOP#"  
04 LBL 05  
05 DSE "LOOP#"  
06 GTO 05  
07 END
```

In this case, the control number is 200.02907, and with DSE, it's being *decremented*: The first time through the loop, DSE "LOOP#" subtracts 7 from 200 to get 193. Then it compares this 193 to 029. Since 193 is *not* Equal to *or less than* 29, no skipping takes place. The GTO 05 is performed, and around you go again.

On the second time around, the number stored in the variable "LOOP#" is 193.02907. So DSE "LOOP#" subtracts 7 from 193 to get 186. But 186 is still greater than 29, so no skip. And here you go again – around and around the loop, subtracting and comparing...

Finally, the value in "LOOP#" has been reduced to 32.02907. This time, when DSE "LOOP#" decrements this value by 7, you get 25.02907. Well, 25 is *less than* 29, so the test is finally satisfied (with a "Yes" answer), so the GTO is skipped, the loop is exited, and that's all she wrote.

Remember:

ISG means "InIcrement (add) and Skip if Greater than."

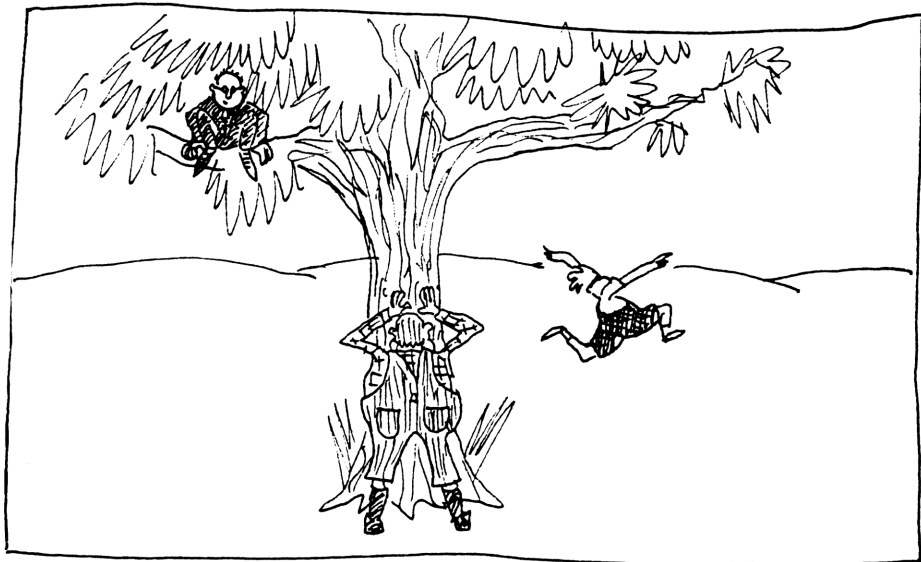
DSE means "Decrement (subtract) and Skip if Equal to or less than."

With that in mind, go for broke with this Loop Counting Challenge...

L. C. C: Rewrite (and make these changes to) the COUNT program just once more (this is the last time – scout's honor):

Rewrite it so that it uses ISG. The ending value should still be specified in the X-register (and negatives still mean an extra pause), but the amount of the increment should now be specified in the Y-register.

Test your finished solution by asking for an ending value of 39, with two pauses per cycle, counting by three's.



Solution: (One of many possibilities):

Keystrokes

1 0 0 0

÷

X<Y

1 0 0 0 0 0

÷

+

[delete 13 0]

RCL 0 0

CONVERT ▾ IP

[delete lines 19-24]

PGM.FCN ▾ ISG

Test It: EXIT 3 ENTER 3 9 +/- XEQ COUNT

Display

00 { 0-Byte Prgm }

01 LBL "COUNT"

02 CF 00

03 X<0?

04 SF 00

05 ABS

06 1000

07 ÷

08 X<>Y

09 100000

10 ÷

11 +

12 STO 00

13 LBL 01

14 RCL 00

15 IP

16 PSE

17 FS? 00

18 PSE

19 ISG 00

20 GTO 01

21 END

The first five steps are nothing new. They test for a negative number and adjust flag 00 accordingly. For your trial run (with -39 as the ending value), flag 00 will be *set*, requesting two pauses per loop cycle.

Lines 06 and 07 begin creation of the loop-control number. As you recall, the ending value in a loop-control number is the first three digits to the right of the decimal point. So, dividing the given ending value (currently sitting in the X-register) by 1000 moves it over to those 3 decimal places: $39 \div 1000 = .039$

Next (lines 08-10), the contents of the X- and Y-registers are exchanged so that you can horse around a bit with the increment value. Since the increment value in a loop-control number is always found at the fourth and fifth decimal places, you need to divide by 100,000 to move it there. In your trial example: $3 \div 100,000 = .00003$

Line 11 adds the contents of the X- and Y-registers together to form a complete loop-control number: $0 \text{ (implied)} + .039 + .00003 = 0.03903$

Line 12 stores the loop-control number in register 00, and *then* you're ready to enter the loop itself, denoted by the LBL 01 at line 13.

Line 14 recalls the loop-control number and line 15 extracts the Integer Part of the number (in this case, 0) and then displays it for one or two PSE's depending on the status of flag 00 (lines 16-18).

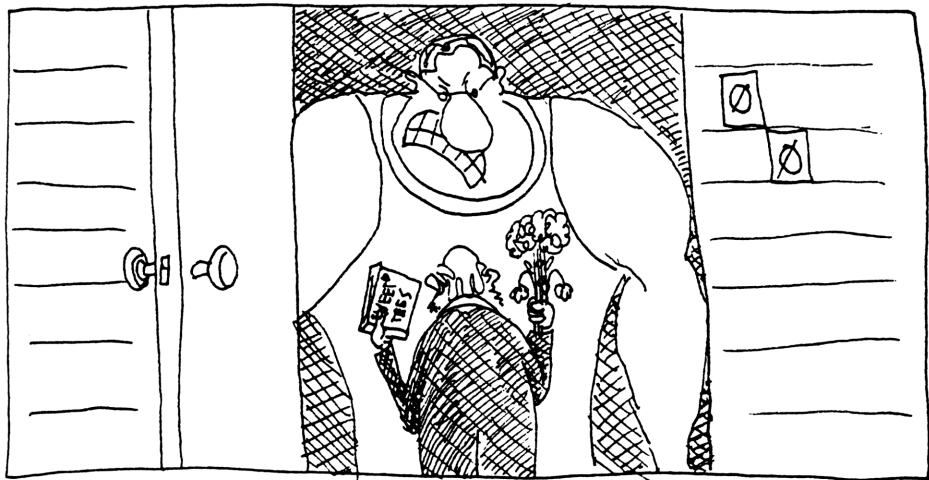
Finally, line 19 and its ISG 00 increment the integer portion of the loop control value each time through, then compares this with the fractional portion (first three digits), and decides (correctly) when to skip the GTO 01, thus ENDing the count and the program.

Indirect Addressing

So much for conditional testing and loop counters. It's time to look at the most subtle – yet, in many ways, most powerful – way to have your computer make automatic decisions: What *is* this stuff – "indirect addressing?" What's wrong with *direct* addressing (whatever that is)?

Here's how it works: A bashful young man wants to go on a blind date. Naturally, he calls a dating service for shy people – and naturally, he isn't given the address of the shy young lady, because this is much too personal to divulge to just anyone. Instead, he's told the address of her brother's home. There (after meeting with the brother's approval, of course), he'll be given her actual address. Thus, the dating service tells him *indirectly* how to get to his date's home – by giving him the address to go to in order to *obtain* her address.

This is much like a "RCL INDirectly" instruction in your HP-42S. Look, for example, at the instruction `RCL IND 00`. The brother's address is 00 (he "lives" in register 00). So the number *stored* in register 00 is the actual address of the shy young lady herself.



Challenge: Put the value 29.7 into the X-register and the value 3 in register 01. Then execute the function **STO IND 01**. What happens?

Answer: The set-up is: **3 STO 01 29.7** Then **STO • IND 01**

STO IND 01 says: "STOre the value currently in the X-register into the register *whose address is indicated by the contents of register 01*."

The HP-42S makes a copy of the 29.7 currently in the X-register and takes it over to look at register 01, where it finds a 3. So it then puts the 29.7 into register 03.

Notice that the **IND** functions are in the **□** "Menu."

Challenge 2: Starting with the above results – using indirect addressing only – store the number 29.7 into register 00.

Solution 2: **0 RCL • IND 01**

STO • IND • ST Y

With a 0 in the X-register, 29.7 is indirectly recalled from register 03. Then it's stored into the *register whose number is stored in the Y-register*. Since 0 is stored in the Y-register, 29.7 is stored in register 00!

Using Indirect Addressing To Make Decisions

To be sure, just because you understand how indirect addressing works doesn't tell you what it's good for, does it? How does it help you to make automatic decisions in your programs?

To begin with, keep in mind that indirect addressing works for other functions besides STO and RCL.* For example, suppose you have a program with 26 little subroutines – each with a numeric label and a RTN statement – one for each letter of the alphabet. Each subroutine calculates how much space and ink that letter requires on a certain type of printer. The program might look like this:

01 LBL "LETTERS"	•
02 LBL 01	•
•	•
•	LBL 25
•	•
RTN	•
LBL 02	•
•	RTN
•	LBL 26
•	•
RTN	•
LBL 03	•
•	END
•	

*The Operation Index in your Owner's Manual gives a complete listing of those functions that can indirectly address.

Each of those sections will do a different calculation, but for each input (each letter), there's only one section that applies.

Question: What's an easy way to tell the HP-42S to choose the proper section for a given letter?

Answer: Key in the number of the letter you want (1 for A, 2 for B, ..., 26 for Z) and then press **XEQ** **◻** **◻** **◻** **ST** **◻**, which is the instruction XEQ IND ST X.

To calculate the amount of ink needed for the letter K, for example, you would key in the number 11 (K being the eleventh letter). Now, when you use XEQ IND ST X this tells the HP-41 to "execute the section whose label is indicated in the X-register." In the X-register, it finds the number 11, so it executes the section with **LBL 11**.

Also, remember that you can execute ALPHA labels indirectly. That is, you could have labelled each of these subroutines with its own letter.

In that case, to execute the K routine indirectly, you'd need to key a K into the ALPHA-register, *then* **AST** *Or it into a data or stack register* and *then* use XEQ IND. You must store an ALPHA label name somewhere other than the ALPHA-register like this, because, unfortunately, there is *no* XEQ IND ALPHA function.

One of the best uses of indirect addressing is in combination with loop counters, so that you accomplish a great deal of work in very few instructions....

Challenge: Write a short program (10 steps) to store the integers 0 through 19 in registers 0 through 19, respectively.

Solution: Here's one way to do it:

<u>Keystrokes</u>	<u>Display</u>
	00 (0-Byte Prgm)
INDY	01 LBL "INDY"
	02 0.019
	03 LBL 06
	04 ENTER
	05 IP
	06 STO IND ST X
	07 R↓
	08 ISG ST X
	09 GTO 06
	10 .END.

Follow the logic of this program:

At line 02, the number .019 goes into the X-register. This will be the loop-control number for the ISG loop (as a loop control number, .019 is also 0.01901) – and it will also act as the index for indirect storage.

Notice that every time the local label at line 03 is reached, the loop-control number must be in the X-register.

Inside the loop, (line 04) the first thing to do is to send a copy of the loop-control number into the Y-register. Then line 05 extracts the integer portion of the original in the X-register. If the loop control number is 0.019, for example, the IP function puts the number 0 in the X-register.

Line 06 is an elegant masterpiece: `STO IND ST X` says "store a copy of the value in the X-register into the register indicated by the X-register." If that number is 0, then 0 is stored into register 00. If it's 19, then 19 is stored into register 19. The very number being stored also tells the machine where to store it!

Then, to end the loop, line 07 brings down the unmolested copy of the loop-control number from the Y-register to the X-register, and lines 08-10 provide the working guts of the loop counting mechanism, incrementing each time and skipping the `GTO` when the integer portion of the loop-control number finally exceeds 19.







Bonus Question: If you were to use `STO IND ST Y` at line 06 (instead of `STO IND ST X`), how would this affect the program's results?

Bonus Answer: Not at all: *Indirect addressing considers only the positive integer portion*, anyway. For example, as an indirect address, $5 = (-5) = (-5.922) = 5.013$.

Reviewing Your Programming Tools

Whew! In this Programming chapter, you've been introduced to *a lot* of new things. Here's a summary to give you a break and a review now. See if you can still recall and grasp these concepts (and if not, here are the page numbers for quick review):

Programming Basics

- The difference between Run and Program Modes (page 170);
- What the Program Display tells you (page 171);
- Beginning a program in its "own space."  GTO   (page 175);
- Ending a program and the difference between END and .END. (page 178);
- Editing a program with the , , and  keys (page 181);
- Positioning the program pointer to insert or delete a line (page 183);
- Finding out how much total memory you have left (page 168);
- Finding out the memory usage of the current program (page 171);

Programming Structures

- Labels: The differences between global and local (page 219)
- Subroutines: How they're "called" and how they RTN (page 226);
- Loops: When and how to use them (page 220);

Communicating With The User

- Keying data in before eXEQuting the program (page 180);
- Using INPUT to prompt for numerical data (page 188);
- Using AON-PROMPT-AOFF to prompt for ALPHA data (page 196);
- Using MVAR and VARMENU to enter data flexibly (page 199);
- Using VIEW to label numerical results (page 192);
- Creating long messages with the `␣` and `␣` characters (page 204);
- Using PROMPT, AVIEW and their differences (pages 193 and 250);

Making Decisions

- Programmer decisions: Branching with GTO and XEQ (page 223);
- User decisions: Using the Programmable Menu (page 229);
- Machine decisions: Conditional Branching
 - Conditional Tests
 - Value comparisons (page 242);
 - Flags – User, Control, System: How to set, clear, or test them (page 246);
 - Data Type: Finding the type of data in the X-register (page 251)
 - Loop Counters
 - Loop-Control Numbers and how they work (page 252);
 - ISG and DSE functions: When and how to use them with loop-control numbers (page 252);
 - Indirect Addressing: For XEQuting subroutines or for using with loop counters (page 260).

Prove Your Programming Proficiency

1. Suppose that you're in Program Mode, with the program pointer positioned at line 00 of a program labelled "WHATZIT". Explain the difference between the following two keystroke sequences:

a. **EXIT** **R/S**

b. **EXIT** **XEQ** **QUIT**

2. After a subroutine is eXEQuted and the program pointer encounters a RTN, where does the program pointer go next?
3. Remember the Quadratic Formula from 'way back in algebra class? Whenever you have an equation of the form

$$ax^2 + bx + c = 0$$

where a , b , and c are real numbers, you can calculate x from a , b , and c according to the following formula:

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

Write a program, QUAD, that solves this Quadratic Formula for x , given data inputs a , b , and c .

Use variable menus in your input, and restrict the program to real-number solutions for x , or else display a message: "NO REAL SOLUTIONS." Do not allow it to yield complex solutions.

4. Suppose you wanted to graph the following equation:

$$\frac{4}{x^2 + x - 12} = y$$

Write a program that converts an input (x) into an output (y).

Use flags to prevent the program from stopping because of a zero denominator – you can't divide by zero (hint: refer to pages 280-282 of your Owners' Manual for a summary of what each of the flags do). If your program encounters such a situation, have it display the message, "ASYMPTOTE AT X=(fill in the current X-value that caused the problem)."

5. Write a program, **ORG**, that creates a Programmable Menu containing two items: **MATH** and **SCIEN**. When you press **MATH** you should branch to another Programmable Menu containing **ALGEB**, **GEOM**, **CALCU**, and **DIFEQ**. When you press **SCIEN**, you should branch to another Programmable Menu containing **CHEMI**, **PHYSI**, and **EENGI**.

Then (although later you may be able to use this *nested* set of Programmable Menus to organize your various technical formulas and programs on your HP-42S), simply program each sub-menu item to display its full title. That is, pressing **ALGEB** gives the message, "ALGEBRA"; pressing **DIFEQ** gives the message, "DIFFERENTIAL EQUATIONS"; pressing **EENGI** gives the message, "ELECTRICAL ENGINEERING", etc.

(Hint: Since you're creating one main menu and two sub-menus, you're really creating something like a single three-page menu.)

Programming Proficiency Proofs

1. Pressing **[R/S]** always begins running the current program at the line right after the line where the program pointer is pointing. In this case, WHATZIT would begin executing from line 09 – probably crashing or misbehaving since it may have missed out on some essential program steps.

Using **[XEQ]**, on the other hand, resets the program pointer at the the program's intended starting label, thereby insuring that you will get "all" of the program. In general, you should use **[XEQ]** if you want to begin the whole program.

2. When a **RTN** statement is encountered, it sends the program pointer back to the program step immediately following the *most recent* **XEQ** statement that the program pointer encountered. The HP-42S "remembers" (in its own system memory) the order and locations of the **XEQ**'s it encounters. Each time the program pointer encounters an **XEQ** statement, it remembers it and puts this "mental note" on the *bottom of a stack* – just as your data input into the RPN Stack goes in at the bottom (the X-register) – with the previous entries being "bumped up" above it. Then, whenever it encounters a **RTN** statement, the machine *removes and uses* the bottom (most recent) entry from the stack to send the program pointer back to below the most recent **XEQ** command. And the other entries drop down, so that the process can repeat. This "RTN stack" can hold up to 8 entries before you start to pop them off the top (i.e. lose them and crash your program).

3. The solution below gives just the listing in the display. You're adept enough with the keyboard now to know how to key in any of the steps in this program. (Right?)

00 { 148-Byte Prgm }		
01 LBL "QUAD" 1	26 +/-	
02 MVAR "A"	27 STO 01	
03 MVAR "B"	28 X<>Y	
04 MVAR "C"	29 +	
05 "INPUT COEFF"	30 STO 02	
06 I"CIENTS" 2	31 RCL 01	
07 CF 21	32 RCL 00 5	
08 RVIEW	33 -	
09 VARMENU "QUAD"	34 STO 03	
10 STOP	35 RCL "A"	
11 EXITALL	36 2	
12 FIX 02	37 x	
13 RCL "B"	38 STO÷ 02	
14 X↑2	39 STO÷ 03	
15 4	40 "X = "	
16 RCL "A" 3	41 ARCL 02	
17 x	42 I" OR X = " 6	
18 RCL "C"	43 ARCL 03	
19 x	44 PROMPT	
20 -	45 GTO "QUAD"	
21 X<0?	46 LBL 09	
22 GTO 09 4	47 "NO REAL "	
23 SQRT	48 I"SOLUTIONS"; " 7	
24 STO 00 5	49 RVIEW	
25 RCL "B"	50 .END.	

After the beginning label (Section 1), Section 2 sets the the variable menu with menu items for the three coefficients a , b , and c . It also prompts the user to "INPUT COEFFICIENTS" (notice that you must ensure that Flag 21 is clear before you assume that `AVIEW` will not halt program execution).

Section 3 then calculates the *discriminant* of the formula – the part of the formula under the radical: $(b^2 - 4ac)$, preparing to determine if real solutions exist. If the discriminant is negative, then the solutions are complex (the square root of a negative number is imaginary). Otherwise, the solutions are both real.

Section 4 tests the discriminant. If it's negative, the program jumps to `LBL 09` (Section 7) to give an appropriate error message.

If the discriminant is positive (and *only* then), the program continues on to Section 5, which finishes calculating the two solutions. The first solution is `STOred` in register 02, the second in register 03.

Section 6 gathers together an answer display in the ALPHA-register. The program stops at the `PROMPT` and the solutions are displayed. To repeat the program, you must press `[R/S]` once more and the `GTO "QUAD"` takes you back to the beginning once again, isolating this display from the `LBL 09` one that follows.

Section 7 displays the "NO REAL SOLUTIONS" message. The `LF` and two spaces are included to make a cleaner two-line display (but are cosmetic and completely optional). Notice, however, that from this output, too, a simple `[R/S]` will restart the program.

4. Here's one possible solution making use of flag 25, one of the control flags:

00 (63-Byte Prgm)	1	11 FC? 25	4
01 LBL "EQ"		12 GTO 03	
02 SF 25	2	13 RCL "X"	
03 4		14 VIEW "Y"	5
04 INPUT "X"		15 STOP	
05 X \uparrow 2		16 GTO "EQ"	
06 RCL+ "X"	3	17 LBL 03	
07 12		18 "ASYMPTOTE AT X="	
08 -		19 ARCL "X"	6
09 \div		20 CF 21	
10 STO "Y"		21 AVIEW	
		22 .END.	

After the beginning label (Section 1), Section 2 sets flag 25, which tells the calculator to ignore the first error it comes to. That is, if it encounters one, it ignores it but also *clears* flag 25.

Then section 3 takes "X" as an input, calculates and stores "Y".

Next, section 4 tests flag 25. If there has been an (ignored) error – such as division by zero – during the calculations in Section 3, flag 25 will have been cleared by this point, in which case the test at line 11 will send the program jumping to LBL 03 (Section 6) to display the asymptote message (mind the flag 21 with AVIEW!).

Thus, Section 5 displays "Y" if it is calculable and Section 6 (at LBL 03) displays what it means if "Y" is not calculable.

5. Here's one very straight-forward solution:

00 { 262-Byte Prgm }		26 LBL C	
01 LBL "ORG"		27 CLMENU	
02 LBL A		28 "CHEMI"	
03 CLMENU		29 KEY 1 XEQ 11	
04 "MATH"		30 "PHYSI"	
05 KEY 1 GTO B		31 KEY 2 XEQ 12	
06 "SCIENCE"		32 "EENGI"	3
07 KEY 2 GTO C	1	33 KEY 3 XEQ 13	
08 KEY 9 GTO 99		34 KEY 9 GTO A	
09 MENU		35 MENU	
10 STOP		36 STOP	
11 GTO A		37 GTO C	
12 LBL B		38 LBL 01	
13 CLMENU		39 "ALGEBRA"	
14 "ALGE"		40 GTO 14	
15 KEY 1 XEQ 01		41 LBL 02	
16 "GEOM"		42 "GEOMETRY"	
17 KEY 2 XEQ 02		43 GTO 14	
18 "CALCU"		44 LBL 03	
19 KEY 3 XEQ 03		45 "CALCULUS"	
20 "DIFEQ"	2	46 GTO 14	4
21 KEY 4 XEQ 04		47 LBL 04	
22 KEY 9 GTO A		48 "DIFFERENTIAL "	
23 MENU		49 F"EQUATIONS"	
24 STOP		50 GTO 14	
25 GTO B		51 LBL 11	
		52 "CHEMISTRY"	

53 GTO 14		61 LBL 14	
54 LBL 12		62 RVIEW	5
55 "PHYSICS"		63 RTN	
56 GTO 14	4	64 LBL 99	
57 LBL 13		65 CLMENU	6
58 "ELECTRICAL "		66 EXITALL	
59 F "ENGINEERING"		67 .END.	
60 GTO 14			

Section 1 (The beginning global LBL "ORG" and LBL A) sets up the first (or main) menu, containing the two choices, **MATH** and **SCIEN**. As you've done before (pages 229-231), you set up a Programmable Menu in three stages: the menu key assignments (lines 04-08), the MENU statement (line 09), and the trapping statements (lines 10-11).

Section 2 (LBL B) sets up the first sub-menu, the one for **MATH**. Notice that in line 22, the **EXIT** key (KEY 9) takes you back to the main menu. This is logical: **EXIT**ing a sub-menu should put you back at the main menu. And **EXIT**ing the main menu should put at the Stack Display – exactly what LBL 99 does in Section 6! Then Section 3 (LBL C) sets up the second sub-menu in a pattern identical to that for the first sub-menu.

Section 4 contains all of the individual "action" labels, each containing only the output ALPHA-strings for each course name and then a GTO 14, which takes you to Section 5, where the tiny output subroutine lives (mind the flag 21 with RVIEW!).

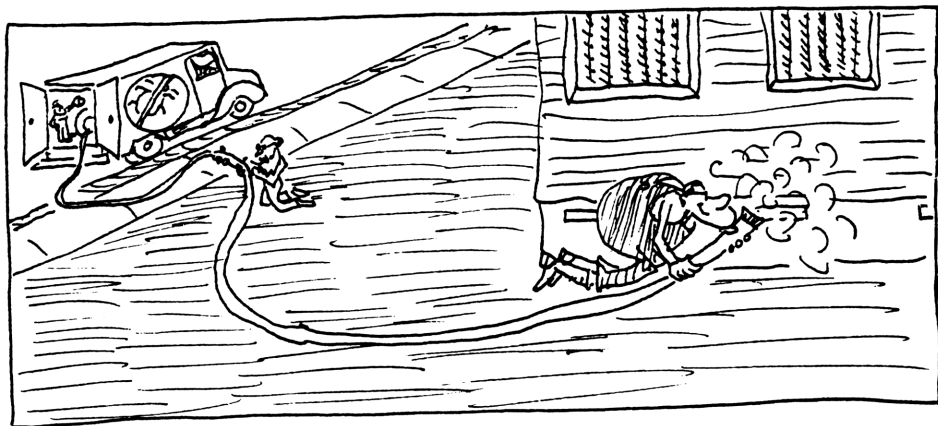
Finally, Section 6 "tidies up" before EXITing.

Programming BIG: A Case Study

Up to this point in the Course, you've had many small example programs to write, each one illustrating a new strategy, function or technique. However, none of these programs approached the size and complexity of a really big program – one that consumes a good-sized chunk of your total possible available memory.

So, to close off this Programming chapter, what you have here is an example of one of these behemoths. The idea is for you to:

- see an example of how to organize and structure complex programs;
- learn how to store larger amounts of data very compactly – an important necessity for large programs which use lots of memory;
- give you a practical review of most of the things you've already learned about programming – all together in one program;
- show you some of the real-life programming tricks and cautions needed to make a large, well-thought out, and interesting program actually run properly; this is called "de-bugging" a program.



But before you get started on this new adventure, you ought to clean out your workshop a bit. You still have some older, not-so-eternally-useful sample programs and variables lying around, cluttering up the place and occupying valuable workspace.

Do This: Clear out the programs, FIRST, CYL, NCYL, HOWDY, GREET, SPHERE, and DENSITY from your storage area. Then clear out the variables that these programs used: H, R, L, W, M, VOLUME, AREA, DEN and HI.

Solution: Press **CLEAR** twice, and choose either **CLP** or **CLV**. Now search through the resulting menu, using the **▲** and **▼** keys until the desired programs or variables are visible, then press the corresponding menu keys to clear them. Repeat this procedure until you've cleared out all of the programs and variables listed above.

When you've finished press **CATALOG MEM** and hold down the **MEM** key to see how much memory you have left. Before proceeding farther, you should have at least **3500 Bytes** available (if you still need more room, clear out the matrix variables, MATT, TAM, and MAT).

OK, now that you have some extra elbow room...

The Challenge

("Good morning, Mr./Ms. Phelps....")

Your mission, should you choose to accept it, is to modify the program MUSIC (the one with DO-RE-MI) so that you can record, name, save and play back any tune that you play on the "keyboard." Of course, the program should also let you simply "practice" on the keyboard, without recording at all – just as it does now.

The program should be flexible and convenient. It should offer you choices, prompt you with instructions, and monitor your inputs with error-trapping routines to avoid "crashes."

In recording a song, every note you press will be taken as one "beat" (one count). In 4/4 time for example, each note you press would be taken as a quarter note. To record a half note, you must simply press the same note twice in a row, etc. Of course, you'll need to add a "no-tone" to your keyboard so that you can include "rests" in your songs.

Then, when your recorded songs are played back, each beat will be played in steady, tight succession (no matter how haltingly you have "recorded" it), so that two or more identical notes played in succession will sound as much as possible like single, longer notes. Admittedly, this won't make for very sophisticated rhythms or dynamics, but the HP-42S isn't exactly capable of symphonic music anyway, since its TONE's are all the same length and you're limited to one octave of a major scale.

("Good luck... ..this page will self-destruct in 5 decades....")

Stage One: Planning

This is the most important stage of all. Without doing the work here, you run the risk of wasting lots of your valuable time doing things that you don't need, or doing them in ways that are useless later on in the program. This is one of the new aspects of large programs: it is difficult to envision them all at one glance.

The most important part of planning is what you *don't* do: You *don't* start out by writing out actual program steps or keying them into your machine – there'll be time enough for that later on. The first step, then, is to list the main tasks of your program and to consider your basic strategies. For example, the new MUSIC program will:

- Allow you to practice on the keyboard without recording – just as the current MUSIC program will. All right – visualize this as one major routine within the program, and call it the "PRACTICE" routine.
- Allow you to record as you play the notes – for later playback. Call this routine "RECORD."
- Allow you to name and store a recorded song. Call this routine "SAVE."
- Allow you to recall a stored song and prepare it for playing. Call this routine "LOAD."
- Allow you to play back the LOADED song. Call this routine "PLAY."
- Allow you to quit the program. Call this routine "EXIT."

Now, from the description of the challenge, you know that a "song" will consist of an ordered collection of tones – one for each beat – right? And you're going to need some kind of tight, efficient playback loop in order to play these beats to make the song sound as realistic as possible. But there's no way to know how many beats a song will have. And anyway, how are you going to quickly "read" and "play" TONES in a tight loop?

Well, from your loop practice programs (pages 252-259), you know you can "loop" quite efficiently with the *numbered storage registers* (whose numbers can be increased as necessary), *and an ISG loop and indirect addressing*. In fact, you can even use indirect addressing to play the TONES themselves (e.g. `TONE IND 00` will sound the tone whose number is now sitting in the register 00)! ...OK...

First Major Design Decision: To play back a recorded song, its "notes" will need to be located in the numbered storage registers.

Hmmm...then how are you going to store recorded songs? You might well have several of them. Are you going to put them all into numbered storage registers? You could do this, but the REGS matrix would get pretty big. And how and where are you going to remember the name of each song? Looks like you want to store each song into its *own variable, with its own name*, right? And obviously, the type of variable to hold an ordered collection of data is a matrix – similar to the REGS matrix: the first beat of the song is the first element, etc....All right....

Second Major Design Decision: Each song will be stored as a collection of beats within a unique, named matrix variable.

Hmm...should you rethink your First Major Design Decision? Maybe

you can play back each song with a program loop that quickly reads and "plays" the beats directly from a song's *matrix* – instead of first having to LOAD those beats into the numbered REGS matrix. Well, you *could* do that, but it might not be as fast a playback loop since you'd need more commands to sound the TONEs (you can't `TONE IND ij` to an element `ij` in a matrix). Besides, it's not that inconvenient to need to LOAD a song, since you can play only one at a time anyway.

OK – individual storage. But won't that use a lot of memory? Matrices tend to do that, you know...unless...you *compact the data*. For example, a collection of 12 beats, initially in registers 01 through 12, can be saved as either a **12 x 1** matrix of reals (one element for each beat), or as a **2 x 1** matrix of ALPHA strings, one element for each 6 beats. Real numbers and ALPHA strings use the same amount of memory, so a **2 x 1** matrix of strings uses far less memory than a **12 x 1** matrix of reals!

1	R ₀₁	can become	1	or	["115566" "5.4433"]
1	R ₀₂		1		
5	R ₀₃		5		
5	R ₀₄		5		
6	R ₀₅		6		
6	R ₀₆		6		
5	R ₀₇		5		
"."	R ₀₈		"."		
4	R ₀₉		4		
4	R ₁₀		4		
3	R ₁₁		3		
3	R ₁₂		3		

Third Major Design Decision: Each song's beats will be compacted into 6-character ALPHA strings within that song's named matrix.

One more consideration: The problem didn't mention anything about editing an existing song. What if you want to correct a mis-recording or lengthen a tune, etc.? Why didn't the challenge include this?

Probably the sophistication of the program and the resulting music doesn't warrant it. Besides, it's probably easier to re-record a song entirely than to "go in and fix" some wrong notes; how do you quickly tell the machine to insert, delete or change individual notes in an already-recorded song? Sure – you *could* program this, but it would be *a lot* more code and no more convenient than simply making a fresh recording of the whole song.

So if you can't edit a song, then notice that once a song is named and saved, the only real reason for LOADING it is to PLAY it (sure – you can rename it, but you can also do that – and clear it, too – outside the program, with standard HP-42S commands, since each song is just an ordinary matrix variable). You can't use the program to do much else with it. *As a user, you'd never need to select a routine called LOAD separately from the routine called PLAY.* So although LOAD is a separate routine in the program, this fact can be invisible to you, the user.

Fourth Major Design Decision: The user's basic choices of activities within the program will be:

PRACTICE RECORD SAVE PLAY EXIT

Clearly, you're talking about some kind of menu here – the "Main" menu of choices for the user. After all, for convenience and flexibility – both for variables and program selections – menus are hard to beat, aren't they? And the existing MUSIC program uses them already!

As you can see, the whole point of writing a program is to make it useful and convenient to the user, so in designing it, try to visualize it from the user's point of view: What will you see in the display at each point in the program? What information will you need to input – and how will you do this? In what order would you want to do things? And what mistakes might you make? ...Concentrate now, and visualize....

You know the first thing you'll encounter is that Main menu. Then what? What could you do next? Anything except SAVE, right? There's no recorded song to SAVE yet. You could PRACTICE for hours without having anything to SAVE. Only after you've RECORDED something should you be able to SAVE it, right? And you'll need to be prompted to input a name to store it under (*mental notes*).

And you shouldn't be allowed to unintentionally destroy any current unSAVED recording – by RECORDing or PLAYing (which means LOADING) another – until you've either SAVED that current one or acknowledged to the program to forget it and RECORD over it, right? (*another mental note*).

However, you *should* be allowed to PLAY the RECORDing you've just made *before* you decide to SAVE it. There's no sense in having to name and store a song that might not even be what you want, just so you can hear it to find out. So when you select PLAY, the program must offer to PLAY the current (and as yet unnamed) song *or* to SAVE it in order to LOAD and PLAY an already-named song (*yet another mental note*).

This visualization gets tricky at times, doesn't it?

Stage Two: Writing The Program

Section by Section

On the basis of your planning, Major Design Decisions and mental notes, you've now conceived of a program that can be broken into sections so that the actual programming is more manageable:

Section 0: *Initializations.* To begin, you must establish the proper settings for any necessary flags, counters, variables, etc.

Section 1: *The Main Menu.* This menu will offer the main choice of activities (PRACTICE, RECORD, SAVE, PLAY, or EXIT).

Section 2: *The PRACTICE Menu.* This will resemble your previous MUSIC program – just a Programmable Menu of the notes of the scale, sounded as the user selects them.

Section 3: *The RECORD Menu.* This will "look and feel" just like the PRACTICE Menu, but each beat will be recorded into a numbered storage register as the keyboard sounds it.

Section 4: *The musical subroutines.* Here are the actual TONE commands in their respective subroutines – just as you used them in your first MUSIC program (but with an added subroutine for a "Rest").

Section 5: *The SAVE routine.* This routine will transfer a song (the collection of beats) from the numbered registers, where it was originally stored upon recording, into a matrix of ALPHA strings – named according to the user's preference – in compact form, with six beats per string.

Section 6: *The LOAD routine.* This routine will be the opposite of the SAVE routine, "unpacking" a stored, named song-matrix into the numbered registers, ready for PLAYing.

Section 7: *The PLAY routine.* This routine will actually run through the numbered storage registers, "read" and sound the notes (beats) of the currently loaded song.

Section 8: *Utility subroutines.* In every complex program, you're likely to have certain functions or sections of code that you'll want to use more than once. In that case, it's more efficient to turn each such section into a subroutine with a LBL and RTN statement so that you can easily call it whenever you need it.

Section 9: *The EXIT routine.* Although you may use this routine only once during a program, it's important to identify and treat it separately anyway. Here you "clean up," exit menus, restore inconvenient system parameters, etc.

On these next few pages, you'll see listings and explanations for each **Section**—often breaking a **Section** down, line by line. *Some functions and techniques here will be new.* If you want to read more about these and understand more fully why they are appropriate choices here, you may want your HP-42S Owner's Manual handy also – opened to the Operation Index (somewhere around page 310).

Section 0: *Initializations*

```
00 { 697-Byte Prgm }  
01 LBL "MUSIC"  
02 FIX 00  
03 CF 00  
04 CF 01  
05 CF 21  
06 SF 26  
07 CF 27  
08 CF 29  
9 "      "  
10 ASTO "NAME"
```

No matter how good a programmer you are, it's not usually possible to foresee all the initialization steps you're going to need. So although these steps must come at the beginning of program execution, often you'll start this list with "whatever you can think of," then return to amend it later, whenever you think of something else.

What you see here, therefore, didn't just "jump out" as "immediately obvious;" these are the final results of such an amendment process. *The point is, you should always have an opening section defined for just such a purpose, so that it's easy to find and amend as you need to.*

Now, what's going on here in these initialization steps?

First, notice that you have absolutely no need for any kind of *numerical display* in this program; everything of interest to the user will appear as either a message, a menu item, or a sound. But you *do* need to format numbers as single numerals (by ARCLling) – for your compacting process when you SAVE a song – without any extra decimal places or even a decimal point. So you FIX 00 and CF 29 (flag 29 controls the presences of the decimal point and digit separators – the commas at the thousands, millions, etc.).

Next, notice that *you can use the same "keyboard" routine* both for RECORDing and PRACTICing – if you tell the machine which one you're doing. You'll use flag 00 to indicate this: When this flag is clear, you're just PRACTICing; when it's set, you're RECORDing. Start in "PRACTICE mode" – with flag 00 *clear*.

Similarly, you'll notice that you need some status indicator to help the machine remember when there's an unSAVED but RECORDED song (so it doesn't let you inadvertently RECORD over something you forgot to SAVE). That'll be flag 01, which will be set only when there is such an unSAVED song. So start with flag 01 *clear*.

Lines 05 - 07 ensure that your sound is turned on and any CUSTOM menu is turned off, and that AVIEWS don't halt program execution.

Finally, since you can have only one song currently loaded into the numbered registers, you'll probably need just one variable to remember that song's name: "NAME". And you'd better put something in there to indicate that there's no song yet: " " (a blank name).

Section 1: *The Main menu*

11 LBL A	20 KEY 3 GTO E
12 "CURRENT SONG: "	21 "SAVE"
13 ARCL "NAME"	22 KEY 4 XEQ H
14 F""	23 "PLAY"
15 AVIEW	24 KEY 6 XEQ I
16 CLMENU	25 KEY 9 GTO 99
17 "PRAC"	26 MENU
18 KEY 1 GTO B	27 STOP
19 "REC"	28 GTO A

Section 2: *The PRACTICE Menu*

29 LBL B	50 LBL D
30 CF 00	51 CLMENU
31 LBL C	52 "TI"
32 "DO↓"	53 KEY 1 XEQ 07
33 KEY 1 XEQ 01	54 "DO↑"
34 "RE"	55 KEY 2 XEQ 08
35 KEY 2 XEQ 02	56 "REST"
36 "MI"	57 KEY 6 XEQ 09
37 KEY 3 XEQ 03	58 KEY 7 GTO C
38 "FA"	59 KEY 8 GTO C
39 KEY 4 XEQ 04	60 KEY 9 GTO A
40 "SO"	61 MENU
41 KEY 5 XEQ 05	62 STOP
42 "LA"	63 GTO D
43 KEY 6 XEQ 06	
44 KEY 7 GTO D	
45 KEY 8 GTO D	
46 KEY 9 GTO A	
47 MENU	
48 STOP	
49 GTO C	

Sections 1 and 2 should look familiar to you in some form or other:

Section 1 (at **LBL A**) is the Main Menu, which you build and display just like you've seen other Programmable Menus built and displayed (recall, for example, pages 229-232): Each choice has a **KEY** associated with it (though not all **KEY**'s are used). Notice that it was logical to put the **EXIT** routine (labelled **99**) on **KEY 9** which is the **(EXIT)** key. As for the choices to **GTO** some labels (lines **18** and **20**) but **XEQ** others (lines **22** and **24**), these will be more self-explanatory as you go on. In programming for yourself, you might well begin by using all **GTO**'s and then deciding later that **XEQ**'s make more sense. That's fine.

Notice one new thing: The name of the currently recorded song will be displayed above the menu (lines **12 - 15**), helping the user to remember. Of course, when the program starts, that name will be all blanks – as you've just arranged in the initialization section.

Section 2 (at **LBL B**) is the basic "keyboard menu," where the program pointer will go if the user chooses the **PRMC** selection (**KEY 1**) from the Main menu. Here's another Programmable Menu – two pages, in fact – similar to that in the first **MUSIC** program, with three differences:

- When you want to **(EXIT)** from this menu (**KEY 9**, line **60**), you won't exit the program entirely, because this is not the "last layer of the program." Instead, you'll be sent back to the Main menu, at **LBL A**.
- There's an extra menu item – a "**REST**" – on the menu's second page.
- Lines **30 - 31**: What are they all about?

To understand those two lines, it's best to consider **Section 3**, the RECORD menu at the same time (shown opposite here)....

Remember that flag 00 is what you're using to indicate whether the basic keyboard menu is being used for PRACTICE or to RECORD. Whenever you go to that menu via LBL B, it will be for PRACTICE, and so the CF 00 *should* be encountered, right?

But notice that (among other things) the RECORD menu (at LBL E) will *set* flag 00 (at line 69), and then send the pointer to the keyboard menu *via* LBL C, *thereby missing step 30*. By providing two different *entry points* to the same routine, you control which circumstances will encounter *all* the steps of that routine rather than just a portion.

Then, once within the keyboard menu, since flag 00 might be *either* set or clear (depending on how you got to this menu), you don't want to mess with it anymore. So in *restarting* the menu loop in cases of mis-strokes or changing pages (e.g. at lines 49, 58, and 59), you don't send the pointer back to LBL B, but rather to LBL C, thus *preserving the current status* of flag 00.

So how does flag 00 actually start the RECORD process? It's nowhere in this menu is it? No – actually it's in the "sound" subroutines (01 – 09) *called by* this menu. In those routines, if flag 00 is set, *the beats will be stored as well as played*. You'll see those routines in a moment.

Section 3: *The RECORD Menu*

64 LBL E	70 SF 01
65 FS? 01	71 "NONAME"
66 XEQ 80	72 ASTO "NAME"
67 FS? 01	73 1.999
68 GTO A	74 STO 00
69 SF 00	75 GTO C

Now notice what else this RECORD Menu does: First, it tests whether an unSAVED song is currently stored in the numbered registers, by testing flag 01 (this is what you've defined flag 01 to mean, right?): If flag 01 is set, the user shouldn't be allowed to RECORD over the unSAVED song without a warning. So you XEQ 80 – an error message to let the user acknowledge the danger and decide if the current song is worth SAVing. Depending upon his/her decision, that error routine will then adjust flag 01 and RTN to line 67, *which then tests it again*. If it's *still* set, this means the user did *not* want to forget it without SAVing it, so back you go to the Main menu (LBL A).

Thus, a user can RECORD a new song *only* if flag 01 is *clear* by step 67. If so, the program then *sets* flags 00 and 01, to signal, respectively, that this is now a new RECORDing session and that as of now, there is a new and therefore unSAVED song in memory. It also gives this song the name "NONAME" to remind the user via the message at the Main menu.

Finally, before entering the keyboard routine, lines 73 - 74 start an ISG loop counter in register 00, to help store and count each beat in successive numbered registers (up to 999 beats in any song).

Section 4: *The keyboard's musical subroutines (labels 01-09)*

76 LBL 01	85 LBL 04	94 LBL 07
77 1	86 4	95 7
78 GTO F	87 GTO F	96 GTO F
79 LBL 02	88 LBL 05	97 LBL 08
80 2	89 5	98 8
81 GTO F	90 GTO F	99 GTO F
82 LBL 03	91 LBL 06	100 LBL 09
83 3	92 6	101 ". "
84 GTO F	93 GTO F	102 ASTO ST X
		103 SF 25
104 LBL F	112 RCL "REGS"	121 LBL G
105 TONE IND ST X	113 DIM?	122 ISG 00
106 FC? 00	114 X<>Y	123 RTN
107 RTN	115 1	124 "SONG LENGTH "
108 SF 25	116 +	125 F"LIMIT"
109 STO IND 00	117 X<>Y	126 TONE 0
110 FS?C 25	118 DIM "REGS"	127 AVIEW
111 GTO G	119 RCL ST Z	128 PSE
	120 STO IND 00	129 RTN

Section 4 is the collection of musical subroutines themselves. You've seen much of this code before, too: Each of the individual subroutines merely loads its appropriate TONE number (a "Rest" is a ". " *character*) into the X-register, then sends the program jumping down to LBL F, where the TONE is sounded *by indirectly addressing the X-register* at line 105.

Notice how the "Rest" routine (LBL 09) sets the *error-ignore flag* (25) so that if the program tries to TONE a ". ", it doesn't crash.

That's what happens whether you're PRACTICing or RECORDing.

But after that, if you're just PRACTICing, then, since flag 00 is clear, the test at line 106 will RTN the pointer back to the keyboard menu. Otherwise, it stays around to *store that beat's TONE value into the register indicated by the counter in register 00.*

Pretty slick, eh?

Notice the two *error traps* here:

The first trap (at line 110), again uses flag 25, the error-ignore flag. As you can see, this flag is useful in that it not only lets you ignore the first error, but it then clears itself to tell you that it has indeed encountered an error and that your one "mulligan" has therefore been "used up."

So you as a programmer know that if flag 25 has been cleared after the attempt to STO IND 00, this indicates that you've run out of registers in the REGS matrix, in which case, before being sent on to LBL G, you re-DIMension REGS with one more element (lines 112 - 118) and then carry out a successful beat-indirect-storage (lines 119 - 120).

Finally, if you've recorded a song with 1000 beats or more, you'll fail the ISG test at LBL G and thus be sent on to the error message below, which warns the user before RTNing back to the keyboard menu.

Notice also that at the end of the RECORDing session, the counter in register 00 will be left "pointing to" (i.e. its integer value will be the number of) the register *following* the register containing the last stored beat (in preparation for the next beat – which never came).

Section 5: *The SAVE routine*

130 LBL H	154 XEQ 81
131 "SAVE SONG AS..?"	155 CLA
132 R0N	156 1.006
133 PROMPT	157 ENTER
134 AOFF	158 LBL 10
135 ASTO "NAME"	159 ARCL IND 00
136 RCL 00	160 SF 02
137 IP	161 ISG ST Y
138 STO 00	162 GTO 11
139 1	163 ASTO ST X
140 -	164 STOEL
141 6	165 CF 02
142 ÷	166 I+
143 1.4	167 CLA
144 +	168 1.006
145 RND	169 ENTER
146 1	170 LBL 11
147 DIM IND "NAME"	171 ISG 00
148 INDEX IND "NAME"	172 GTO 10
149 1	173 ASTO ST X
150 STOIJ	174 FS? 02
151 RCL 00	175 STOEL
152 STOEL	176 CF 01
153 I+	177 RTN

The overall objective here is to transfer the notes of the current song from the numbered registers (where they were RECORDED, one beat per register) to a named matrix – in compacted form.

There are a lot of details and little programming maneuvers you may not have seen before, so look and think carefully....

The first thing to do is to prompt the user for the name under which this current song should be **SAVED**. This name is then stored in the variable, **"NAME"**. That's what lines **131 - 135** are all about.

Next job: Create and **DIMension** a new matrix variable by that name (lines **136 - 147**). This matrix, which will contain **ALPHA** strings, will be **n x 1**, and you need to figure out what **n** must be – i.e. how many elements this matrix must have.

Hmm... you'll need one string for every six beats in the song, right? But you can't just divide the number of beats by six; unless that division just happens to come out even (no remainder), you'll need to round *up*. In such a case, the last element in the matrix will contain a string that's not fully six characters long, right? No big deal – but you'll need to do this rounding and do it right.

And another thing: After you've saved this song (and "erased" it), suppose you want to **PLAY** it later. You'll need to somehow load it back into the numbered registers, set up a loop counter, then run through that loop, read each beat and sound it, right? (All that will be in the **LOAD** and **PLAY** routines.) *But how are you going to know then exactly how many beats were in that song?* Sure – you can guess within six, simply by looking at the dimension of the matrix it's stored in – but you need to know *exactly* to set up your **PLAY** loop properly.

Well, you *could* go through an ugly process, where you take the last element in the matrix and break down its string to figure out how long it is and therefore how many extra beats will be in the song. Or (much more simply), you could just designate the first element in the matrix as a special element – the number that tells you the song's length....

This is what you're going to do: As you observed on the bottom of page 293, when the recording is finished, register 00 will contain a perfectly good indicator of how long the song is: that register's integer portion stopped counting just beyond the end of the song! Why not just store that value as the first element in your song's matrix?

OK, then this means you'll need one extra element in the matrix for this information – this song-length number. Thus, you'll need to divide that song-length by six, round any remainder up and then add one.

How do you do this? See lines 136 - 147: You get the song length from register 00, take its Integer Portion (put this back into register 00 for a moment – you'll see why soon), then subtract one (because the count went one past the last register), then divide by 6. Now, an easy trick for rounding *up* is to add .4 and then use RND, which will round to the *nearest* integer (either up or down). You *don't* want to add .5 because this could cause a rounding up with no remainder in the division – not good. But since you need to round up and add one, just do it all at once: add 1.4 and RND....Voilà! The proper number of rows for the matrix!

Now put in a 1 for the single column in the matrix and DIMension it with the name indicated in the variable "NAME".

OK – your matrix variable is properly sized and named. Next, you must store into its first element the song-length value you've kept in register 00. To store elements into a matrix variable, you must first make that matrix the "current one" by INDEXing it – which you can do indirectly (i.e. without knowing the matrix's real name) via the value contained in the variable "NAME". An INDEXed matrix will stay INDEXed until you edit or INDEX another matrix or use EXITALL.

And once a matrix is INDEXed, you can do a great number of manipulations to it, via a pair of *index pointers* – a row pointer and a column pointer. You *position* these pointers with the `STOIJ` command and with the values in the Y- and X- registers (1 and 1 here) which represent the desired row and column pointer positions, respectively. So now you're positioned at the first row in the first (only) column, ready to store an element value, which you do by recalling register 00 and using `STOEL`.

Now you're ready to do a loop with the actual beat tone values. First, you move the matrix row pointer to the next row, using `I+`. Then you transform the value in register 00 into an `ISG` counter: "count from one to one-less-than-this-value (this transformation is done by routine 81).

Now, the challenge here is actually to construct a loop *within* a loop: You need to build a six-character string from each six beats, then store this string into the matrix, so the *inner* loop will be this string-building loop, which must complete every six times through. Its `ISG` counter (starting at 1.006) will "live" in the Y-register (the X-register is a "way-station" for the strings being transferred to the matrix, since you can't `ASTOEL` – only `STOEL`). The outer, "overall" loop is controlled by the `ISG` counter in register 00, which is the length of the song itself; when this loop is finished, the whole `SAVE` routine is finished, at which point, you clear flag 01 to indicate that the current song is now safely `SAVED`.

You're pretty good with loops by now. See if you can follow a "hypothetical song" with, say, 16 beats, through this double loop....*Notice* the temporary use of flag 02 to signal whether the inner loop has just then completed and successfully stored a full string (in which case flag 02 will be clear), or whether a final storage of a partial string is necessary upon exiting the outer loop (in which case flag 02 will still be set).

Section 6: *The LOAD routine*

178 LBL I	199 ". "
179 "PLAY WHAT SONG?"	200 ASTO "TEMP"
180 CF 23	201 CLA
181 AON	202 I+
182 PROMPT	203 RCLEL
183 AOFF	204 ARCL ST X
184 ASTO "TEMP"	205 LBL 12
185 FC? 23	206 ATOX
186 GTO J	207 X#0?
187 FS? 01	208 GTO 13
188 XEQ 80	209 CLA
189 FS? 01	210 I+
190 RTN	211 RCLEL
191 RCL "TEMP"	212 ARCL ST X
192 STO "NAME"	213 ATOX
193 INDEX IND "NAME"	214 LBL 13
194 1	215 48
195 ENTER	216 -
196 STOIJ	217 X<0?
197 RCLEL	218 RCL "TEMP"
198 XEQ 81	219 STO IND 00
	220 ISG 00
	221 GTO 12

The idea behind **Section 6** ("LOAD") is to "unpack" a previously SAVED song into the numbered registers, in preparation for PLAYing. Remember that this LOAD routine is the "invisible" first part of the PLAY routine, as you decided back on page 282.

With PLAY, as with SAVE, you must begin by prompting for the song's name (lines 179 - 183). But notice that you'll allow the user to indicate the current song simply by pressing **[R/S]** without keying in *any* name (a very useful and common technique). You accomplish this with flag

23, which is set whenever the HP-42S experiences ALPHA input.

If the user does mean to PLAY the currently recorded song, so be it: off you go to the actual PLAY routine at LBL J. If it's a different song to be played, you'd better repeat the kind of error trap (routine 80) you used in the RECORD routine – to prevent the user from accidentally erasing (LOADing right over) an unSAVED song that was meant to be "a keeper." As you know, the status of flag 01 *after* this routine will tell you of the user's intentions, so you test it again (line 189) and either abort the mission by RTNing to the Main menu (to let the user do a SAVE) or forget the current song and go on to LOAD and PLAY a new one (whose name has been saved in the variable "TEMP", just in case).

Assuming you do go on, the rest of the LOAD routine is quite straightforward, now that you've been through the logic of a SAVE routine: You INDEX the intended matrix, position its pointers at the first element, then recall this and use it to build an ISG counter for register 00.

Then it's a loop to recall successive elements out of the matrix, into the ALPHA-register, then use a function called ATOX to remove the leftmost character from the ALPHA register and send its *character value* (the characters "1"-"9" have *character values* 49-58; a "." has a character value of 46) to the X-register.* There these values are converted by subtracting 48. A negative value here indicates a ".", so you substitute the character itself, which has been stored in "TEMP".

Notice the test at line 207 to check whether the ATOX failed due to an empty ALPHA-register (i.e. that string has been exhausted). If so, lines 209 to 213 refill the ALPHA-register and then do a successful ATOX.

*These character values are found in the table on pages 288-290 of your Owner's Manual.

Section 7: *The PLAY Subroutine*

222	LBL	J	225	LBL	14
223	RCL	00	226	SF	25
224	XEQ	81	227	RCL	IND 00
			228	TONE	IND ST X
			229	ISG	00
			230	GTO	14
			231	RTN	

Here, *finally*, at **Section 7**, is the tiny (tinny?) heart of the "record player." You'll notice that there was no final RTN at the end of the LOAD subroutine, since right after you LOAD a fresh song into the numbered registers, you want to go ahead and PLAY it.

And what is this PLAY routine?

Well, first, you must make an ISG counter out of the song-length value in register 00 (remember that it was this value that was stored, as is, into the first element of the song's matrix). To do this, of course, you use that subroutine 81, as you've done before.

Then it's nothing more than a simple ISG loop, starting at LBL 14, where you recall the beat-value in each successive register and TONE that value through indirect addressing. Notice that you must also set flag 25 each time through the loop, just in case you encounter a "Rest" value, which can't be TONEd, of course.

Section 8: *Utility Subroutines*

232 LBL 80	242 LBL 81
233 "ERASE CURRENT "	243 IP
234 T"SONG?"	244 1
235 CF 23	245 -
236 AON	246 1E3
237 PROMPT	247 ÷
238 AOFF	248 1
239 FC? 23	249 +
240 CF 01	250 STO 00
241 RTN	251 RTN

These two subroutines are just small routines that you use more than once during the program. Subroutine 80 is the message that gives the user one last chance to avoid erasing an unSAVED recording. Subroutine 81 builds an ISG counter and places it in register 00.




Section 9: *The EXIT Subroutine*

252 LBL 99	255 SF 29
253 CLMENU	256 FIX 02
254 EXITALL	257 END

Try to "EXIT cleanly," – restore the calculator to a reasonably convenient state. Besides using CLMENU and EXITALL, since this program cleared flag 29 and set FIX 00 (to omit the decimal point and all decimal places) – fairly unusual conditions – it's best to "undo these" now.

Stage Three: Debugging

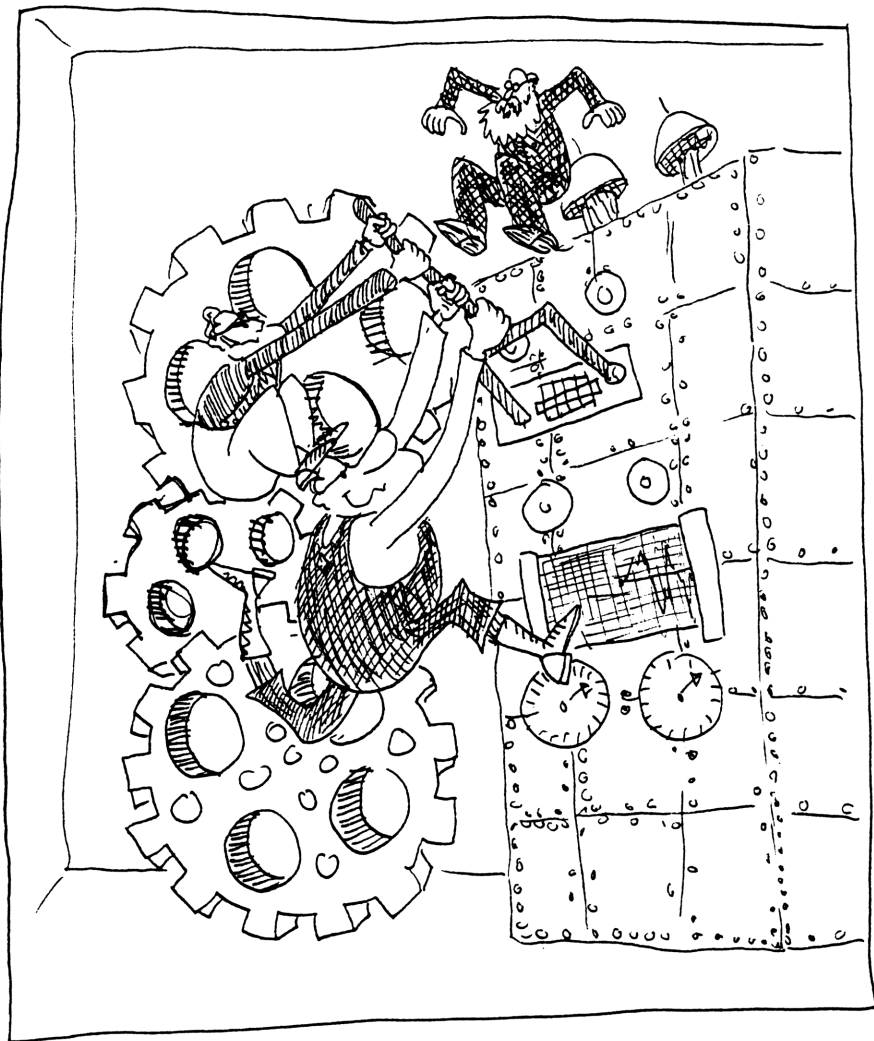
Programs don't just "write themselves;" they don't leap from your mind onto the page and into the machine by magic. And in developing them, you'll make a lot of mistakes in your reasoning, logic, and design. You'll need to some testing and "debugging" Here are some rules of thumb:

- To make things easier, key in and debug by **sections** (another major reason why you develop in **sections**).
- Plan and even write down some typical examples and their intended results and use these to test each section. If those all seem to work all right, try to purposely "crash" the program by giving strange responses or selecting menu items in unexpected (and illogical) combinations. This will highlight any needs for error traps and expose other flaws in your logic.
- Remember that you can stop a program at any time with the  key. And then you can view the contents of any register or variable – *without disturbing those contents or the Stack* – with VIEW and AVIEW. Remember also that you can do single-step execution by moving the pointer (and executing the code it traverses) one line at a time, using the  () key while in Run Mode.
- If you really get stuck, take a break or turn your attention to another section of the program. Often, you'll hit a mental block by pounding your head against some knotty (naughty?) little error, but if you turn your energies elsewhere, the mental block may clear – or your thoughts on another section may lead directly to the solution.

One more thing: No program *design* is ever perfect.

Sure, you may get the bugs out of some working version, but after you've developed it, keyed it in, debugged it, and tried it a few times, you'll almost always notice (or think of) certain things that could be better. For example, in this MUSIC program:

- Because it takes less time to generate and ignore an error than it does to generate a normal TONE, the duration of a "Rest" is only about half as long as that of a sounded note. Thus, when recording a song, you need to put in twice as many beats for rests as you do for any other note. You could experiment with the PLAY routine to try to find a way to even out the duration of a rest without unduly slowing down the PLAY loop.
- It is not immediately obvious that you need to press **[R/S]** instead of **[ENTER]** after you enter the name of a song. Since this is different than what you normally do, you may want to write an error-trapping routine that finds out when **[ENTER]** has been chosen and prompts the user to press **[R/S]** instead.
- It may seem inconsistent and unwise to have a simple **[R/S]** in one case signify "PLAY the current program" and in another "erase the current program." This could lead to unintended erasures.
- To be more "responsible," you could, in the Initializations section, actually test and remember the current display setting (from flags 28 - 29, 36 - 41, etc.), so that in the EXIT routine, you could restore exactly what the user had set his/her display to before this program was ever run, instead of arbitrarily choosing how to leave things.



SOLVING PROBLEMS: Using Your Power Tools

Congratulations! You've finished building your first customized "power tool"—that MUSIC program!

But you also have some very useful power tools already built into the machine (and after having worked so hard to make MUSIC, you can certainly appreciate these more now, eh?). By way of re-introduction (and look back to the Big Picture on page 18, if you wish):

BASE performs math in several bases and handles integer arithmetic and Boolean algebra (binary logic). This is especially useful for electrical engineers and computer programmers.


MATRIX is very useful, as you've already seen, for organizing, storing, and manipulating large collections of data at once. Matrices are used in business to help make planning and decisions, in science and engineering to do calculations that are impossible or too time-consuming to do any other way, and in statistics to perform detailed analyses of whole collections of data.


STAT performs many calculations in one- and two-variable statistics, including regression and forecasting according to any one of four curve models.

$\int f(x)$ calculates the definite integral of a function that you program. This is essential for most technical fields.

SOLVER solves any specified equation for any unknown variable, an essential ability in any kind of numerical analysis.

The BASE Menu

The functions of the  BASE Menu can be divided into three categories: its special display modes, its integer arithmetic, and its logic functions (Boolean algebra).

The most commonly used functions are the special display modes. When you first press  BASE, you'll see four mode choices grouped on this menu list: **HEXM**, **DECI**, **OCTM**, and **BINM**. Each of these displays the X-register in a different number *base*. The *base* of a numbering system is the number of unique symbols (digits) used to represent numbers.

For example, the DECimal number system is base-10; it has ten digits*— 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. All numbers are represented by combinations of those ten digits and no others.

But in base-8 (the OCTal number system) there are only *eight* digits — 0, 1, 2, 3, 4, 5, 6, and 7. So how do you write "nine" in base-8?

Well, how do you write "eleven" in base-10? By using *two consecutive digits* (11). Each digit stands for a different quantity. In base-10, the right-most "1" is the "ones" place (10^0) and the next is the "tens" place (10^1) and the next one the "hundreds" place (10^2), etc.

Similarly, in base-8, you have the "ones" (8^0) place, the "eights" (8^1) place, the "sixty-fours" (8^2) place, etc. So to write 9 in base-8 you need one "eight" and one "one": 11_8 (the subscript means the number is being expressed in a base other than base-10).

*Your ten fingers are also called "digits." This is no coincidence — how did you learn to count?

Try This: Convert 469 to base-8 (OCTal), using your HP-42S.

Solution: Key in $\boxed{4}\boxed{6}\boxed{9}$ and press $\boxed{\text{BASE}}$ (if you haven't already done so). Then $\boxed{\text{OCTM}}$. Answer: 725_8

You can check this by calculating 7 "sixty-fours" plus 2 "eights" plus 5 "ones": $7(8^2) + 2(8^1) + 5(8^0) = 469$.

Get the idea?

So, you'd expect base-16 (HEXadecimal) to have sixteen digits, right? Well, it does! The letters A-F are recruited to stand as the six extra symbols ("numerals"). And that explains the menu item $\boxed{\text{AlphF}}$: You use it to key in the "alphabetic digits" in a HEXadecimal number.

For Example: Convert 725_8 to HEXadecimal, then add $7CF_{16}$ to it. Express the answer in DECimal.

Solution: Press $\boxed{\text{HEXM}}$ to display 105, then press $\boxed{\text{AlphF}}\boxed{7}\boxed{\text{C}}\boxed{\text{F}}\boxed{+}$. The display shows 9A4, which is the answer in HEXadecimal. Finally, press $\boxed{\text{EXIT}}\boxed{\text{DECM}}$ to get the Answer: 2,468

(Again, you could check this: $725_8 = 469$ – already proven above – and $7CF_{16} = 7(16^2) + 12(16^1) + 15(16^0) = 1,999$. And $469 + 1,999 = 2,468$)

But bases *really* get interesting when you see how computers actually count "when nobody's looking:" They count in BINary (base-2), using just two digits, 0 and 1 (and you'll often hear Binary digITS called by their shortened name: BITS).

This system works just like all the others – except that there are so many fewer symbols available. The positional "place"-values in the BINary system are:

$$\begin{array}{ccccccccc} \dots(2^4)\text{'s} & (2^3)\text{'s} & (2^2)\text{'s} & (2^1)\text{'s} & (2^0)\text{'s} & & & & \\ & & & & & \text{or} & & & \\ \dots 16\text{'s} & 8\text{'s} & 4\text{'s} & 2\text{'s} & 1\text{'s} & & & & \end{array}$$

So the largest binary number you could write in five bits would be 11111_2 , which is $1(2^4) + 1(2^3) + 1(2^2) + 1(2^1) + 1(2^0)$, or 31.

The number of digits (bits) available to represent a binary number is called the *Word Size*. In the case above, for example, the Word Size was 5. Your HP-42S cannot display a number in *any* of the HEXadecimal, OCTal, or BINary modes if the BINary version would require a Word Size greater than 36. So the *range* of values displayable on your HP-42S in *any* of the HEX, OCT or BIN formats is

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$, or $2^{36} - 1$, or 68,719,476,735, with actual values extending from -34,359,738,368 to 34,359,738,367

If you ask your HP-42S to so display a value outside this range, it will instead show <Too Big>, telling you that, while this number is still *stored* properly, it cannot be *displayed* in any mode *other* than decimal.

BINary mode is also used in computers to convert to OCTal and HEXadecimal: Since BINary has an "eights" place, the number 11110000110010001101_2 can be converted into OCTal simply by regrouping the "bits" into groups of threes (right to left):

011	110	000	110	010	001	101
3	6	0	6	2	1	5

Similarly, you can convert to HEXadecimal form: Just collect the bits into groups of four and change each group into a character in HEXadecimal:

1111	0000	1100	1000	1101
F	0	C	8	D

Each OCT digit is three bits and each HEX digit is four bits!

Thus, $1111\ 0000\ 1100\ 1000\ 1101_2 = 3606215_8 = F0C8D_{16}$

(Verify this if you wish!)

Also, you know that bytes are units of memory in your calculator, right? But a *byte* is nothing more than eight bits (the equivalent of two HEX digits)! Each ALPHA character uses one byte of memory – one two-digit HEX number.

Sure enough – if you look at the Character codes in your Owner's Manual (pages 288-291) you'll see that each character has its own two-digit HEX code. There are 255 possible characters representable by 8 bits (though the HP-42S doesn't use them all).

Negative Numbers In The BASE Menu

There are some other limitations you should know about when doing base conversions on the HP-42S.

First of all, truly negative numbers are displayed *only* in DECimal mode. In the other three bases, changing the sign of a number produces something quite different.

To understand these differences, you need to focus for the moment on base-2. To represent negative values, the "sign" of a number must be represented somehow; you need to use one of the 36 available bits (the bit at the farthest left): a 0 there means a positive number and a 1 means a negative number.

For example, the number

000 000 000 000 000 000 000 000 000 011 101₂

is equivalent to 29 in base-10. Since the 36th bit is clearly a zero here (though the display doesn't show leading zeroes), the value is positive.

But how do you suppose you change the sign of this number, transforming it into its opposite – the BINary equivalent of -29?

100 000 000 000 000 000 000 000 000 011 101?


Well...not exactly....

Try It: Convert -29_{10} to BINARY.

Result: 111 111 111 111 111 111 111 111 111 111 100 011.

(Press  **SHOW** and hold it down to see the entire number).

Yoicks! Has the world gone completely wacko?

No, it's just a different form of logic. Negative numbers are represented in the  **BASE** Menu by something called the "2's-complement" of its positive binary number.

The easiest way to understand this is to envision the tape counter on a cassette recorder (most of these tape counters have only 3 digits, so imagine for a moment that you're working with a Word Size of 3 instead of 36 – the argument is the same):

Suppose you reset a tape counter to 000 and then press the rewind key so that the counter moves *backward* by one unit. You'd see the number 999 – and you'd know this is really a -1, right?

Right. Now imagine that this three-place counter has, on each of its wheels, instead of 10 symbols, only two: 0 and 1. OK, you reset this unique counter to 000 and then, again, move the tape backward by one unit. What would appear? It would show 111, wouldn't it?

In the world of binary tape counters, 111 is equivalent to -1. The *negative* of 001 ("one") is 111. The *negative* of 010 ("two"), is 110, etc.

Hold that thought. Now, notice that in the binary world, zeroes and ones are "complementary." Since there are only two possible values for any digit, the complement of a binary number is simply that number with its digits exchanged: You change the zeroes to ones and the ones to zeroes. So 001 becomes 110, 11101 becomes 00010, etc.

That's not a very startling observation, but notice that you'll get the *negative* of a binary number (as in the tape-counter analogy) if you *add 1 to its complement*: The negative of 001 is its complement, 110, plus 1, or 111! The negative of 010 is its complement, 101, plus 1, or 110.

In 36 bits, the negative of

000 000 000 000 000 000 000 000 000 000 011 101

is its "complement + 1:"

(111 111 111 111 111 111 111 111 111 111 100 010) + 1

or 111 111 111 111 111 111 111 111 111 111 100 011

— which is exactly the answer that your calculator gave you in the last example! This process of reversing each digit, then adding 1, is called taking *the 2's complement* of a number.

Then, of course, once you have a negative number in BINary format, you (and your HP-42S) can simply read its digits to get the corresponding number in OCTal and HEXadecimal:

77777777743₈ and FFFFFFFE3₁₆

Fractions And Integers In Other Bases

Now that you have some feel for how integers can be represented in various number bases...

Try This: Key in 47.35 and convert it to HEX. Convert back to DEC, then key in 47 and convert it to HEX.

Results: 47.35 converts to $2F.$ while 47 converts to $2F.$ The only difference is the period ($.$).

While you're in the **BASE** Menu and in any mode other than DECimal, all fractions are completely ignored *in the display*. The HP-42S places a period after the integer part to remind you that the number displayed value has a fractional part that is *not* being displayed.

Furthermore, while in the **BASE** Menu, the arithmetic operations (+, -, x, and ÷) *completely ignore fractions and work only on the integer portions of the numbers. This is true even in DECimal mode.*

For Example: From the **BASE** Menu, add 5.78 and 4.89.

Answer: 9.00. Yep. When in the **BASE** Menu, your calculator treats this problem as "5 plus 4," completely ignoring the fractional parts – no rounding or anything. This is *integer arithmetic*.

The **LOGIC** Menu

The third major set of features on the **BASE** Menu is in the **LOGIC** Menu. There you can manipulate BINARY numbers in a way that is even simpler than adding and subtracting.

George Boole began the computer revolution unknowingly in the middle of the 19th century when he developed a system of logic in which everything spoken or written could be considered as either true or false (or "0" or "1", if you prefer). Then he went on to define different ways in which to combine statements that are true and false, developing a "two-valued," or *binary* system of logic.

It wasn't until 100 years after George Boole invented this binary logic that the first primitive electronic computers were made. The man who really made the connection between logic and computer science was Claude Shannon. He developed the idea of "switching circuit theory," the idea being that electronic switches being ON or OFF could exactly represent Boole's system, where every assertion is either TRUE or FALSE.

Well, the rest is history: To this day, Boole's system – called Boolean Algebra, or binary math – is the actual way that your calculator (and nearly every computer) performs arithmetic.

To get an idea of how this Boolean Algebra works try the following:

Example: Perform the following Boolean operation: 37 AND 14

Solution: Press **MODE** from the **MODE** Menu. Then **3** **7** **ENTER** **1** **4** **AND**. Answer: 4.00

Which makes no sense at all until you see it at the BINary level:

When the computer does a logical operation (such as AND) with two BINary values, *it treats each pair of corresponding bits independently:*

$$\begin{array}{r} 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 100\ 101\ (37) \\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 001\ 110\ (14) \\ \hline 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 100\ (4) \end{array}$$

Think about it this way: If you have two verifiable statements, (i) "I am male," and (ii) "I have blue eyes," each of these statements is either true or false. If true, assign it a value of 1; if false, 0.

Now consider the statement, "I am male AND I have blue eyes."

This statement is true (1) only if *both* the individual statements are true (1). Mathematically, then, (1 AND 1) = 1. But (1 AND 0) = 0, (0 AND 1) = 0, and (0 AND 0) = 0, since if either individual statement is false (0), then the entire combined statement is false.

The point is, *to perform an AND on two BINary numbers, your machine performs an AND on each pair of corresponding bits, and places the result in the corresponding bit in the BINary answer.*

There are other logical operators, too: **OR**, **XOR**, and **NOT**. Using the same analogy (two verifiable statements), the logic of each of these is:

- if *at least one* (it could be both) of the individual statements is true, then the *combined OR* statement is true:

$$(1 \text{ OR } 1) = 1 \quad (1 \text{ OR } 0) = 1 \quad (0 \text{ OR } 1) = 1 \quad (0 \text{ OR } 0) = 0$$

Thus, $(37 \text{ OR } 14) = 47$.

- if one or the other (but *not both*) of the statements is true, then the *eXclusive OR* is true:

$$(1 \text{ XOR } 1) = 0 \quad (1 \text{ XOR } 0) = 1 \quad (0 \text{ XOR } 1) = 1 \quad (0 \text{ XOR } 0) = 0$$

Thus, $(37 \text{ XOR } 14) = 43$.

- NOT is simple operation that takes the complement (*not* the 2's complement – just a simple flipping of bits), replacing 1's with 0's and 0's with 1's. NOT operates on only one number at a time and has the effect of reversing the sign and subtracting one:


$$\text{Thus, } \text{NOT}(1) = -2 \quad \text{NOT}(-1) = 0 \quad \text{NOT}(37) = -38 \quad \text{NOT}(-14) = 13$$


O Brave New World, that has such operators in't...

And what good are these operators?...Well, certain electronic circuits can *simulate* them (open/closed = 1/0) – then combine in various ways to build (among other things) *arithmetic* operators!

Your HP-42S can multiply and add only because it can AND and OR!

Solving Problems In MATRIX

You've already had a good introduction to the  MATRIX Menu. You know how to use the Matrix Editor to enter and edit matrices (pages 142-147), how to use them to store data in a compact form (pages 280-281), and how to manipulate them within programs (pages 294-297).

Now, it's time to use matrices and the  MATRIX Menu to solve some problems. But before you do anything too complicated, make sure you understand the details of matrix multiplication.

Matrix Multiplication

Back on page 155, you observed the following essential rule:

The order of multiplication is important: the number of columns in the first matrix must match the number of rows in the second matrix.

For Example: Find $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A} = \begin{bmatrix} 3 & -4 & 1 \\ -1 & 2 & 7 \end{bmatrix}$
and $\mathbf{B} = \begin{bmatrix} 2 & 9 \\ -3 & 0 \\ 4 & -2 \end{bmatrix}$

Solution: $\mathbf{C} = \begin{bmatrix} 22 & 25 \\ 20 & -23 \end{bmatrix}$

Here's why: Element 1:1 of **C** = $[(3)(2) + (-4)(-3) + (1)(4)] = 22$

How so? Because this is the *sum of the products* that result when each element of *row* 1 of matrix **A** is multiplied by the corresponding elements of *column* 1 in matrix **B**.

And element 1:2 of **C** would be the *sum of the products* resulting when the elements of *row* 1 of **A** are multiplied by the corresponding elements of *column* 2 of **B**: $[(3)(9) + (-4)(0) + (1)(-2)] = 25$

Get the idea? Ah, but why do any of it by hand with an HP-42S handy?

Go For It: DIMension and enter **A**: **MATRIX** **2** **ENTER** **3** **▼** **DIM**
ALPHA **A** **ALPHA** **EDITN** **⏏** **3** **→** **4** **+/-** **→** **1** **→**
1 **+/-** **→** **2** **→** **7** **→** **EXIT**.

Now for **B**: **3** **ENTER** **2** **DIM** **ALPHA** **B** **ALPHA** **EDITN** **⏏** **2**
→ **9** **→** **3** **+/-** **→** **0** **→** **4** **→** **2** **+/-** **→** **EXIT**.

Now, to multiply **AB**, the *first* matrix (**A** here) must be in the Y-register and the *second* matrix in the X-register:

RCL **⏏** **RCL** **⏏**. Now **⊗**. You'll find the solution matrix in the Matrix Editor (**MATRIX** **EDIT**, etc.).

From the above definition of matrix multiplication, you can see why the order might matter: To reverse the matrices would mean combining totally different sets of rows and columns, wouldn't it? (Use the HP-42S to find **BA**: **RCL** **⏏** **RCL** **⏏** **⊗** **MATRIX** **EDIT**)

Simultaneous Equations

One of the most common and most powerful uses for matrices is to solve systems of linear equations. For example suppose you want to find x , y , and z such that these equations are *all* true, *simultaneously*:

$$4x - 3y + 2z = 40$$

$$5x + 9y - 7z = 47$$

$$9x + 8y - 3z = 97$$

To begin with, you need to be able to rewrite the problem so that it becomes *one* equation involving matrices. To do this, notice that there are three kinds of numbers in these equations – the coefficients of the variables, the variables themselves, and the right-hand side constants (40, 47, and 97). Each of these groups can be placed in its own matrix:

$$\mathbf{A} = \begin{bmatrix} 4 & -3 & 2 \\ 5 & 9 & -7 \\ 9 & 8 & -3 \end{bmatrix}$$

Coefficients

$$\mathbf{X} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Variables

$$\mathbf{B} = \begin{bmatrix} 40 \\ 47 \\ 97 \end{bmatrix}$$

Constants ("answers")

Notice that if you follow the rules of matrix multiplication that you've just reviewed, when you find \mathbf{AX} , you'll get this matrix:

$$\mathbf{AX} = \begin{bmatrix} 4x - 3y + 2z \\ 5x + 9y - 7z \\ 9x + 8y - 3z \end{bmatrix}$$

Aha! Comparing this matrix to the system of equations above, it's obvious that another way to write those equations is: $\mathbf{AX} = \mathbf{B}$

Thus the three equations of three variables have thus been reduced to *one* (matrix) equation with *one* (matrix) variable, \mathbf{X} !

Well, all this was discovered a long time ago. And now, to solve for the solution matrix, **X**, on paper, you'd probably use techniques named after the dead mathematicians who did the discovering (Gaussian elimination or Cramer's Rule). But can't you just hear the HP-42S snorting and pawing the ground, eager to do battle with it?)

Charge! Press **MATRIX** **SIM2** and key in **03**. The machine will automatically create and dimension three matrices: **MATA** (the *square* coefficient matrix), **MATB** (the *one-column* constant matrix), and **MATX** (the *one-column* variable matrix). With three unknowns, **MATA** will be **3 x 3**; **MATB** and **MATX** will be **3 x 1**.*

So now you'll see a menu with those matrices, **MATA**, **MATB**, and **MATX**. The idea now is to fill in the values in the known matrices (**A** and **B**) and solve for the unknown matrix (**X**). The keystrokes for the example problem would be:

MATH **4** **→** **3** **+/-** **→** **2** **→** **5** **→** **9** **→** **7** **+/-** **→**
9 **→** **8** **→** **3** **+/-** **→** **EXIT** **MATB** **4** **0** **→** **4** **7** **→**
9 **7** **→** **EXIT** **MATH**...(you'll be sent right to the Matrix Editor to see the solution)...

Victory: $\text{MATX} = \begin{bmatrix} 10 \\ 2 \\ 3 \end{bmatrix}$ That is, $x = 10$, $y = 2$, and $z = 3$

*As you may or may not remember, you must *always* have the same number of equations as variables if you hope to get an answer. This means that you can solve a system of three equations in three variables, or a system of five equations in five variables, etc.

Ready to try one on your own?

Solve:

$$\begin{array}{rcl} a + b + c + d + e & = & -1 \\ 3a - 2b - 2c + 3d + 2e & = & 13 \\ 3c + 4d - 4e & = & 7 \\ 5a - 4b & + & e = 30 \\ c & - & 2e = 3 \end{array}$$

Solution: First, MATA is a 5×5 matrix (notice how zeroes have been filled in where there are no coefficients otherwise):

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 3 & -2 & -2 & 3 & 2 \\ 0 & 0 & 3 & 4 & -4 \\ 5 & -4 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -2 \end{bmatrix}$$

MATB is the **5 x 1** matrix:

$$\begin{bmatrix} -1 \\ 13 \\ 7 \\ 30 \\ 3 \end{bmatrix}$$

And the solution matrix, MATX, is:



$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}$$

Starting with **MATRIX** **SIM** **05**, proceed as on page 320
MATH.... $a = 3$, $b = -4$, $c = 1$, $d = 1.213 \times 10^{-12}$ (nearly 0^*), $e = -1$

Now, aren't you glad that your calculator did that for you?

*When you substitute the answer back into any of the equations, it becomes obvious that d must, in fact, be exactly zero. The difference is due to the rounding done during calculation.

The STAT Menu

The  STAT Menu is a power tool you'll be studying more closely than you did the  MATRIX Menu, since this is the first look you've had at it.*

First Lesson: Some of the numbered registers (i.e. elements in the built-in REGS matrix) are used as *STATistical registers*. They are normally registers R_{11} through R_{23} (thirteen in all) – unless you specifically reassign them.

Here's are the STATistical registers in their standard "locations:"

Σx	R_{11}
Σx^2	R_{12}
Σy	R_{13}
Σy^2	R_{14}
Σxy	R_{15}
n	R_{16}

$\Sigma \ln x$	R_{17}
$\Sigma (\ln x)^2$	R_{18}
$\Sigma \ln y$	R_{19}
$\Sigma (\ln y)^2$	R_{20}
$\Sigma \ln x \ln y$	R_{21}
$\Sigma x \ln y$	R_{22}
$\Sigma y \ln x$	R_{23}

Second Lesson: These registers differ from other numbered registers because they *automatically change their contents* after certain operations; you don't need to refer to ("address") these registers directly to change the data being accumulated in them.

And what operations cause these effects? Just two: $\Sigma+$ and $\Sigma-$.

*Unless, of course, you've already mastered the entire Menu on your own. If so, skip to page 350.

But before you start messing with Σ^+ and Σ^- , you ought to spend a moment considering the basic ideas behind statistics. For in no other area is your calculator capable of generating more garbage than in the Σ^+ Menu. A little knowledge here can be a *very* misleading thing.

What Are Statistics, Anyway?

To put it as directly as possible: "Statistics are calculation methods in which you use *collections* of numeric observations to detect patterns or correspondences among real-world events or phenomena." In other words, statistics are used to *discover* what's *important* or *consistent* about a collection of otherwise confusing observations.

For example, everybody's familiar with the idea of an *average*. If you want to predict the amount of rain that's going to fall this year, one good way is to add up all the rainfall amounts for the past few years and then divide by the number of years in your sample.

In other words, you're doing this calculation: $\frac{\sum x}{n}$

where each individual x is the rainfall amount for one of those past years, \sum means to *sum* all the individual x 's together, and n is the number of years of data (the number of x 's) that you're using.

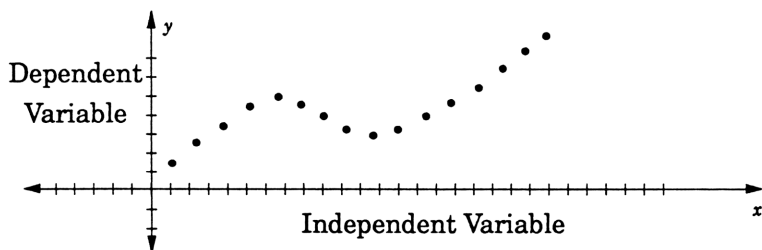
Thus, an average is one simple kind of statistic. And this begins to explain two of the statistical registers in the collection (n and Σx): Those registers are where you store the *number* of data (n) and the *sum* of those data (Σx), respectively. Then it's easy to find the average: just divide the contents of R_{11} by the contents of register R_{16} , right?

But what about all of those other registers? Why would you need $\sum x^2$? or any "y-data" (not to mention $\ln y$)?

Well, those accumulated quantities are necessary also – for many really interesting and useful tests about your accumulated data.

Suppose, for example, that someone suggests that your annual amount of rainfall *depends* on the average daily temperature for that year. How could you test this? Clearly, you now have *two* sets of observations (rainfall and temperature), the one supposedly depending on the other.

It's just like the graph of a mathematical function, where each y-value (the *height* of any point on the curve) is a function of (i.e. it *depends upon*) the corresponding x-value (the horizontal position of that point):



This is the purpose of *two*-variable statistics – to analyze relationships between an independent variable (the *x*-data) and an allegedly dependent variable (the *y*-data).

And this is why you have those other statistical registers on your HP-42S: It can do either one *or* two-variable statistics – and the formulas require not just $\sum x$ and $\sum y$ but also $\sum x^2$, $\sum y^2$, $\sum xy$, $\sum \ln x$, $\sum \ln y$, etc. So as you key in collections of observations – your statistical data – the machine will keep running tallies of *all* these different quantities.

Entering Some Statistical Data

Before you start accumulating and analyzing a new set of statistical data, you should always make sure that your STATistical registers are cleared from any previous data.

To do this, select the **■ CLEAR** Menu, then press **CLX** to clear all of the statistical registers (no matter where they're located!)....Now, FIX 00 and then...

Example: Suppose that you've owned a small bookstore for the past 20 years. The last 11 years' sales figures are:

<u>Year</u>	<u>Sales</u>
1978	\$170,000
1979	220,000
1980	215,000
1981	240,000
1982	145,000
1983	190,000
1984	230,000
1985	240,000
1986	245,000
1987	180,000
1988	270,000

You want to use the statistical capabilities of your HP-42S to tell you some things about this sales history. How do you do this?

Solution: The idea is to key in all these values *as statistical data*. When you do so, your HP-42S will *not* remember the individual values themselves, but it *will* accumulate* them (in the STAT registers, of course) in the form of those various *sums*, from which it can then calculate all sorts of useful statistics for you.

So here you go: Press $\boxed{1}\boxed{7}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$.

What happens? The X-register now shows a 1., telling you how many data points you've entered. It does this simply by recalling the current (updated) value in *n*-register (R_{16} , if located normally). This makes sense, right? That's what the statistical number, *n*, is *supposed* to measure—the number of data points accumulated so far.

So keep going:

$\boxed{2}\boxed{2}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{2}\boxed{1}\boxed{5}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{2}\boxed{4}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{1}\boxed{4}\boxed{5}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{1}\boxed{9}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{2}\boxed{3}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{2}\boxed{4}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{2}\boxed{4}\boxed{5}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{1}\boxed{8}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$
$\boxed{2}\boxed{7}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$

*It's better to use this term, "accumulate," rather than "store." It's not quite accurate to say that you "store" your data in the STAT registers. After all, once you enter them with the $\boxed{\Sigma+}$ key, you *cannot retrieve them again as individual values*; it's only their *cumulative sums* that are stored.

As you do this, keep in mind what's actually going on inside your calculator: Every time you press $\boxed{\Sigma+}$, the machine looks for *two* values – one in the X-register and one in the Y-register.*

Then it adds each x -value to the Σx register, and the *square* of that value to the Σx^2 register. It does the same for the y -data, and then adds the *product* of the x -value and the y -value to the Σxy register. It then proceeds to calculate the natural logs of x and y (the LN function) and accumulate those respective sums, squares, and products into their appropriate statistical registers.

Finally, it increments the n -register by one, to count the total number of data that you have now accumulated in this way – and recalls this new n -value to the X-register to help you keep your place!

All this – every time you press that one little $\boxed{\Sigma+}$ key!

Thus, if you've done everything right, at this point you should see 11. displayed in the X-register, since you now have 11 years' worth of sales data accumulated in the STAT registers.

*If you're working only with one-variable statistics ("x-data") the machine will still accumulate whatever meaningless numbers it happens to find in the Y-register, but this is harmless exercise and will not affect the results of your x -analyses.


Checking and Editing Your Data

...Hmm... but what if you're not sure that you've done everything right (or, what if you're *sure* that you've *not* done everything right)?

The first clue that you may have messed up would be that the X-register didn't end up with 11. in it (if not, then just start over on page 326). But assuming you got *that* much right, how *else* might you check the accuracy of your entered data?

Try This: Enter the  STAT Menu. You should see:



Now press  SUM and you should get 2,345,000. – if you've correctly keyed in your data. This is your 11-year total sales revenue. If you see this, then you can probably assume that everything's all right. If not, then go back and start again at the top of page 326.

This is one of the drawbacks to *cumulative* statistics (as opposed to storing the data values themselves): If you've mis-entered something, there's not much to do except to start over – unless you know exactly *how* you went wrong.

Even if it's all correct, what if you simply want to *amend or edit* the data list (add or subtract something, replace a figure, etc.)?

"Uh.....hmmm...can that even be done? How can you change individual values that aren't even stored as such?" Well, it's not obvious, but you *can* – by "backing out" ("un-entering") any values you want to delete, then entering any new or additional values.

Like This: Suppose you just discovered an accounting error for the years 1981-1984. The sales values for those years should actually be (respectively): \$236,000, \$154,500, \$209,000 and \$223,000. How would you edit your statistical accumulation to reflect these recent findings?

No Sweat: First, use $\Sigma-$ key to "un-enter" your sales for the years 1981 through 1984 (you're literally *subtracting* from each of the statistical registers the same amounts that the $\Sigma+$ key originally added – effectively erasing those data values from your cumulative totals):

240000 $\Sigma-$ 145000 $\Sigma-$ 190000
 $\Sigma-$ 230000 $\Sigma-$

(Notice how the value in the X-register confirms that you are indeed subtracting data values.)

Now add in the new figures (and the order never matters, does it?): 236000 $\Sigma+$ 154500 $\Sigma+$ 209000 $\Sigma+$ 223000 $\Sigma+$. (If you did all this correctly, there should be 11. once again sitting in the X-register, and the **SUM** value should be 2,362,500.).

Just Plain Mean

All right – now that you have the corrected set of sales figures all accumulated properly into the STATistical register of your HP-42S, what do you want to know about your income?

Might as well start with something simple....

F'rinstance: What was your average (also known as your *mean*) yearly sales over, say, the last ten years?

Solution: First you need to adjust your data. Currently, it has data for the last *eleven* years. To eliminate the oldest year's data (that for 1978), key in $\boxed{1}\boxed{7}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma-}$.

Press $\boxed{\text{MENU}}$ (from the $\boxed{\text{STAT}}$ Menu, of course). This calculates both the means of both the x -values and the y -values, and places each in its appropriate stack register (in this case, of course, you're concerned only with the X -register, because you only have one set of data).

Answer: 219,250.

Not bad.

A Moving Average

Time Flies: Now suppose it's the end of 1989 – and you're curious how this year's sales (\$234,600) affected your ten-year *moving* average. That is, you're always concerned with the *most recent* ten-year period, so you need to do your editing trick on the STAT registers once again – subtracting your 1979 sales and adding your 1989 sales – and *then* find the average of *those* ten values.

The keystrokes? By now these ought to be...

Old Hat: Press $\boxed{2}\boxed{2}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma-}$, then $\boxed{2}\boxed{3}\boxed{4}\boxed{6}\boxed{0}\boxed{0}\boxed{0}\boxed{\Sigma+}$ (notice that you can do this right at the $\boxed{\text{STAT}}$ Menu).

OK, now you've just updated the data (i.e. you've moved the average forward) – and you can tell from the *n*-value in the X-register that you still have ten values, so that checks.

Now find that average: press $\boxed{\text{MEMN}}$.

Result: 220,710.

Does this answer make sense?

Sure it does: You deleted a value of \$220,000 and replaced it with a value of \$234,600. You'd therefore expect the 10-year average to go up – and it did.

A Weighted Average

All right, now you've seen a plain old, garden-variety, vanilla-flavored average – and one that "moves" in time (a moving average). But what's a "weighted average?"

Try This: Suppose that your *monthly* sales revenue for 1989 went something like this (and this adds up to \$234,600, which is what you used for your 1989 *yearly* total):

January	\$ 29,000
February	10,500
March	17,200
April	15,500
May	10,500
June	9,700
July	9,700
August	15,500
September	34,500
October	15,500
November	29,000
December	38,000

What was your *monthly* average income?

Strategy: You *could* just key all this into the STATistics register and find the mean of x , exactly as you've done before. Or, you could observe that since there are *repeated* values here, it might be easier to use a *weighted mean* (**WMN**)).

Compare: First, do it the "old-fashioned" way: Press **CLEAR** **CLΣ**
2**9****0****0****0****Σ+** **1****0****5****0****0****Σ+** **1****7****2****0****0****Σ+**
1**5****5****0****0****Σ+** **1****0****5****0****0****Σ+** **9****7****0****0****Σ+** **9****7****0****0****Σ+**
1**5****5****0****0****Σ+** **3****4****5****0****0****Σ+** **1****5****5****0****0****Σ+**
3**8****0****0****0****Σ+**. Then **STAT** **MEAN**. Result: 19,550.

Yes, But: Since *the order of data entry never matters*, use a weighted mean to key in each different *value* just once – but note also *how often that value occurs in your collection*:

<u>Monthly Sales</u>	<u>Frequency Of Occurrence</u>
9,700	2
10,500	2
15,500	3
17,200	1
29,000	2
34,500	1
38,000	1

To "tell this" to your HP-42S, you key in *two* numbers for each item. The first (the *y*-value) is the frequency (or *weight*); the second (the *x*-value) is the sales value itself:

CLEAR **CLΣ** **2****ENTER** **9****7****0****0****Σ+** **2****ENTER** **1****0****5****0****0****Σ+**
3**ENTER** **1****5****5****0****0****Σ+** **1****ENTER** **1****7****2****0****0****Σ+**
2**ENTER** **2****9****0****0****0****Σ+** **1****ENTER** **3****4****5****0****0****Σ+**
1**ENTER** **3****8****0****0****0****Σ+** **MMN** Result: 19,550
 Look familiar?

So that's a weighted average: For every x -value you key in, you must also specify (with a corresponding y -value) how "heavily" that x -value should be "weighted" – i.e., how many times it can be considered to have "occurred" in this data set. ...Get the idea? Want to do another?

Sure: A chemist knows that any sample of "pure" copper she uses is actually composed of two isotopes (varieties that differ slightly in mass): ^{63}Cu (62.93 g/mol) and ^{65}Cu (64.93 g/mol). She knows that about 69.2% of all copper atoms are ^{63}Cu and 30.8% are ^{65}Cu . She needs to use a weighted mean to find the "average" mass to assume for her sample of copper.

Hmm: She has a data set that looks like this:

<u>Isotope</u>	<u>Mass (x)</u>	<u>Frequency of Occurrence (y)</u>
^{63}Cu	62.93 g/mol	69.2%
^{65}Cu	64.93 g/mol	30.8%

Now, to calculate the weighted mean:

DISP **FIX** **02** for a more meaningful display setting, then
CLEAR **FLC** **69.2** **ENTER** **62.93** **Σ^+** **30.8** **ENTER**
64.93 **Σ^+** **MIN**. Result: 63.55 g/mol.

This means that the chemist can safely *assume* that the best mass to use for any random sample of copper would be 63.55 g/mol – the *weighted average* of the masses of the two isotopes.

Deviating From The Average

Of course, any average can tell you only a certain amount about your data. To say that you "sold a monthly average of \$19,550 worth of books in 1989" makes it sound as if you could count on that level of sales every month, steady and predictable.

Sure, it *might* have happened that way. Or you *might* have sold the entire year's total (\$234,600) during the Christmas season rush – and sold nothing the rest of the year.

The point is, you *can't tell* merely by looking at the average, because no matter how uneven and irregular your sales actually were, as long as they add up to \$234,600 for the year, your monthly average will always calculate out to be the same number. An average just doesn't tell you anything about the *distribution* of your values.

Well, after a quick glance at your actual monthly income history for the year, you *know* what your sales distribution was: It wasn't smooth-and-regular, nor was it "all-in-one-rush." In the summer months, sales were slow (everybody was at the beach), while in the fall and winter, your book sales were brisk – accelerating as Christmas approached.

So exactly *how* even or uneven was it? How can you measure this from a set of values?

That's one of the uses of the *standard deviation* statistic – to measure how widely dispersed or varied from the average your actual data values are.

To get the idea,

Try This: Key a set of *identical* values into the STATistical registers, say, four 4's: **CLEAR** **CLΣ** **4** **Σ+** **4** **Σ+** **4** **Σ+** **4** **Σ+**

What's the average of this set of values? It's 4., of course (verify it with **MΣN**, if you wish).

And what's the *standard deviation* – the measure of how far a "typical value" tends to vary from this average?

To find out, press **SDΣV**. Result: 0.

Since every value is an average value, there's no variation from that average; the standard deviation is zero.

Now This: Key in these values to the STAT registers: 11, 0, 1, 4 (press **CLEAR** **CLΣ** **1** **1** **Σ+** **0** **Σ+** **1** **Σ+** **4** **Σ+**).

Find the average: **MΣN**. Result: 4. – same as before.

But now find the *standard deviation* of this set of data: **SDΣV**. Result: 5.

Compared to the average itself, this is a large deviation – telling you that the actual values vary quite widely from that average.

This is the way to measure how "even" or "uneven" your book sales revenues were over the year 1989: Take the standard deviation of your monthly sales levels for that year....

Like So: (Unfortunately, because you've since entered other statistical data, you'll now need to re-key in the twelve values again; see page 332 if you want to review the list.)

CLEAR **CLX** 29000 **Σ+** 10500 **Σ+** 17200 **Σ+**
15500 **Σ+** 10500 **Σ+** 9700 **Σ+** 9700 **Σ+**
15500 **Σ+** 34500 **Σ+** 15500 **Σ+**
29000 **Σ+** 38000 **Σ+**.

Just to refresh your memory, find the average monthly sales again (press **MEMN**). Result: 19,550.00.

Now, to test out the "evenness" of the sales, press **SDEV**.
Result: 10,247.35

That's quite a large deviation in comparison to the average itself – so it was quite an up-and-down year for your store, sales-wise.

And if you were to compare this to similar calculations for other years, you could tell immediately in which years you had smoother, more uniform sales – the years with smaller standard deviations!

Now, to be more exact, there are *two kinds* of standard deviations – and they mean different things, because they use different formulas.

The formula programmed into your HP-42S assumes that your data is really a *sample* of a much larger population – as if you were catching fish to get an idea of the typical weight of any such fish in the ocean. This is useful, obviously, when you can't actually catch and weigh every fish in the ocean.

But for other situations (as with your monthly sales figures for 1989), you *can* actually analyze all the possible data – an entire *population*. And the standard deviation formula for an entire *population* is slightly different than for a sample* (and the *population* formula is *not* programmed into your HP-42S). But you *can* get your HP-42S to calculate the population standard deviation – by adjusting your data a tad.

Like This: To get the population (true) standard deviation for your 1989 monthly income, you add one extra entry to your STAT accumulation – the *average* for the actual data. So you'd first calculate this average (press **MEMN**), then enter it as a "13th value:" **($\Sigma+$)**. Now you can use the built-in formula, **SDEV**, to find σ ... Result: 9,811.

This is somewhat *less* than the sample standard deviation, which is usually the case.

*Statisticians use different symbols for sample and population standard deviations. Sample standard deviation is abbreviated as *s*. Population standard deviation is abbreviated as σ .

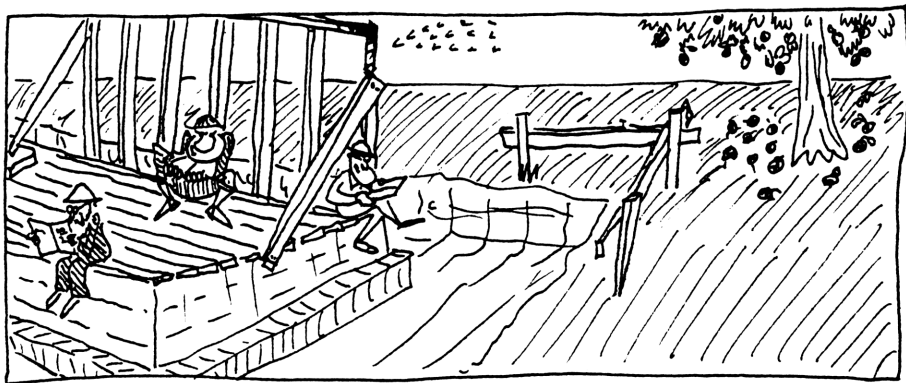
Two-Variable Statistics

Up to now, the information you've gathered from your bookstore data has involved only *one* variable – the book sales figures themselves. You now know the yearly and monthly averages of recent sales, and you know that your monthly sales vary considerably from their average.

But do you have any idea *why* it varied like that?

"Sure – because more customers come into bookstores in the fall and winter than in the spring and summer."

OK, but why? Does the average person suddenly become more interested in books in September?



Is that when everyone can afford the time or the money to shop for books? Would it have anything to do with the weather – the amount of rainfall or the temperature outside?

Well, any of these ideas are possible explanations. But how would you go about testing whether or not a particular explanation is a good one?

Getting Trendy: Regression and Estimation

Here's where you really start to use two-variable statistics to help you answer questions and test possibilities. And one very useful toolset is already built into your HP-42S: regression and estimation.

Regression is a general term in statistics for finding a mathematical relationship (a *dependence*) between two (or more) variables. If you think of the two variables as x and y , then regression is the process of finding the equation that is transforming each x datum into a y datum.

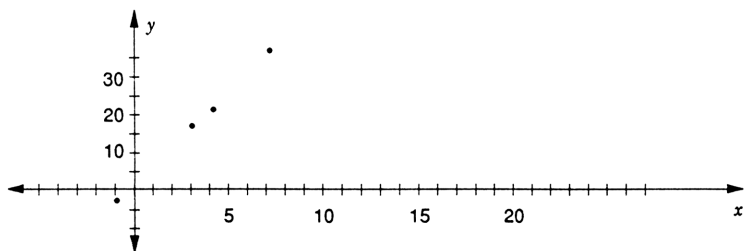
Take, for example, the equation, $y = 5x + 2$. This formula transforms each x into a y by first multiplying the x by 5 and then adding 2 to it.

But that's when you know the equation already. Regression analysis doesn't begin with an equation. It begins with two sets of data, the x -values and the y -values, and attempts to find the *one* equation that comes the closest to turning each of the x -values into their corresponding y -values. In the simple example above, regression would begin with a set of data such as:

x	y
3	17
4	22
7	37
-1	-3

and would then figure out (mathematically, somehow) that if each x were multiplied by 5 and then incremented by 2, the result would be the corresponding y -value.

To solve such a tricky problem, what your calculator does, essentially, is plot the x - and y - values on an imaginary graph, like the one below:



It then "draws" the *best-fitting curve* – the curve that gets as close as possible to as many of the data points as it can. Then the equation it wants will be the equation of that curve.

In the simple example graphed above this "curve" is actually a straight-line (indeed, when regression is restricted to finding the best-fitting *line* for a set of data, it is called *linear regression*). But whether or not the curve is linear, you can often find that curve, with the help of a very "smart" item in the **STAT** Menu: **CFIT**, or Curve FITting:

Do This: Enter the data from the example on the previous page into the STAT registers, then press **CFIT**.

Here's How: **CLEAR** **CLX** **1** **7** **ENTER** **3** **Σ+** **2** **2** **ENTER** **4** **Σ+**
3 **7** **ENTER** **7** **Σ+** **3** **+/-** **ENTER** **1** **+/-** **Σ+** **CFIT**.

Remember that you must enter the y -value first, then key in the x -value.

The display should now look like this:



This new sub-Menu can be used to find the "equation" that is the solution to a regression problem.

Try It: Use your HP-42S to find the regression "equation," assuming that the best-fitting curve is a line.

Like So: First, you must tell the calculator your assumption about the *shape* of the best-fitting curve. To do this, choose **MODL**. This selects one of the four MODEL assumptions that your calculator can work with. You'll probably see LINear Function selected (**LINF**) already; if not, select it now (the other three MODELS are LOGarithmic Function, EXPonential Function, and PoWeR Function, which you'll see shortly).

Now, remember that the general equation for a line is

$$y = mx + b$$

where **m** is the *slope* of the line and **b** is the *y-intercept*. Your calculator will use this notation convention, also.

So, **EXIT** back to the **CFIT** sub-Menu and choose **SLOPE**. Result: 5. Then choose **YINT**. Result: 2.

Thus the best-fitting equation of this linear regression is: $y = 5x + 2$

Which is exactly what you knew it should be. Remember that the number, m , is called the *slope* of the line, because it determines its "steepness" as it appears on the graph. For every 1 unit you change the x -value, you change the y -value by " m " units. So the higher the m -value, the steeper the slope of the line. And if m is negative, the line slopes "downhill" to the right; if m is positive, the line slopes "uphill" to the right (as it does in the last example).

As for b , what y -value do you get when x is zero?... $y = b$, right?

But x is zero *only along the y -axis*, so that point on the line would also be on the y -axis, and thus the line would cross the y -axis at that point. So b is called the *y -intercept* value – because this is the value (the "height") at which the line "intercepts" the y -axis.

The "slope" and " y -intercept" are also key values in MODEls other than LINear, although their interpretation isn't quite so simple as in the LINear case. The equations for the other three options are:

- Logarithmic: $y = b + m(\ln x)$
- Exponential: $y = be^{mx}$ or $\ln y = \ln b + mx$
- Power: $y = bx^m$ or $\ln y = \ln b + m(\ln x)$

Using the STAT registers (including the registers containing logarithmic data), your HP-42S can find a best-fitting equation using any of these models – you just select the MODEl most likely to "fit" your data.

OK, you've been through an ideal example with that trivial little linear equation. What about the real world – your sales data, for example?

You need to decide on a second variable for your data. Didn't you say you might have some ideas as to why your sales figures vary from year to year – some other factor – some *variable* condition upon which your sales seems to *depend*?

What's your best guess? Christmas? Well, it probably has an effect, but it isn't exactly a variable – it comes around every year.

Weather?...It's possible: People read more (it's an indoor sport) as the weather gets bad. But what *measurable* aspect of weather do you think matters most? Temperature?...Rainfall?...Hmm... a rainy spell in the summertime sends people indoors to read, even when the temperature remains relatively high. Rainfall might be a good variable to test.

First, you'd better state your theory (the one you're about to test statistically) in so many words, just so the folks at home know what it is you're trying to prove:

'The amount of annual book sales over the past ten years (1980-1989) depended upon the amount of rainfall in the vicinity of my bookstore. The rainer it was during any particular year, the more book sales I had that year.'

OK, now when you say "depended upon," you're saying that the rain and nothing else was by-and-large responsible for your varying book sales. You're saying that other factors (bus strikes, the competition's summer sale, heat waves, MTV, tax audits, cash crises, the school schedule, etc.) didn't *really* cause these variations.

Sure, those other factors may have caused small fluctuations in your sales, but you're claiming here that they were insignificant compared to the governing condition – the *independent variable* – rainy weather.

Fine. And now what do you mean by the phrase, "the rainier it was, the more sales I had?" Do you mean that if the rainfall doubled, your sales doubled? Tripled? If the rainfall is an inch more in any given year, do you generate \$10,000 more in sales? What?

Well, that's a tough one. You've only just now suggested that there is *any* kind of numeric connection – a *correlation* – between rain and sales, and now you're already trying to decide what the correlating relationships might be.

Remember regression? Why not let your HP-42S decide that for you? Why not try a simple, Linear model, and see how well it correlates – or fits – your data?

Ready to test your theory? All right: To put it a little more mathematically, what you're really proposing on the previous page is this:

'For the last ten years, the annual book sales of my bookstore has been directly proportional (on a linear model) to the amount of rainfall. That is:

$$y_{\text{sales}} = mx_{\text{rainfall}} + b$$

*where **m** and **b** are constants.'*

Test It: Here are your rainfall and sales data for the last ten years:

<u>Year</u>	<u>Rainfall (inches)</u>	<u>Sales (\$)</u>
1980	17.0	215,000
1981	25.2	236,000
1982	19.2	154,500
1983	18.5	209,000
1984	21.5	223,000
1985	25.0	240,000
1986	28.3	245,000
1987	12.5	180,000
1988	25.9	270,000
1989	22.0	234,600

Is there a linear correlation between annual rainfall and annual book sales?

Analyze: Press **2** **CLEAR** **CLX** **2** **DISP** **FIX** **0** **2**, then enter all the data (y-data first in each set):

2 1 5 0 0 0 ENTER 1 7 Σ^+
2 3 6 0 0 0 ENTER 2 5 . 2 Σ^+
1 5 4 5 0 0 ENTER 1 9 . 2 Σ^+
2 0 9 0 0 0 ENTER 1 8 . 5 Σ^+
2 2 3 0 0 0 ENTER 2 1 . 5 Σ^+
2 4 0 0 0 0 ENTER 2 5 Σ^+
2 4 5 0 0 0 ENTER 2 8 . 3 Σ^+
1 8 0 0 0 0 ENTER 1 2 . 5 Σ^+
2 7 0 0 0 0 ENTER 2 5 . 9 Σ^+
2 3 4 6 0 0 ENTER 2 2 Σ^+

(Now check the **SUM** – there should be 215.10 in the X-register and 2,207,100.00 in the Y-register. If not, then you've made an entry error and need to start over.)

Next, press **CFIT SLOPE** to find m . Result: 5,269.78

Then find the y-intercept, **YINT**. Result: 107,356.95

So your equation is: $y_{\text{sales}} = 5,269.78x_{\text{rainfall}} + 107,356.95$

So, are you right? Is the rain driving your book sales? That is, *how well does this line fit your data?* Unlike an ideal example, not all the data points are *exactly on* this line of regression. You know that the line is the *best-fitting* line for the given data, but that doesn't really tell you everything.

You need to use a measure of "fitness" to tell you how *closely* the line fits your data points. This measure is called the *correlation coefficient*, r , a number between -1 and 1: If r is close to 1 (for a positive m) or -1 (for a negative m), the fit is good, and your theory seems valid. If r is nearer to 0, then the fit is not good, and your theory sucks.

So, test the fitness of the regression line of your rainfall theory: **CORR**. Result: 0.76

Hmm... not bad. 0.76 indicates a fairly good linear correlation, though it could be stronger (suggesting possibly that although rainfall is a significant factor in your book sales, it doesn't explain all fluctuations).

How about trying other MODELS? Maybe rainfall affects sales in a pattern more nearly logarithmic or exponential than linear. You can find out – and without re-entering any data!

Explore: Find the correlation coefficients for your rainfall/book sales theory for logarithmic, exponential, and power curves.

Discover: Select **MDL**, then press **LOGF**, **EXIT** back to the previous sub-Menu, and select **LOGR** to find the correlation coefficient for the logarithmic model: 0.74.

Repeat this procedure (**MDL** **EXPF** **EXIT** **EXPR**) to find the correlation coefficient in the exponential model: 0.73.

Similarly, the power model yields a coefficient of 0.71.

It looks like the linear model did the best job after all!

Question: How can you make sure you *always* use the best model?

Answer: Select **BEST** on the **MDL** sub-Menu. The calculator will check all four models for you and select the one with the highest correlation coefficient for that particular set of data. Try it now....It picks the linear model, as it should!

All right, assuming that your linear correlation is more or less correct, if your cousin (the Old Farmer's Almanac expert) were to whisper conspiratorily that next year you'd have 30.4 inches of rain, what would you project for your bookstore sales next year?

Soothsay: Use your HP-42S to forecast your future sales based on this "tip" about the rainfall.

Like So: You need to use the ForeCaST X and the ForeCaST Y functions on your **EDIT** sub-Menu. If you enter an x-value, and press **FCSTY**, you'll find out what the y-value would be, based on the regression equation.

So, key in your cousin's rainfall prediction and choose **FCSTY**. Result: 267,558.38. Not bad!

But what you *really* want to know is how much rain you'd need to get to that magic sales plateau of \$300,000.

Simple: **3000000** **FCSTX**: Result: 36.56





"Hmm.....maybe a branch store in Seattle...."

Well, at least now you know how to use some of the built-in statistical functions and tests on your HP-42S. As you can see, your calculator does the hard part – accumulating the data – for you, so that you can concentrate on choosing the correct test!

SOLVER and Integrate:

Power Tools That Use Programs

Yes, you heard right. These final two power tools are *really* powerful, because they allow – nay, require – you to customize them with (simple) programs of your own.

To use  , for example, your program must calculate the *function* that you want to integrate. To use  , your program must create an *equation*. In both cases, these programs are actually very straightforward – far simpler than that MUSIC mess, for instance.

However, there *are* a few minimum requirements for these programs:

- Each must begin with a global label.
- Each must make use of variable menus (recall pages 199-203 and MVAR and VARMENU statements).
- Each should be as short and simple as possible – without ALPHA messages, PROMPTs, or other instructions that stop or use the display. These programs may be run many times over (like a loop) as your HP-42S calculates with them; don't slow it down needlessly.

These rules aren't too tough, are they? All of the programming you need here should seem quite easy to you (but if you do need to review programming a bit, reread pages 173-209 now).

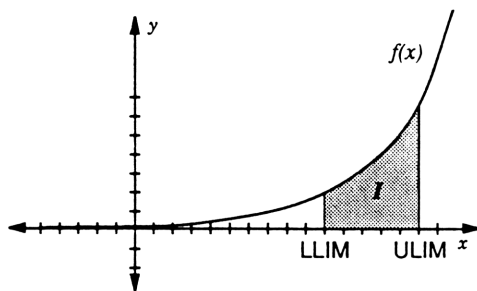
The $\int f(x)$ Menu

The $\int f(x)$ Menu is used to calculate the *definite integral* of a function.

As you may (or may not) recall, the definite integral of a function is, graphically speaking, an area or region, I , whose four "sides" are defined by four things:

- The function, $f(x)$, usually defines the "top side" of the region. This "side" is usually not a straight line, but a curve of some sort.
- The x -axis (the horizontal axis on a graph) usually defines the "bottom side" of the region.
- The Lower LIMit (LLIM) is a vertical line defining the *left* side.
- The Upper LIMit (ULIM) is a vertical line defining the *right* side.

So the graph of a typical definite integral might look like this:



Mathematically, the region of the definite integral is defined like this:

$$I = \int_{\text{LLIM}}^{\text{ULIM}} f(x) \, dx$$

where the values of the two limits (LLIM and ULIM) appear on the integral (\int) sign, the function, $f(x)$, follows that sign, and the variable of integration, x in this case, is indicated by what follows the d .

All of this means that to calculate a definite integral on your HP-42S, you'll have to define for it:

- A function
 - The limits of integration (LLIM and ULIM)
 - The variable of integration
-

Example: Use your calculator to solve this definite integral:

$$I = \int_{x=7}^{x=12} x(\sqrt{x^2 - a^2})^n \, dx$$

where $a = 5$ and $n = 3$.

Solution: First, you write a short program, using menu variables and calculating the function, $f(x)$ under the integral sign:

$$x(\sqrt{x^2 - a^2})^n$$

To do this, of course, you must **▀** **GTO** **••**, then go into Program Mode, **▀** **PRGM**, to key in this new program:

01 LBL "DISC"	The global label
02 MVAR "X"	The three variables in the function are defined as menu variables.
03 MVAR "A"	
04 MVAR "N"	
05 RCL "X"	This section actually calculates the function. At the end of line 14, the current value of the function will be sitting in the X-register.
06 X \uparrow 2	
07 RCL "A"	
08 X \uparrow 2	
09 -	
10 SQRT	
11 RCL "N"	
12 Y \uparrow X	
13 RCL "X"	
14 x	

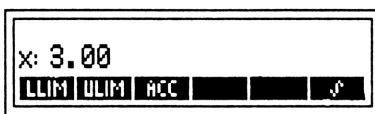
That's it – you've written the program, so **▀** **EXIT** Program Mode and select **▀** **f(x)**. You'll see this message:

"Select f(x) Program"

You must now select *which* program ("function") you want to integrate (since you might have had several different functions stored in memory). Of course, you have only the one, so choose **DISC**....

Now you're told to "Set Vars; Select fvar" and you're shown a variable menu for the function DISC, complete with variables, **X**, **A**, and **N**.

Enter values for those variables you know: You know that $a = 5$ and $n = 3$, so key in **5** **A** **3** **N**. Then you press the menu key of your *variable of integration* – in this case, x : **X**. As soon as you select your variable of integration, you'll see this menu:



Next, you enter your limits. For this problem, your LLIM is 7 and your ULIM is 12: **7** **LLIM** **12** **ULIM**.

Now, there's only one more item to deal with before you can press **✓** to calculate the integral: You must use **ACC** to tell the machine how "accurate" you want the result of your integral calculation.

"Don't I want it calculated as accurately as possible?"

Not necessarily.

The more accurate you want your result, the more time it will take your calculator. Besides, the limits you key in may already be approximations; you can't really expect an integral more accurate than the approximations you're using to calculate it, right?

Think of the Accuracy Factor as a tolerance for error: Could you live with a 1% error in your answer or does it need to be more like 0.01%? In the first case, you'd use 0.01 as your Accuracy Factor; in the second, you'd use 0.0001.

Choose 0.01 for now, and key it in (press $\square \square 0 \square 1$ \square ACC). Then (finally!) press $\square \int$ to calculate the integral....

Result: 30,320.15

The $\square \int$ function puts something in the Y-register, too: This the *uncertainty of computation*. Check it out: press $\square \text{X} \square \text{Y}$ and see that the uncertainty of computation for this integration is 301.57.

This means the final result could (and probably should) be expressed as: $30,320.15 \pm 301.57$

So the actual ("exact") answer is *somewhere between 30,018.58 and 30,621.72*.

Try narrowing down the margin of error by keying in 0.0001 as the Accuracy Factor and then recalculating the integral: $\square \square \square \square 0 \square 1$ \square ACC $\square \int$.

Result: 30,331.29 $\pm (\square \text{X} \square \text{Y})$ 3.03. Sure enough – your error has been cut by a factor of a hundred – but it takes a lot longer, doesn't it?

Spikes, Asymptotes, And Other Beasties: Know Your Limits!

The integration routine that you've just learned works very well for all "well-behaved" smooth functions – and as long as you pay attention to some of the basic rules for definite integrals.

Try This: Solve the same integral as the previous example, except this time use different limits: $LLIM = 4$ and $ULIM = 7$.

Solution: Just key in the new limits: $\boxed{4} \boxed{LLIM} \boxed{7} \boxed{ULIM}$ and press $\boxed{\text{↵}}$ to recalculate!...

...Aaargh! What beastie is this? "Invalid Type" ?
What happened?

Well, when you changed the limits, you created a situation where your calculator was trying to take the square root of a negative number: When $x = 4$, the function is a complex number – a no-no within definite integrals.

You need to stick to real numbers!

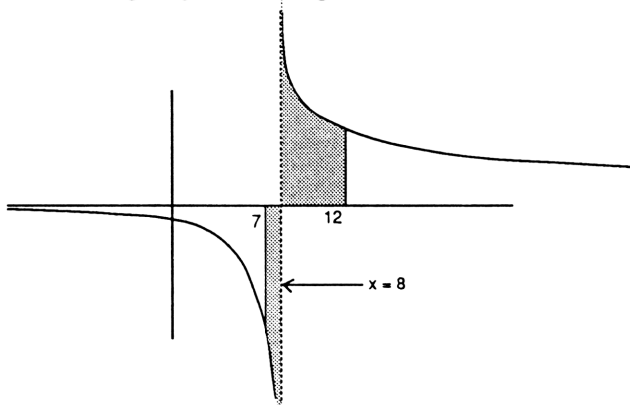
And complex numbers aren't the only problems to watch out for....

Question: Why can't the HP-42S help you with this integral?

$$I = \int_{x=7}^{x=12} \frac{x^2}{x-8} dx$$

Answer: Within those limits, ($x = 7$ and $x = 12$), there's a value ($x = 8$) where the function is *undefined*.

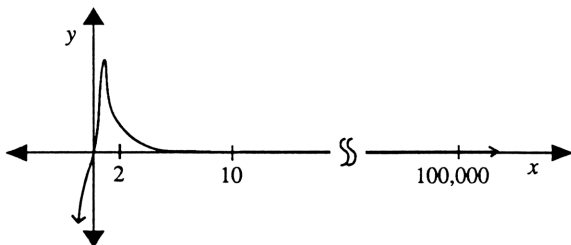
All definite integrals must use *continuous* functions within the limits specified (in this case, the graph clearly shows an asymptote at $x = 8$). *Since the function is not continuous over this interval the HP-42S will give you a wrong result.*



See? the *area is unbounded* at an asymptote, so a definite integral is undefined for this range – *even though the calculator will give you an answer*. You must use your mathematical insight to know it's "bogus!"

The other "bogies" to watch out for are the "spikes." Certain functions that are real, continuous, and usually well-behaved still "go ballistic" once in awhile,* varying enormously over very small intervals.

For Example: The graph of the function, $f(x) = x(2^{-x})$, has a sharp spike between $x = 0$ and $x = 2$:






If you choose limits too far apart – such as $x=0$ and $x=100,000$ – the calculator will give you a misleading answer, because, in doing the integration, it selects a number of *sample* points from within the limits you set. With too large a range, its sample points may "miss" the region with the spike – thus grossly underestimating the area under the curve!

The only way to avoid this is simply to be sure you know if your function has any steep spikes. And if you need to calculate and integral with a spike, you might want to reduce your interval of integration and/or your Accuracy Factor (to allow for more sample points within the interval).




*doesn't everybody?

The SOLVER Tool

The  SOLVER Menu works similarly to the  $f(x)$ Menu, except that  SOLVER uses your program as an *equation* rather than a *function*.


What's the difference? Well, with any algebraic expression – any combination of variables, constants and operators – a *function* is a process that *assigns* values to the *variables* and then computes a single, unique *numeric result* – called the "value of the function." For example, in your function, DISC, its algebraic expression is $x(\sqrt{x^2 - a^2})^n$. The function is the process that assigns a value to x , "plugs this into" the expression and blindly computes the result – "the value of the function DISC at a given x ."


An *equation*, on the other hand, is a process that *assigns* this resulting value of the function ahead of time, then "backtracks" to compute the value of the variables necessary to produce that result. So to use the above function as an *equation*, the machine would *assign* it a value, then "backtrack" through the expression to compute what x must have been to produce that value!

So you see, the only difference between the assumptions of  $f(x)$ and  SOLVER is that  SOLVER *assigns* a resulting value to the specified function. And that assigned value is always *zero*.

"Hmm...But what if my function is *already* in the form of an equation – and it's not equal to zero?"


No problem – you simply rearrange it: If you have $3x^2 + 4x = 35$, for instance, you could instead write this as $3x^2 + 4x - 35 = 0$,...right?

Any equation can be rearranged so that it is set equal to zero. And that's the strategy you must use when programming for the  **SOLVER**.

Try This: Write a program for  **SOLVER** to solve for any variable in the Ideal Gas Law: $PV = nRT$
where P = pressure of the gas
 V = volume of the gas
 n = number of moles of the gas
 R = ideal gas constant
 T = the temperature of the gas (in Kelvins)

Solution: First, *rearrange the equation so that it is equal to zero:*
 $PV - nRT = 0$ Now write the program, complete with a global label, menu variables, and the straightforward "calculation-description" itself:

01 LBL "GAS"	09 x
02 MVAR "P"	10 RCL "N"
03 MVAR "V"	11 RCL "R"
04 MVAR "N"	12 RCL "T"
05 MVAR "R"	13 x
06 MVAR "T"	14 x
07 RCL "P"	15 -
08 RCL "V"	

Once you've created a program for the equation you want to solve, you can use  **SOLVER** to solve for *any one* of its variables – after you key in known values for the other variables.

Try It: Use **SOLVER** to find out what pressure a gas is under if 10.0 moles (n) of it are contained in a 8.2 Liters (V) jar at a temperature (T) of 273 Kelvins (0°C or 32°F). The Ideal Gas Constant, R , is 0.0821 L-atm/mol- $^{\circ}\text{K}$.

Solution: **EXIT** Program Mode, then press **SOLVER** to see the menu of global labels and "Select Solve Program" (choose **Gas**, of course).

Next, you'll see a variable menu with all the variables of the Ideal Gas equation. Just key in the known values:

10 **N** **8.2** **V** **273** **T** **0.0821** **R** **P**

then press **P** to solve for the unknown pressure.

Result: 27.33 (atm.)

Not bad, eh? And there's no end to the "what-if" calculations you can do once you have an equation programmed for **SOLVER**!*

For Example: What pressure results if you raise the temperature to 300K? What temperature results if you maintain pressure at 27.33 atm but expand the gas to 12 liters?

Solutions: **300** **T** **P**. Result: 30.04 (atm).

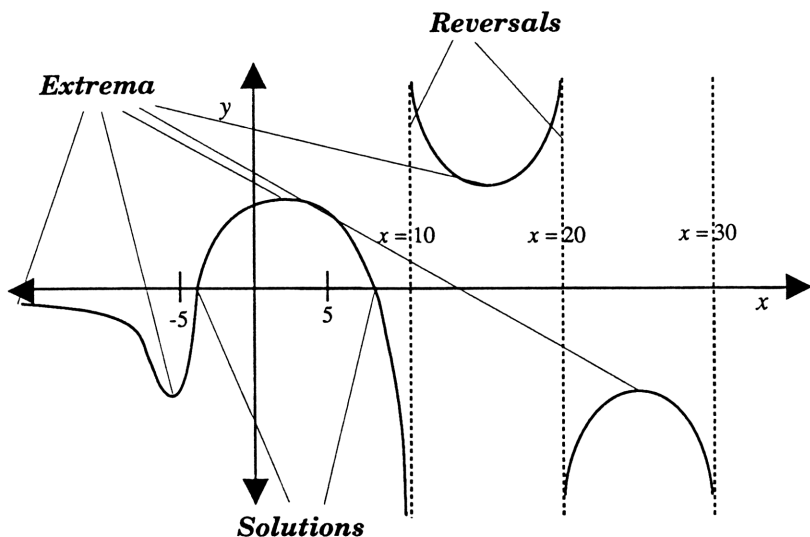
27.33 **P** **12** **V** **T**. Result: 399.46 (K)

*And remember (from page 203) that you can always press **■** and hold down a menu-key to see what value is currently stored. This keeps you from getting lost when doing lots of "what-iffing."


What SOLVER Is Really Doing

As handy as **SOLVER** sounds so far, you should also know that it gives back a whole lot more information... than just the answer. In fact, it fills up the stack with information....

As with **INT(x)**, **SOLVER** works with a "mental" graph of a function, such as the one shown below:



To the Solver, a "solution" occurs anytime the curve crosses the x -axis. (After all, that's when the value of the function becomes zero – just as your machine requires, right?)

The function illustrated above has *two* solutions – one somewhere between -5 and 0, the other somewhere between 5 and 10....hmmm...so, which solution will it give you when you use  **SOLVER**?





Guessing The Answer: Telling Where To Look






Whenever you're working with an equation that has more than one solution (or otherwise behaves strangely like this example graph), you should enter two "guesses" as to the answer. Your machine will then begin looking for an answer *between* those two guesses.








For Example: The equation $x^2 + x - 6 = 0$ has two solutions. Use your calculator to find them. (Hint: one solution is positive, and the other is negative.)

Solution: First, enter the program for this equation:



01 LBL "TWO"	05 X [↑] 2
02 MVAR "X"	06 +
03 RCL "X"	07 6
04 ENTER	08 -

Now,  Program Mode and press   for the variable menu ( is the only variable).

Key in two guesses, to suggest a "search area" for a solution:    . Now press  (yep – same key again) to find it: 2.00.


Now, guess a range for the negative solution:
      and press  again....
Result: -3.00

Don't Be Misled By False Solutions

As you can tell from this previous example, just as with  $\int f(x)$, you have to know something about your functions to get reliable solutions. It may take some exploration, and fortunately, the  **SOLVER** is particularly handy for this exploration.

But you can still misinterpret the results if you aren't careful! There are several types of situations that can lead the Solver to produce an answer in the X-register that isn't exactly a solution!

If you refer back to the graph on page 362, you see the two most common types illustrated there – Reversals and Extrema.

Your  **SOLVER** seeks solutions by repeatedly trying new values for the variable it's solving for. It tries to make the result of each successive trial closer and closer to zero. It figures that it has gone too far when the result of one trial was a positive value but the result of the very next trial was negative (or vice versa). It *assumes* that it just "crossed" the x -axis between those two trial points and that the solution (also known in algebra class as a "root" or a "zero") is there somewhere in between. It then narrows its search and quickly pinpoints the point that yields up a zero value – and reports this as a solution.

The problem comes when the program encounters a *Reversal* point. Reversals are typically caused by vertical asymptotes where one side is *increasingly* positive and the other side is *increasingly* negative as x approaches the asymptote. So your calculator can be fooled into reporting the asymptote value as a solution – even though the graph did not actually cross the x -axis! *Reversals are false solutions!*

A second type of false solution can occur when the calculator discovers that the function is no longer looking like it will ever cross the x -axis. This happens on the graph whenever the function "bends" in such a way that values that had been getting more and more positive suddenly start getting more and more negative (or vice versa).

Each of these "bends" in the graph of a function occurs at an *Extremum*. Sometimes these Extrema are useful things to know – but they aren't solutions to the equation, because the curve is *not* crossing the x -axis.


The Good News

You don't need to worry too much about these false solutions. Indeed, you can use them to help you find out more about your equation.

This is because your HP-42S keeps you informed about what it's doing.

It tells you when it has encountered a "Sign Reversal" or an "Extremum". It also tells you when one or more of your *guesses* happen to be impossible solutions (i.e. they tell the calculator to divide by zero or some other impossible thing). In such cases, your machine will display "Bad Guess(es)".

If by any chance your equation yields the same value no matter what trial value is used, your calculator will wonder, "Constant?," suggesting to you that your function is a constant (i.e. a number).

Sometimes the answer in the X-register is actually less interesting than what's in the other Stack registers. Every time you use  SOLVER, all *four* Stack registers receive some part of the "answer:"

In the X-register is the "solution," of course – false or otherwise;

In the Y-register is the previous trial value – the "next best "solution. Comparing this with the X-register gives you some idea about what the "neighborhood" of the solution looks like.

In the Z-register is the value of the function when the "solution" is used as a trial. Of course, if the solution is an exact solution, then the value in Z should be zero (or very, very close to zero, in the case of irrational number solutions).

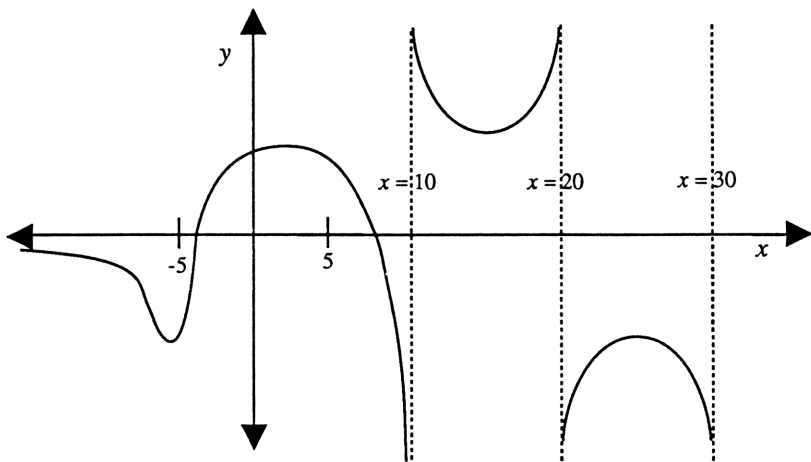
In the T-register is a number – a integer between 0 and 4:

- 0 means: "There's a real solution in the *x*-register."
- 1 means: "The 'solution' in the *x*-register is more likely just the approximate location of a sign reversal (an asymptote, maybe)."
- 2 means "The 'solution' in the *x*-register is more likely just the approximate location of an extremum." (Remember that the value of the function at this extremum is stored in the Z-register).
- 3 means: "One or more of the guesses caused an error during the calculation; you should try some new ones."
- 4 means: "The function seems constant in this search interval."


Test your understanding of **SOLVER**'s behavior and of the information it provides you:


Quizzette: Assume that your function has a graph like the one below. For each of the following sets of guesses, what type of response will **SOLVER** give – solution, reversal, extremum, bad guess, or constant?

1. -1 and 5
2. 10 and 19
3. 19 and 21
4. 5 and 35
5. 5 and 9
6. -8 and -10
7. 21 and 29
8. 5 and 19



- Solutiette:**
1. Extremum. In this range, the calculator will discover the bump.
 2. Bad Guess. 10 is an asymptote. The function is undefined at $x = 10$.
 3. Reversal. The asymptote at $x = 20$ is crossed here.
 4. Bad Guess. The function is undefined at $x = 35$; the asymptote at $x = 30$ limits the function from getting out that far.
 5. Solution. Finally! Make well-chosen guesses and you'll arrive at a solution.
 6. Extremum. This is called an asymptotic extremum because the function gets closer and closer to the x -axis without ever getting there.
 7. Extremum. This is a straight-forward example of an extremum.
 8. Solution? Assuming that it begins its search at the first guess, 5, it'll arrive at a solution before any of the other beasts.

Moral Of The Story: Use your own knowledge of the function to narrow down your search for solutions *before* using  **SOLVER**.

Finally, *did you know* that you can solve an equation (using  SOLVER) that includes an integration in it?

Hmm: Junior Beaver arrived home from Hydro Tech one spring, and found his father all in a dither about ducks "ruinin' my dam again! That's the third year in a row them ducks've landed on th' pond so thick that the water level rose above my dam !"

Junior knew that something had happened the last few years to flood the dam, but he had his doubts about this theory. "Pop, how much does the water have to rise to cause a problem?"

"I always leave 'bout half a meter of "just-in-case" at the top. It's hard to find enough building material to do more 'n that."

"Then I don't think the ducks could possibly be at fault. Let me run some calculations to see," Junior suggested. He then set out with his trusty HP-42S to test the validity of his father's theory, taking these measurements:


- The surface of the pond – roughly rectangular – measured 70 meters by 13 meters.
- The pond was 3.5 meters deep (at its deepest) and trough-shaped – like a tall tin can, halved lengthwise and laid flat.
- The average volume of a duck's bottom (d.b.) in water was 2.25 Liters and its area on the surface averaged (450 cm^2) .*

*For these sensitive and personal measurements, Junior Beaver wishes to thank several duck acquaintances who prefer to remain anonymous. They know who they are.

Then he derived the formula equating the total volume of n ducks' bottoms, $nV_{d.b.}$, to the rise in the pond level:*

$$1000 \int_{h=D}^{h=D+I} L \left(\frac{(x^2 + 4D^2)^2}{64D^2} \right) \text{Acos} \left(1 - \frac{8hD}{x^2 + 4D^2} \right) - \frac{L}{2} \left(\frac{x^2 + 4D^2 - 8hD}{8D} \right) \sqrt{h \left(\frac{x^2 + 4D^2 - 4hD}{D} \right)} dh$$

L = length of the pond, x = the beginning width of the pond, h = the actual maximum depth of the pond, D = the beginning maximum depth of the pond (before ducks), and I = the increase in pond level due to the n ducks' bottoms.

But Junior had trouble using this formula with  **SOLVER** to find h . Your job is to help him do this.

Hmm: You'll need *two* programs. The first one is for the function under the integral (call it "POND"):

01 LBL "POND"	17 ÷	33 RCL "H"
02 MVAR "D"	18 STO "R"	34 -
03 MVAR "X"	19 X↑2	35 x
04 MVAR "H"	20 RCL "L"	36 RCL "R"
05 MVAR "L"	21 x	37 2
06 RAD	22 1	38 x
07 RCL "X"	23 RCL "H"	39 RCL "H"
08 X↑2	24 RCL "R"	40 -
09 RCL "D"	25 ÷	41 RCL "H"
10 X↑2	26 -	42 x
11 4	27 ACOS	43 4
12 x	28 x	44 x
13 +	29 RCL "L"	45 SQRT
14 RCL "D"	30 2	46 x
15 8	31 ÷	47 -
16 x	32 RCL "R"	

* You probably want to take Junior's word for this – unless you, too, are an Eager Beaver!

Then, to get an equation *using this integral within a* **SOLVER** *function*, you need to rearrange it like this:

$$nV_{db.} - \left(1000 \int_{h=D}^{h=D+I} \text{"POND"} \right) = 0$$

Thus, the **SOLVER** program for this would be:

```

01 LBL "BEAV"
02 MVAR "N"      09 PGMINT "POND"  16 1000
03 MVAR "V"      10 RCL "D"      17 X
04 MVAR "D"      11 STO "LLIM"   18 RCL "N"
05 MVAR "I"      12 RCL "I"      19 RCL "V"
06 MVAR "X"      13 +            20 X
07 MVAR "L"      14 STO "ULIM"   21 X<>Y
08 MVAR "ACC"    15 INTEG "H"    22 -

```

This **SOLVER** program is just like others you've seen except for the instructions on lines 09 and 15. You'll find these when you're in **PRGM** Mode and you choose **f(x)**. You'll see a menu with **MVAR** and two new options: **PINT** and **INTEG**.

You use **PINT** (at line 09) to tell your **SOLVER** program which other program you want to integrate. Then you use **INTEG** to tell your **SOLVER** program to actually begin the integration using the specified variable as the variable of integration.

The program lines 10-14 store the limits of integration, LLIM and ULIM, and the ACCuracy factor. Remember that when a program is used to do the integration, you need to have the program store the limits of integration because you won't be keying them in from the keyboard.

Junior wants to use his programs now to test his theory. First he must figure out how many ducks can fit on the lake: The total surface area of the lake is 9,100,000 cm² (70 m x 13 m). Each duck occupies an average of 450 cm², so about 20,222 ducks – packed in like sardines – could fit onto the pond's surface area.

Now, he presses **SOLVER** and chooses **BEAN**. Then, he enters the following values for his variables:

$N = 20,222$ ducks

$X = 13$ meters

$V_{db} = 2.25$ Liters

$L = 70$ meters

$D = 3.5$ meters

$ACC = 0.01$

When he's finished with the entry of data, he solves for I – the amount the water will rise due to the ducks – by pressing **I**:

Result: **0.01** (meters).

Junior grins triumphantly. Even if the pond were packed solid with ducks, the water level would rise only about 1 centimeter, not nearly enough to swamp his dad's extra 50 centimeters of dam (in fact, even if all 20,222 ducks were to dive for food at the same moment, this still wouldn't be enough to do it)!*

*It turned out, however, that the ducks weren't entirely blameless: After some further exploration, Junior and his Dad discovered that the real problem was that his Dad was building his dam each year on a deeper and deeper layer of duck droppings accumulating on the bottom of the pond. The droppings were not as firm a base as the actual pond bottom, so the dam was settling and slipping a little more each year. By pure coincidence, the resulting swamping happened around the same time as the annual arrival of the ducks.

Powerful Problems

- Using the **BASE** Menu, find the HEXadecimal solution to the following:

$$\frac{16D3_{16} - 7143_8}{100100_2} - 61_{10} = ?_{16}$$

- Can you use the **SIM** function in the **MATRIX** Menu to solve a system of 6 equations in 5 variables? How would you go about it?
- Your calculator has built-in statistical analysis functions only for two-variable regressions. How could you make it do "multivariate linear regression" to analyze data with more than two variables (say, 4 independent variables and one dependent variable)?
- The volume of the intersection of a sphere with a radius of 6 cm and a cone with an aperture angle of $\pi/3$ can be found by solving the following:

$$V = \int_{\theta=0}^{\theta=2\pi} \int_{\phi=0}^{\phi=\frac{\pi}{3}} \int_{\rho=0}^{\rho=6} \rho^2 \sin\phi \, d\rho \, d\phi \, d\theta$$

Find the volume of this intersection using the power of **∫f(x)**.

- Find all six real roots of this polynomial:

$$x^6 - 45x^5 - 447x^4 + 4293x^3 + 1782x^2 - 29952x - 27040 = 0$$

Powerful Solutions

1. The sequence of keystrokes: **BASE** **HEXM** **1** **6** **HEX** **0** **3** **EXIT**
DECM **7** **1** **4** **3** **—** **BINM** **1** **0** **0** **1** **0** **0** **÷** **DECM** **6** **1** **—** **HEXM**.

Result: FFFFFFFF

2. Yes, but you want to ignore one of the equations. Remember from page 319 that **SIMR** calculates a solution matrix with matrix division, which involves an inversion of a matrix, and – as you learned back on page 152, only square matrices can be inverted. This means that you can use **SIMR** only for systems that have matching numbers of equations and variables.
3. As you remember (page 345), the linear regression model for one independent variable and one dependent variable is $y = a + bx$. If there were four independent variables, the regression equation becomes $f = a + bx + cy + dz + ew$ where f is the dependent variable, x , y , z , and w are the independent variables, and a , b , c , d , and e are the coefficients of regression.

So how do you solve one equation in five variables?

Obviously, you can't. But by combining the powers of your **STAT** and **MATRIX** power tools you can transform this one equation into a system of *five* equations in five variables.

The idea is to create a single *matrix* equation of the type $AX = B$.

You need to accumulate, using the STATistical registers, various values and then store them in MATA and MATB so that the matrix equation looks like this:

$$\begin{bmatrix} n & \sum x & \sum y & \sum z & \sum w \\ \sum x & \sum (x^2) & \sum xy & \sum xz & \sum xw \\ \sum y & \sum xy & \sum (y^2) & \sum yz & \sum yw \\ \sum z & \sum xz & \sum yz & \sum (z^2) & \sum zw \\ \sum w & \sum xw & \sum yw & \sum zw & \sum (w^2) \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} \sum f \\ \sum fx \\ \sum fy \\ \sum fz \\ \sum fw \end{bmatrix}$$

The *coefficient of determination*, which tells you how good a fit your regression equation is, can be calculated with this formula:

$$R^2 = \frac{a\sum f + b\sum fx + c\sum fy + d\sum fz + e\sum fw - \frac{1}{n}(\sum f)^2}{\sum (f^2) - \frac{1}{n}(\sum f)^2}$$

All of this can be done by combining the power of the STATistical registers and the matrix math.

Of course, the best way to do this is to write a program. You could then use it to: enter data, accumulate it into statistical registers (and others, too), create the matrices, use **SIMR** to solve for the "solution matrix," and calculate R^2 automatically. And since it's a program, it would allow you to use prompts and menus to keep everything straight.

This is just one example of how programming can vastly extend the power of your power tools. Indeed, there are few problems that are totally beyond the abilities of your calculator – as long as there is enough memory available for the problem.

4. The triple integral can be solved with a series of small programs. First, rearrange the integral for a clearer order of calculation:

$$V = \int_{\theta=0}^{\theta=2\pi} \left(\int_{\phi=0}^{\phi=\frac{\pi}{3}} \left(\int_{\rho=0}^{\rho=6} \rho^2 d\rho \right) \sin\phi d\phi \right) d\theta$$

Your program needs to calculate the innermost integral first, use this result as a constant in the next integral, calculate the second integral, and use that result a constant in the final integral. Of course, you need to make sure that the correct limits are stored before each integration. Note: In order to calculate a multiple integral, you must *not* use $\int f(x)$. If you try to use $\int f(x)$, you'll get the message "Integ(Integ)" which is your calculator's way of telling you that it doesn't do windows.

The program, TRIP1, calculates this triple integral by calling up integration subroutines three separate times.*

Integration Subroutines:

01 LBL "RHO"	01 LBL "PHI"	01 LBL "THETA"
02 MVAR "P"	02 MVAR "Q"	02 MVAR "H"
03 RCL "P"	03 RCL "P"	03 RCL "Q"
04 X↑2	04 RCL "Q"	
	05 SIN	
	06 x	

*This basic format could be used to find any triple integral – all that you'd need to change would be the specific integration subroutines (RHO, PHI, and THETA) and the limits of integration in the main program so that they match the new problem.

Main Program:

```
01 LBL "TRIP1"          15 STO "P"
02 MVAR "RADIUS"        16 PGMINT PHI"
03 MVAR "ANGLE"         17 RCL "ANGLE"
04 VARMENU "TRIP1"      18 STO "ULIM"
05 STOP                 19 INTEG "Q"
06 RAD                  20 STO "Q"
07 PGMINT "RHO"         21 PGMINT "THETA"
08 0                    22 PI
09 STO "LLIM"           23 2
10 RCL "R"              24 x
11 STO "ULIM"           25 STO "ULIM"
12 0.01                 26 INTEG "H"
13 STO "ACC"            27 STO "VOLUME"
14 INTEG "P"            28 VIEW "VOLUME"
```

First, you input the radius of the sphere and the aperture angle of the cone (and line 06 puts into RADian mode). Then you call "RHO" and store the limits of the first integration and the ACCu-racy factor (0.01) to be used throughout the problem.

After the first integration, the result is stored in "P", to be used in the next integration. Then a new upper-limit is stored (the lower limit and accuracy factor remain the same), and the result of the second integration is stored in "Q". Again, the upper-limit for the next integration is changed (to 2π), and "THETA" is inte-grated. Finally, the answer is stored and viewed as "VOLUME".

Now press **[XEQ] [TRIP1] [6] [RADI] [] [π] [3] [÷] [ANGLE] [R/S]**, to calculate the triple integral:

Answer: VOLUME=226.22 (cm³).

5. The trick in this problem is to be systematic in your search for multiple solutions, using whatever shortcuts you can remember from your algebra class. First, of course, you need to write the program for this polynomial equation:

01 LBL "POLY"	13 4	24 RCL "X"
02 MVAR "X"	14 Y↑X	25 X↑2
03 RCL "X"	15 447	26 1782
04 6	16 x	27 x
05 Y↑X	17 -	28 +
06 RCL "X"	18 RCL "X"	29 RCL "X"
07 5	19 3	30 29952
08 Y↑X	20 Y↑X	31 x
09 45	21 4293	32 -
10 x	22 x	33 27040
11 -	23 +	34 -
12 RCL "X"		

Now, before actually trying to find the solutions, you should see what information you can glean from the equation to help you narrow your search somewhat:

- According to Descartes' Rule of Signs you can expect either 1 or 3 positive and either 1 or 3 negative real solutions.
- Values for x lower than the lowest solution yield will positive y -values. And values for x higher than the highest solution will also yield positive y -values. This is a property of polynomials of *even* degree (in this case, the degree is 6), *whose highest-degree coefficient is positive in sign* (the x^6 term has a positive coefficient).

Now begin your search: **[SOLVER]** **[POLY]** gets you to the variable menu. Enter your two guesses, then press **[X]** to begin the search for solutions (when entering guesses with no good idea where to begin or what the graph looks like, it's a good idea to use the same number for both guesses. Here is one suggested searching sequence. Of course, you can use any method you want):

<u>Guesses</u>	<u>Result</u>	<u>Interpretation</u>
0, 0	- 1.00	Press [R)] [R)] [R)] to see a 0.00 in the T-register, so this is a true solution.
1, 1	- 2.00	Again, confirm that this is a root. Now, from Descartes Rule of Signs, you know there must be a third negative root.
2, 2	4.00	A positive root!
8, 8	5.00	A second positive root! There's a third positive root, too! Notice that when $x=8$, the sign in the display is negative, so the next root is <i>greater than</i> 8.
64, 64	52.00	That's it! Now find the negative one.
-4, -4	- 2.00	Nope – you have that one already!
-16, -16	-13.00	Ahhh! There it is – the sixth root.

In sum, the roots are: -13, -2, -1, 4, 5, and 52.



Foundation Completed

"The Book Stops Here"

That's about it – the end of the Easy Course, the tools put away – for now. But you have only just begun. You've just finished your apprenticeship and become a fully-trained HP-42S tinkerer.

But of course, it doesn't mean you've seen it all. As we said at the start, this book doesn't cover all of the many uses of your machine. For example, here are a couple of topics you can now explore on your own:

- Display graphics;
- Printer operations;

Of course, most of the topics you did cover in this Easy Course have plenty more exciting functions and variations to try. Explore them – enjoy them – and get the most out of your HP-42S!

How Did You Like This Book?

Do you find yourself wishing we had covered other things? More of the same things? Did any mistakes, typos, or other little mysteries leap out and grab you by the lapels? Please let us hear from you. Your comments are our only way of knowing whether these books help or not – and we always read our mail!

Grapevine Publications, Inc.

P.O. Box 118

Corvallis, Oregon 97339-0118 U.S.A.

Here are other great books

HP-28S Software Power Tools Utilities

Whether or not you're experienced with the HP-28S, you'll find this book to be a great collection of advice, good habits and sound programming principles.

- **Directory Manipulation**
- **Display Formatting**
- **Graphics/Plotting**
- **General Object Manipulation**
- **Printer Manipulation**
- **Program Development**
- **Sorting/Searching**

Learn clear, clean and convenient methods for arranging and naming the objects and directories in your HP-28S's memory, so that as you gather tools and routines, "there's a place for everything—and everything in its place."

An Easy Course In Using The HP-28S

If you're looking for a clear, straightforward explanation of the powerful HP-28S, 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-28S your favorite calculating tool.

You'll learn about this and more:

- | | |
|---------------------------------|--------------------|
| • The Display | • Menu keys |
| • Posting Memos | • Keyboards |
| • Real Numbers | • Flags |
| • Strings | • Lists |
| • Complex Numbers | • Matrices |
| • Algebraic Objects | • Vectors |
| • Programs | |
| • User Defined Functions | |

HP-28S Software Power Tools Electrical Circuits

Here's the solutions books you've been waiting for! First, you build a friendly and easy-to-edit description of your circuit, which may have any of the following elements in series or in parallel:

- | | |
|--------------------------------------|---------------------|
| • Resistors | • Capacitors |
| • Inductors | • Impedences |
| • Independent voltage sources | |
| • Independent current sources | |

You can do either mesh or nodal analysis, construct general networks or ladders, and use a host of small utility routines to do side calculations as you wish. And naturally, you can vary the frequency of your sources and plot results, either on the display or the infrared printer. Every routine is explained, and every piece of the program is documented!

Here's a list of all our other books:

- **An Easy Course In Using The HP-22S**
- **An Easy Course In Using The HP-27S**
- **An Easy Course In Using The HP-19B**
- **HP-19B Pocket Guide: Just In Case**
- **An Easy Course In Using The HP-17B**
- **HP-17B Pocket Guide: Just In Case**
- **An Easy Course In Using The HP-12C**
- **HP-12C Pocket Guide: Just In Case**
- **An Easy Course In Programming The HP-41**
- **An Easy Course In Programming The HP-11C and HP-15C**
- **The HP Business Consultant Training Guide**
- **Using Your HP-41 Advantage**
- **Statics For Students**
- **An Easy Course In Using The HP-16C**

Grapevine Publications, Inc.

P.O. Box 118

Corvallis, OR 97339-0118

To Order:

Call our Toll-Free Line for the location of the GPI dealer nearest you, **OR**

Charge the books to **VISA** or **Mastercard**, **OR**

Send this Order Form to: **Grapevine Publications, P.O. Box 118, Corvallis, OR 97339**

_____ copies of	An Easy Course In Using The HP-42S	@ \$22.00 ea. = \$ _____
_____ copies of	An Easy Course In Using The HP-32S (Availability TBA)	@ \$22.00 ea. = \$ _____
_____ copies of	An Easy Course In Using The HP-22S	@ \$22.00 ea. = \$ _____
_____ copies of	An Easy Course In Using The HP-19B	@ \$22.00 ea. = \$ _____
_____ copies of	The HP-19B Pocket Guide: Just In Case	@ \$ 5.00 ea. = \$ _____
_____ copies of	An Easy Course In Using The HP-17B	@ \$22.00 ea. = \$ _____
_____ copies of	The HP-17B Pocket Guide: Just In Case	@ \$ 5.00 ea. = \$ _____
_____ copies of	The HP Business Consultant Training Guide (18C)	@ \$22.00 ea. = \$ _____
_____ copies of	An Easy Course In Using The HP-12C	@ \$22.00 ea. = \$ _____
_____ copies of	The HP-12C Pocket Guide: Just In Case	@ \$ 5.00 ea. = \$ _____
_____ copies of	An Easy Course In Using The HP-28S	@ \$22.00 ea. = \$ _____
_____ copies of	HP-28S Software Power Tools: Utilities (Available Sept. 25, '90)	@ \$18.00 ea. = \$ _____
_____ copies of	HP-28S Software Power Tools: Electrical Circuits	@ \$18.00 ea. = \$ _____
_____ copies of	An Easy Course In Using The HP-27S	@ \$22.00 ea. = \$ _____
_____ copies of	An Easy Course In Programming The HP-41..	@ \$22.00 ea. = \$ _____
_____ copies of	Computer Science on Your HP-41 (Using Advantage)	@ \$15.00 ea. = \$ _____
_____ copies of	Using Your HP-41 Advantage: Statics	@ \$12.00 ea. = \$ _____
_____ copies of	An Easy Course In Programming The HP-11C&15C	@ \$22.00 ea. = \$ _____
_____ copies of	An Easy Course In Using The HP-16C	@ \$22.00 ea. = \$ _____

(Prices valid through February 5, 1990)

Subtotal = \$ _____

SHIPPING INFORMATION:

For orders **less than \$10.00** **ADD \$ 1.00** \$ _____

For all other orders – Choose one: **Post Office** shipping and handling **ADD \$ 2.50** \$ _____

(allow 2-3 weeks for delivery)

UPS shipping and handling **ADD \$ 3.75** \$ _____

(allow 7-10 days for delivery)

International Mail **Surface Post \$4.50** **ADD \$ 4.50** \$ _____

(allow 6-8 weeks for delivery)

Air Parcel (Please contact us for correct amount or add \$10 per book to Canada and Mexico. Add \$25 per book to all other countries. We will refund any cash excess, or charge exact shipping cost to credit cards. Allow 2-3 weeks delivery)

TOTAL AMOUNT: _____ **\$** _____

PAYMENT:

Your personal check is welcome. Please make it out to **Grapevine Publications, Inc. OR**
(International Check or Money Order must be in U.S. funds and drawn on a U.S. bank)

Your **VISA** or **MasterCard** #: _____ Exp. date: _____

Your signature: _____ Phone: () _____

Name _____

Shipping Address _____

(Note: UPS will not deliver to a P.O. Box! Please give a street address for UPS delivery.)

City _____ State _____ Zip _____

For Orders Only
call: **1-800-338-4331**
(In Oregon 754-0583)

Reader Comments

We here at Grapevine love to hear feedback about our publications. It helps us write books tailored to our readers' needs. If you have any specific comments or advice for our authors after reading this book, we'd appreciate hearing them!

Which of our books do you have?

Comments, Advice and Suggestions:

May we use your comments as testimonials?

Your Name:

Profession:

City, State where you live:

How long have you had your HP calculator?

Please send Grapevine Catalogs to the following people:

Name _____

Address _____

City _____ State _____ Zip _____

and

Name _____

Address _____

City _____ State _____ Zip _____

An Easy Course In Using The HP-42S

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-42S

If you have an HP-42S, you can do a lot, right? *"Right!"*

Calculate a hairy triple integral? *"I can do that!"*

Solve a system of 6 linear complex equations? *"I can do that!"*

Customize your HP-42S menus, just for you? *"I can do that!"*

*...How am I going to **do** that?*

Take this Easy Course on the HP-42S. It'll show you how!

First, you'll read about math and functions, the Stack, the CUSTOM menu and each of the variable types (real/ALPHA, Complex and Matrix). Then you'll learn to *program* the HP-42S and use its built-in application tools to solve tough problems with ease and understanding.

All this – with diagrams and examples, quizzes and solutions – in Grapevine's inimitable style and clarity. It's amazing how the right explanation can turn a complex machine into a powerful and friendly tool!

ISBN 0-931011-26-4



FROM THE PRESS AT

GRAPEVINE PUBLICATIONS, INC.

P.O. Box 118 • Corvallis, Oregon 97339-0118 • U.S.A. • (503) 754-0583



HP Part # 92206M