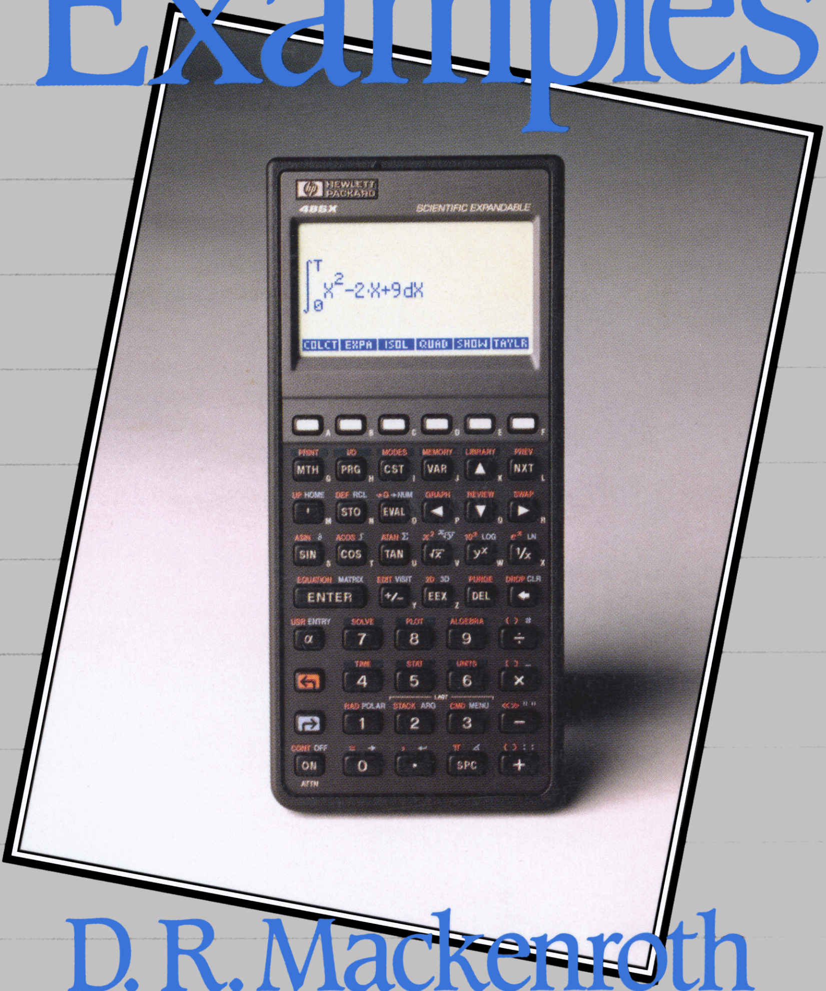
HEWLETT
PACKARD

HP 48 Programming Examples



D. R. Mackenroth

HP 48 Programming Examples

Hewlett-Packard Press Series

Binstock *et al., Programming for the Laserjet II in C*

Dowden, *Inside the EISA Computers*

Eppler, *A Beginner's Guide to SCPI*

Hewlett-Packard, *HP-GL/2 Reference Guide*

Hewlett-Packard, *The Ultimate Guide to the vi and ex Text Editors*

Kasper/Arns, *Graphics Programming with PHIGS and PHIGS Plus*

Kobara, *Visual Design with OSF/Motif*

Mackenroth, *HP 48 Programming Examples*

McMinds, *Mastering Motif Widgets*

Monday/Robinson, *Using Your HP 95LX: Practical Examples and Applications*

Rosenberg, *KornShell Programming Tutorial*

Schoonover/Bowie/Arnold, *GNU Emacs: UNIX Text Processing and Programming*

HP 48 Programming Examples

D. R. Mackenroth

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam
Bonn Paris Milan Madrid Sydney Singapore
Tokyo Seoul Taipei Mexico City San Juan

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warrantees or representations, nor does it accept any liabilities with respect to the programs or applications.

The publisher offers discounts on this book when ordered in quanity for special sales. For more information please contact:

Corporate & Professional Publishing Group
Addison-Wesley Publishing Company
One Jacob Way
Reading, Massachusetts 01867

ISBN 0-201-56325-8
1 2 3 4 5 6 7 8 9 10 MA 9594939291
First printing, October 1991

Contents

8. Fun and Games

Preface

I undertook to write these programs for the Hewlett-Packard HP 48 calculator because I thought it would be fun and it would be easy.

Fun because the HP 48 has every possible feature and function a scientific programmer could want. And easy because I already had a healthy dollop of experience—both as the writer of owner's manuals for early Hewlett-Packard programmables like the HP 25 and HP 97, and as the author of several books on programming in BASIC.

Turns out I was half right.

Creating programs for the HP 48 *is* fun, simply because this calculator has everything going for it. It's chock full of mathematic and scientific functions, along with a panoply of graphics, sound, and timekeeping features. And almost everything you can do from the keyboard can be done from a program.

Truth to tell, though, it wasn't easy at first. I made every mistake in the book—and some that probably hadn't been invented yet. Moreover, I was immutably tied to tired old programming techniques that failed to take full advantage of the HP 48's power.

But now it can be told: The HP 48 *is* remarkably easy to program—once you nail down a few principles. In these pages, you'll profit from what I so painfully learned. And I can now proselytize for HP 48 programming with the zeal of the newly converted.

What I hope you'll find in these pages is a source of practical programming knowledge:

- **Application programs** from the fields of calculus, statistics, business, and more.
- **Fun programs** from the realms of music, games, and health.
- **In-depth explanations** of many commands, and plenty of program annotation explaining every step.
- **Detailed step-by-step procedures** for creating, saving, and running the programs.
- **Structured programming** techniques that make it easier to construct large applications from tiny "building-block" subprograms.

- **Hints and tips** for getting the most from the calculator's display and menu.
- **Graphics programming** details that show how to draw pictures—and how to animate them, too.
- **A proven management strategy** for handling programs and directories.
- **Index entries** that lead you to explanations of key commands and techniques.

In short, this is the book I wish I'd had when I was learning to program the HP 48 calculator!

News Flash: Companion Disk Available! Now there's no need to laboriously type in program instructions. If you have an IBM-compatible or Macintosh personal computer, simply purchase the companion disk that contains all the application programs for *HP 48 Programming Examples*. Use the handy order form that accompanies this book.

Acknowledgements

This book is the brainchild of Don Cole of Hewlett-Packard, and to him I extend profound thanks. As my erstwhile manager and present-day mentor, Don brought me under the HP Press umbrella and gave generously of time and resources. Thanks also to John Wait of Addison-Wesley, and to Bob Silvey and Steve Beitler of Hewlett-Packard for their efforts in making HP Press a reality.

Special appreciation goes to HP's Hank Schroeder. Hank played a major part in every phase of this book, from first inchoate outline to complete, final manuscript. As the key reviewer, he was instrumental in making the programs easier to use and the text easier to understand.

Kudos also to Dennis York, Ron Brooks, and Tom Hender for their thorough and incisive review and debugging efforts. And thanks to Sandy Allen, who not only brought much-needed consistency to punctuation and grammar, but also goaded author and reviewers toward better on-time performance.

This book was produced using Hewlett-Packard's HP Tag text markup language. Paul Johnson supplied the software, hardware, and know-how for getting HP 48 programs and screens onto the printed page. Human intervention was present in the form of Jim Woods, who took my rough penciled drawings and magically transmogrified them into illustrations that are clear and precise. My special gratitude goes to Anne Shatrau Todd, who set up an easily modified template of HP Tag files for me to follow, and who was a source of knowledge and guidance at every turn.

As the primary contact at Addison-Wesley, Alan Apt was everything an author could hope for, offering aid and encouragement just when it was needed. Sheri Dearing, Kathleen Manley, and Shirley McGuire (in Oregon, Massachusetts, and Colorado, respectively) made certain that all parts of the publishing puzzle came quickly and smoothly together.

Finally, to everyone else who helped, including those unsung heroes at HP Calculator Support and the typists at Laser Works: thanks.

D.R. Mackenroth
Mountain View, California

1

Introduction

Welcome to *HP 48 Programming Examples*! Whether you're looking for a cookbook of instant applications, a source of information about HP 48 functions, or a means by which to learn how to program, this book is for you. You'll find *HP 48 Programming Examples* is friendly and easy to use, with all example programs and procedures thoroughly explained.

Why Program the HP 48, Anyway?

If you use your HP 48 calculator for the same purpose over and over—or if you want to lug around the full set of manuals at all times—you may never need to write a program. After all, with over 500 number-crunching commands built in, your HP 48 has plenty of functionality. Whatever you want to do, mathematics-wise, chances are there's an HP 48 function to fill the bill.

Nevertheless, programming can make the calculator a lot easier to use and more fun to have around. Here's how:

- **Detailed prompts:** Programming is a timesaver if, like most students and not a few engineers, you're switching applications several times a day. With just a few program lines, you can add prompts and labels to accompany your most-used functions. This "in-line user documentation" can tell you what to enter, when to enter it, and what your output is. By adding meaningful prompts such as `"Input the tangent"` or `"Calculate the number of wobulators and enter it in the display; then press CONT"`, you'll always do the right thing.

- **Explanatory text:** A program can include titles, remarks, clarification, examples, moral support—in short, anything that makes it easier to understand and use. You can display up to seven lines of text at a time, or scroll it past the reader, so you'll never again ask "What did that program do?"
- **Decisions:** Need to pick an alternative in a hurry? Programming is how you get to take advantage of the decision-making powers of the HP 48. Flags and structures such as IF ... THEN, CASE ... END, and DO ... UNTIL can automatically choose the best selection, then proceed.
- **Repetitive tasks:** With loops like FOR ... NEXT and START ... STEP, a program can replace recurring and tiresome keystroke routines, freeing you for more creative activity.
- **New functions:** Even with all the built-in power of the HP 48, from time to time you're sure to need something that's just a little different. Programming your own functions gives you the performance you want—and does it your way.

The Programs

There are programs here from the worlds of finance, calculus, statistics, and more—including real-world solutions and classroom aids. You'll also find fun applications such as music, games, and graphics. There's even a chapter with programs for health!

Because this book emphasizes structured programming, even long applications are a series of short, easy-to-enter subprograms. You can type the keystrokes for any subprogram in a few minutes, test it, then go on to the next one. (Naturally, if you can enter the program lines in a computer, then download them, so much the better.)

How to Use This Book

This book is *self-contained*. All you need is your calculator, and away you go. You can jump around—it's not necessary to read the chapters, or enter the programs, in any particular order. But you may want to begin with chapter 2, "Introduction to HP 48 Programming." This valuable chapter gives you step-by-step procedures for keying in a program, running it, and saving it, and it also explains the techniques and advantages of structured programming. Furthermore, chapter 2 offers a wealth of hints and tips for better HP 48 programs.

Most programs in this book are preceded by an *explanation and keystroke example* that illustrate the principles involved. You can whip through this example in a few moments, so you'll be better prepared to understand the actual program. And occasionally referring back to this preliminary description makes it easy to follow even long, detailed programs to their logical conclusion.

Programmers and HP 48 users can employ these programs in many different ways:

- **Use the entire program.** Each one is complete. Refer to chapter 2 if you need help on how to enter a program.
- **Modify the program.** No need to begin at ground zero every time you want to do something new on your HP 48. Just study and adapt these programs for your own ends.
- **Learn to program by doing it.** Study the explanations for the programs, and you'll learn valuable programming techniques. Don't be afraid to experiment either!
- **Pick up hints and tips.** Whether you're already a sophisticated programmer, or you're trying to learn how to use the calculator, you'll find the code explanations an invaluable source of tips and tricks for HP 48 functions and programming.

You can think of this book as a rich source of prewritten programs, or as templates to alter or augment. The important thing is not to feel bound by the programs in this book. Use them, study them, change them—they'll help you be a better HP 48 user, and a better programmer, too.

2

Introduction to HP 48 Programming

Programming releases the full power of the HP 48, and lets you completely command the calculator. You can create everything from simple, job-specific programs that handle repetitive tasks, to complex, detailed, applications with multifarious features.

This chapter contains hints and tips for programming, a few simple precepts to follow that will ease your life as an HP 48 programmer. You'll also find an explanation of the *structured programming* techniques used later in the example programs. Structured programming is an ordered, logical approach that makes it easier to create even long, complex applications.

A Quick Review

In the HP 48, a *program* is an object, similar to a number, a variable name, a list, an array, or any other object. It can be placed on the stack, stored using a variable name, and evaluated (executed) repeatedly.

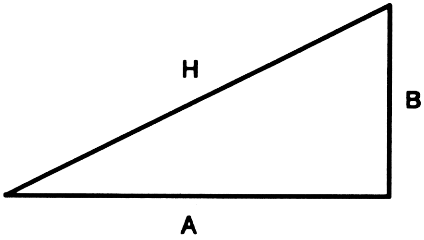
A program begins with the open program bracket, «, and ends with the close bracket, ». Between the brackets are HP 48 objects and commands that are not executed until the program is run.

Writing a Program

To write a program, press ↰ « », then press keys for the commands and other objects in the program. In many cases, the keys you press are the same ones you would press to make the calculation from the keyboard.

Here's an example: The Pythagorean theorem computes the hypotenuse of a right triangle, if the two sides are known. The formula is:

$$H = \sqrt{A^2 + B^2}$$



Here's how to enter a program to find the hypotenuse, H, of a right triangle, assuming the values for sides A and B are on the stack:

↰ « »
↰ x^2
↰ SWAP
↰ x^2
+
$\sqrt{x}$

As you key in the HP 48 commands, they aren't executed. Instead, the command name is written into the program. The keystrokes create the following program.

Program Instructions	Comments
«	Beginning of program.
SQ	Squares the number A in level 1 of the stack.
SWAP	Exchanges the contents of stack levels 1 and 2.
SQ	Squares the number B now in level 1 of the stack.
+	Adds the squares of A and B.
√	Takes the square root to find the hypotenuse.
»	End of program.

Now that you've entered the program, your next task is to save it under a name.

Saving a Program

To save the program, you create it and press ENTER to place it on the stack. Then you enter a name and press STO. The program is stored in the current directory.

For instance, to save the program under the name Hyp:

Keystrokes	Display	Comments
ENTER	« SQ SWAP SQ + √ »	Puts program on the stack.
' Hyp STO		Stores the program as *Hyp*.

If you look at the VAR menu after you save the program, you see the program name in a label, like this:

If the name is a long one, then you see only the first four or five letters. Even if you enter part of the name as lowercase letters, the name in the menu is seen as all uppercase. Within the HP 48, though, the program retains its name just as you entered it. If you enter

HYPOT, that's the program name; and if you enter Hypotenuse, then *that's* the name, even though you see HYPOT on the menu display.

Running the Program

There are several ways to run a program:

- Press the menu key (in the VAR menu).
- Type the program name in the command line, then press ENTER.
- Place the program name in level 1 of the stack, then press EVAL.
- Use the unquoted program name in another program.

As a simple illustration, here's how to find the hypotenuse of a triangle with side A of 3 units and side B of 4 units:

Keystrokes	**Display**	**Comments**
3 ENTER	3.00	Enters side A on the stack.
4	4	Puts side B on the stack.
HYP	5.00	Calculates the hypotenuse.

With sides A and B entered onto levels 1 and 2 of the HP 48 stack, pressing HYP executes the complete program. The values for sides A and B are "used up" by the program, and the result (the hypotenuse, 5) is left on the stack when the program is done.

That's HP-48 programming in a nutshell. Now let's look at some ways you can handle programs, and make them easier to understand and use.

Conventions

Here's a word about the conventions used in this book. When a program (such as *Hyp*) changes the HP 48's stack, you'll see a table of arguments and results, like this:

Arguments	Results
2: side A	2:
1: side B	1: hypotenuse H

This means that the program *expects* two arguments to be on the stack; in this case, side A is in level 2 and side B is in level 1. The program consumes these two arguments (leaving the rest of the stack unchanged), and *returns* a value for the hypotenuse, H. The net result is that two values are gobbled up by the program, and a single value is left on the stack when it's done.

If you're recording and documenting your own programs (and you should), an arguments-results table is a good thing to provide.

Many programs in this book have a *neutral* effect on the stack; they expect nothing and leave nothing. When the stack is unchanged by a program, the table of arguments and results isn't needed.

Because of the structured programming techniques used in this book—in which one program may call several other programs—intermediate results are sometimes present when a program is called. When such a result is present on the stack, but isn't used by the program, you'll often see it in italic type, like this:

Arguments	Results
2:	2: *tagged number of wobulators*
1: *tagged number of wobulators*	1: wobulator efficiency factor

In this example, the program in question doesn't need anything on the stack when it's executed; and it adds the "wobulator efficiency factor" to the stack. However, the "tagged number of wobulators" is also present on the stack so it appears in italic type.

Now for the operation of *this* program, it doesn't matter whether that "tagged number of wobulators" is on the stack or not; this particular program runs equally well with or without it. But in the overall flow of things, the quantity will be used by a later program, so I've included it for clarity's sake. (You can omit these unused-but-present quantities in your own documentation, or underline them or something.)

Command Names

In this book, as in the HP 48 calculator, commands are always capital letters:

```
PDIM COS RESET
```

Variable Names

HP 48 variables—and that includes programs, of course—can have astonishingly long, detailed names, and can contain letters and digits. You *can't* use dashes or periods though!

In this book, programs and subprograms are shown with an initial capital letter, followed by lowercase, or by caps and lowercase:

```
Sine1 Getvals AddSine
```

This is just a convention that is used in this book, and you don't have to follow it. You can make your variable names all uppercase letters, all lowercase, or any combination.

Names of Local Variables

Finally, *local* variables follow the same convention used in the owner's literature; that is, they're all lowercase:

```
a c max
```

You don't have to follow any of these conventions. But I find they work for me, especially in differentiating among variables and commands in long listings.

Structured Programming

Structured programming is good practice with any language from C to Cobol. But it's even more important when you're working with the HP 48. That's because the very act of punching all those tiny keys to enter a single long program is, frankly, arduous, and the frustration is great if you make a mistake.

In programming, the natural tendency is hit the ↰ « » key and begin throwing in commands. But as the program grows, it becomes harder and harder to find any errors that occur. Moreover, long programs are more difficult to edit, because you have to use keys like ▼ to move through the steps.

In structured programming, you break down each application into a series of tasks or jobs that are done in order. Perhaps you break down those tasks into simpler tasks. You arrange the tasks in the order to be performed, always working to reduce each task into a series of smaller ones.

You finally wind up with a list of very simple jobs, each of which can be accomplished in a few keystrokes. The list itself is your main program, and those individual simple jobs, or tasks, become your subprograms.

For instance, here's a list of tasks for an improved version of the Pythagorean theorem program:

1. Get the value of side *A*.
2. Get the value of side *B*.
3. Compute the hypotenuse, *H*.
4. Label the output.

Writing a Main Program

Using the list of tasks, you can write a main program that shows them all. Here's the task list for the Pythagorean theorem calculation, shown as a main program:

```
«
GetA
GetB
ComputeH
Label
»
```

With this program in the command line, save the program as *Hypotenuse*:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Hypotenuse STO	Stores the program as *Hypotenuse*.

You have created and saved the main *Hypotenuse* program. Even though you save it with a name that includes both uppercase and lowercase letters, the menu display shows only uppercase characters:

```
HYPOT
```

See how easy it is to understand the main *Hypotenuse* program? The program consists of nothing more than calls to other programs. In this book, we call them "subprograms," but there's nothing special about them. An HP 48 subprogram (or "subroutine") is actually just another program. Its name shows up in the menu display.

Using structured programming techniques, you break down your main application into a series of bite-sized subprograms. You can write each of the subprograms individually, test it to make sure it works correctly, then include it in the main program.

What happens if you run *Hypotenuse* now? Remember, none of the subprograms have been created yet. *Hypotenuse* is just a "shell" that shows individual subprograms and the order in which they will be executed. However, there's no problem in running *Hypotenuse*. When a program like *Hypotenuse* tries to execute a subprogram that is not available in the current directory (or a higher one), the calculator simply puts the name on the stack.

Keystrokes	Display
HYPOT	'GetA' 'GetB' 'ComputeH' 'Label'

Your next step is to write and store the individual subprograms.

Top-Down or Bottom-Up?

As you write your main program and subprograms, you have a choice of at least two basic strategies:

- Top-down programming.
- Bottom-up programming.

Top-Down Programming: In top-down programming, you write the main program—the task list—first. In it, you include all the names of the subprograms, in the order they're called. Using descriptive names such as *GetA* and *ComputeH* tells you exactly what each subprogram will do; it's like a notepad.

Bottom-Up Programming: In bottom-up programming, you create the subprograms first, then write the main program. The HP 48 is particularly well suited to this technique, because to "write" a subprogram name into the main program, you simply press its VAR menu key.

Whether you're using top-down or bottom-up programming, the procedure for writing individual subprograms is the same. You write the subprogram, store it, and test it before moving on to the next subprogram.

Let's enter the individual subprograms for the *Hypotenuse* program.

The GetA Subprogram

The *GetA* subprogram pauses execution, and gets a single value from the user. It takes nothing from the stack, and leaves a single number (the value of side *A* of a right triangle) on the stack when it finishes running.

Writing the Subprogram: You enter subprograms just as you do any program. (In fact, a subprogram is just another program in the eyes of your HP 48.)

Program Instructions	Comments
«	Begins subprogram.
"Input A"	Prompt string for INPUT command.
""	Command-line string (an empty string) for INPUT.
INPUT	Pauses program to get one value.
OBJ→	Converts from object to a number.
»	Ends subprogram.

Saving the Subprogram: To save this subprogram as *GetA*:

Keystrokes	Comments
ENTER	Puts program on the stack.
' GetA STO	Stores the program as *GetA*.

Testing the Subprogram: *GetA* doesn't need anything on the stack, and it leaves a single value there. Before going on to the next subprogram, you should verify that the current one works as it's supposed to. Press GETA to test it.

Keystrokes	Display	Comments
GETA	Input A	*GetA* prompts for input and halts.
3 ENTER	3.00	*GetA* leaves the value of *A* on the stack.

The GetB Subprogram

The *GetB* subprogram is similar to *GetA*; it gets a single value from the user, taking nothing from the stack. It leaves the value of side *B* on the stack.

Program Instructions	Comments
«	
"Input B"	Prompt string.
""	Command-line string.
INPUT	Pauses program to get one value.
OBJ→	Converts from object to a number.
»	

To save this program as *GetB*:

Keystrokes	Comments
ENTER	Puts program on the stack.
' GetB STO	Stores the program as *GetB*.

As with *GetA* (or any subprogram) you should test *GetB* before continuing. *GetB* should pause and prompt for input of a single value, then leave that value on the stack.

The ComputeH Subprogram

This is where the hypotenuse of a right triangle is computed. *ComputeH* expects values for sides A and B to be in levels 2 and 1 of the stack. It plugs these values into the Pythagorean theorem to compute the value of the hypotenuse, H. A single value (H) is left on the stack when the program is finished.

Program Instructions	Comments
«	
SQ	Squares the number in level 1 of the stack.
SWAP	Exchanges the contents of stack levels 1 and 2.
SQ	Squares the number now in level 1 of the stack.
+	Adds the squares.
√	Takes the square root to find the hypotenuse.
»	

Keystrokes	Comments
ENTER	Puts program on the stack.
' ComputeH STO	Stores the program as *ComputeH*.

To test *ComputeH*, you need to supply it with two values on the stack, then see that the correct hypotenuse is calculated. If you write and test your subprograms in the order they're called, the expected values should already be on the stack now, which makes testing of *ComputeH* easy. But if you like, you can supply the two values manually:

Keystrokes	Display	Comments
3 ENTER	3.00	Enters value for A.
4	4	Enters value for B.
COMP	5.00	Calculates correct hypotenuse.

The Label Subprogram

The *Label* subprogram takes the hypotenuse and adds a tag for easy identification. It removes one value from the stack, and returns that value, tagged with the letter "H."

Program Instructions	Comments
«	
"H"	Puts the letter H on the stack.
→TAG	Tags stack level 2 with the contents of level 1.
»	

Keystrokes	Comments
ENTER	Puts this program on the stack.
' Label STO	Stores the program.

To test this subprogram, make sure one number is on the stack, then press LABEL:

Keystrokes	Display	Comments
5 LABEL	H: 5.00	Labels hypotenuse as H.

Running the Complete Hypotenuse Program

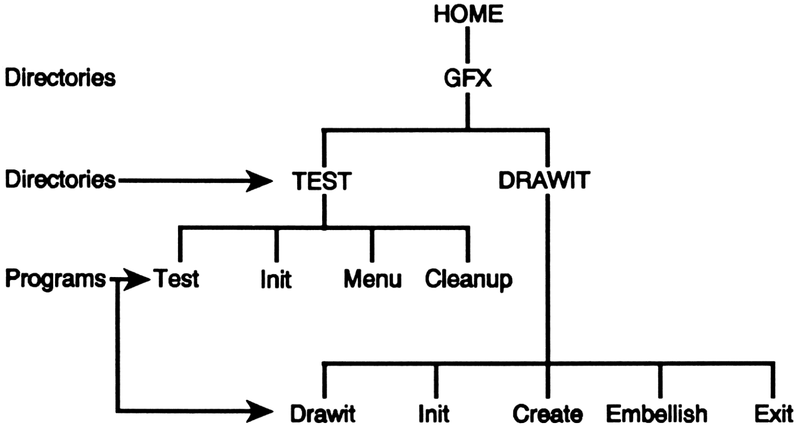
Now comes the acid test—running the *Hypotenuse* program. Because we've practiced structured programming, and tested each individual program in turn, there should be no surprises now. Run *Hypotenuse* to find the hypotenuse of a triangle with side A of 3 meters and side B of 4 meters:

Keystrokes	Display	Comments
HYPOT	Input A	Prompts for side A.
3 ENTER	Input B	Prompts for side B.
4 ENTER	H: 5.00	Program calculates and labels the hypotenuse.

Structured programming lets you create a long, fully-formed HP 48 application out of simple, easily-tested subprograms. Instead of trying to build a single long program, it's so much easier to create a group of "subprograms" or "subroutines," each consisting of just a few keystrokes. Then you can test each subprogram individually and get it working correctly before you proceed to the next one. When all subprograms are written, you combine them into one bodacious, world-beating application.

A Top-Down Programming Strategy

Using structured programming techniques means shorter, simpler individual programs—but it also means you'll have a lot of programs, variables, and other objects floating around. There are a number of strategies for managing programs. Here's one that works for me.



Each complete "application" is placed in its own directory. For instance, in the figure, *Test* and *Drawit* are applications, so each has a directory. (I make the directories all capital letters, though this isn't really important.)

Within the directory, there's a main program, named with the same name as the directory. It's always the first program in the directory, so it shows up on the left of the menu label display when you press (VAR). The other programs and objects in the directory are all subprograms—that is, they're all called, in one way or another, by the main program.

Pressing the main program key starts the application running. From here on, the main program takes over, calling each subprogram in turn, and prompting for input when necessary. Often, the main program displays a menu of choices. The whole point is that once you press the main program key, that program takes over completely.

For instance, if you're in the HOME directory and want to run *Test*, you press GFX TEST to get into the TEST directory. Then press TEST to run the *Test* program. From that point, *Test* takes over, prompting you for data, displaying menus for choices, and so on. You don't have to make any decisions about what subprograms to run, and in what order.

A Painless Programming Procedure

If you're careless when programming, you can easily wind up with your program in the wrong directory; or worse, you might even hit ATTN and lose all those valuable keystrokes. Here's a safe and pain-free procedure for creating an application program and its individual subprograms:

1. Before doing anything else, *make sure you're in the directory where you want to place the program*:

   - Use ↱ HOME for the home directory.
   - Use ↰ UP for the next level up.
   - Use VAR, NXT, and the menu keys to move downward through subdirectories.

2. Create the name of the program, and make it a directory. For example:

   ' TEST ↰ MEMORY CRDIR

3. Get into that new, now-empty directory:

   TEST

4. Store only the program brackets as a program with that name:

   ↰ « »
   ENTER
   ' Test
   STO

5. Now you have a “shell” program that you can fill in with keystrokes. Just use VISIT to get into the program and begin programming:

   ' TEST ↱ VISIT

Hints for Entering Programs

The HP 48’s tiny keyboard, while well-designed, isn’t going to win any touch-typing contests. Here are some helpful hints for getting your programs into the calculator.

Keying in Alphabetic Characters

To follow the conventions used in this book, you’ll need to use both uppercase and lowercase alphabetic characters. To get to locked alphabetic characters, you press the α key twice. To then switch between uppercase and lowercase, press ↰ α.

For instance, here’s how to enter lowercase alphanumeric characters, and how to switch back and forth between uppercase and lowercase:

- α α ↰ α gives you alpha-locked lowercase alpha, so the next characters you type are lowercase, like abc.
- ↰ α gets you back to uppercase alpha, so you can enter letters as DEF.
- ↰ α gets you back to lowercase alpha again: ghi.
- α gets you out of alpha mode.

With alpha mode locked, just use ↰ α every time you want to switch from uppercase to lowercase or vice versa.

Use Menus and Commands to Help

Use built-in menus to help you. To avoid the tedium of typing, use the menus and their commands as typing aids to put keystrokes into your programs. Some menu commands, such as those in the (PRG) BRCH menu, help you remember what options are available for commands such as FOR and WHILE.

Use All the Alpha Keys

Remember, you're not limited to the characters shown on the face of the HP 48. There's a whole slew of special characters, including ones such as x̄ and ∞, that you can use to make your program's user instructions, screens, and output labels even more accurate and easy to read.

These special characters are shown in the owner's manual and on the back of the calculator's quiçk reference guide. To "type" one of these special characters, make sure the calculator is in alpha mode, then press the right or left shift key ((↱) or (↰)), followed by the key that gives your the special character. For instance, here's how you'd get the string `"From here to ∞"` into the display:

Keystrokes	**Display**	**Comments**
(↱) (" ")	`""`	Starts a text string.
(α) (α)		Selects alpha mode.
F	`F`	Enters first letter as uppercase.
(↰) (α)		Switches to lowercase alpha.
rom here to	`"From here to "`	Enters next letters as lowercase characters (end with a space).
(↱) (CST)	`"From here to ∞"`	Types the infinity symbol.

Program Names and How They're Used

Commands in programs must be capital letters because that's what the HP 48 expects. If you enter the word Eval into a program, the calculator won't execute the EVAL command; instead, it will look for a program or other object named *Eval*.

The HP 48 is picky about program names, too. If you enter a program and name it, say, *Init*, that name appears in the menu display as INIT . Well, that's the way the HP 48 displays the name, but internally it's still *Init*. And if you try to call a subprogram called *INIT* (or *INit*, or *init*) from another program, you're going to come up empty- handed. Your calling program *has* to call *Init*. Don't say you haven't been warned!

Actually, it's pretty easy to make sure you call a subprogram by its correct name: just press its menu key whenever you want to "write" the name. Menu keys make it easier to manage variables and programs with long descriptive names because they can "key in" the program or variable name for you.

What about length, anyhow? The menu key display shows only four or five characters (depending on how fat the characters are), so you may want to limit variable and program names to that length. Personally, I find it's easier to use longer, more meaningful names, even though I don't see the entire name in the menu display.

But think about what happens if you want to select *Program3* from the keyboard from among the following programs:

Program1
Program2
Program3
Program4
Program5
Program6

The menu display looks like this:

PROG PROG PROG PROG PROG PROG

You're better off naming these programs *Pgm1*, *Pgm2*, etc.

Another consideration, if you're moving programs to and from the HP 48 and your personal computer, is how those names will look in the new environment. A Macintosh computer, of course, doesn't limit names; but your DOS-based PC limits HP 48 names to 8 characters. So the HP 48 variable *Mygraphics*, for instance, becomes *MYGRAPHI.* (not *MYGRAPHI.CS*) when it's brought into the DOS environment.

Faster Character Deletions

When it comes to character-by-character deletions, ← is somewhat faster than DEL. Move the cursor to the *last* character you want to delete and use ← to go backwards through the program commands.

Working with Directories

Particularly with structured programming, where each main program is made up of several subprograms, the number of programs in your HP 48 can become very large. Directories are essential for managing programs and other objects—and for preserving your sanity as well.

Creating a Directory

To create a directory:

1. Press ' to start the directory name.
2. Key in the alphabetic characters for the name. (Or place the name on the stack.)
3. Press ↰ MEMORY CRDIR to create the directory.

Examining a Directory's Contents

To see just the abbreviated names of objects in a directory, press VAR followed by the directory's menu key to get into the directory. Then use NXT to review the names shown by the menu keys.

To display the names of menu pageful of objects in a directory, press ↰ REVIEW; this shows you the object names on the HP 48's liquid crystal display screen. When you're done examining one page, press ATTN to return to the stack display.

You can also see the full names and contents of every program in the directory, viewing them all at once. Here's what to do:

1. Press ↰ UP to get to the directory *above* the one in which the programs are located.
2. Press ↱ followed by the directory name. For instance, to see the contents of a directory called SFX, press ↱ SFX. This brings the directory and its contents into the stack. You'll see the heading DIR, followed by the name and contents of each individual program.
3. Press ▼ to scroll down through the programs. The display shows the word DIR, followed by the first program name, followed by the program itself. The display might begin something like this:

   ```
   DIR
   Alarm
   «
   DO 200.00 1 BEEP 0.50 WAIT
   ```

4. When you've finished, press ATTN (the ON key) or ENTER.

Reordering a VAR Menu

Each time you create and name a new object, including a program, it goes to the beginning of the current directory's VAR menu list. If you're doing top-down programming—in which you create the main program first—your main program will always be at the tail end of the directory list, making it somewhat hard to find.

It's an easy matter, though, to *reorder* variables in the menu list so you see the main program first, followed by the first program called, and so on. Here's how to do it:

1. Use ↰ MEMORY VARS to create a list of all variables in the directory.
2. Press ↰ EDIT (or ▼) to edit the list. It's easiest to erase variable names with DEL or ◀. Use the VAR menu and press individual menu keys to save time when you want to "type" names.
3. Press ENTER to put the edited list on the stack.
4. Use ↰ MEMORY ORDER to order the list. ORDER places programs and other objects into the menu in the order you've

chosen. Objects not in the list aren't deleted. They're simply added at the end, so their menu keys will appear later, perhaps on a later page of the menu.

There's an even easier procedure to use. Suppose you have a large number of variables, but care about seeing only the first few; the order of the rest doesn't matter. In this case, create a list using just those variable names, and use it as the argument for ORDER:

1. Press ↰ { } to start the list.
2. Use the VAR menu, and press the first menu key you want to see, then the second, and so on.
3. Press ENTER to place the list on the stack.
4. Press ↰ MEMORY ORDER to order the list. The variables in the list will now appear beginning on the left side of the first page of the VAR menu; all other variables in this directory will appear after them.

If you have a large number of variables, this second procedure can be significantly faster, since the HP 48 is actually reordering a small number of variables instead of all of them.

Working with Programs

Editing and copying an existing program is usually easier than writing an entirely new one. And there are plenty of times when you'll want to copy a program from one directory to another, rename a program, or move it.

Viewing and Editing a Program's Contents

To view and edit the contents of a program—or any variable—you can:

1. Press ' and the variable name.
2. Press ↱ VISIT to get into the program.
3. Do your editing.
4. Press ENTER to save the edited program.

When using VISIT to edit a program, remember this:

- To exit and save your changes, press ENTER.
- To exit without saving the changes (preserving the previous program version), press ATTN (the ON key).

Deleting Programs

To purge a program (or any other object):

1. Press ' to put a pair of tick marks in the command line.

2. Press the menu key for the program. This puts the program name between the tick marks (for example, 'Bigprog').

3. Press ↰ PURGE.

The safest way to purge a directory is to use PURGE, which means that you must first delete all the individual programs and other objects in it. To delete several programs at once, you can put them into a list, then purge the entire list:

1. Press ↰ { } to begin the list.

2. Press the menu keys for the individual programs or other objects. Put them between the braces. For example, the display might look like this: { Bigprog Littleprog Testprog }

3. Press ↰ PURGE. All the objects in the list are deleted.

Copying, Renaming, and Moving Programs

On the HP 48, copying, renaming, and moving a program or other object are quite similar. The general procedure is as follows:

1. To begin, put the program contents into the stack by pressing the ↱ key, followed by the program's menu key.
2. Then press the ' key, and enter a name between the ' ' tick characters. (If you're copying, you can simply press the program's menu key to enter the name.) Now the program's contents are in level 2 of the stack, and the ultimate name is in level 1.
3. If it's a move or a copy, use ↰ UP and VAR to get to the directory where you want to place the program.

4. Press [STO] to store the program under the name you specified (that is, the name between the tick marks).

5. If you're renaming or moving the program, purge the original program. (Go back to its original directory, press ['] followed by the program name, then press [↰] [PURGE].)

Example: Renaming a Program. To rename a program, you give it a different name within the same directory. For example, here's how to rename the program *Myway* to *Yrway*:

1. Press [↱].

2. Press the MYWAY menu key. This brings the *contents* of that program into the stack.

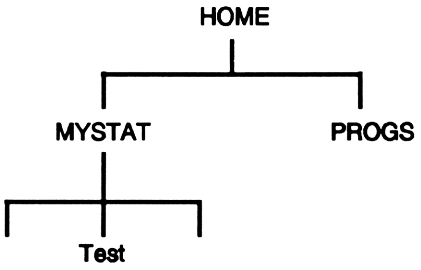
3. Press ['] to place double tick marks in the command line.

4. Enter the name Yrway between the tick marks.

5. Press [STO] to store the program under that name. At this point, there are two copies of the same program: *Myway* and *Yrway*.

6. Press ['] MYWAY again. Then press [↰] [PURGE] to delete *Myway*, leaving only *Yrway*.

Example: Moving a Program. Moving is similar to renaming. For instance, suppose you want to move the program *Test* from a directory called MYSTAT to the directory called PROGS:



The procedure is as follows:

1. Press [↱] [HOME] followed by the MYSTAT menu key to get into the MYSTAT directory.

2. Press ↱ TEST to put the contents of the *Test* program on the stack.

3. Press ' and enter the new program name. If it's going to be *Test*, you can press TEST instead of typing the characters.

4. Press ↱ HOME followed by PROGS to get into the PROGS directory. Even though you've changed the current directory, the contents of *Test* are still in level 2, and the name is in level 1.

5. Once you're in the new directory, press STO to store the program there.

6. Now you have two copies of the same program. If you don't need the original one, go back to the MYSTAT directory, bring the original *Test* into the stack by pressing ' TEST , then press ↰ PURGE to delete it.

What about Entire Directories?

You can copy, rename, and move entire directories at once. Just bring the entire contents of the directory into the stack with ↱ followed by the directory name. Then move or rename those contents as you would a single program.

The safest way to erase a directory is to use ↰ PURGE to purge all programs and other objects from that directory, Then purge the empty directory itself.

To erase a *full* directory all at once, put its name on the stack, then execute the PGDIR command (↰ MEMORY NXT NXT PGDIR. However, be *very* careful using this command—make sure you know what's in the directory before you erase it!

Debugging

The HP 48 has several features that help in writing and debugging programs.

Using Subprogram Calls

Remember that if a subprogram doesn't exist when called, the HP 48 does not produce an error. Instead, it merely puts that name on the stack. As you write structured programs with many calls to subprograms, you may want to use top-down programming and purposely leave some subprograms undefined until they're written.

Single-Step Debugging

An invaluable aid in debugging is running a subprogram one step at a time to see how it works. After each step, you usually see the stack, so you can examine—or even change—the current value before proceeding.

To single-step through, say, a subprogram called *Test*:

1. Press (') TEST to put 'Test' on the stack.
2. Press (PRG) CTRL DBUG.
3. Press SST to single step through the program.
4. Use KILL to abandon program execution. This doesn't affect the program itself.

SST appears on the menu line, of course. What if your program displays its own custom menu at some point, and you no longer see SST? Just press (PRG) CTRL to get back to the program control menu, then resume single-stepping with SST.

Single Step and Subprograms

When single-stepping through a main program, remember that subprograms are *executed* when called by the SST key. Thus, each press of SST executes the next subprogram. To see every step of the main program *and* the steps of each individual subprogram, use SST↓ instead.

General Programming Hints

Keep these hints and tricks in mind as you write your own programs.

How to Halt a Runaway Program

If you want to stop a running program and return to the stack display, press ATTN (the ON key). You can use this technique almost any time, even to halt a runaway program or an endless loop.

If you're really in trouble—a program has taken over the keyboard and has locked out all keys, for example—press ON and the C top-row key *at the same time*.

How to Manually Test a Flag

Need to know the status of a system flag or a user flag? Here's how you can determine it quickly:

1. Enter the flag number.
2. Execute PRG TEST NXT NXT FS? . If the flag is set, you'll see a 1 on the stack. If it's clear, you'll get a 0.

For instance, to test system flag −56, which gives the status of the error and BEEP command beeps:

Keystrokes	**Display**
56 +/-	`-56`
PRG TEST NXT NXT FS?	`0.00`

Thus, user flag −56 is clear, which means error beeps and the BEEP command are on.

How to Manage Memory Usage

With a calculator as powerful and flexible as the HP 48, there are often dozens of ways to accomplish any objective. In this book, we've haven't always taken the most direct or memory-saving route. Instead, we've opted for clarity, even if it chews up a few more bytes.

This is particularly true when it comes to variable names and user instructions, which we've made long and descriptive.

To check the available memory, use ⬅ MEMORY MEM . If this command shows you're running low on memory, or if you start getting Out of Memory or Insufficient Memory messages, you'll have to take action. Here are some tricks you can use to save those valuable bytes:

- Slash variable names to reduce the number of characters.
- Reduce user instructions.
- Use local variables, which go away after you've used them, instead of global variables (which stick around).
- Don't use recursive programs. (A recursive program is one that calls itself.)

Another option, if your calculator permits it, is to add memory. The procedure is fast and easy, and need not affect the programs and variables you now have in your HP 48.

Never Assume Anything

Never assume the calculator will be "set up" a particular way when you run a program. If your program needs degrees mode or should display answers to two decimal places, be sure to specifically set all those parameters—or prompt the user to do it.

Consider Different Object Types for Different Tasks

Remember the different types of objects available in the HP 48. For instance, a program is surrounded by « and », while a list has { and }. A list lets you combine objects—including programs—for manipulation.

Remember the Order of Evaluation

Keep the following evaluation rules in mind when you're entering formulas. The HP 48's order of precedence, from first to last, is as follows:

1. Parentheses (working from inside out).
2. Functions (SIN, LOG, etc.).
3. Factorial.
4. Power and square root.
5. Negation, multiplication, division.
6. Addition and subtraction.

If in doubt, use the Equation Writer to enter equations. It takes care of most precedence problems for you.

Maintain Good Housekeeping

Use local variables whenever you can. Local variables are used only by the current program, and vanish upon exit. They keep your directories from filling up.

If your program generates lots of variables, including those that are automatically generated by the calculator's own functions (such as PPAR), you may want to write a short routine to purge them upon exiting. Otherwise, your directories are going to be so full that looking for a particular program name will be like finding a needle in a haystack.

Besides purging variables, your programs and subprograms should take into account what they do to the stack. Sometimes, of course, you'll want a result or series of results left on the stack. But if your program has left unneeded items on the stack, you can DROP the stack to eliminate them upon exit.

Don't Use Loops Unless They're Necessary

An HP 48 "program" doesn't have to run continually to be effective. For instance, if you've programmed computers in another language, such as BASIC, you know that to get a menu display, you need to set up a continuous loop. With the HP 48, however, you can eliminate menu loops, since the calculator itself can display menus.

Put Messages in Catalogs

If you're writing long programs with lots of messages, you may want to do what the pros do: make up a "catalog" of messages, and call each one by its number when you need it. By putting all your messages in one place (a list works particularly well), you make it easier to check meaning, spelling, and grammar. It's also easier to compare them for consistency, or even translate them into other languages.

A Final Word

Remember, most (all!) of the programs in this book can be modified or improved. And you're encouraged to do so. Modifying and changing programs is a great way to learn.

3

Business and Finance

Dedicated business calculators (especially those from Hewlett-Packard) are well-known in the business and financial community. And with the right programs, the HP 48 can handle virtually anything that can be done on one of these dedicated business machines. Thanks to the HP Solve application, TVM (time, value, money) calculations are remarkably easy, and the added benefits of the HP 48's sophisticated graphics and analysis can help you create a peerless business calculator.

This chapter shows a few programs for use in business and finance. As you'll see, the programs in this chapter aren't complex, yet they're very useful.

Hypertext, The Information Dispenser

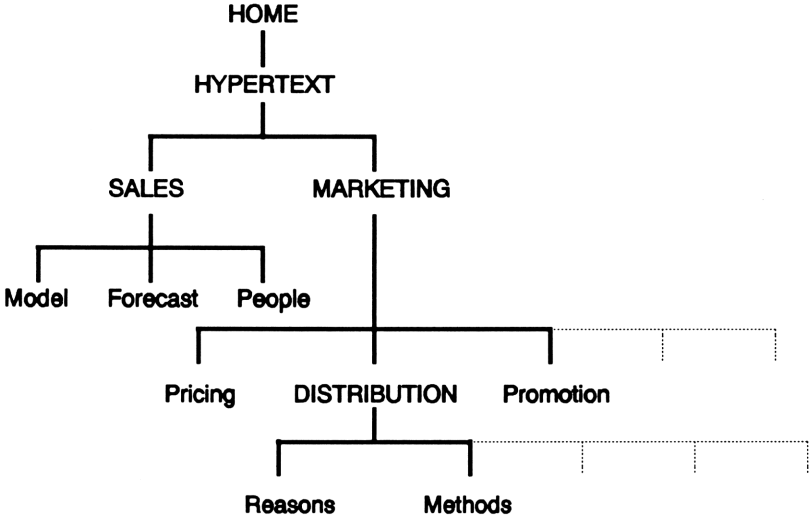
Giving an important speech or presentation? Need a few memory-jogging notes to get you through your next meeting? The HP 48 can be a pocket-sized dispenser of valuable and timely information, a veritable fount of knowledge at your fingertips. With its excellent memory and comparatively large display, this calculator can handle, organize, and display all kinds of data. You begin by writing a series of "notecards" containing the notes, text, graphics, or formulas you'll need. Then when you go to that meeting or presentation, you whip out the HP 48, go into the HYPERTEXT directory, and start pressing menu keys—which you've thoughtfully prearranged in the order you'll need them. The menu display means you can take topics in any order, too, ready to field questions or provide answers.

The Hypertext Directory

Begin by creating a directory for HYPERTEXT, then get into that directory:

' HYPERTEXT
↰ MEMORY CRDIR
VAR
HYPER

The HYPERTEXT directory is where you'll put all the subdirectories and individual "notecards." Inside this directory, you can create a subdirectory for each topic. Within each topic, you can place notecards—and more subdirectories, if desired—in the order you'll need them.



If you use a structure like that shown in the figure, for instance, you merely stride into that marketing meeting, pull out your HP 48, and press the MARK key. This gets you into the MARKETING subdirectory. Within this subdirectory, your notecards are laid out in order for your presentation:

PRICI DISTRI PROM

You press the PRICI key, and key pricing information appears frozen on the HP 48's display. You confidently begin your speech, referring to your notes as necessary. Distribution is a deeper subject, so when you press DISTRI, you have several entries for it. It's like outlining, only you're using directories and subdirectories instead of topics and subtopics.

Get the idea? Now let's look at an example of a hypertext notecard.

Pricing—Displaying Text

Pricing is a notecard that displays text. It's really a short program that pauses to display one screen of text, moves to the next screen when you press a key, then ends. *Pricing* has no overall effect on the stack.

Program Instructions	**Comments**
«	
CLLCD	Clears calculator display.
"PRICING DECISION: 1.Define tgt mkts. 2.Estimate mkt potntl. 3.Develop positioning. 4.Design mktng mix. 5.Estimate price elasticity of dmnd."	Be sure to end each line with an endline character ((⮫) (↵)) to get a display as shown here.
1 DISP	Displays text string beginning on row 1.
7 FREEZE	Freezes display until next key press.
0 WAIT	Waits for a key press.
DROP	Throws away the key address generated by WAIT.
CLLCD	

Program Instructions **Comments**

```
"6.Estimate rlvnt csts.
7.Analyze environment
factors.
8.Set pricing objctvs.
9.Develop price
structure."

1 DISP

7 FREEZE

0 WAIT

DROP

»
```

To save this notecard program as *Pricing*:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Pricing STO	Stores the program.

Pricing is a typical notecard program. It begins with a CLLCD (*clear LCD*) command that gets rid of any previous information from the calculator display. Then a long text string is placed on the stack. If you enter this string by typing, be sure to place an endline character (press ↱ ↵) at the end of each text line shown in the listing.

With a text string on the stack, the number 1 is entered next. This puts the necessary arguments for DISP on the stack: the text string in level 2, and the number 1 in level 1. When DISP is executed, it displays the text beginning on the left side of row 1 in the display—in other words, it fills up the calculator's display screen.

How large can we make each screen of text? DISP can display up to seven rows of text on the screen at once. The number of words you can get in each row depends on whether the letters themselves are fat (like M and D) or thin (such as I and t). A good rule of thumb is to use about 17-20 characters per row.

Some words may appear to trail off the edge when you're typing them in. Because of the different size of the characters in the display, though, some of these "lost" characters may actually be visible when the program is run.

After the DISP command, we execute `7 FREEZE`, which freezes the entire display until the next key press—or until the next DISP or graphics command. This is an important point. We need some way to halt the program, allowing display of that first screen, before we move on to display the second *Pricing* screen.

For this we use WAIT, with 0 as its argument. WAIT normally pauses execution for a specified number of seconds; thus, `3 WAIT` gives you a three-second breather, `10 WAIT` pauses a program for 10 seconds, and so on. The 0 argument works a little differently, though. The `0 WAIT` command waits for the press of any key before continuing. If you don't press a key, the program will be paused forever.

Here's another advantage to using WAIT here: the key you press isn't executed. It's swallowed up by WAIT, so you don't have to worry about annoying error beeps during your critical presentation.

Using `0 WAIT` does have one effect that we need to handle, however. Although it doesn't execute the key you press to continue, WAIT *does* put the keyboard address of the key on the stack. So if you press the [ENTER] key, you wind up with `51` on the stack. For this reason, right after executing `0 WAIT`, we use DROP. This drops the stack one level, which gets rid of that key address.

In the case of *Pricing*, there are two screens of information, so the program goes through the entire sequence twice. It clears the display with CLLCD, displays a window of text using `1 DISP`, freezes the display with FREEZE and waits for a key press with WAIT. When you press any key to continue, the program executes DROP to rid the stack of the key address.

Pricing shows two screenfuls of text, then ends. You can, if you like, put all the notes for an entire 30-minute presentation in one program, so that successive presses of any key bring up screen after screen. But this makes it difficult to change the order, or to refer back to your earlier notes. You're usually better off with many programs containing short one- or two-screen displays.

Running the Pricing Notecard

To run a notecard such as *Pricing*, just press the menu key and view the text.

Program Prompt or Display **Your Action**

PRICI

```
PRICING DECISION:
1.Define tgt mkts.
2.Estimate mkt potntl.
3.Develop positioning.
4.Design mktng mix.
5.Estimate price
elasticity of dmnd.
```

Now hit any key to see the next screenful of text for this notecard.

Program Prompt or Display **Your Action**

[ENTER] (or any key)

```
6.Estimate rlvnt csts.
7.Analyze environment
factors.
8.Set pricing objctvs.
9.Develop price
structure.
```

Figuring Depreciation

In business, when you purchase an asset like a building, a computer, or a vehicle, its value gradually lessens over a period of time. This lessening of value is known as depreciation, and it's used in figuring business expenses and for tax purposes.

To figure depreciation, you need to know the *cost* of the asset, its *life* (how many years before it has no value left), and the *method* of figuring depreciation. For tax purposes, an asset falls into a class with a specified life; for instance, a computer's class life is five years, while the class life for a building is 31.5 years.

There are several different formulas for depreciation. The simplest is called the *straight-line* method. Using straight-line depreciation, an asset loses a fixed percentage each year of its life. For example, a computer with a life of five years loses 20% of its value each year. Thus, if the computer costs $1000, it depreciates by $200 the first year, $200 the second year, and so on. The formula for the amount of straight-line depreciation for any one year is:

$$\frac{cost}{life}$$

Another method of depreciation uses the *declining-balance* formula, in which the undepreciated balance (known as the book value) is reduced by a certain percentage each year. This method is often used by businesses because it places the largest amount of depreciation in the first years of ownership. The formula for one year's amount of double-declining-balance (or 200% declining balance) depreciation is:

$$\frac{2c}{n}\left(1-\frac{2}{n}\right)^{y-1}$$

where

c = cost of the asset
n = number of years in class life
y = this year

For the $1,000 computer with a class life of five years, this results in a depreciation of 40%, or $400, the first year, 24% the second year, and so on.

In figuring depreciation, it's also useful to know how much of an asset's value (its book value) remains after deductions. For double-declining-balance depreciation, this is given by the following formula:

$$c\left(1-\frac{2}{n}\right)^{n}$$

The *Depreciation* program and its associated subprograms let you choose your type of depreciation. The program then prompts you to input the cost, class life, and the year of the asset's life in which the depreciation occurs. Finally, you see a display with everything you need to know.

The Depreciation Directory

To create a directory for the depreciation programs, then get into that directory:

' DEPRECIATION
↰ MEMORY CRDIR
VAR
DEPRE

The DEPRECIATION directory will hold all programs and subprograms for calculating depreciation. Any objects (including programs and equations) that you enter and save will be placed in this directory.

The Main Depreciation Program

The main *Depreciation* program changes the key menu to show only two keys: STLIN and DDBAL. Depending on which key you press, *Depreciation* calls three subprograms to figure straight-line depreciation, or it calls three subprograms to figure depreciation using the double-declining-balance method. The main program takes nothing from the stack and leaves nothing there.

Program Instructions	Comments
«	Start of *Depreciation* program.
{	Starts key list used by TMENU.
{	Begins defining list for first menu key.
"STLIN"	First menu key display.
«	Begins defining program for first menu key.
Getvals	Calls subprogram to get needed values.
Stline	Calls subprogram to figure straight-line depreciation.
Output	Calls subprogram for output.
»	Ends first key procedure.
}	Ends list for first menu key.

Program Instructions	**Comments**
`{`	Begins defining list for second menu key.
`"DDBAL"`	Second menu key display.
`«`	Begins defining program for second menu key.
`Getvals`	Gets needed values.
`Ddbalance`	Figures double-declining balance depreciation.
`Output`	Shows output.
`»`	Ends defining program for second menu key.
`}`	Ends defining list for second menu key.
`}`	Ends key list for TMENU.
`TMENU`	Uses the above list to create two menu keys.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Depreciation STO	Stores the program as *Depreciation*.

The main *Depreciation* program uses the TMENU (*temporary menu*) command to define two menu keys that will appear in the display. For its argument, TMENU needs a list that contains menu key labels and definitions, in the following format:

{ {"label1" *object1*} {"label2" *object2*}...}

The list used by TMENU is actually made of several internal lists. Each list contains a label for the key and an object—that is, an action to be taken if that key is pressed.

In the case of *Depreciation*, the list specifies just two keys. If you press STLIN, you call the subprograms *Getvals*, *Stline*, and *Output*. Press DDBAL and you summon *Getvals*, followed this time by *Ddbalance*, and finally *Output*. Notice that each key can execute only a single object. Thus, to call more than one subprogram with each key, you have to place each set of subprogram names in its own program; that is, you place the names between program brackets within the list.

Getvals—Labeling Input from the Keyboard

The *Getvals* subprogram is the first one called, whether you press STLIN or DDBAL. *Getvals* prompts for keyboard input of values for cost, class life, and the year. It tags these values and leaves them on the stack for later use by *Stline* or *Ddbalance*.

Arguments	Results
3:	3: tagged cost
2:	2: tagged life
1:	1: tagged year of life

Program Instructions	**Comments**
«	
"Input the cost or basis of the asset"	Prompt string for INPUT.
":Cost: "	Tag (on command line during INPUT).
INPUT	Pauses for input of cost, displaying above messages.
OBJ→	Converts keyboard input into a tagged number.
"Input the class life in years"	Prompt string.
":Life: "	Command-line tag.
INPUT	Pauses for input of class life.
OBJ→	Converts to tagged number.

Program Instructions	Comments
"Enter this year (1 for 1st, 2 for 2nd, etc.)"	Prompt string.
":This year: "	Command-line tag.
INPUT	Pauses for input of year.
OBJ→	Converts to tagged number.
≫	

When you enter the commands for *Getvals*, be sure to place (↱ ↵) to end lines within the message strings as shown. Otherwise, when you run the program, lines of message text may extend past the edges of the display screen.

To save this subprogram:

Keystrokes	Comments
ENTER	
' Getvals STO	Stores the program as *Getvals*.

Getvals is really nothing more than three INPUT commands, one after another. Together, they leave three tagged values on the stack when *Getvals* is finished.

Let's look at how one of these INPUT commands works. INPUT takes as its argument two strings from the stack. In level 2 is the prompt string, which in this case can be up to three lines of text. In level 1 is the command-line string. When INPUT is executed, it takes these two quantities from the stack, displays them, and waits for input.

When you enter a number from the keyboard and press ENTER, the INPUT command actually combines the command-line string and your input. By placing a leading and trailing semicolon around the command line, we create a tagged object—without using the →TAG command.

Thus, for a three-year old asset with a cost of $1,000 and a class life of five years, *Getvals* leaves the following quantities on the stack:

Stack Level	Contents
Level 3:	Cost: 1000.00
Level 2:	Life: 5.00
Level 1:	This year: 3.00

Because each of these quantities has been converted to a number with OBJ→, you can use them as you would any number. You can add, subtract, multiply, divide, just as if the number stands alone. As you'll see, you can easily convert a tagged value to a string, too.

Stline—Using Local Variables in Formulas

The *Stline* subprogram takes three values (cost, life, and year) from the stack, and uses them to figure depreciation by the straight-line method. It leaves four quantities on the stack: a string, "St line"; the tagged cost of the asset; the amount of this year's depreciation; and the amount remaining to be depreciated.

Arguments	Results
4:	4: "St line"
3: tagged cost	3: tagged cost
2: tagged life	2: tagged depreciation
1: tagged year of life	1: tagged remaining value

Program Instructions	Comments
`«`	
`→ cost life year`	Creates local variables.
`  «`	Begins defining procedure for local variables.
`  "St. line"`	Places this string on the stack.
`  cost`	Places the value of this variable on the stack.
`  'cost/life' EVAL`	Calculates depreciation and places it on stack.
`  "This year" →TAG`	Adds tag to depreciation value.
`  'cost-year*cost/life' EVAL`	Calculates remaining value.
`  "Remaining" →TAG`	Adds tag to remaining value.
`  »`	Ends defining procedure for local variables.
`»`	Ends program.

To save the program:

Keystrokes	Comments
ENTER	Puts subprogram on the stack.
' Stline STO	Stores the *Stline* subprogram.

Using local variables in this subprogram makes it much easier to follow than if we'd done all these depreciation calculations on the stack. The program begins by taking three quantities from the stack and creating the local variables *cost*, *life*, and *year*. Because we've used meaningful names, anyone who comes across this program, say, 100 years from now will know what values it needs.

In order to create local variables, we have to *immediately* follow their declaration with a defining procedure. Here, as is often the case, the defining procedure is a program, enclosed between program brackets.

The defining program first places the string "St line" on the stack. Our *Output* program is going to use this, and it will also use the value for *cost*, which we place on the stack after the string.

The formula for any year's depreciation is the cost of the asset divided by the number of years of its life. It's a simple matter to turn this into the formula 'cost/life', then perform that calculation with the EVAL command. We tag the result using the "This year" tag and the →TAG command.

Similarly, the formula for the asset's remaining value (another quantity that's prized in accounting and tax circles) is given by the cost minus the total depreciation to date; or 'cost- year*cost/life'. Again, within the program we use EVAL to get a result from this easy-to-read formula, and we tag the result with the string "Remaining" for later use.

Ddbalance—Different Names for Local Variables

Ddbalance is very similar to *Stline*; it takes the cost, number of years of total life, and age in years from the stack. (They were placed there by *Getvals*, remember?) *Ddbalance* uses these quantities to calculate the amount of depreciation by the double-declining-balance method, and returns four quantities to the stack: a string describing the method, the cost, the depreciation for this year, and the amount remaining for depreciation in future years.

Arguments	Results
4:	4: "200 DB"
3: tagged cost	3: tagged cost
2: tagged life	2: tagged depreciation
1: tagged year of life	1: tagged remaining value

Program Instructions	Comments
`«`	Begins program.
`→ c n y`	Creates local variables.
`  «`	Begins defining procedure for local variables.
`  "200 DB"`	Puts string on stack.
`  c`	Puts value of *c* (*cost*) on stack.
`  '2*c/n*(1-2/n)^(y-1)’`	Formula for this year's depreciation.
`  EVAL`	Calculates depreciation, returns result to stack.
`  "This year" →TAG`	Tags depreciation result.
`  'c*(1-2/n)^y' EVAL`	Calculates remaining value.
`  "Remaining" →TAG`	Tags remaining value.
`  »`	Ends defining procedure.
`»`	Ends program.

Keystrokes	Comments
ENTER	
' Ddbalance STO	Stores *Ddbalance* program.

As in the *Stline* program, *Ddbalance* converts the three values on the stack to local variables before doing anything else. Those values were placed on the stack by the preceding program, and they're always in the same order, of course. But compare *Ddbalance* and *Stline*; notice that although each expects the same values from the stack, we use different variable names to refer to them within the subprogram. For instance, in *Stline* the variable is *cost*, while in *Ddbalance* it's *c*.

Here the local variables are called *c* (for *cost*), *n* (for *total number of years of life*), and *y* (for *this year*), to correspond to the formulas we're using. Because local variables are lost when the program is finished running, you can use whatever names you want within the program.

As with *Stline*, when *Ddbalance* is finished running, it leaves four quantities on the stack: a string (this time the string is "200 DB"); the tagged cost of the asset; the tagged amount of depreciation for this year; and the tagged remaining value of the asset after this year. Now we're ready for the *Output* subprogram.

Output—Combining Answers into a Single Display

The *Output* subprogram takes four values from the stack, converts them to local variables, and combines them to create a detailed, easy-to-read output display. It also leaves the tagged values for this year's depreciation and remaining value on the stack.

Arguments	Results
4: identifying string	4:
3: tagged cost	3:
2: tagged depreciation	2: tagged depreciation
1: tagged remaining value	1: tagged remaining value

Program Instructions	Comments
`«`	Begins program.
`→ method cost depcn rvalue`	Creates local variables.
`  «`	Begins defining procedure for local variables.
`  "DEPRECIATION`	Begins text string for use by DISP.
	Blank line (that is, a newline character)
`  "`	End of text string.
`  cost`	Puts local variable *cost* in stack.
`  →STR`	Converts to a string.
`  +`	Adds the two strings.

Program Instructions	Comments
"	(Put a newline character after the quotation mark.)
" +	Adds the newline character to the string.
":Method: " +	Adds this word to the string.
method	Places the string from *method* in the stack.
+	Adds the two strings.
"	(Again, put a newline character after the quotation mark.)
" +	Adds the newline character to the string.
depcn	Places the value of *depcn* in the stack.
→STR +	Converts value to a string, and adds it.
"	(Put a newline character after the quotes.)
" +	Adds another newline.
rvalue	Gets the value of *rvalue*.
→STR +	Converts to a string, and adds to the display string.
CLLCD	Clears the calculator's liquid crystal display.
1 DISP	Displays string beginning on line 1.
7 FREEZE	Freezes all parts of display until next key press.
depcn rvalue	Leaves two values on the stack.
»	Ends defining procedure for local variables.
»	Ends program.

To save the *Output* subprogram:

Keystrokes	Comments
ENTER	
' Output STO	Stores the subprogram.

In this program, we take the four quantities passed from the preceding program (either *Stline* or *Ddbalance*) and put them in a long string for display by DISP. The DISP command uses as its argument a string in stack level 2, and a number in level 1. The string is what's displayed, and the number tells DISP *where* (that is, what line of the display) to begin showing it.

To create the display string, we begin with a headline, "DEPRECIATION". Before we end this string, though, we add a couple of newline characters (↱ ⏎), which has the effect of adding a blank line before beginning a new one.

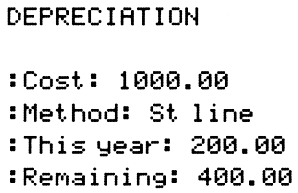
Next we bring the value of the *cost* variable into the stack. The value is a tagged number, so both the tag and the value appear. When the →STR command is executed, it converts the *entire* quantity—including its tag—to a string. If *cost* is $1000, the stack looks like this:

Stack Level	Contents
Level 2:	"DEPRECIATION
	"
Level 1:	":Cost: 1000.00"

Thus, when + is executed, the string in level 1 is added to that in level 2, creating one single long multi-line string. We add another newline character, then bring the rest of the variables into the stack and add them to the string, too.

Notice that we add the word "Method" to the string before adding the value of the variable *method*. Also notice that because "Method" is *already* a string, we don't have to use →STR on it.

After we've added all the necessary labels and the four variables to the display string, we execute 1 DISP to exhibit that string beginning with line 1 of the HP 48's display. The display looks something like this:



```
DEPRECIATION

:Cost: 1000.00
:Method: St line
:This year: 200.00
:Remaining: 400.00
```

If we didn't throw in a FREEZE command here, you'd see the display only for a fleeting moment. By adding 7 FREEZE before exiting the program, we guarantee the display will remain until the next key press.

Just before exit, we use the local variables ***depcn*** and ***rvalue*** to place these two quantities on the stack. They're tagged, of course, for easy identification, so you can use them in other calculations.

Running the Depreciation Program

To run the *Depreciation* program, just hit the DEPRE key. This presents you with a pair of menu choices, STLIN and DDBAL. Then you can use the program to compare depreciation using these two alternatives.

For instance, suppose you purchased a computer for $3250, and you want to see how much depreciation to allow in the first year. Its life is five years.

Program Prompt or Display	Your Action
	DEPRE
	STLIN
Input the cost or basis of the asset	
:Cost:	3250 ENTER
Input the class life in years	
:Life:	5 ENTER
Enter this year (1 for 1st, 2 for 2nd, etc.)	
:This year:	1 ENTER
DEPRECIATION :Cost: 3250.00 :Method: St line :This year: 650.00 :Remaining: 2600.00	

Now compare the amount you can take in the first year using straight line depreciation with that allowed by the double-declining-balance method.

Program Prompt or Display	Your Action
	DDBAL
Input the cost or basis of the asset	
:Cost:	3250 ENTER
Input the class life in years	
:Life:	5 ENTER
Enter this year (1 for 1st, 2 for 2nd, etc.)	
:This year:	1 ENTER
DEPRECIATION :Cost: 3250.00 :Method: 200 DB :This year: 1300.00 :Remaining: 1950.00	

After these two depreciation calculations, the following values are left on the stack:

Stack Level	Contents
Level 4:	This year: 650.00
Level 3:	Remaining: 2600.00
Level 2:	This year: 1300.00
Level 1:	Remaining: 1950.00

You may have noticed that using double-declining-balance depreciation, the asset is never fully depreciated to zero. For this reason, tax authorities usually permit you to switch to the straight-line method any time during the life of the asset. You make the switch in the first year the amount of depreciation is greater using straight-line.

Compound Interest Amount

Who in our society hasn't been exposed to the vagaries of compound interest? Whether it's a savings account, an auto loan, or a "revolving" credit balance that seems to grow exponentially, compound interest touches all of our lives in some way.

In compound interest, the interest earned on a sum (the principal) is added to that sum, and interest is then earned on the entire amount. The formula for computing compound interest is:

$$A = P\left(1 + \frac{r}{m}\right)^{mt}$$

where

P = amount of principal
r = interest rate per year, expressed as a decimal
m = number of compounding periods per year
t = time; that is, number of years
A = amount accumulated at the end of t years

Looking at this equation, you can see that it won't be too difficult to solve for the amount, A. But what if you want to know the number of years, t, that it will take to accumulate a certain sum, or the effective interest rate, r, given an amount and principal?

When you have complex equations with many interacting variables, the easiest way to handle them is with the HP Solve application. With HP Solve, you enter and store the equation, then see a menu of that equation's variables in the display. HP Solve lets you enter the known variables, then solve for an unknown. It's a lot easier than trying to write a program to do the same thing.

Keyboard Example of HP Solve

For instance, let's suppose you want a million dollars when you retire 20 years from now. Here's how you could use HP Solve to find the total amount you'd need to invest now, assuming an interest rate of 7.5% compounded quarterly:

↰ EQUATION
A ↰ =
P × ↰ ( ) 1 + r ÷ m
▶ ▶ y^x ↰ ( ) m × t ▶

Store this as the current equation:

ENTER

↰ SOLVE

STEQ

Then use the SOLVR menu of variables to find the amount. First input the known variables:

Keystrokes	Display	Comments
SOLVR		Displays SOLVR menu.
1000000 A	A: 1,000,000.00	Enter amount.
.075 R	r: 0.08	Enter interest rate.
4 M	m: 4.00	Enter compounding periods per year.
20 T	t: 20.00	Enter time in years.

Then solve for the principal:

Keystrokes	Display	Comments
↰ P	P: 226250.87	The required principal.

This shows you need to invest $226,250.87 in order to have a million dollars 20 years from now.

The Compound Directory

To create a program for compound interest, begin by creating a directory for the *Compound* program and its subprograms:

' COMPOUND

↰ MEMORY CRDIR

VAR

COMP

The COMPOUND directory will hold all programs and subprograms for this application . Any objects (including programs) that you now enter and save will be placed in this directory.

The Main Compound Program—Programming HP Solve

Using the HP Solve application means you don't have to write complicated programs. With HP Solve, the hard part—the linking of many variables in a formula so you can solve for any of them—is done for you. However, you can surround HP Solve with a shell that automatically loads the correct formula, provides user instructions, and cleans up when it's done.

The advantage of programming an equation is simplicity: you press one of the VAR keys (for example, COMP for the *Compound* program) and you get:

- The correct equation.
- The HP Solve SOLVR menu of variables for that equation.
- User instructions on the screen.

The disadvantage is that all those variables used in the formula remain in your VAR list when you're through. However, you can purge the variables with a cleanup routine.

Now let's look at the main compound interest program, *Compound*. This main program merely calls a series of subprograms in order. By itself, *Compound* takes nothing from the stack and leaves nothing there, although as you'll see, HP Solve leaves its results on the stack, ready for you to use them in other calculations.

Program Instructions	**Comments**
«	
`Init`	Initializes calculator and stores equation.
`Msg1`	First part of user instructions.
`Msg2`	Second part of user instructions.
`Cprompt`	Halts execution and displays SOLVR menu.
`Cleanup`	Purges variables and shows VAR menu again.
»	

To save this main compound interest program as *Compound*:

Keystrokes	Comments
(ENTER)	Puts program on the stack.
(') Compound (STO)	Stores the program as *Compound*.

Now let's look at the individual subprograms and what they do.

Init—Specifying a Built-In Menu

The *Init* subprogram "writes" the compound interest formula into the stack and stores it as the current equation. Then *Init* displays the HP Solve menu of variables—which, of course, are the variables from that current equation. *Init* also sets the display mode to show two decimal places, and ensures that the time doesn't appear during display of user messages. This subprogram has no overall effect on the stack.

Program Instructions	Comments
`«`	
`'A=P*(1+r/m)^(m*t)'`	Compound interest formula.
`STEQ`	Stores formula as current equation.
`30 MENU`	Displays HP Solve SOLVR menu.
`-40 CF`	Turns off display of time.
`2 FIX`	Shows numbers to two decimal places.
`»`	

To save this subprogram:

Keystrokes	Comments
(ENTER)	Puts program on the stack.
(') Init (STO)	Stores the program as *Init*.

This subprogram begins by "writing" the compound interest formula into the display and placing it on the stack. When you surround the formula with tick marks ('), it means the formula won't be evaluated by a running program. Instead it's merely placed on the stack.

Next, *Init* executes the STEQ (*store equation*) command. STEQ stores the formula as the current equation. This equation and its variables are usable within the current COMPOUND directory, but don't affect operations and variables in the rest of the calculator. Thus, the current equation affects only the current directory; other directories may have different current equations.

With the equation stored, *Init* executes `30 MENU`. The MENU command lets you call up any of the built-in calculator menus; and `30 MENU` specifies HP Solve's SOLVR menu. It's just as if you'd pressed ↰ (SOLVE) SOLVR.

Clearing system flag −40 with the `-40 CF` instruction keeps the internal timer display from appearing in the middle of some later messages. And the `2 FIX` statement specifies that numbers will be shown to two decimal places (that is, as dollars and cents).

If you press INIT to run this program, you'll find that the HP 48 switches to its HP Solve application, and you see the menu of HP Solve variables at the bottom of the display:

```
  A      P      R      M      T    EXPR=
```

In fact, this is all you really need to run the HP Solve application. You can plug known values into those variables, and use the ↰ key followed by the variable key to solve for a value, just as in the keyboard example. But we're going to use additional subprograms to include some memory-jogging instructional messages. We'll also add a cleanup routine, so that each time you run *Compound*, you'll start with a clean slate of variables.

Msg1—Displaying a Message without FREEZE

The *Msg1* subprogram displays the first of two user messages. It waits for you to press any key before continuing, and has no overall effect on the stack.

Program Instructions	Comments
«	
CLLCD	Clears previous messages from display.
"COMPOUND INTEREST Enter each known quantity, then press its menu key. To solve for a quantity, press the [←] key, followed by the menu key."	Message for display by DISP.
1 DISP	Displays message beginning on line 1.
-1 WAIT	Waits for key press.
DROP	Throws away key address from key press.
»	

To save the *Msg1* program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Msg1 STO	Stores the program.

To begin, a CLLCD command clears the HP 48's liquid crystal display of any previous messages or information. Then *Msg1* places a seven-line message string on the stack, followed by the number 1. (As usual, place endlines, ↱ ↵, after each line of the message string.) The message itself tells users the name of the program, and how to run it.

DISP takes two quantities from the stack, the message string from level 2 and the number from level 1. Since the number is 1, DISP displays the message string beginning on line 1 of the calculator's screen—that is, at the top. This seven-line message pretty well fills up the available display area.

With the message string now displayed, WAIT gives you time to read it. Because nothing happens between the execution of DISP and the WAIT, the display isn't updated, so we don't need to use FREEZE here.

Both `0 WAIT` and `-1 WAIT` will wait for the press of any key before continuing. Here we've used `-1 WAIT`, so that along with the message, you also allow the user-key menu to be updated. This lets you see the latest menu—that is, the menu of SOLVR variables.

After reading the message displayed in *Msg1*, you press any key to continue to the next subprogram. However, with 0 or −1 as its argument, WAIT not only waits for you to punch a key, but also returns a number representing the address of the key that was pressed. That's why we need to DROP the stack before exiting *Msg1*; the DROP "throws away" that useless address.

Msg2—Displaying a Second Message

Msg2 displays a detailed explanation of each of the quantities called by the list of SOLVR variables. Like *Msg1*, it has no effect on the stack.

Program Instructions	**Comments**
`«`	
`CLLCD`	Clears previous display.

`"P=Principal amount`
`r=yearly interest rate`
`as decimal (.07,etc)`
`m=no. of compounding`
`periods per year`
`t=no. of years`
`A=Amount after t years"`
	Text string for display by DISP.

`1 DISP`	Displays text beginning on line 1.
`0 WAIT`	Waits for a key press.
`DROP`	Throws away key address left by WAIT.
`»`	

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Msg2 STO	Stores the program.

Although the message itself is different, *Msg2* is very similar to *Msg1*. The only real difference lies in the use of WAIT. We still want to display the message while waiting for a key press, so we can use either `0 WAIT` or `-1 WAIT` here. Now in *Msg1*, we needed `-1 WAIT`, so that the menu keys could be updated while waiting. Because those keys have now been updated, though, we don't need to use `-1 WAIT` here; we can use `0 WAIT` instead. `0 WAIT` merely waits for you to press any key, while maintaining the previous list of menu keys.

At this point in the *Compound* program, we want to give control back to the user temporarily, to allow the use of the SOLVR menu of variables. So the next subprogram that is called is *Cprompt*.

Cprompt—A Good PROMPT Application

The *Cprompt* subprogram temporarily suspends execution of *Compound*, giving complete keyboard control back to the user. *Cprompt* provides a message to explain what's happening. By itself, *Cprompt* has no overall effect on the stack, but when execution is resumed, it's highly likely that a computed value will be present there.

Program Instructions	**Comments**
`«`	

`"Now in HP Solve.`
`Press CONT when done."`
	Prompt string for use by PROMPT.

`PROMPT`	Displays prompt string, returns control to keyboard.
`»`	

Keystrokes	**Comments**
ENTER	Puts this short program on the stack.
' Cprompt STO	Stores *Cprompt* program.

At this point in *Compound*, it's time to put values into the SOLVR variables and make some compound interest calculations. Temporarily returning control to the keyboard like this is a perfect application for PROMPT or HALT. And as it happens, *Cprompt* is really nothing more than a PROMPT command.

PROMPT takes as its argument a text string, which it displays at the top of the HP 48's LCD. PROMPT also returns control to the keyboard—but it expects that when you're done pressing keys, you'll press ⬅ CONT to continue. If you forget to press CONT, you're likely to find the word HALT in the status area.

When you finally do resume execution, *Compound* calls *Cleanup* to take care of some housekeeping before it ends.

Cleanup—A Programmatic Purge of Variables

After you're done computing compound interest, you're left with the menu of SOLVR variables in the current directory, along with all of *Compound*'s subprograms. *Cleanup* purges the variables, and redisplays the VAR menu. It doesn't affect the stack.

Program Instructions	**Comments**
«	
{ A P r m t }	Makes a list of variables.
PURGE	Deletes those variables.
2 MENU	Displays the VAR menu again.
»	

Keystrokes	**Comments**
ENTER	
' Cleanup STO	Stores the *Cleanup* program.

The PURGE command deletes objects from the HP 48. The PURGE command, which is available on the keyboard as ⬅ PURGE, can take a single variable name as its argument, or it can take a list of names. In the *Cleanup* subprogram, we supply PURGE with a list of variable names—the very names that form the SOLVR list of variables, left over when you're done making compound interest calculations.

Using PURGE to clean up is an excellent way to get rid of unneeded objects before exiting a program. You can use PURGE in this manner to erase SOLVR variables and delete reserved names, such as PPAR and ΣDAT, that have been created in the current directory by the HP 48.

Finally, *Cleanup* executes `2 MENU`. This does nothing more than display the updated list of variables; it's exactly as if you'd pressed the VAR key now. However, changing back to the normal VAR menu on exit provides "visual feedback" that something has happened, and makes it easy for the user to find COMP for running the program again.

Running the Compound Interest Program

Now let's run that compound interest program to see how much we'd need to invest now in order to have a million dollars 20 years from now. Again, assume an interest rate of 7.5%, and quarterly compounding. Press COMP to begin the program.

Program Prompt or Display	**Your Action**
	COMP
COMPOUND INTEREST Enter each known quantity, then press its menu key. To solve for a quantity, press the [←] key, followed by the menu key.	ENTER (or any key)

Now you see the next message.

Program Prompt or Display	**Your Action**
`P=Principal amount`	
`r=yearly interest rate`	
`as decimal (.07,etc)`	
`m=no. of compounding`	
`periods per year`	
`t=no. of years`	
`A=Amount after t years`	(ENTER) (or any key)
`Now in HP Solve`	
`Press CONT when done.`	

The HP 48 is now in HP Solve, all ready for you to key in quantities for the different variables. The menu of SOLVR variables is shown at the bottom of the display:

```
[ A ] [ P ] [ R ] [ M ] [ T ] EXPR=
```

Using this menu, you can store the known values in their variables, and solve for *P*. The amount, *A*, is $1,000,000.00. Interest rate *r* is 7.5% (that is, .075), and the time, *t*, is 20 years. Number of compounding periods (*m*) is 4 per year.

Keystrokes	**Display**
1000000 [A]	`A: 1,000,000.00`
.075 [R]	`r: 0.08`
4 [M]	`m: 4.00`
20 [T]	`t: 20.00`

Then use (↰) before the menu key to solve for the variable *P*:

Keystrokes	**Display**
(↰) [P]	`P: 226250.87`

The answer is shown in the display. To continue the program, and clean up the current directory:

(↰) (CONT)

That does it! You've quickly calculated the necessary principal value. And the VAR menu keys for the current directory are once again displayed.

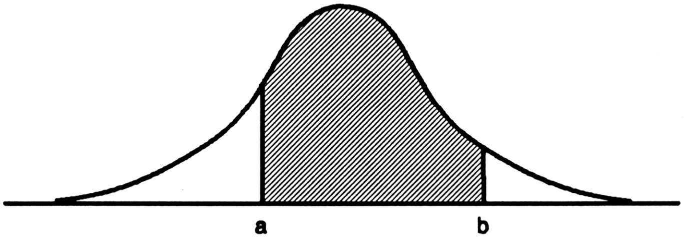
4

Statistics Programs

The HP 48 has several statistics functions built into the calculator. Like other functions, you can use them individually or as parts of a program. This section illustrates a few techniques for getting the most from the statistics features of the HP 48.

Normal Probability

A normal distribution, with the mean in the middle, produces a bell-shaped probability curve. The probability that a given individual *x* is between *a* and *b* can be computed using calculus, of course, but most statistics texts use a method that involves finding the standard score, and then using a table to determine probability.



The HP 48's UTPN function gives the probability of something *above* a particular point—the probability that you'll be in the top 10 percent of your class, for instance. It requires three quantities in the stack.

Stack Level	Contents
Level 3:	Mean, $\bar{x}$
Level 2:	Variance, s^2
Level 1:	Value of x

Given these quantities, the UTPN function returns the upper-tail probability; that is, the probability represented by the area under the curve from x to the right.

Keyboard Example

To find the probability of the area from a to b, we need to subtract the upper-tail probability of a from that of b. For instance, intelligence quotients are normally distributed with a mean ($\bar{x}$) of 100 and a standard deviation (s) of 10.

What good is knowing the probability of a normal distribution? Suppose that a hot new electronic dating service, *Compumeet*, intends to provide you with an introduction to someone chosen completely at random. You can use the following procedure to determine the probability that Mr. or Ms. Right has an IQ between 95 and 105:

Keystrokes	Display	Comments
100 ENTER	100.00	Enter the mean, $\bar{x}$.
10 ENTER	10.00	Enter the standard deviation, s.
↰ x^2	100.00	Square it to get the variance, s^2.
95	95	Enter the value for a.
MTH PROB NXT		
UTPN	0.69	Compute the upper- tail probability for a.
↱ LAST ARG	95.00	Recall levels 1, 2, and 3 (the last arguments) to the stack.

Keystrokes	Display	Comments
↰ DROP	100.00	Drop the stack to leave mean and variance in levels 2 and 1, respectively.
105 UTPN	0.31	Compute upper-tail probability for *b*.
−	0.38	Subtract upper-tail probability of *b* from that of *a* to get probability between *a* and *b*.

The probability is .38, or about 38 percent that a person chosen at random will have an IQ between 95 and 105.

The NormalProb Directory

To create a directory for the normal probability program and subprograms, then get into that directory:

' NORMALPROB
↰ MEMORY CRDIR
VAR
NORM

The NORMALPROB directory will hold all programs and subprograms for calculating normal probability. Any objects (including programs) that you enter and save will now be placed in this directory.

The Main NormalProb Program

Here's the main program for computing the probability from *a* to *b*. It prompts for the quantities as they're needed. The program determines probabilities of individual values for a normal population. The *NormalProb* program requires nothing on the stack when it begins; when finished, it leaves your tagged *a* and *b* values in levels 3 and 2, and it leaves the calculated probability in level 1 of the stack.

Arguments	Results
3:	3: tagged value for a
2:	2: tagged value for b
1:	1: tagged probability

Program Instructions	Comments
`«`	
`Init`	Initializes HP 48 calculator.
`Message`	Displays program title and user instructions.
`Getmean`	Prompts user to enter the mean.
`Getsdev`	Prompts for standard deviation.
`Geta`	Prompts for point *a*.
`Getb`	Prompts for point *b*.
`Compute`	Computes normal probability.
`Label`	Labels the output.
`»`	

To save the main *NormalProb* program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' NormalProb STO	Stores the program as *NormalProb*.

This is about as straightforward an example of structured programming as you'll find anywhere. *NormalProb* calls several subprograms in turn, and each one performs a single task. *Init*, of course, sets up the HP 48 for this calculation. The *Message* subprogram displays some user instructions, explaining what the program does.

Getmean, *Getsdev*, *Geta*, and *Getb* all prompt the user to input quantities needed for the actual probability calculation. Each of these subprograms gives the user an opportunity to verify (or change) the value entered. Finally, *Compute* uses the UTPN command to calculate normal probability, and *Label* tags and displays the final answer.

Init—Answers to Two Decimal Places

The first program called by *NormalProb* is *Init*, which initializes the HP 48 calculator to display answers to two decimal places. *Init* doesn't affect the stack.

Program Instructions	**Comments**
`«`	
`2 FIX`	Sets HP 48 to display answers to two decimal places.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Init STO	Stores the program.

Although this subprogram as now written merely sets the display mode for the calculator, you could also use it for other settings as well.

Message—Halting for a Display

The *Message* subprogram halts to display a main title and some brief user instructions. It has no overall effect on the stack.

Program Instructions	Comments
«	
CLLCD	Clears the calculator's liquid crystal display.
"NORMAL PROBABILITY Probability between two points on normal distribution. You'll be prompted to enter left point, then right point. Do CONT now."	Text string for display by DISP. Place endline characters (↱ ↵) at the end of each line.
1 DISP	Displays text beginning on line 1.
7 FREEZE	Freezes the entire display until next key press.
HALT	Halts program until ↰ CONT is pressed.
»	

To save this subprogram:

Keystrokes	Comments
ENTER	
' Message STO	Stores the program in the current directory.

The *Message* subprogram begins with a CLLCD command, which clears any previous information from the calculator display. If you don't use CLLCD, the later DISP command simply writes over the existing display; and any areas unaffected by the new text or message remain just as they were. We use CLLCD to guarantee a clean slate.

Then *Message* places a long text string in the stack for later use by the DISP command. This string is so long, in fact, that when displayed, it fills the HP 48's screen. As you enter the *Message* subprogram, be sure to press ↱ ↵ at the end of each text line, to make sure the entire message is displayed with no scrolling.

The DISP command requires two arguments on the stack: a text string in level 2, and a number in level 1. When executed, DISP places that text string in the calculator display, with the first character of text beginning on the row specified by the number. Thus, "Text" 1 DISP would place the word Text in row 1 (the top row) of the display, while "Hello" 4 DISP would put this familiar greeting in the center row.

The 7 FREEZE command freezes the entire display, preventing it from being updated until the next key press. Without this command, you'll see the message displayed by DISP all right, but it will immediately disappear as soon as the stack is updated. Thanks to FREEZE, the stack can be working and changing behind the scenes, while you read the message frozen in the calculator display.

There are a couple of ways to pause an HP 48 program, but in *Message* we've chosen to HALT it. The HALT command suspends program execution and returns control to the keyboard; to continue the program, you press ↰ CONT.

With HALT, you don't have to continue the program now. Instead, you can go off and use the statistics keys to input data, or compute mean or standard deviation, or perform other calculations. Then you can come back and continue on to the next subprogram.

Getmean—Verifying Input with PROMPT

The next subprogram call is to *Getmean*, which prompts the user for input of the mean value. Before the subprogram continues, the user can double-check the value, and change it if necessary. *Getmean* takes nothing from the stack, and leaves the tagged value of the mean there when it has completed execution.

Arguments	Results
1:	1: tagged mean

Program Instructions	Comments
`«`	
`"Enter the mean (x̄)"`	Prompt string for INPUT.
`":x̄: "`	Command-line string.
`INPUT`	Pauses for input, displaying prompt and command-line strings.
`OBJ→`	Converts user input to a tagged object.
`"Press CONT if x̄ is OK"`	String for PROMPT.
`PROMPT`	Waits for CONT.
`»`	

Keystrokes	Comments
ENTER	Puts program on the stack.
' Getmean STO	Saves the program.

This subprogram uses INPUT to get a value for mean from the user, and PROMPT to allow an extra verification (or input) before continuing.

INPUT requires two strings as it arguments, so *Getmean* begins by placing two text strings on the stack. The first is `"Enter the mean (x̄)"`, which is the prompt string for INPUT; this is what will be seen in row 1, at the top of the display, when INPUT is executed. The second text string is `":x̄: "`. This text appears on the command line, with the cursor right after it, and whatever is input by the user will be added to this string.

When the INPUT command is executed, the program halts, displays the prompt and command-line strings, and waits for the user to input a value for the mean. (What it's actually waiting for is the next press of ENTER.)

See the colons at the beginning and end of the command-line string? Because of these colons, the value that's INPUT is added to the string to create the *form* of a tagged object, like this: "**:** *tag***:** *object*". Then the OBJ→ command removes the quotation marks from the string and converts it into the tagged object. When this tagged object is displayed on the stack, the leading colon is dropped, so you see something like this:

```
x̄: 100
```

Now the subprogram is halted again, this time by PROMPT. As used here, the PROMPT command takes as its argument the string "Press CONT if x̄ is OK". When executed, PROMPT displays this prompt string and halts execution, returning control to the keyboard. The user can now verify whether or not the value input for mean is correct. If the value is correct, the user presses ↰ CONT to continue.

Notice the difference between HALT (used in *Message*) and PROMPT. Both cause the program to halt and return control to the keyboard until CONT or SST is pressed. However, HALT doesn't display a prompt string, but does display the word HALT in the status line.

The next three subprograms, *Getsdev*, *Geta*, and *Getb*, all function the same way—with INPUT followed by PROMPT.

Getsdev

Getsdev prompts the user to input a value for standard deviation. This subprogram takes nothing from the stack, and adds a tagged standard deviation.

Arguments	Results
2:	2: *tagged mean*
1: *tagged mean*	1: tagged standard deviation

Program Instructions	Comments
`«`	
`"Enter the standard deviation (s)"`	Prompt string for INPUT.
`":s: "`	Command-line string for INPUT.
`INPUT`	Waits for keyboard input of standard deviation.
`OBJ→`	Converts command-line string and value to tagged object.
`"Press CONT if s is OK"`	Prompt string for PROMPT.
`PROMPT`	Halts execution and waits for ↰ CONT.
`»`	

Keystrokes	Comments
ENTER	Puts program on the stack.
' Getsdev STO	Stores the program.

Getsdev gets the value for standard deviation from the user. Like *Getmean*, this subprogram includes an extra verification step (the PROMPT command) to make sure the right value has been entered.

Geta

The *Geta* subprogram prompts the user to input a value for *a*, the left-hand point. This subprogram takes nothing from the stack, and adds a suitably tagged value for *a*.

Arguments	Results
3:	3: *tagged mean*
2: *tagged mean*	2: *tagged standard deviation*
1: *tagged standard deviation*	1: tagged value for a

Program Instructions	Comments
`«`	
`"Enter the left point (a)"`	Prompt string for INPUT.
`":a: "`	Command-line string for INPUT.
`INPUT`	Waits for keyboard input of value.
`OBJ→`	Converts command-line string and value to tagged object.
`"Press CONT if a is OK"`	Prompt string for PROMPT.
`PROMPT`	Halts execution and waits for ↰ CONT.
`»`	

Keystrokes	Comments
ENTER	Puts program on the stack.
' Geta STO	Stores the program.

Getb

This subprogram prompts the user to input a value for *b*, the right-hand point. *Getb* takes nothing from the stack, and puts a tagged value for *b* there.

Arguments	Results
4:	4: *tagged mean*
3: *tagged mean*	3: *tagged standard deviation*
2: *tagged standard deviation*	2: *tagged value for a*
1: *tagged value for a*	1: tagged value for b

Program Instructions	Comments
`«`	
`"Enter the right point (b)"`	Prompt string for INPUT.
`":b: "`	Command-line string for INPUT.
`INPUT`	Waits for keyboard input of value.
`OBJ→`	Converts command-line string and value to tagged object.
`"Press CONT if b is OK"`	Prompt string for PROMPT.
`PROMPT`	Halts execution and waits for ↰ CONT.
`»`	

Keystrokes	Comments
ENTER	Puts *Getb* on the stack.
' Getb STO	Stores the program *Getb*.

After *Getb* and the previously called subprograms have been executed, all four needed values are on the stack. Now the actual probability computation can take place.

Compute—Local Variables and Stack Calculations

The *Compute* subprogram handles the computational chores in *NormalProb*. It accomplishes the same thing you saw in the earlier keyboard example: with the values for mean, standard deviation, point *a*, and point *b* on the stack, *Compute* calls UTPN twice to find the difference between the probability of *a* and the probability of *b*. This subprogram takes four values from the stack, and returns three: a tagged *a*, a tagged *b*, and the probability a random value lies between *a* and *b*.

Arguments	Results
4: tagged mean	4:
3: tagged standard deviation	3: probability
2: tagged value for a	2: tagged value for a
1: tagged value for b	1: tagged value for b

Program Instructions	Comments
`«`	Begins main program.
`→ m s a b`	Converts four values on stack to local variables.
`  «`	Begins defining procedure for local variables.
`  m`	Places mean (*m*) on stack.
`  s SQ`	Squares standard deviation (*s*) and places on stack.
`  a`	Places left-hand point (*a*) on stack.
`  UTPN`	Computes upper-tail probability for left-hand point.

Program Instructions	**Comments**
m	Places mean on stack.
s SQ	Squares standard deviation to get variance.
b	Places right-hand point (*b*) on stack.
UTPN	Computes upper-tail probability for right-hand point.
-	Subtracts to find probability from *a* to *b*.
a	Places *a* on stack.
b	Places *b* on stack.
»	Ends defining procedure.
»	Ends main program.

To save the *Compute* program:

Keystrokes	**Comments**
ENTER	
' Compute STO	Stores this program.

When *Compute* is called, four quantities are on the stack. We need to use some of these quantities more than once, and we want to return two of them (*a* and *b*) to the stack along with the final calculated value for probability. So we turn them into local variables, which can be summoned again and again within the defining procedure.

Within the *Compute* program, → m s a b takes four quantities (mean, standard deviation, *a*, and *b*) from the stack and converts them to local variables. The defining procedure, which is a program, follows immediately.

Now we use those local variables to calculate the two upper-tail probabilities we need. The mean, local variable *m*, is put on the stack first, followed by the standard deviation, *s*. The standard deviation is squared to get the variance. Next, the value for the left-hand point, represented by local variable *a*, is put onto the stack.

With the three values are in place, we can execute the UTPN command. This command, remember, returns the upper-tail probability for point *a*.

The next step is to perform the same calculation for point *b*. Within the defining procedure, a local variable can be used as often as it's needed. So we simply put *m* and *s* on the stack again, square *s*, and put *b* on the stack. Then we execute UTPN again to find the second upper-tail probability. We want the probability from *a* to *b*, so we subtract P(*b*) from P(*a*).

It would be nice if the user could see the values for *a* and *b* along with the probability, so we return these two values to the stack before exiting *Compute*. Local variables *a* and *b* are still tagged numbers, so they appear in the stack with their tags still attached.

Label—Using Local Variables to Rearrange the Stack

The final subprogram called by *NormalProb* is *Label*, which adds a tag to the calculated value for probability. *Label* also rearranges the values so the user has a more useful stack display upon exit. This subprogram takes three values from the stack and returns three.

Arguments	Results
3: probability	3: tagged value for a
2: tagged value for a	2: tagged value for b
1: tagged value for b	1: tagged probability

Program Instructions	**Comments**
`«`	Begins main program.
`→ p a b`	Creates local variables.
`  «`	Begins defining procedure for local variables.
`  a b p`	Puts local variables back on stack in different order.
`  "P(a→b)"`	Text string that is tag for *p* (probability).
`  →TAG`	Combines text string and value for *p*.
`  »`	Ends defining procedure.
`»`	Ends program.

Keystrokes	**Comments**
ENTER	
' Label STO	Stores the subprogram *Label*.

The *Label* subprogram takes three values off the stack and converts them to local variables. Within the defining procedure, the three variables are put back on the stack in a different order. This brings the value for probability (*p*) into level 1, where it can easily be used in other calculations.

Notice that with a slight modification to *Compute* (placing the values for *a* and *b* on the stack before computing probability), you could eliminate some of the steps of this *Label* program. But as shown here, *Label* illustrates how local variables make it easy to change the order of quantities on the stack.

We also want to label the probability, so the user knows what this value represents. Thus, we now place a text string on the stack; this puts the calculated value for probability in level 2 and the text string in level 1, ready to be combined.

Stack Level	Contents
Level 2:	Calculated probability
Level 1:	"P(*a*→*b*)"

Now →TAG is executed. →TAG combines the value in level 2 with the tag in level 1 to create a single tagged object. Like all tagged objects, this one can be used in calculations involving addition, multiplication, and so on. And because the value for probability is now clearly labeled, the user knows exactly what it means: the probability between *a* and *b*.

Running the NormalProb Program

Run the *NormalProb* program to find the probability that a random Mr. or Ms. Right possesses an IQ between 95 and 105. Press NORM to start the program.

Program Prompt or Display	Your Action
	NORM
NORMAL PROBABILITY Probability between two points on normal distribution. You'll be prompted to enter left point, then right point. Do CONT now.	⬅ CONT
Enter the mean (x̄) :x̄:	100 ENTER
Press CONT if x̄ is OK 1: x̄: 100.00	⬅ CONT

Now continue in this manner. Enter each needed variable when prompted, verify each one, then press ⬅ CONT to continue.

```
Program Prompt or Display             Your Action

Enter the standard
deviation (s)
:s:                                   10 ENTER

Press CONT if s is OK
2:       x̄: 100.00
1:       s: 10.00                     ⬑ CONT

Enter the left
point (a)
:a:                                   95 ENTER

Press CONT if a is OK
3:       x̄: 100.00
2:        s: 10.00
1:        a: 95.00                    ⬑ CONT

Enter the right
point (b)
:b:                                   105 ENTER

Press CONT if b is OK
4:       x̄: 100.00
3:        s: 10.00
2:        a: 95.00
1:       b: 105.00                    ⬑ CONT

3:        a: 95.00
2:       b: 105.00
1:   P(a→b): 0.38
```

The probability is about 38 percent that someone you meet has an IQ between 95 and 105.

Hypothesis Tester

Statistics is often used to make assumptions about a *population* (all members of a group) based on what is known about a *sample* (a few members of a group). In hypothesis testing, we take a sample, test the sample, then determine whether that sample is representative of the population.

The classic approach to hypothesis testing involves finding a test statistic (a so-called z score), and comparing this to the test statistic for the desired confidence level. A more modern approach, and one that works well on the HP 48, is the *probability value* or *p-value* method, in which the calculated probability is compared directly to the confidence level. We'll use the p-value approach in this section.

Let's start with an example to show how hypothesis testing works. A certain coach claims that in an annual school-wide test of general knowledge, athletes actually score higher than the general student population. In particular, the coach points to scores achieved by 40 members of the football team, whose mean score ($\overline{x}$) was 75.3, with a standard deviation of 13.0. The school average, (μ), for the same test given year after year is 72. At the .05 level of significance, do the football team's test results prove the coach right or wrong?

Well, "right or wrong" is an elusive concept, especially in statistics. Instead of proving right or wrong, what we do is prove or disprove a null hypothesis. To do this, we follow a step-by-step procedure:

1. Formulate two hypotheses, a null and an alternate.
2. State the level of significance, a probability value called α.
3. Calculate the critical value.
4. Calculate the test probability, or p-value.
5. Compare the p-value to α, and accept or reject the null hypothesis.

Formulating the Hypotheses: We first formulate two hypotheses, a *null* hypothesis and an *alternate* hypothesis. For the coach's claim, the hypotheses might look like this:

- *H_0, null hypothesis*: The mean of athletes' scores is less than or equal to 72.
- *H_A, alternate hypothesis*: The mean of athletes' scores is greater than 72.

Notice that the alternate hypothesis, H_A, is what you want to prove, but you don't ever really prove it. Instead, what you do is either *reject* or *fail to reject* the null hypothesis, H_0.

The null hypothesis always has an equals sign in it. It might be something like "Scores = 100" or "IQ > = 130." If there is only an equals sign in the null hypothesis, you'll do a *two-tail test*. If the null hypothesis includes the words "greater than or equal to" or "less than or equal to," you'll do a *one-tail* test.

Determining the Level of Significance: The level of significance is called alpha (α), and it represents the probability of error of a certain type. The level of significance is really a measure of how much risk you're willing to take. If error would have serious consequences, use a small value for α; if you can accept more risk, use a larger value.

Typical values for α are .05 and .01. We'll give the coach some leeway and use .05 for our illustration.

Calculating the Critical Value: Most illustrations of hypothesis testing focus on the standard normal distribution—the familiar bell curve—and its critical value, z. But there are several other formulas available, depending on the size of the population and how much information is known about it. We're going to use the Student's t distribution, because it works for small samples (of 30 or fewer) and you can use the sample standard deviation, s, instead of the population standard deviation, σ. Another reason for choosing the Student's t distribution is that for samples of 30 and above, the critical values (called t values) are about the same as z, the critical value used with normal distribution.

The formula is:

$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$

For our illustration, then:

$$t = \frac{75.3 - 72}{13/\sqrt{40}} = 1.6055$$

Calculating the P-Value: To calculate the probability, or p-value, you use the critical value and the *degrees of freedom*.

The degrees of freedom, or df, are given as the sample size less one. Thus, the degrees of freedom in our illustration are given by:

$$df = 40 - 1 = 39$$

In the pre-HP 48 days, you then took t and df, and used a table in the back of a statistics book to find the p-value. Today, though, you can use the calculator's UTPT function. With the degrees of freedom in stack level 2, and the value for t in level 1, executing UTPT gives the probability that a random variable is greater than t. (The probability returned by UTPT for this calculation is 0.0582.)

Accepting or Rejecting the Null Hypothesis: To determine whether to accept or reject the null hypothesis, you compare the calculated p-value to α.

- If the p-value is less than or equal to α, reject the null hypothesis.
- If the p-value is greater than α, fail to reject the null hypothesis (which implies that you accept the alternate hypothesis).

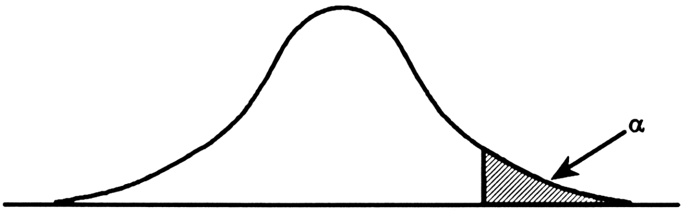
Because the p-value (.0582) is greater than α (.05), we fail to reject the null hypothesis—which means there is insufficient data to accept the coach's claim that the football team athletes have higher scores.

One-Tail or Two-Tail? If the null hypothesis contains a "less than or equal to" or "more than or equal to" statement, as in the previous example, it's known as a one-tail test. However, if the null hypothesis has an equals sign only, the alternate hypothesis has two conditions: one above and one below the distribution.

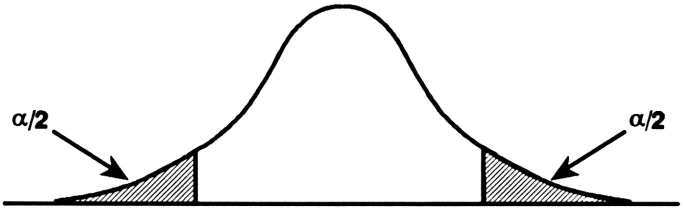
For instance, if the coach had said the mean scores for the football team were *the same* as that of the entire school, there is a chance for rejection both above and below the hypothesized mean. The hypotheses look like this:

- *Null hypothesis*: The mean of athletes' scores is equal to 72.
- *Alternate hypothesis*: The mean of athletes' scores is greater than 72 or less than 72.

In this case, you need to divide α by two, or double the calculated p-value, to test the hypothesis.



One-tail test



Two-tail test

Keyboard Example

Here are the keystrokes used to calculate the *p*-value in our one-tail example:

Keystrokes	Display	Comments
.05 ENTER	0.0500	Enter α.
40 ENTER	40.0000	Enter sample size n.
1 −	39.0000	Calculate degrees of freedom (*df*).
75.3 ENTER	75.3000	Enter sample mean ($\overline{x}$).
72 −	3.3000	Subtract hypothesized population mean (μ).
13 ENTER	13.0000	Enter sample standard deviation (s).

Keystrokes	Display	Comments
40 [√x]	6.3246	Square root of n.
[÷] [÷]	1.6055	Calculate t.
MTH PROB		
[NXT] UTPT	0.0582	Calculate p-value.

With both α and the p-value left on the stack, you can easily see which is larger, and make your conclusions about whether to accept or reject the null hypothesis.

The Hypotest Directory

To create a directory for the hypothesis testing programs, then get into that directory:

['] HYPOTEST
[↰] [MEMORY] CRDIR
[VAR]
HYPOT

The HYPOTEST directory will hold all programs and subprograms for hypothesis testing. Any programs that you enter and save will be placed in this directory.

The Main Hypotest Program

Hypotest presents you with a temporary menu of choices:

MSSG 1TAIL 2TAIL EXIT

By pressing one of these menu keys, you select whether to:

- See a message of user instructions (MSSG).
- Perform a one-tail hypothesis test (1TAIL).
- Do a two-tail hypothesis test (2TAIL).
- Exit from the application (EXIT).

Hypotest runs different subprograms, depending on which menu key is pressed. It has no overall effect on the stack.

Program Instructions	Comments
«	
Init	Initialize HP 48.
{	Begins list for TMENU (outer list).
{"MSSG" Message}	List for first menu key.
{"1TAIL"	Begins list for second menu key; menu key label.
«	Begins program executed when you press 1TAIL.
Getvars	Gets variables.
T1	Calculates *p*-value for one-tail hypothesis.
Getalpha	Gets level of significance.
Test	Compares *p*-value to level of significance; rejects or fails to reject.
>>	Ends program executed by 1TAIL.
}	Ends list for second menu key.
{ "2TAIL"	Begins list for third menu key.
«	Begins program executed when you press 2TAIL).
Getvars	Gets variables.
T2	Calculates *p*-value for two-tail hypothesis.
Getalpha	Gets level of significance.
Test	Compares *p*-value to alpha; reject or fail to reject.
»	Ends 2TAIL program.
}	Ends list for third menu key.

Program Instructions	Comments
`{ }`	Fourth menu key is blank.
`{ }`	Fifth menu key is blank.
`{"EXIT" Exit}`	List for sixth menu key.
`{`	Ends list for TMENU (outer list).
`TMENU`	
`»`	

To save the main *Hypotest* program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Hypotest STO	Stores the program.

The main *Hypotest* program begins by executing *Init*, a subprogram that readies the HP 48 for things to come. Then *Hypotest* puts a long list—it's really a "list of lists"—on the stack, and executes TMENU.

When executed, the TMENU command uses the information in that long list to create a temporary menu across the bottom of the display. TMENU requires as its argument a list of lists that looks like this:

`{ {"label1"` *object1* `} {"label2"` *object2* `}...}`

Each inner list consists of a label and an object. The label (it's the word in quotation marks) is what you see in the menu display. The object is what is executed when you press that key. The first inner list is assigned to the left menu key (at key address 11), the second list to the second key (address 12), and so on. Only *one* object can follow the label.

If you press the key labeled MSSG, the *Message* subprogram is executed. *Message* displays some user instructions and information about the *Hypotest* application.

Here's an instance where we've placed user instructions separate from other parts of the program. You can press MSSG to get help or jog your memory; but if you want to do several tests without having to view the message each time, you can go directly to 1TAIL or 2TAIL.

Init—Initializing to Two Decimal Places

The *Init* subprogram is always called when you run *Hypotest*. It merely sets the HP 48 to display answers to two decimal places. It does not affect the stack.

Program Instructions	Comments
«	
2 FIX	Sets HP 48 to display numbers to two decimal places.
»	

To save this short subprogram:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Init STO	Stores the program.

If you want to see your answers to more decimal places, change this subprogram to execute 3 FIX or 4 FIX.

Message—Display without Stopping

If you press the MSSG menu key, *Hypotest* calls the *Message* subprogram. A message fills up the display, explaining a little about hypothesis testing. When you're done reading the message, you press any of the menu keys (or any key) to continue. *Message* has no overall effect on the stack.

Program Instructions	Comments
`«`	
`CLLCD`	Clears HP 48's display.

`"HYPOTHESIS TEST:`
`Tests null hypothesis`
`at α level of`
`confidence. Choose`
`one-tail if H0`
`contains ≤ or ≥,`
`two-tail if = only."`
	Text for display by DISP. Be sure to put endline characters (↱ ↵) at the end of each line.

`1`	Row where text will begin.
`DISP`	Displays text beginning on LCD row 1.
`7 FREEZE`	Freezes display until next key press.
`»`	

To save this subprogram:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Message STO	Stores the program.

Message begins with a CLLCD (*clear LCD*) command, which clears the calculator's display of any previous information. Then *Message* puts a long text string into the stack. This text string includes everything between the quotation marks; if you don't want to have to scroll to see the message, be sure to place endline characters (press ↱ ↵) at the end of each line as shown.

The text string is one of the two arguments required by DISP. The second argument is the number 1, which is placed on the stack next. This value tells DISP where to display the text string. When DISP is executed now, it displays the long text message, beginning on row 1 of the calculator's LCD.

The final command executed by the subprogram is `7 FREEZE`. With the number 7 as its argument, FREEZE prevents updating of any of the display areas until the next key press. Because FREEZE is the final command of this subprogram, and because no other subprograms are executed after *Message* until you press a menu key, the *Hypotest* program *appears* to be paused here.

At this point, the main *Hypotest* program has already been run. (It displays the temporary menu, remember?) *Message* is also finished running. You see the results: *Message*'s text message in the display, along with *Hypotest*'s temporary menu.

If you've programmed in computer languages—or even worked with some of the other program examples in this book—you know that it's often necessary to place a HALT or WAIT or some similar command after a message, to give the user time to read the text before a program continues to some other task. No need to do that here, though. The *Hypotest* program isn't actually "running" while *Message*'s text is being displayed. Instead, it merely displays a menu of top-row keys which, if pressed, perform the different functions of the *Hypotest* program. Interestingly, all other keys on the keyboard are active, too.

When you now press another key, such as 1TAIL, that key is *executed* and the display updated. You don't need to use CONT or perform any intermediate steps.

Now let's see what happens when you press 1TAIL or 2TAIL.

Getvars—Labeling Values on INPUT

Getvars is the first subprogram called when you press the 1TAIL or 2TAIL menu key. *Getvars* gets four of the five values needed for the hypothesis test, tags each with an appropriate label, and puts them on the stack.

Arguments	Results
4:	4: tagged $\bar{x}$
3:	3: tagged μ
2:	2: tagged s
1:	1: tagged n

Program Instructions	Comments
`«`	

`"Enter the sample`
`mean (x̄)"`
	Prompt string for INPUT.

`":x̄:"`	Command-line string for INPUT.
`INPUT`	Waits for input of mean from the keyboard.
`OBJ→`	Converts keyboard input to a number.

`"Enter the`
`hypothesized`
`population mean (μ)"`
	Prompt string.

`":μ:"`	Command-line string.
`INPUT`	Waits for keyboard input.
`OBJ→`	Converts to number.

`"Enter the sample`
`standard deviation (s)"`
	Prompt string.

`":s:"`	Command-line string.
`INPUT`	Waits for keyboard input.
`OBJ→`	Converts to number.

`"Enter the sample`
`size (n)"`
	Prompt string.

`":n:"`	Command-line string.
`INPUT`	Waits for keyboard input.
`OBJ→`	Converts to number.
`»`	

Keystrokes	Comments
ENTER	Puts program on the stack.
' Getvars STO	Stores the program as *Getvars*.

Getvars contains four INPUT commands, which cause the subprogram to stop four times. You are prompted to enter the appropriate value each time.

INPUT takes two arguments: a prompt string from level 2 of the stack, and a command-line string from level 1. For the first INPUT command, the prompt string in level 2 is "Enter the sample mean (x̄)". By adding an endline character ((⮫) (↵)) after the word sample, you ensure that you'll be able to view the entire prompt message on two lines. (Without the added endline character, the prompt message will trail off the screen like this....) When shown by INPUT, the prompt message begins on the top row of the display, and is shown on rows 1 and 2. Your prompt messages can be shown on as many as three rows.

(If you have trouble typing the characters for $\bar{x}$ and μ, look at the table of Alpha Mode left- and right-shifted keys in the owner's manual or the quick reference guide. To type $\bar{x}$, for instance, you first make sure you're in alpha-entry mode; then press (⮫) (1/x) together.)

As for the INPUT's level 1 argument, a number of options are available. Here we've chosen to add only the command-line string ":(x̄:):".

When the first INPUT command is executed, here's what you see:

```
Enter the sample
mean (x̄)

:x̄:
```

The value you enter from the keyboard is combined with the command-line string by INPUT. By placing colons before and after the $\bar{x}$, we ensure that when you input a value for the mean, then execute OBJ→, the resulting object is a number that's tagged with the $\bar{x}$ label, like this:

```
x̄: 75.30
```

This tagged number can be used in mathematical formulas as if it didn't have a tag at all. In fact, if you run the *Getvars* subprogram by pressing 1TAIL or 2TAIL, you'll never see the tagged result. However, labeling values with tags is extremely useful in debugging.

The other INPUT commands function the same way: A prompt string is placed in level 2 of the stack, a command-line string is put in level 1, and INPUT is executed. At the end of *Getvars*, the four tagged values (for mean, the hypothesized μ, standard deviation s, and sample size n) are on the stack, ready for the calculations performed by *T1* or *T2*.

T1—Calculations Made Easy with Local Variables

If you pressed the 1TAIL menu key, the next subprogram executed after *Getvars* is *T1*. The *T1* subprogram calculates the *p*-value for a one-tail test. It takes four values (sample mean, hypothesized mean, sample standard deviation, and sample size) from the stack, and returns the *p*-value probability.

Arguments	Results
4: tagged $\bar{x}$	4:
3: tagged μ	3:
2: tagged s	2:
1: tagged n	1: tagged p-value

Program Instructions	Comments
«	Begins subprogram.
→ x u s n	Creates local variables.
«	Begins defining procedure for local variables.
'n-1' EVAL	Calculates degrees of freedom.
'(x-u)/s*√n' EVAL	Calculates *t*.
ABS	Ensures *t* is positive.
UTPT	Calculates *p*-value.
"1T p-value" →TAG	Labels the answer.
»	Ends defining procedure.
»	Ends subprogram.

To save this subprogram as *T1*:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' T1 STO	Saves the program.

T1 begins by taking all four quantities from the stack and converting them to local variables. As a shorthand, we've used the local variable x to identify the quantity $\bar{x}$, and u to identify μ. You could just as easily use the actual characters $\bar{x}$ and μ as the local variable names.

The defining procedure for the local variables must begin right after they are created. Here, as in most cases, the defining procedure is a program. The local variables have meaning only within this inner program.

The UTPT command requires the degrees of freedom in stack level 2, and the value for t in level 1. The statement 'n-1' EVAL computes the degrees of freedom and places it on the stack. Then the formula for t is entered and evaluated the same way. With the appropriate values in stack levels 2 and 1, UTPT computes probability—the p-value.

There's no law that says you have to use local variables for these calculations, of course. You could also have manipulated the stack, as we did in the keyboard example. But using local variables makes these calculations much more straightforward and easy to understand—all the more important when you don't add explanatory comments within your program code.

With the calculated p-value now on the stack, we want to label it. So we place the string "1T p-value" in level 1, moving the p-value up to level 2. The →TAG command then tags the value from level 2 with the tag from level 1. If you compare this procedure with that for INPUT, you'll notice that you don't need any colons for →TAG's string; the colon is added automatically.

T2

If you press 2TAIL to choose a two-tail test, the only difference from pressing 1TAIL is that *T2* is called instead of *T1*. The *T2* subprogram is almost the same as *T1*; it takes four values from the stack ($\bar{x}$, μ, s, and n) and returns the p-value. However, the p-value is double the probability calculated by UTPT, since this is for a two-tail test.

Arguments	Results
4: tagged $\bar{x}$	4:
3: tagged μ	3:
2: tagged s	2:
1: tagged n	1: tagged p-value

Program Instructions	**Comments**
`«`	Begins subprogram.
`→ x u s n`	Creates local variables.
`  «`	Begins defining procedure for local variables.
`  'n-1' EVAL`	Calculates degrees of freedom.
`  '(x-u)/s*√n' EVAL`	Calculates *t*.
`  ABS`	Ensures *t* is positive.
`  UTPT`	Calculates probability.
`  2 *`	Doubles calculated probability for two-tail p-value.
`  "2T p-value" →TAG`	Labels the answer.
`  »`	Ends defining procedure.
`»`	Ends main program.

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' T2 STO	Stores the program.

As you can see, within the defining procedure, *T2* takes the probability calculated by UTPT and doubles it to get the *p*-value. This accounts for the two-tail test.

Except for the call to *T1* or *T2*, pressing `1TAIL` or `2TAIL` executes all subprograms in the same order. The next call is to get the value of α.

Getalpha—INPUT with a Blank Command Line

Getalpha performs the final task necessary to get ready for the actual hypothesis test: it gets the level of significance, α, from the user. Getalpha takes nothing from the stack, and leaves the untagged value for α.

Arguments	**Results**
2:	2: *tagged p-value*
1: *tagged p-value*	1: α

Program Instructions	**Comments**
`«`	
`"Enter the desired level of significance (α)"`	Prompt string for INPUT.
`""`	Command-line string (blank) for INPUT.
`INPUT`	Waits for input of significance level (α).
`OBJ→`	Converts to number.
`»`	

Keystrokes	Comments
ENTER	
' Getalpha STO	Stores the Getalpha program.

Getalpha is similar to any of the INPUT commands used in *Getvars*. A prompt string is placed in level 2 of the stack, a command-line string is put in level 1, and INPUT is executed. The INPUT command halts execution until you press ENTER to signify that a number has been entered on the command line. (To enter the α character in the prompt string, make sure the calculator is in alpha mode, then press the ↱ and A keys.)

The major difference in how Getalpha uses INPUT is on the command line. Using the characters " " as the command-line string results in a completely empty command line, so when INPUT is executed, the command line is blank. You enter a number on this line in response to INPUT, and the number is placed on the stack alone, with no tag attached.

After Getalpha has been run, both the *p*-value and α are on the stack. It's time to perform the hypothesis test.

Test—Comparing Local Variables

Test compares the *p*-value and α, and determines whether to reject or fail to reject the null hypothesis. *Test* takes two values (*p*-value and α) from the stack, and returns the two values in reverse order.

Arguments	Results
2: tagged p-value	2: tagged α
1: α	1: tagged p-value

Program Instructions	Comments
«	Begins subprogram.
→ p a	Makes p-value and α local variables p and a.
«	Begins defining procedure for local variables.
IF 'p≤a'	Begins test structure.
THEN	If p is less than or equal to a,
a	puts a on the stack,
Reject	and calls the subprogram *Reject*.
ELSE	If p is greater than a,
a	puts a on the stack,
Failreject	and calls the subprogram *Failreject*.
END	End of test structure.
a	Puts the value of a on the stack.
"α"	Puts text string "α" on the stack.
→TAG	Tags the value of α.
p	Puts the tagged p-value on the stack.
»	Ends defining procedure for local variables.
»	Ends *Test* subprogram.

Keystrokes	Comments
ENTER	Puts *Test* program on the stack.
' Test STO	Stores the program.

The *Test* subprogram takes the p-value and α from the stack and turns them into local variables p and a. As soon as those local variables are created, the defining procedure (a program) begins.

The heart of the defining procedure is the IF ... THEN ... ELSE structure. Remember the rules for hypothesis testing using the p-value?

- If the p-value is less than or equal to α, reject the null hypothesis.
- If the p-value is greater than α, fail to reject the null hypothesis (and accept the alternate hypothesis).

You can easily see these conditions in the IF ... THEN ... ELSE construction. If ***p*** is less than ***a***, or equal to ***a***, it means that our assumption was correct. We can reject the null hypothesis (what we're trying, in fact, to disprove), and accept the alternate hypothesis. In this case, *Test* puts the value of ***a*** on the stack and calls the subprogram *Reject*.

If ***p*** is greater than ***a***, the THEN condition is false, and the ELSE condition is true. This means we have insufficient evidence to reject the null hypothesis. So *Test* places the value of ***a*** on the stack and calls the *Failreject* subprogram.

Test, then, is going to call either *Reject* or *Failreject*. Each of these subprograms takes the value of α from the stack, and displays a rather detailed message explaining the decision.

After executing the IF ... THEN ... ELSE structure, *Test* again places ***a*** on the stack, tags it as α, then places ***p*** on the stack. The variable ***p***, recall, was tagged earlier with the label "p-value". And through all of these machinations, being passed from one subprogram to the next, that value retains its tag. Thus, after you press either `1TAIL` or `2TAIL`, you wind up with the tagged α and p-value on the stack for your perusal or further use.

Reject—Adding a Calculated Value to the Display

Reject is called by *Test* if the p-value is less than or equal to α. It combines α with text strings to display a comprehensive message explaining the decision to reject the null hypothesis. *Reject* removes one value from the stack.

Arguments	Results
1: α	1:

Program Instructions	Comments
`«`	Begins subprogram.
`→ a`	Takes α from stack, creates local variable *a*.
`  «`	Begins defining procedure for local variable.
`  CLLCD`	Clears HP 48's LCD.
`  "REJECT:`	Begins first text string.
`  At the "`	Ends first text string.
`  a →STR`	Converts *a* into second text string.
`  +`	Combines first and second text strings.
`  "`	Begins third text string.
`  level of significance,`	
`  there is sufficient`	
`  evidence to reject the`	
`  null hypothesis (and`	
`  accept the alternate)."`	Ends third text string.
`  +`	Adds third string to combined first and second strings.
`  1 DISP`	Displays resulting long string.
`  3 FREEZE`	Freezes main display and status area.
`  0 WAIT`	Waits for a key press.
`  DROP`	Throws away address of the key that was pressed.
`  »`	Ends defining procedure for local variable.
`»`	Ends subprogram.

To save this subprogram in the current directory:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Reject STO	Stores the program as *Reject*.

Reject starts by taking the value of α from the stack and creating a local variable called *a*. Then the defining procedure for that local variable begins. The defining procedure is itself a program.

Within the defining procedure, the HP 48's display is cleared by CLLCD, then a short string is placed on the stack: `"REJECT:■At the "`. (The ■ stands for the endline character.) If these words seem rather incomplete, it's because this is actually the first of three text strings. *Reject* combines all three strings into a single long, highly detailed display. To make your displays look like the ones shown for this example, be sure to enter everything between quotation marks just as shown, and type endline characters, ↱ ↵, at the end of each line.

With the first string in the stack, *Reject* now places the value of *a* (that is, α), on the stack, too. The variable *a* is a number, so we use the `→STR` command to convert it to a string. Now two strings are on the stack, so it might look like this:

Stack Level	**Contents**
Level 2:	`"REJECT:■At the "`
Level 1:	`"0.05"`

Two strings are now on the stack. *Reject* executes + to add them together, leaving the resulting string in level 1:

Stack Level	**Contents**
Level 1:	`"REJECT:■At the 0.05"`

A third string is now placed on the stack by *Reject*. This string begins with quotation marks (") followed immediately by an endline character. (Press ↱ ↵ to type it.) Several lines of text follow, explaining the outcome and meaning of the test. Then *Reject* adds this string to the combined first and second string. The result is a long, detailed string that includes both text entered by the subprogram and a value you keyed in earlier.

The 1 DISP command displays that text, beginning in row 1 of the HP 48's liquid crystal display. Here's an example of the result you can expect:

```
REJECT:
At the 0.05
level of significance,
there is sufficient
evidence to reject the
null hypothesis (and
accept the alternate).
```

See how we left a fairly long space to insert the α value? This prevents subsequent words from trailing off the screen if you happen to be showing numbers like α with a lot of displayed digits (for example, as 0.0500000).

In order to hold the display on the screen until the next key press, we use 3 FREEZE. With 3 as its argument, FREEZE prevents the updating of anything in the display except the menu area until the next key press.

Reject is an instance where we want to leave the message on the screen until a key press, but don't want the key press to do anything. That is, we want to be able to press any key to continue, but not have that key affect the stack or the calculator. Thus the next command, 0 WAIT, which waits for the press of a key. With an argument of 0, WAIT "swallows up" the next key press so that—except for ATTN—the key doesn't perform its usual function. You can press ENTER or STO or 1, and not worry about generating an error.

0 WAIT does do one thing in response to a key press, though: It returns the address of the key. We don't need that address, so before exiting *Reject*, we execute DROP to drop the stack one level, effectively expunging the key address from the stack.

Failreject

The *Failreject* subprogram is just like *Reject*, except, of course, the displayed message is different. *Failreject* is called by *Test* if the calculated p-value is greater than α. It takes one quantity (α) from the stack, and returns nothing.

Arguments	Results
1: α	1:

Program Instructions	**Comments**
`«`	Begins subprogram.
`→ a`	Takes α from stack, creates local variable *a*.
`  «`	Begins defining procedure for local variable.
`  CLLCD`	Clears HP 48's LCD.
`  "FAIL TO REJECT:`	
`  At the "`	Begins text string.
`  a →STR`	Converts *a* to text.
`  +`	Adds the two strings together.
`  "`	Begins another text string. (↱ ↵ after ".)
`  level of significance,`	
`  there is insufficient`	
`  evidence to reject the`	
`  null hypothesis."`	
`  +`	Adds strings together.

Program Instructions	Comments
`1 DISP`	Displays complete text string.
`3 FREEZE`	Freezes main display and status area.
`0 WAIT`	Waits for a key press.
`DROP`	Gets rid of key address.
`»`	Ends defining procedure.
`»`	Ends subprogram.

To save this subprogram:

Keystrokes	Comments
ENTER	
' Failreject STO	Stores the program.

Notice that we can't say "accept the null hypothesis." All that the hypothesis test tells us is that either we have enough information to reject the null hypothesis, or else we don't have enough information to reject it. Hence the headline "FAIL TO REJECT".

Exit—Changing Menu Displays

There's one more menu option to consider. If you run *Hypotest* and press the EXIT key, the *Exit* subprogram is run. *Exit* simply changes the menu back to the ordinary VAR menu, to give an indication that you're done with *Hypotest*'s temporary menu for now. *Exit* has no effect on the stack.

Program Instructions	Comments
`«`	
`0 MENU`	Changes back to previous menu display.
`»`	

Keystrokes	Comments
ENTER	Puts *Exit* program on the stack.
' Exit STO	Stores the program.

As written now, *Exit* merely gives you some visual feedback when you're done. It changes the displayed menu from the temporary menu back to the previous VAR menu, but doesn't otherwise affect the calculator.

Although it may seem trivial, having an exit routine like this is always a good idea. It gives you a place to reset flags and add other cleanup commands as your application grows.

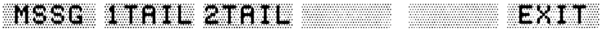
Running the Hypotest Program

Let's try running *Hypotest* to check out that coach's contention. The hypotheses are as follows:

- ***H_0*, *null hypothesis***: The mean of athletes' scores is equal to or less than 72.
- ***H_A*, *alternate hypothesis***: The mean of athletes' scores is greater than 72.

The mean of football players' scores is 75.3. Standard deviation is 13, and sample size is 40. Test at a level of significance of 0.05.

To get started, press the HYPOT key to fire up *Hypotest*. You see the display of menu keys:



First look at the message:

Program Prompt or Display	Your Action
	MSSG
HYPOTHESIS TEST: Tests null hypothesis at alpha; level of confidence. Choose one-tail if H0 contains ≤ or ≥, two-tail if = only.	

The null hypothesis contains the words "equal to or less than," so this is a one-tail test. Press 1TAIL.

Program Prompt or Display	Your Action
	1TAIL
Enter the sample mean ($\bar{x}$) :($\bar{x}$):	75.3 ENTER
Enter the hypothesized population mean (μ) :μ:	72 ENTER
Enter the sample standard deviation (s) :s:	13 ENTER
Enter the sample size (n) :n:	40 ENTER
Enter the desired level of significance (α)	.05 ENTER

Now you see the answer in the display.

Program Prompt or Display **Your Action**

```
FAIL TO REJECT:
At the 0.05
level of significance,
there is insufficient
evidence to reject the
null hypothesis.
```

Although the coach may be able to sway the alumni and fans, the school's statisticians will remain unconvinced by the claim that football players' scores are higher than the school average.

When you exit the program, the values for alpha and the calculated *p*-value are left on the stack:

Program Prompt or Display **Your Action**

EXIT

```
2:              α: 0.05
1:    1T p-value: 0.06
```

To see what happens with a little more allowable error, change the desired level of significance to 0.10 instead of 0.05. What happens to the coach's hypothesis now?

5

Calculus and Analytic Geometry

If ever a calculator was built for calculus, the HP 48 is it. With its full complement of built-in functions, including derivatives, integrals, and curve drawing, the HP 48 is astonishingly well-equipped—right out of the box—to handle most calculus problems. It's so powerful and sophisticated, in fact, that it's ready to go, with no programming whatsoever needed on your part.

Nevertheless, you may want to incorporate this powerful arsenal of calculus features into a program. You can add a "shell" of comprehensive user instructions to an existing feature, including detailed prompts—for example, a prompt for the limits over which a curve is drawn. Naturally, you can include calculus commands in other programs, too, using them as you would any mathematical function. This chapter illustrates a few ways the HP 48 can help you get the most from a calculus class or a calculus-intensive application.

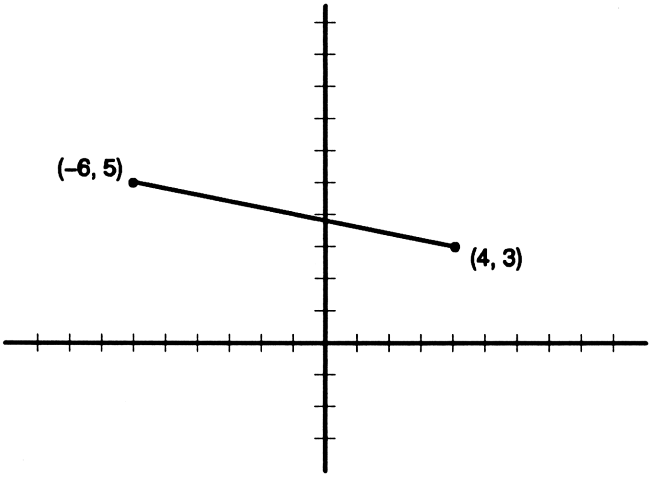
Distance Between Two Points

One of the first things required by almost any basic mathematics class—including first-quarter calculus—is to find the distance between two points in the Cartesian coordinate system. You can laboriously grind it out using the Pythagorean theorem, of course. But an easier way is to use the HP 48's remarkable ability to convert between rectangular and polar coordinates, and to subtract rectangular coordinates.

Keyboard Example

When the HP 48 is in rectangular mode, numbers in parentheses—that is, complex numbers—represent points in the Cartesian system. Thus (1,2) is a point 1 unit along the x-axis and 2 units along the y-axis.

You can actually key in numbers in this format, using the parentheses and comma on the calculator. To find distance, you subtract two such points in rectangular mode, then switch to polar mode and read the distance. For example, consider the distance between the points (4,3) and (−6,5) in the Cartesian coordinate system.



To find the distance between the points (4,3) and (−6,5), you can use the following procedure.

Keystrokes	Display	Comments
⬅ MODES NXT NXT DEG		Sets degrees mode.
➡ POLAR		Switches to rectangular mode; press until the R∡Z annunciator is *off*.
⬅ ( )	( )	
4 ⬅ , 3 ENTER	(4.00,3.00)	Enters first point. (You can use SPC instead of the comma if you want.)
⬅ ( )	( )	
6 +/− ⬅ , 5	(-6,5)	Enters second point.
−	(10.00,-2.00)	Subtracts rectangular coordinates.
➡ POLAR	(10.20,∡-11.31)	Distance and angle.

When you switch to polar mode with ➡ POLAR, you see the distance (10.2 units), as well as the angle in degrees between the two points.

That's not too difficult. But if you didn't do it every day, you could easily forget the procedure. So let's see how we can accomplish the same thing in a program, one that will prompt for input and give us the correct result every time.

The Distance Directory

To create a directory for the Cartesian distance programs, then get into that directory:

['] DISTANCE
[⬅] [MEMORY] CRDIR
[VAR]
DISTA

The DISTANCE directory will hold all programs and subprograms for calculating distance. Any objects (including programs and equations) that you enter and save will be placed in this directory.

The Main Distance Program

The main *Distance* program consists of nothing more than calls to seven subprograms. In the course of the program, you are asked to input point 1 and point 2; the program leaves the value for distance on the stack.

Arguments	Results
1:	1: tagged distance

Program Instructions	Comments
«	
Init	Initializes for rectangular and degrees modes.
P1get	Gets first point.
P2get	Gets second point.
Compute	Subtracts second point from first.
Change	Changes to polar mode.
Display	Displays the distance only.
»	

To save the main *Distance* program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Distance STO	Stores the program.

The program begins with an initialization routine, *Init*, that places the calculator in rectangular and degrees modes. It then calls a pair of subprograms, *P1get* and *P2get*, to prompt for the two Cartesian points. These points are subtracted, by *Compute* and the result (still in rectangular mode) is passed to *Change*. The *Change* subprogram switches to polar mode, which automatically converts any complex number in the stack to polar mode. Finally, *Display* tags and displays the result.

Init—Using Flags to Guarantee Status

The *Init* subprogram clears several system flags to ensure that HP 48 status is the way you want it before actual calculations begin. *Init* has no overall effect on the stack.

Program Instructions	**Comments**
«	
-15 CF	
-16 CF	Ensures rectangular mode.
-17 CF	
-18 CF	Ensures degrees mode.
-19 SF	Ensures that →V2 creates complex numbers from real numbers.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Init STO	Stores the program as *Init*.

Flags −15 and −16 are used together to specify the type of coordinates assumed by the HP 48. To specify rectangular mode, coordinate system flags −15 and −16 must both be clear. By specifying rectangular mode here, *Init* guarantees that any subsequent operations involving complex numbers, such as (3,5), assume the value is a Cartesian coordinate. (Later in the program, we'll change the status of flag −16, which will change HP 48 status to polar—and any complex numbers to polar form.)

Flags −17 and −18 are also used together. They specify the angle mode for trigonometric functions. When both are clear, the calculator is in degrees mode.

Setting flag −19 ensures that the →V2 command we use later creates a complex number from two real numbers (instead of creating a two-dimensional vector). Interestingly, the setting of this flag has no effect on the final output of the *Distance* program. However, it will affect your intermediate results.

P1get—Multiple Inputs from One Prompt

After initialization, the next subprogram called by *Distance* is *P1get*. This subprogram prompts for input of the first point, and lets the user enter one coordinate on one line, and the second coordinate on another line. *P1get* takes nothing from the stack, and leaves a complex number—two coordinates in parentheses.

Arguments	Results
1:	1: (x1,y1)

Program Instructions	Comments
`«`	
`"Enter point P1"`	Prompt string for use by INPUT.
`{`	Beginning of command-line list for INPUT.
`":x1:`	Prompt for *x1*. (Add a ↱ ↵.)
`:y1:"`	Prompt for *y1*.
`{ 1 0 }`	Places cursor in first row of command line.
`}`	End of command-line list for INPUT.
`INPUT`	Prompts for input, using prompt string and command-line list.
`OBJ→`	Converts resulting string into its component objects.
`→V2`	Combines two real numbers into complex number.
`»`	

To save the *P1get* program:

Keystrokes	Comments
ENTER	
' P1get STO	Saves the program.

P1get makes use of the INPUT command to prompt for and process input from command-line rows. INPUT takes as its arguments a prompt string from level 2 and a command-line string from level 1. The command pauses program execution with the cursor on the command line, so that you can enter the needed value or values. When you enter the data and press ENTER, the program continues.

Let's take a close look at those prompt and command-line strings. *P1get* begins by placing the string `"Enter point P1"` on the stack. This is the prompt string used by INPUT.

The next item placed on the stack is a long list. This list contains a command-line string and an inner list with two numbers in it. The complete list is also used by INPUT. Here's what those different items mean:

- The string `":x1:■:y1:"` appears on the command line when INPUT is executed. If you place an endline ((↱) (↵)) after `:x1:` as you're supposed to, this string will occupy *two* rows of the display. (The ■ indicates the newline character.)
- The list `{ 1 0 }` specifies the row and column where the cursor will be placed on the command line. The first number in the list, 1, specifies row 1 (the top row of the two). The second number in the list, 0, specifies that the cursor will be at the end of that row. Because the number 1 is positive, you will see an *insert* cursor instead of a replace cursor, although this doesn't really matter here.

Thus, when you run *P1get* and INPUT is executed, you see the following display:

```
Enter point P1:

:x1:
:y1:
```

The cursor is initially at row 1, the row labeled `:x1:`. You enter the x-coordinate for the first point here, then use (▼) to move to the bottom row, where you enter the y-coordinate. Then you press (ENTER) and the program continues.

INPUT combines your keyboard input with the command-line string, and places the entire quantity on the stack. Let's say you enter values for the point (4,3). After INPUT is executed, the stack contains the following string:

```
":x1:4■:y1:3"
```

Next we want to take that string and convert it to its component objects. So we execute OBJ→. This command converts the string into two tagged objects:

```
x1: 4.00
y1: 3.00
```

With these two numbers now on the stack, all that remains is to combine them into a complex number. The →V2 command takes the two values from level 2 and level 1 of the stack, strips off the tags, and places them together in a set of parentheses. This command even adds the comma between them for you.

```
(4.00,3.00)
```

The setting of flag −19 determines what is done by →V2. If this flag is clear, →V2 creates a two-dimensional vector from two real numbers on the stack; if it's set (we set it in *Init*, remember?), →V2 creates a complex number. The overall *Distance* application gives identical answers regardless of the setting of flag −19, but the intermediate results may be different.

With the coordinates for the first point now on the stack, in parentheses, we're ready to get the second point.

P2get

With the exception of its prompt and command-line strings, *P2get* is identical to *P1get*. It takes nothing from the stack, and leaves the coordinates for the second point there. After running *P2get*, the stack contains both pairs of coordinates.

Arguments	Results
2:	2: (x1,y1)
1: (x1,y1)	1: (x2,y2)

Program Instructions	Comments
`«`	
`"Enter point P2"`	
`{`	
`":x2:`	Add a ↱ ↵
`:y2:"`	to this prompt for *y2*.
`{ 1 0 }`	
`}`	
`INPUT`	Prompts for input.
`OBJ→`	Converts resulting string.
`→V2`	Combines into complex number.
`»`	

Keystrokes	Comments
ENTER	
' P2get STO	Saves the subprogram.

After *P1get* and *P2get* are called, the stack has both points, in coordinate form, all ready to be combined.

Compute—Using Local Variables for Math

The next subprogram called by *Distance* is *Compute*. It finds the difference between two sets of coordinates, and puts the difference on the stack as a complex number. *Compute* gets two complex numbers from the stack, and returns a single complex number.

Arguments	Results
2: (x1, y1)	2:
1: (x2, y2)	1: (x,y)

Program Instructions	Comments
`«`	Begins program.
`→ p1 p2`	Creates local variables.
`  «`	Begins defining procedure (a program) for local variables.
`  'p1-p2' →NUM`	Subtracts point 2 from point 1.
`  »`	Ends defining procedure.
`»`	Ends program.

To store this subprogram as *Compute*:

Keystrokes	Comments
ENTER	Places subprogram on the stack.
' Compute STO	Stores subprogram.

The *Compute* subprogram simply subtracts one point from the other. With coordinates for the two points on the stack, you could very easily subtract with −. But we'll turn the two points into local variables and subtract them algebraically. This will illustrate how local variables are used, and also will make the entire *Distance* program easier to understand.

Compute offers a very simple illustration of how to create and use local variables. The subprogram expects a pair of complex numbers on the stack. The expression `→ p1 p2` takes these two complex numbers and turns them into local variables.

In creating local variables, the value from stack level 1 is used by the *final* variable in the list. The value from stack level 2 is used by the next-to-last variable, and so on. Thus, the complex number representing the coordinates for point 2 is assigned to *p2*, and the complex number representing point 1 is assigned to *p1*.

In order for local variables to be created, the defining procedure must *immediately* follow the declaration. Here, as in most cases, we're using a program as the defining procedure. Those local variables, *p1* and *p2*, exist only within the defining procedure—that is, between the internal set of program braces, « and ».

To use the local variables, we place them into an algebraic object, the equation `p1-p2`. Then we evaluate that object. Some other programs in this book use EVAL for this purpose, but here we've used →NUM to actually evaluate the equation into a numerical result.

The result, which is placed on the stack, is the difference between *p1* and *p2*—and since both of these quantities are complex numbers, the result is also a complex number.

The HP 48 is still in rectangular mode, so the complex number left on the stack by *Compute* represents the distance in the x-direction and the distance in the y-direction between points *p1* and *p2*. The next thing to do is convert this quantity into polar distance and angle.

Change—A Programmatic Rectangular-to-Polar Conversion

The *Change* subprogram takes a complex number representing rectangular distances, and converts it to polar distance and angle. In fact, *Change* converts *every* complex number on the stack to polar.

Arguments	Results
1: (x,y)	1: (distance, angle)

Program Instructions	Comments
«	
-16 SF	Changes from rectangular to polar mode.
»	

Keystrokes	Comments
ENTER	
' Change STO	Stores the program as *Change*.

Recall that to convert a rectangular quantity to polar from the keyboard, you simply press ↱ POLAR. This command isn't programmable, but you can accomplish the same thing with system flag −16.

When flags −15 and −16 are both clear, the HP 48 is in rectangular mode. To change to polar/cylindrical mode, we set flag −16 with the SF command. Notice that if you have other complex numbers elsewhere on the stack, they'll be changed to polar magnitude and angle, too.

Display—Tagging Output

The *Display* subprogram gets a polar complex number from the stack, throws away the angle portion, and tags the distance, leaving it on the stack.

Arguments	Results
1: (distance, angle)	1: tagged distance

Program Instructions	Comments
`«`	
`V→`	Separates polar vector into its elements (distance and angle).
`DROP`	Gets rid of the angle.
`"Distance" →TAG`	Creates tagged object for easy identification.
`»`	

To save the *Display* program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Display STO	Stores it.

The *Display* subprogram makes use of the V→ command to take apart the complex number. In polar mode, with a complex number in level 1, V→ does the same thing as ↰ 2D from the keyboard: the magnitude is returned to stack level 2 and the angle is returned to level 1. Thus, after V→ is executed, the stack contains two quantities:

Stack Level	**Contents**
Level 2:	magnitude
Level 1:	angle

For our purposes, we don't care about the angle. So *Display* uses DROP to drop the stack, throwing away the angle and leaving the magnitude—that is, the polar distance between the two points—in level 1.

To make the output easier to read, we label it with →TAG. This command takes two arguments: the value for distance from level 2, and the tag from level 1. It combines them to form a tagged object. This tagged value for distance is just like any other number—you can add, subtract, multiply, or divide it—except that it now has a label, making it easy to identify.

Running the Distance Program

Run the distance program and see if you come up with the same value as calculated from the keyboard. For our illustration, we want the distance between the points (4,3) and (−6,5). Press DISTA to begin the program.

Program Prompt or Display	**Your Action**
	DISTA
Enter point P1	
:x1:	4 ▼
:y1:	3 ENTER
Enter point P2	
:x1:	6 +/- ▼
:y1:	5 ENTER

```
1: Distance: 10.20
```

There's the correct distance, all right. Armed with this little program, you're ready to calculate distance on any Cartesian grid.

The *Distance* program has been kept simple on purpose. You might want to add a call to a cleanup subprogram that resets the calculator for rectangular mode (by clearing flag −16 again), and performs other tasks.

Slope of a Straight Line

Most of the HP 48's commands are programmable, but the SLOPE menu key (in the GRAPHICS FCN menu) is one that isn't. To find the slope of a line, you can use this GRAPHICS FCN menu, but if you just need to find several slopes in a hurry—particularly if the lines are straight—that's like using a cannon to blast a flea.

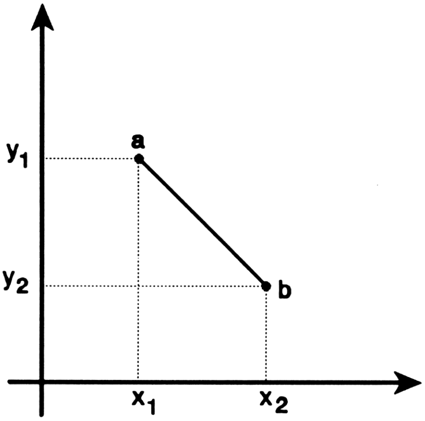
Keyboard Example

To find the slope of a line between points *a* and *b*, the following formula is used:

$$m = \frac{y2 - y1}{x2 - x1}$$

where

m is the slope.
x1 and *y1* are the coordinates of point *a*.
x2 and *y2* are the coordinates of point *b*.



Thus, to find the slope of the line connecting points (2,8) and (12,3), you could use the following procedure:

Keystrokes	Display	Comments
3 ENTER 8 −	-5.00	Calculates *y2−y1*.
12 ENTER 2 −	10.00	Calculates *x2−x1*.
÷	-0.50	Slope, *m*, of line.

This example uses the stack to calculate slope, and it works pretty well. Now let's see how we can accomplish the same thing using a program—with a few enhancements that make the whole procedure a "no-brainer."

The Slope Directory

To create a directory for the *Slope* programs, then get into that directory:

' SLOPE
↰ MEMORY CRDIR
VAR
SLOPE

The SLOPE directory will hold all programs and subprograms for calculating slope. Any objects (including programs and equations) that you enter and save will be placed in this directory.

The Main Slope Program

The *Slope* program is a fast, simple way to calculate the slope of a straight line. It prompts for the input of both coordinates of one point, followed by the coordinates of the second point. Then it displays the slope, labeled with a tag for easy identification.

The main *Slope* program consists of calls to three subprograms. When viewed as an overall program, *Slope* takes nothing from the stack, and leaves the tagged slope there.

Arguments	Results
1:	1: tagged slope

Program Instructions	Comments
«	
Xy1	Gets point *(x1,y1)*.
Xy2	Gets point *(x2,y2)*.
Findslope	Calculates and tags the slope.
»	

To save this program:

Keystrokes	Comments
ENTER	
' Slope STO	Saves the program as *Slope*.

This simple program is another classic example of structured programming. The main *Slope* program consists entirely of calls to subprograms. *Xy1* prompts for entry of coordinates for the first point, while *Xy2* gets coordinates for the second point. (It actually doesn't matter which point is designated point 1 and which is point 2.) After these two programs have been called, the coordinates for both points are on the HP 48's stack. Then *Slope* calls the subprogram *Findslope*, which handles the actual calculation and display.

Xy1—Checking for Valid Input

The program *Xy1* prompts and waits for input of coordinates for the first point. It allows input of both the x- and y-coordinate in the display, and verifies that the entry is made up of valid objects. *Xy1* takes nothing from the stack and leaves both coordinates there; the coordinates are tagged for identification.

Arguments	Results
`2:`	`2:` tagged x1
`1:`	`1:` tagged y1

Program Instructions	Comments
`«`	
`"Enter x1 and y1"`	Prompt string for use by INPUT.
`{`	Begins command line list.
`":x1:`	Prompt for *x1*. (Add a ↱ ↵.)
`:y1:"`	Prompt for *y1*.
`{ 1 0 }`	Places cursor in first row of command line.
`V`	Checks for proper syntax.
`}`	Ends command-line list for INPUT.
`INPUT`	Prompts and waits for input of *x1* and *y1*.
`OBJ→`	Converts input into its component objects.
`»`	

Xy1 is very similar to *P1get*, used in the *Distance* program. However, it adds a validity checking to the INPUT command.

INPUT uses both a general prompt string and a command-line list. When called by *Slope*, the *Xy1* subprogram places a general prompt string on the stack, then begins the command-line list. This list contains one more element than *P1get*: the letter V. Here's what the different parts of that command-line string mean:

- The string ":x1: ■:y1:" appears on the command line when INPUT is executed. (■ shows location of the newline character.)
- The list { 1 0 } specifies the row and column where the cursor will be placed on the command line. Thus, the cursor is placed in row 1 (the top of two rows), column 0.
- The V verifies that the characters in the result string—that is, the characters you enter in response to INPUT—are a valid object. You get a warning message, and INPUT re-prompts, if you fail to enter a valid object.

Using V here gives an added measure of error prevention right in the INPUT command. When INPUT is executed, the program pauses and waits for input from the keyboard, displaying the following message and prompts just as *P1get* does:

```
Enter point P1:

:x1:
:y1:
```

You enter an x-value at the x1 prompt, then use the down arrow key to go to the y1 prompt and enter another value. Because you've included the V in the command-line string, INPUT checks to see if all parts of your input make up a valid object before continuing.

After INPUT of both points, a string is on the stack that consists of something like the following:

Stack Level	**Contents**
Level 1:	":x1:2■:y1:8"

The OBJ→ command converts the string into two tagged objects, breaking the string where the newline (■) character is placed. The result is that *Xy1* leaves two tagged objects on the stack:

Stack Level	**Contents**
Level 2:	`:x1: 2`
Level 1:	`:y1: 8`

Xy2—More of the Same

The *Xy2* program accomplishes virtually the same thing as *Xy1*, except that it gets the second point *(x2,y2)*. *Xy2* takes nothing from the stack, and places coordinate *x2* in level 2 and *y2* in level 1.

Arguments	**Results**
2:	2: tagged x2
1:	1: tagged y2

Program Instructions	**Comments**
`«`	
`"Enter x2 and y2"`	Prompt string for use by INPUT.
`{`	Begins command line list.
`":x2:`	Prompt for *x2*. (Add a ↱ ↵.)
`:y2:"`	Prompt for *y2*.
`{ 1 0 }`	Places cursor in first row of command line.
`V`	Checks for proper syntax.
`}`	Ends command-line list for INPUT.

Program Instructions	Comments
`INPUT`	Prompts and waits for input of *x2* and *y2*.
`OBJ→`	Converts input into its component objects.
`»`	

After *Xy2* is executed, the stack contains all four coordinates, arranged as follows:

Stack Level	Contents
Level 4:	*x1*
Level 3:	*y1*
Level 2:	*x2*
Level 1:	*y2*

With the four needed points on the stack, all that remains is to calculate the slope.

Findslope—Using Local Variables

Findslope is the last subprogram called by *Slope*. It takes four quantities from the stack and calculates the slope, tagging the result and leaving it on the stack.

Arguments	Results
4: tagged x1	4:
3: tagged y1	3:
2: tagged x2	2:
1: tagged y2	1: tagged slope

Program Instructions	Comments
`«`	
`→ x1 y1 x2 y2`	Converts to local variables.
`  '(y2-y1)/(x2-x1)'`	Defining procedure (an algebraic) for local variables.
`'m'`	Places the letter "m" on the stack.
`→TAG`	Tags the slope with "m".
`»`	

Keystrokes	Comments
ENTER	
' Findslope STO	Stores the program.

This is a case where you *could* use stack operations to calculate the slope. But it's so much easier to understand what's happening if you use local variables instead.

Findslope takes four quantities from the stack, and converts them immediately to the local variables *x1*, *y1*, *x2*, and *y2*. To help you remember how local variables are created, think of their creation as occurring from right to left. That is, *y2* is created from the quantity in level 1, *x2* from level 2, *y1* from level 3, and so on.

As soon as the local variables have been created, their *defining procedure* is executed. Here, the defining procedure isn't a program. Instead, it's the algebraic object `'(y2-y1)/(x2-x1)'`—which is easily recognizable as the formula for slope of a straight line.

Since we're using an algebraic object as the defining procedure, instead of using a program, we don't need the EVAL command. We wind up with a smaller, more concise overall program, one that uses less memory.

The parentheses are necessary, because without them, the HP 48 would use its own order of evaluation, performing division first, followed by subtraction. Without the parentheses in this expression, what you'd wind up with would be the result of the following formula:

$$y2 - \frac{y1}{x1} - x1$$

which isn't the result you want at all. However, the HP 48 evaluates quantities inside parentheses first, then works outward. Even if you're not sure of the order of evaluation, using parentheses can guarantee you get what you want.

Placing the object `(y2-y1)/(x2-x1)` inside single quotation marks in the program prevents each of those variables from being evaluated "on the fly." For instance, if this program "executed" the local variable `x1`, what you'd get would be the value (1 or 3 or 99, or something) assigned to that variable. By putting the variable or expression inside single quotation marks, we make sure it's evaluated all at once.

The entire expression is evaluated, using the current values for local variables *x1*, *y1*, *x2*, and *y2*. The result of executing this algebraic object, the slope of the line, is placed on the stack.

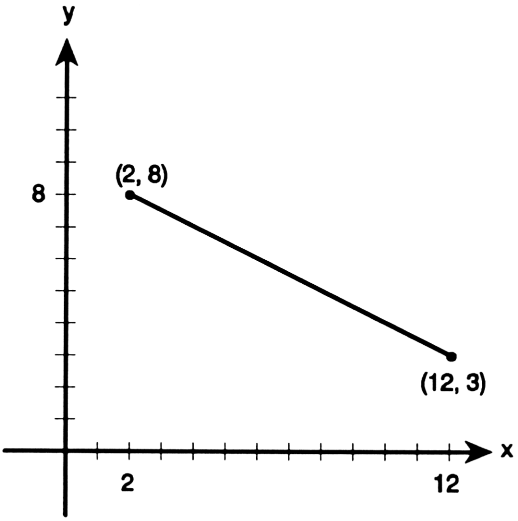
All that's left is to put a suitable label on our output. When the program adds `'m'` to the stack, the stage is set for the →TAG command.

Stack Level	Contents
Level 2:	slope of the line
Level 1:	`'m'`

→TAG combines two objects from the stack to form a single tagged object. The quantity comes from level 2, and the tag from level 1. The tag is usually a string, marked by double quotation marks (" "). Here, however, we've used a name marked with single quotation marks (tick marks), which is also valid. When →TAG is executed, the result is the slope of the line, labeled with an identifying tag of `m`.

Running the Slope Program

The *Slope* program quickly gives you the slope of the line between any two points in the Cartesian coordinate system. Here's an example of finding the slope between a line drawn from point (2,8) to point (12,3).



Press SLOPE to begin the program.

Program Prompt or Display	Your Action
	SLOPE
Enter x1 and y1	
:x1:	2 ▼
:y1:	8 ENTER
Enter x2 and y2	
:x2:	12 ▼
:y2:	3 ENTER
m: -0.50	

The result is a nicely labeled slope. Since the slope in the stack is a tagged number, you can use it in other calculations without modification.

Performing Integration Programmatically

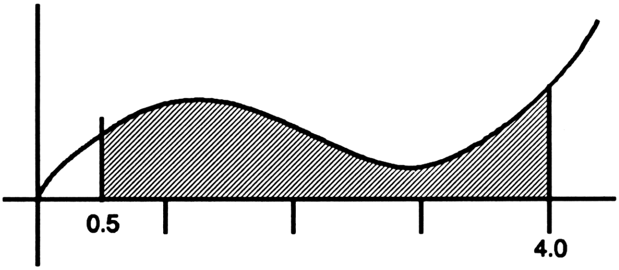
Calculus is one of the things the HP 48 does best. Its calculus functions are so sophisticated and well-designed that you may never need to program them. However, you can easily add a "shell" to an existing function, with user instructions that prompt for the exact input and with a label for the result. The *Integrator* program below is such a shell.

Keyboard Example

Consider the following integral:

$$\int_{0.5}^{4} \left(x^3 - 6x^2 + 9x + 1\right) dx$$

One way to think about this integral is that it's the area under the curve from 0.5 to 4.



The HP 48 has several ways to calculate this integral. One method (the one that is easiest to program) is to place all the necessary quantities in the stack, then use the $\int$ command. For numerical integration, the $\int$ command requires the HP 48's numerical results flag (system flag −3) to be set, and it needs the following quantities in the stack:

Stack Level	Contents
Level 4:	lower limit of integration
Level 3:	upper limit of integration
Level 2:	integrand (the equation)
Level 1:	variable of integration

Before performing the integration, you need to decide the accuracy factor, which is determined by the prevailing display format. Thus, if you want an accuracy of 0.01 (that is, 1%), you specify 2 FIX before performing the integration.

The keystrokes for getting these quantities into the stack and computing the integral are shown below.

Keystrokes	**Display**	**Comments**
⬅ MODES 2 FIX		Specifies accuracy to two decimal places.
3 +/- PRG TEST NXT NXT SF		Sets flag to produce numerical results.
0.5 ENTER	0.50	Enters lower limit of integration.
4 ENTER	4.00	Enters upper limit of integration.

With the limits of integration on the stack, you can enter the integrand—that is, the equation. The easiest way to handle this is to switch to the Equation Writer, enter the equation, and place it on the stack.

Keystrokes	Comments
↰ EQUATION	Switches to Equation Writer.
X y^x 3 ▶ − 6X	
y^x 2 ▶ + 9X + 1	Enters equation into Equation Writer.
ENTER	Places equation on stack.

Now you add the variable of integration, X, and perform the integration:

Keystrokes	Display	Comments
' X ENTER	'X'	Puts variable of integration on stack.
↱ ∫	10.61	Area under the curve.

Frankly, if you have just a single problem to solve, this method is less efficient than several other HP 48 integration options. But the above procedure is easily converted to the *Integrator* program, which can prompt for all necessary values and label the result. It's just the ticket for grinding through long problem sets. Read on!

The Integrator Directory

Before you begin writing the *Integrator* program and subprograms, make a directory for them, then get into that directory:

' INTEGRATOR
↰ MEMORY CRDIR
VAR
INTEG

This INTEGRATOR directory will hold all the necessary programs and subprograms for the *Integrator* program.

The Main Integrator Program

As with other structured programs, the main *Integrator* program is made up entirely of calls to subprograms. When executed in order by *Integrator*, these subprograms prompt for the input of accuracy level, then halt to allow you to input the equation (using the HP 48's Equation Writer, if you want). You are then prompted to enter the limits of integration, and the result is calculated. The program expects nothing on the stack, and leaves the tagged result there.

Arguments	Results
1:	1: tagged answer

Program Instructions	Comments
«	
Init	Initializes HP 48.
Accuracy	Prompts for input of desired accuracy level.
Equation	Prompts for input of equation as function of X.
Limits	Prompts for limits of integration.
Doit	Performs the integration.
Labelit	Tags the answer.
»	

To save this program as *Integrator*:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Integrator STO	Stores the program.

The *Init* subprogram sets up the HP 48 for the other subprograms to come. *Init* is followed by *Accuracy*, which lets you enter the number of decimal places of accuracy to which you want to see the result.

The *Equation* subprogram is executed next. It prompts you to enter the equation as a function of X, then turns over control to the calculator keyboard. You can enter the equation any way you want—in the command line or using the Equation Writer. When done, you use CONT to continue.

Program execution resumes with *Limits*, which prompts you to enter first the lower limit of integration, then the upper limit. With the equation and the limits of integration on the stack (and the calculator set to the correct number of decimal places for the accuracy you want), *Doit* performs the actual integration and produces the answer. Finally, *Labelit* tags the answer with a suitable label for easy identification.

Init—Specifying Numerical Results Mode

The *Init* subprogram consists of a single command that sets user flag −3. It doesn't alter the stack.

Program Instructions	**Comments**
«	
`-3 SF`	Sets numerical results mode.
»	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Init STO	Stores the program.

When user flag −3 is clear, you might wind up with symbolic results instead of the numeric ones you're looking for. So *Init* sets this flag to guarantee you'll get a numeric output.

Init is pretty small right now, and this single instruction could easily be placed in another subprogram. But creating a separate *Init* subprogram to like this makes *Integrator* easy to modify if you want to set other initialization conditions at a later date.

Accuracy—Setting the Number of Displayed Digits

Accuracy prompts you to enter the desired level of accuracy. *Accuracy* uses the FIX command to set the number of displayed digits; it has no effect on the stack contents.

Program Instructions	Comments
«	
"Enter accuracy as no. of decimal places (eg. 3 for 0.001%)"	Prompt-line string for INPUT. (Place endline characters, ⮕ ↵, at the end of each line as shown here.)
""	Command-line string (a blank string) for INPUT.
INPUT	Halts program for user input.
OBJ→	Converts input to a real number.
FIX	Specifies number of displayed decimal places.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Accuracy STO	Stores the program as *Accuracy*.

Accuracy begins by placing two text strings on the stack. Both are required by the INPUT command that follows. The first text string will prompt the user for input, while the second appears on the command line. That second text string is a null string ("").

When INPUT is executed, it displays the two text strings, and waits for input from the keyboard. The first text string occupies three rows in the display. (In fact, this is the maximum number of rows you can display with INPUT.) Nothing is really displayed by the second text string, but it's still needed by INPUT.

In doing numerical integration from the keyboard, you set the accuracy factor by means of the ↰ MODES FIX key. You input the number of decimal places to which you want the answer to be accurate. For instance, if you want an accuracy of 0.001%, you execute 3 FIX .

That's what you do in *Accuracy*, too. When prompted by INPUT, you enter the number of digits of accuracy that you want. The value you input is converted to a real number by OBJ→, then used as the argument for FIX. After *Accuracy*, the HP 48 is set for numerical integration at the accuracy level you desire.

Equation—Temporarily Returning to the Keyboard

The next subprogram, *Equation*, prompts you to enter the equation as a function of X, then returns control to the keyboard. When you've entered the equation, you hit ↰ CONT to continue the program. *Equation* takes nothing from the stack, and leaves your equation there.

Arguments	Results
1:	1: equation as function of X

Program Instructions	Comments
«	
"Enter EQUATION as f(X), then press CONT."	Prompt string for PROMPT.
PROMPT	Displays prompt string, returns control to keyboard.
»	

To save this subprogram:

Keystrokes	Comments
ENTER	
' Equation STO	Stores the program.

When it comes to entering equations, it's hard to improve on the HP 48's way of doing things. Thus, in *Equation* we return control to the keyboard, to allow you to enter an equation in the fastest and easiest manner you know how. You can use the command line, or you can press ⬅ EQUATION and take advantage of the Equation Writer.

Equation places a text string in the stack, then executes the PROMPT command. (If you want your prompt text to fit on the screen, be sure to add an endline character after the word "as.") PROMPT displays the string beginning on the first row of the HP 48's liquid crystal display, and returns control to the keyboard.

You enter an equation. We've specified that the equation must be as a function of X, so if you have an equation in T or s, you'll need to change it to X when you enter that equation. (You could easily specify "f(x)" instead of "f(X)," but "X" is easier to enter from the keyboard than its lowercase cousin.) After you press ENTER to put the equation on the stack, you press ⬅ CONT to continue. The *Equation* subprogram finishes running, and the next subprogram is called by *Integrator*.

Accuracy and *Equation* illustrate an interesting difference between INPUT and PROMPT. With INPUT, your prompt string can occupy three rows of the display. For PROMPT, though, you're limited to two lines.

Limits—Adding to the Stack with INPUT

Limits prompts for the lower and upper limits of integration, and places these on the stack after the equation. It takes nothing from the stack.

Arguments	Results
3:	3: *equation as function of X*
2:	2: lower limit of integration
1: *equation as function of X*	1: upper limit of integration

Program Instructions	Comments
«	
"Enter the lower limit of integration"	Prompt string for use by INPUT.
""	Blank command-line string for INPUT.
INPUT	Prompts and waits for input of lower limit.
OBJ→	Converts input to real number.
"Enter the upper limit of integration"	Prompt string for use by INPUT.
""	Blank command-line string for INPUT.
INPUT	Prompts and waits for input of upper limit.
OBJ→	Converts input to real number.
»	

To save this subprogram:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Limits STO	

Limits illustrates the simple, straightforward use of a pair of INPUT statements. The first INPUT prompts for the lower limit, while the second INPUT asks you to input the upper limit.

As usual, each INPUT requires two strings as its arguments. The first string ("Enter the...integration") is the prompt string, and appears at the top of the central display area. The second string ("") is a null string that is placed on the command line by INPUT. Execution is halted by INPUT until you enter a limit, then press ENTER, whereupon execution resumes.

Doit—Performing Numerical Integration in a Program

Doit is the subprogram that actually performs integration. It takes three quantities from the stack: the equation, the lower integration limit, and the upper limit. *Doit* adds *X*, the variable of integration, to the stack. Then it takes all the necessary quantities and calculates the integral, leaving the answer on the stack.

Arguments	Results
3: equation as function of X	3:
2: lower limit of integration	2:
1: upper limit of integration	1: answer

Program Instructions	Comments
«	Begins program.
→ e l u	Creates local variables.
«	Begins defining procedure (a program) for local variables.
l u e 'X'	Places lower limit, upper limit, equation, and 'X' (the variable of integration) on the stack.
∫	Performs the integration.
»	Ends defining procedure.
»	Ends program.

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Doit STO	Stores the program.

Doit uses local variables to perform integration. It begins by taking three quantities from the stack and turning them into the local variables *e* (the equation), *l* (the lower limit of integration), and *u* (the upper limit).

As soon as the local variables have been declared, they must be defined. *Doit* uses a short program to define them. The program prepares the stack for numerical integration:

Stack Level	**Contents**
Level 4:	lower limit, *l*
Level 3:	upper limit, *u*
Level 2:	integrand (the equation, *e*)
Level 1:	variable of integration, `'X'`

Notice that for the sake of simplicity, *Doit* plugs in `'X'` as the variable of integration. You could easily change this to an INPUT command that asks the user to supply the variable of integration.

With the necessary quantities on the stack, *Doit* executes the $\int$ command. This performs the integration, and produces the answer: the area under the curve from *l* to *u*. All that's left now is to label that answer.

Labelit—Adding a Tag

Labelit simply adds a tag to the answer that's on the stack. It takes a quantity from the stack, tags it with the word "Answer," and places it back on the stack, ready for use.

Arguments	**Results**
1: answer	1: tagged answer

Program Instructions	Comments
«	
"Area"	Puts label on the stack.
→TAG	Labels the answer.
»	

To save the program as *Labelit*:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Labelit STO	Stores the program.

Labelit shows the simple use of the →TAG command to label the answer. When *Labelit* is executed, the numerical result of integration (from *Doit*) is already on the stack. *Labelit* adds a string label. Here the label is "Area", because the answer represents the area under a curve—but of course, any label that helps you identify the answer will do. When →TAG is executed, it adds the label before the numerical result, and automatically includes a colon to separate the two items.

Tagging a number doesn't affect its value at all. You can still move it, store it, or use it in other calculations. In fact, the tag stays right with the number as long as that number is intact. (That is, until you combine it with other numbers in addition, subtraction, or some other operation.)

Running the Integrator Program

To run the Integrator program, just hit the INTEG key and respond to the prompts. For instance, try it for the integral you calculated earlier from the keyboard, but show the answer to four decimal places:

$$\int_{0.5}^{4} \left(x^3 - 6x^2 + 9x + 1\right) dx$$

Program Prompt or Display	Your Action
	INTEG
Enter accuracy as no. of decimal places (eg, 3 for 0.001%)	4 ENTER

At this point, you enter the equation, making sure it's expressed as a function of X. You can enter the equation on the command line, or switch to the Equation Writer application as in the earlier keyboard example. Here's how to enter the equation using the command line:

Program Prompt or Display	Your Action
Enter EQUATION as f(X), then press CONT.	' X y^x 3 − 6 × X y^x 2 + 9 × X + 1 ENTER ↰ CONT
Enter the lower limit of integration	.5 ENTER
Enter the upper limit of integration	4 ENTER
1: Area: 10.6094	

6

Sound and Music

Whether you appreciate them for their own aesthetic value or use them in other applications, sound and music play an important role in programming. Now, the HP 48 is no Stradivarius, but it's capable of some pretty amazing sounds and audio effects. As you'll see in this chapter, you can make your calculator sing like a bird, or you can play it like a piano.

The key to producing sound on the HP 48 is its BEEP command. BEEP takes two arguments: the frequency in Hertz from level 2 of the stack, and the duration in seconds from level 1. For instance, to sound a middle C (523.25 Hertz) for 1 second, you could execute the following:

Keystrokes	**Comments**
523.25 ENTER	Frequency placed on the stack.
1	Duration placed on the stack.
PRG CTRL	
NXT NXT BEEP	Sounds the beep.

In order for BEEP to generate sound, flag −56 must be cleared. (Clearing this flag also enables beeps for errors.) As it happens, the BEEP command has some practical limits, too. For instance, if you specify a frequency higher than about 4400 Hertz, you'll get some kind of a sound, but probably not the frequency you expected.

Well, who wants to sit around plugging in frequencies and durations all day? Better to let a program do it for you—as in the following examples.

The SFX Programs

These SFX (*sound effects*) programs illustrate how the HP 48 can produce some pretty amazing tone patterns. These are just a start, of course. You'll want to experiment with your own sounds.

The SFX Directory

First, make a directory for the sound effects programs, then get into that directory:

(') SFX

(↰) (MEMORY) CRDIR

(VAR)

SFX

The SFX directory will hold all the sound effects programs. Any objects (including programs) that you enter and save will now be placed in this directory.

Alarm—Repeating Until Any Key

The first, and simplest, of the sound effects is a repeating alarm. It could be the dive alarm on a submarine game, the signal that a long program has completed execution, or the first sound you hear when the HP 48 gently wakes you from blissful slumber. The alarm itself is a long tone, followed by a shorter pause, then another long tone, and so on. Here, it keeps on sounding until you press any key on the calculator, but you could easily give it a different sort of exit.

Program Instructions	**Comments**
`«`	
`DO`	Begins DO loop.
`   200 1 BEEP`	Sounds a beep at 200 Hertz for one second.
`   .5 WAIT`	Pauses for one-half second
`UNTIL KEY`	Repeats until a key is pressed.
`END`	
`DROP`	
`»`	

If you've entered the program, you can store it as *Alarm*:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Alarm STO	Stores program as *Alarm*.

The program is nothing more than a DO ... UNTIL loop that is executed over and over again. The loop clause, the "heart" of the program, consists of a BEEP and a WAIT instruction. Each time through the loop, the number 200 is placed in level 1. Then 1 is placed in level 1, bumping the 200 up to level 2. With these quantities on the stack, the BEEP command uses them to sound a 200-Hertz tone for one second.

Next the quantity 0.5 is placed in level 1 of the stack, and WAIT uses it to "wait" for one-half second before continuing. Up to this point, all the quantities placed on the stack have been consumed.

Everything between DO and UNTIL is executed over and over again. Every time that UNTIL is encountered, the test clause (that is, between UNTIL and END) is performed. If the result of the test clause is 0 (false), the loop is executed again. If it's a 1 (true), then execution falls out of the loop. Remember, the test clause is executed just like any other statement. Whatever the test clause puts in level 1 is then evaluated (and consumed) by UNTIL.

For the test clause, we use the KEY statement. Now KEY, remember, is executed just like any other statement; however, if you haven't pressed a key, it returns 0 to stack level 1.

Stack Level	Contents
Level 1:	0

UNTIL evaluates this as "false," and bounces execution back up to the DO, where the loop clause is executed again.

When you press a key—any key—that press is remembered by KEY. Now, when execution comes to the test clause again, KEY is executed just as before. This time, though, you've pressed a key, so the results are different. KEY places the key address in stack level 2, and a 1 in level 1. Let's say you pressed the ENTER key. KEY returns its address (51) in level 2, and 1 in level 1.

Stack Level	Contents
Level 2:	51
Level 1:	1

The UNTIL test gobbles up the 1 from level 1 and evaluates it as "true." The key address drops to level 1, and execution falls out of the loop and continues. Since we have the key address in level 1 when we exit, we add a DROP statement to the program to clean up the stack.

An interesting feature of KEY is that it won't register a key press while the BEEP tone is being executed. To exit by pressing a key, you'll have to hit the key before or after the tone. (Naturally, you can press ATTN to exit any time.)

Klaxon—Waiting for a Specific Key

Ah, the sounds of Paris! The sharp clipping of heels along the Champs-Elysees, the rush of the Metro—and, inevitably, the distinctive two-tone Klaxons of police and emergency vehicles. To replicate the sound of the Klaxon, you can use the BEEP statement with two different frequencies. The following program sounds a

Klaxon, repeating the same series of two tones over and over, until you press user key F.

Program Instructions	Comments
`«`	
`DO`	Start of loop clause.
`   500 .5 BEEP`	Sounds tone at 500 Hertz for a half-second.
`   350 .5 BEEP`	Sounds tone at 350 Hertz for a half-second.
`UNTIL`	End of loop clause, start of test clause.
`   KEY`	Test clause
`END`	
`IF 16 ≠`	Tests to see if key address is not 16.
`   THEN Klaxon`	If key 16 was not pressed, starts the loop again.
`END`	Otherwise, ends the program.
`»`	

For this program to run correctly, you'll need to store it as *Klaxon* (or change the name of the program called by THEN).

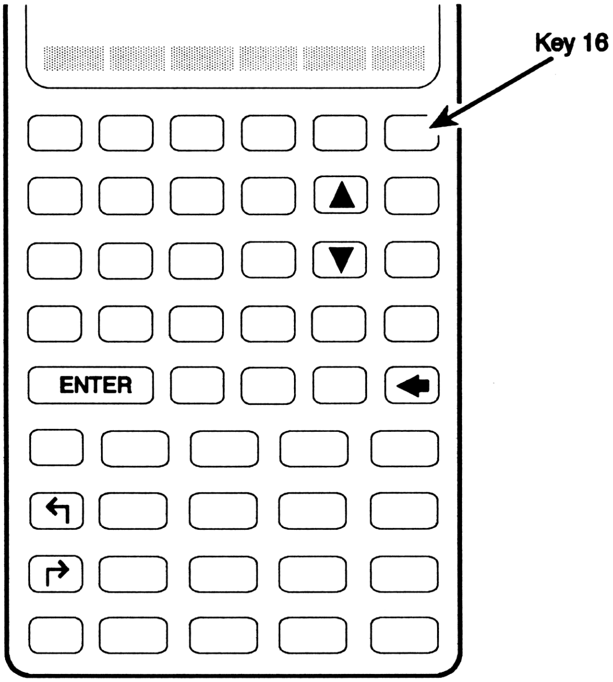
Keystrokes	Comments
ENTER	Puts program on the stack.
' Klaxon STO	Stores program as *Klaxon*.

The main part of the program consists of a DO ... UNTIL loop. Each time through the loop, the quantity 500 is placed in stack level 2 and 0.5 in level 1; and BEEP uses these to sound a 500-Hertz tone for one-half second. Then 350 is placed in level 2 and 0.5 in level 1, and BEEP uses them to generate a lower tone (350 Hertz) for the same length of time.

As in the *Alarm* program, the test clause after UNTIL is a KEY statement. As long as you don't press a key, the KEY statement returns 0 to level 1, UNTIL consumes it and evaluates it as false, and the loop is executed again. When you *do* press a key, its keycode is returned to level 2, and a 1 is returned to level 1. UNTIL evaluates this as "true," and execution falls out of the DO ... UNTIL loop and continues with the next statement after END.

So far, the *Klaxon* is very similar to the *Alarm*. In this program, though, we're going to keep that *Klaxon* sounding until the user presses not just any key, but a *specific* key.

Remember, when execution falls out of the loop, the address of the key pressed is still in level 1 of the stack. We can test this address with an IF ... THEN structure to find out if it's the key we want. In this case, we want the exit key to be user key F—that is, at row 1, key 6 (address 16).



Let's say the user pressed the ENTER key to exit. That means 51 is in level 1 when the DO ... UNTIL ... END loop is exited. The IF statement puts 16 in level 1, bumping 51 up to level 2.

Stack Level	Contents
Level 2:	51
Level 1:	16

Then IF removes both these quantities from the stack, asking the question "Is level 1 unequal to level 2?" Since the answer is yes (true), the true clause after THEN is executed. This clause is `Klaxon` (the name of the program), so the program simply calls itself again. Thus, *Klaxon* calls itself recursively if the user tries to exit by pressing any key on the keyboard except key F (or ATTN).

If the user finally gets smart and hits user key F, KEY returns 16 to the stack, and that's what's left when the DO ... UNTIL ... END loop is exited. When the test `IF 16 ≠` is performed, there's a 16 in level 2 and a 16 in level 1, so the test for *inequality* is false. The THEN clause isn't executed, and the program (finally) halts.

Bird—Eliminating Recursion with Nested Loops

The HP 48 lends itself surprisingly well to a creating multitudes of ornithological sounds. While our fine feathered friends might not be fooled, cheeps and peeps from your calculator can add spice and verve to games and other keyboard activities.

The program below illustrates one way of summoning birdlike sounds from the bowels of the HP 48. It simulates the mating cry of the long-necked doofus, a mythical creature in the turkey family.

```
Program Instructions              Comments

«
DO                                Begins outer DO ... UNTIL loop.
  DO                              Begins inner DO ... UNTIL loop.
    2500 3000                     Start and finish for FOR.
    FOR i                         FOR loop counter goes from 2500 to
                                  3000.
      i .01 BEEP                  Generates .01-second tone at 2500
                                  Hz., 2550 Hz., and so on.
    50 STEP                       Step interval for FOR loop.
    .3 WAIT                       Pauses for 0.3 second.
    2500 3000
    FOR i
      i .01 BEEP                  Another loop like the one above.
    50 STEP
    .3 WAIT
    3000 2500                     Start and finish for FOR.
    FOR i                         This loop goes from 3000 to 2500.
    i .01 BEEP
    -50 STEP                      Negative step.
    .3 WAIT
  UNTIL KEY                       End of inner DO ... UNTIL loop
                                  clause; loop executed until any key is
                                  pressed.
  END                             End of inner DO ... UNTIL
                                  structure.
UNTIL 16 ==                       Outer loop continues until key
                                  pressed is key 16.
END                               End of IF structure.
»
```

Save the program as shown here.

Keystrokes	Comments
ENTER	Puts program on the stack.
' Bird STO	Stores program as *Bird*.

Recursion (the calling of a program by itself, as in the earlier *Klaxon* program) consumes memory. To avoid recursion, *Bird* uses a pair of nested DO ... UNTIL loops. The inner loop waits until any key is pressed, while the outer loop waits until the key is the F key (at address 16).

The inner DO ... UNTIL ... END loop is much like the loops in the other SFX programs. It is repeated over and over until the user presses a key. Within this loop, however, we add three FOR ... STEP loops to create the bird's characteristic call.

FOR takes two numbers from the stack: a *start* and a *finish*. So we begin by placing the numbers 2500 and 3000 in levels 2 and 1, respectively. FOR removes these from the stack and uses them to determine where it is to begin and end. In this case, the loop will go from 2500 to 3000. Since this is a FOR ... STEP loop, we can specify the increment, too. The `50 STEP` instruction specifies an increment of 50, so the loop counter begins at 2500, goes to 2550, then to 2600, and so on.

An interesting feature of FOR (and one we'll put to use here) is that it creates a local variable. The name you use immediately after the word FOR is a local variable that exists between FOR and STEP (or FOR and NEXT) only. We've called it *i*, to keep with the convention found in many programming texts, but you can give it any name you like. In essence, all you're doing with the words `FOR i` is creating a local variable called *i* that contains the value of the loop counter. It's not part of the loop clause, and isn't put on the stack yet.

The first time through the loop, *i* has a value of 2500. The line `i .01 BEEP` puts *i* and .01 on the stack, then uses them to generate a 2500-Hertz tone that lasts .01 second. The next time through the loop *i* is 2550, so the tone is 2550 Hertz. This continues until the loop reaches 3000. Notice that the duration of each tone is very short.

Taken together, these tones create a "chirp" going from 2500 Hertz to 3000 Hertz.

When the FOR ... STEP loop has been executed 11 times, execution continues to the WAIT, which pauses for 0.3 second—just enough time for the long-necked doofus to catch its breath! Then we follow up with another FOR ... STEP loop that's exactly the same as the first, creating a second identical chirp.

The third component of this bird's call is a chirp that starts high and falls, instead of the other way around. For this, we use yet another FOR ... STEP loop; it goes from 3000 Hertz to 2500 Hertz. The STEP increment is negative, so the first time through the loop, *i* is 3000, then next time 2950, the next time 2900, and so on.

Each time through the inner DO ... UNTIL loop, the bird chirps twice from low to high, then once from high to low. Rising-rising-falling, rising-rising-falling, the melodic song of the long-necked doofus continues until you press a key. As in other programs, we've used KEY for the UNTIL test clause, so upon exit from the inner DO ... UNTIL ... END loop, the value of the key address is present in level 1.

Now we make use of the outer DO ... UNTIL ... END loop to check that key address. The outer loop is executed UNTIL that key address is equal to 16. If the key address is *not* equal to 16, execution begins again at the first DO instruction, and you hear the sequence of chirps once more. When you finally press the user key at location 16—that is, the first row, sixth key—execution falls out of the outer loop and the chirping ends.

Using KEY to end execution is pretty effective here—a lot more so than in the *Alarm* or *Klaxon* programs, where you have a long continuous tone. That's because the HP 48 doesn't read its key buffer while BEEP is being executed. The chirps in this program, though, are made up of many very short BEEPs, so there are many more opportunities for the HP 48 to recognize your key press.

Car—Using Local Variables to Control Loops

A powerful sports car pulls away from a stop light. The car accelerates, the engine revving up almost to the tachometer's red line. The driver shifts gears, the engine slows down, then revs again, higher this time. Another shift, and the car's engine revs higher still.

Close your eyes when you run the following program and you just may find yourself at the wheel of that sports car. The program uses three FOR ... STEP loops to simulate three gears. It also features a set of local variables that make it easy for you to change and experiment with different values.

Program Instructions	Comments
`«`	
`100 400 10 .03`	Four quantities placed on stack.
`→ b e s t`	Quantities converted to local variables.
`  «`	Begins defining procedure (here, a program).
`  b e`	Variables for begin and end of loop placed on stack.
`  FOR i`	Begins FOR loop.
`    i t BEEP`	Tone sounds at frequency of loop counter (*i*) for duration of *t*.
`  s STEP`	STEP interval is *s*.
`  .75 WAIT`	Pauses for 0.75 second.
`  b 100 +`	Second loop begins 100 Hertz higher than *b*.
`  e 100 +`	Loop ends 100 Hertz higher than *e*.

```
Program Instructions              Comments
  FOR i
     i t BEEP
  s STEP
  .75 WAIT
  b 200 +                         Third loop begins 200 Hertz higher
                                  than b.
  e 200 +                         It ends 200 Hertz higher than e.
  FOR i
    i t BEEP
  s STEP
  »                               End of local variables.
»                                 End of program.
```

To store the program as *Car*:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Car STO	Stores *Car* program.

We begin by placing four values on the stack, then converting them to local variables. We start with *b* (begin) of 100, *e* (end) of 400, *s* (step) of 10, and *t* (time) of .03. Because all the variables are right at the beginning of the program, you can easily edit the program to change them if you want to experiment. Or you could eliminate the `100 400 10 .03` entirely, and simply make sure that these four quantities are on the stack when the program is run.

Declaring the local variables `b e s t` removes the four numbers from the stack. In order for these local variables to work properly in the HP 48, they must be immediately followed by the defining procedure. Here, the defining procedure is a program.

Within the inner program, the variables *b* and *e* are placed on the stack. This means that level 2 contains the value for *b* (that is, 100), while level 1 has the value for *e* (400).

Stack Level	Contents
Level 2:	100
Level 1:	400

Now the first FOR ... STEP loop is run. It runs from 100 to 400, using a STEP increment of *s* (that is, 10). The variable *i* contains the latest incremental value; so the first time through the loop, *i* is 100, the second time it's 110, the third time 120, and so on.

Each time through the loop, BEEP sounds a tone at a frequency of *i* Hertz for a duration of *t* seconds. So you hear a 100-Hertz tone for 0.03 second, then a 110-Hertz tone for the same time, then a 120-Hertz tone, and all the rest of the tones up to 400 Hertz. The rising tones come in rapid succession—not unlike the sound of an accelerating motor.

When the tones for "first gear" have been generated, the program WAITs for three-quarters of a second to simulate the shifting of gears. Then the program places on the stack the beginning and ending values for another FOR ... STEP loop. This time we add 100 to the value of *b* and *e*, so that the loop goes from 200 to 500. The higher values mean that the range of tones for "second gear" is somewhat higher than for first gear. The loop uses the same STEP interval, *s*.

For "third gear," we execute a third FOR ... STEP loop, with the range of tones slightly higher still. The program shown here doesn't repeat itself as the other sound effects do—after all, an automobile accelerates only once, then it's gone—but you can easily add a DO ... UNTIL loop if you want to hear the roar of the engine over and over. Also, you can play around with the values for *b* and *e*, until you have that motor purring!

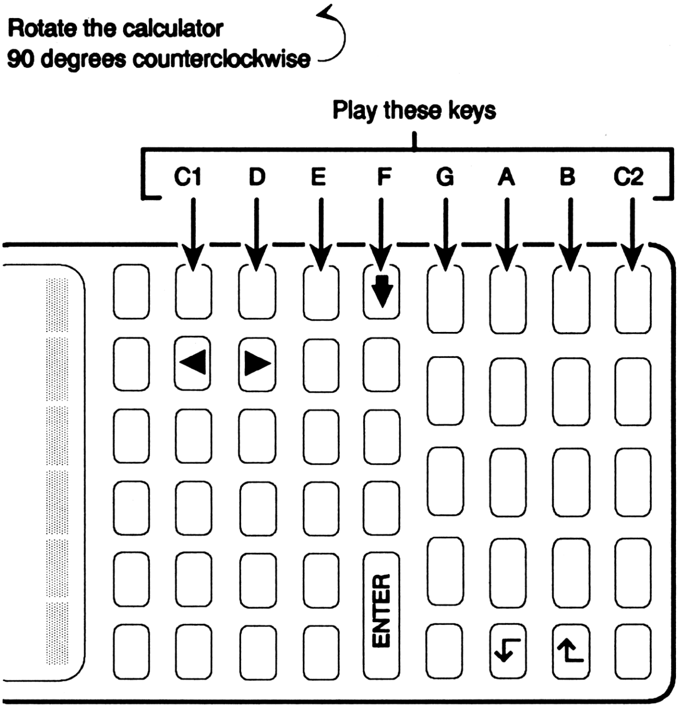
The HP 48 as a Piano

As you know, all it takes is the right values in stack level 1 and level 2, and the HP 48 can play any note you'll find on many musical instruments. Moreover, you can redefine the HP 48 keyboard so that each key plays a specific note. Does that sound like a piano or organ? Read on!

Music is highly mathematical; the same note in the next-higher octave, for instance, is approximately double the frequency of that note in the current octave. The following table shows the relationship of frequencies and notes.

Musical Note	Frequency in Hertz	Musical Note	Frequency in Hertz
C	130.81	Middle C	523.25
D	146.83	D	587.33
E	164.81	E	659.26
F	174.61	F	698.46
G	196.00	G	783.99
A	220.00	A	880.00
B	246.94	B	987.77
C	261.63	C	1046.50
D	293.66	D	1174.70
E	329.63	E	1318.50
F	349.23	F	1396.90
G	392.00	G	1568.00
A	440.00	A	1760.00
B	493.88	B	1975.50

The programs in this section convert your HP 48 into a one-octave piano. The piano's "keys" are all the calculator keys along the right side of the HP 48 keyboard (except for the top-row user key). That is, they're the keys addressed by location codes of 26, 36, 46, 55, 65, 75, 85, and 95. To play the piano, you rotate the calculator so the display is on your left, then play the eight black keys of what is now the top row. Except for those eight keys and an exit key, all other keys on the keyboard are "locked out" when you're playing the piano.



The Piano Directory

Begin by creating a directory to hold the *Piano* program and subprograms. Then get into that directory:

' PIANO
↰ MEMORY CRDIR
VAR
PIANO

The PIANO directory will hold all programs and subprograms for the HP 48 piano. All the programs and subprograms you write now should be saved here.

The Main Piano Program

Following the tenets of structured programming, the main program consists almost entirely of calls to other objects—including subprograms and a list of keys. When *Piano* has been run, it leaves a list on the stack containing the notes you've played—a timeless record of your composition.

Arguments	Results
1:	1: list of musical notes

Program Instructions	Comments
`«`	
`Message`	Calls *Message* for general user instructions.
`Init`	Calls *Init* for initialization.
`Menulist TMENU`	Uses the list *Menulist* to create a temporary menu.
`Dokeys`	Locks out all keys except the ones for *Piano*.
`»`	

After a general message to the user and an initialization routine, this main program creates a temporary menu using the TMENU command and an external list in *Menulist*. Then it calls the program *Dokeys*, which prevents you from using any keys except those assigned for the piano.

Once you've entered this main program, you can store it as *Piano*:

Keystrokes	**Comments**
ENTER	Puts the program on the stack.
' Piano STO	Stores program as *Piano*.

Now let's take a closer look at those individual subprograms and how they work.

Message—Displaying User Instructions

The *Message* subprogram is the first object called by the main *Piano* program. It displays a general message to the user, then waits for a press of any key before continuing. *Message* has no overall effect on the stack.

Program Instructions	**Comments**
«	
"PIANO Turn the HP 48 a quarter-turn counter- clockwise, then play the 8 top keys. Any key to continue."	Message that is displayed by DISP.
CLLCD	Clears the LCD display.
1 DISP	Displays message beginning in the first line of the display.
7 FREEZE	Freezes the entire display.
0 WAIT	Waits for press of any key.
DROP	Gets rid of the key address returned by WAIT.
»	

To save the subprogram:

Keystrokes	Comments
ENTER	
' Message STO	Stores program as *Message*.

The subprogram begins by creating a message to the user; this message is placed in level 1 of the stack. Next, the CLLCD instruction clears the entire calculator display. (It doesn't affect the stack, though.)

A 1 is placed on the stack, so that the message is in level 2 and the 1 is in level 1. Then DISP is executed. DISP means "display the message that's in level 2, beginning at the display line specified in level 1." Thus, the message to the user is put in the display, beginning with line 1.

The next program line, `7 FREEZE`, freezes the entire display, showing the user message. Now, if we let the program simply continue, you'd see the message for only a split-second; FREEZE doesn't halt the program. So we place 0 in level 1 and execute a WAIT instruction. With 0 as the argument, WAIT suspends execution until the next key press.

When you press a key, any key, the subprogram continues execution. WAIT leaves the key's address in level 1 of the stack, so we do a DROP to clean things up before exiting.

Init—Setting User Mode

The next subprogram called is *Init*, which performs initialization. This subprogram doesn't affect the stack.

Program Instructions	Comments
«	
-62 SF	Sets flag for User mode.
-56 CF	Enables audible BEEPs.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Init STO	Stores the program.

The HP 48 lets you create a user keyboard, assigning any function to any key on the keyboard. This user keyboard is active whenever the calculator is in User mode. To place the calculator into User mode from a program, you can set system flag −62, as in this *Init* routine. (We'll clear the flag when we exit the program, to return the calculator to its normal operation.)

We're going to use BEEP to create the musical "notes," so the next thing we need to do is make sure the BEEPs are audible. Thus, we clear system flag −56. When this flag is cleared, it enables both the BEEP command *and* error beeps.

Menulist—Specifying a Menu Key and Action

Menulist is the argument for the TMENU command. TMENU, remember, requires for its argument a "list of lists"; each of the internal lists contains a key name and its action. In the case of the *Piano*, our temporary menu contains just one key, the choice for exiting the program. Hence, *Menulist* is very simple:

Program Instructions	Comments
`{`	Starts list for use by TMENU.
`{ "EXIT" Exit }`	Only one user key is specified.
`}`	End of list.

To save the list, follow the same procedure as for a program:

Keystrokes	Comments
ENTER	Places the list on the stack.
' Menulist STO	Stores it as *Menulist*.

When used as the argument for TMENU, *Menulist* creates a temporary menu in the display. Only one key is shown: user key A, which has the label EXIT. All other keys in the display are blank.

When you're finished playing the piano, there's only one way you can exit, and that's by pressing the EXIT key. When you press EXIT, it calls the subprogram *Exit*, which gets the program out of User mode and restores the normal keyboard and menu. More about *Exit* in a moment.

Dokeys—Redefining the Entire Keyboard

Dokeys is what makes the piano possible. It's where most of the real work takes place. *Dokeys* assigns functions to those HP 48 keys you want active, and turns off *all* other keys—including even ATTN! It also prepares the HP 48 to record your notes for future reference or shipment to Nashville.

Program Instructions	Comments
«	
S DELKEYS	Disables all unassigned keys.
Keylist STOKEYS	Assigns functions only to keys in *Keylist*.
'SKEY'	
11.1	
ASN	Assigns user key A, needed for exit.
""	Places null string on the stack.
»	

To save this subprogram:

Keystrokes	**Comments**
ENTER	
' Dokeys STO	Stores *Dokeys*.

Dokeys begins by placing S in level 1, then executing DELKEYS. This S DELKEYS statement has the effect of "killing" all standard key definitions on the keyboard. If you could stop the program after this step, not a key would be active, and all you'd get would be error beeps.

Here's a conundrum, though: The use of DELKEYS means that if you hit a non-piano key while you're playing, it's not executed—but it does produce an error beep. Because flag −56 controls both BEEP and error beeps, you get them both, or you get neither. (To eliminate the error beep that comes from hitting the wrong key, you can eliminate the S DELKEYS instruction from your program. In this case, however, all those extraneous keys are active—they aren't locked out.)

Now that we've turned off all standard keys , we can set up the keyboard the way *we* want. That's a job for STOKEYS. This command takes as its argument a list of key definitions and locations, one after another, like this:

{ *definition1 location1 definition2 location2 ...* }

Our list of definitions and locations is called *Keylist*. You'll see the details of *Keylist* in a moment; for now, it's enough to understand that this list assigns the correct "note" to each of the keys we'll use for the piano. STOKEYS, with a list of definitions and locations, is perfect for multiple key assignments like the ones we need here.

Before we examine *Keylist*, see how we use ASN to provide an exit from the user keyboard. ASN, remember, lets you reassign a *single* standard key. As the arguments for ASN, you need 'SKEY' in level 2 of the stack, and the three-digit key address in level 1. We want to make sure user key A is still active, so we place its address, 11.1 (or 11.0, or simply 11) in level 1.

Finally, *Dokeys* places "" in level 1 of the stack. This is a so-called "null string" (that is, a string with nothing in it). When you press the first "piano" key, not only do you hear the note, but that key is also written into the display by adding it to the null string. Other keys you press are also added. Thus, when you're done, level 1 of the stack contains a complete record of every note you played.

Keylist—Defining Specific Keys

To understand how the piano actually makes music, you have to understand *Keylist*. This list contains all the information needed to turn HP 48 buttons into piano keys. *Keylist* consists of a definition followed by a key location, another definition followed by another key location, and so on. STOKEYS, in the *Dokeys* subprogram, uses *Keylist* as its argument, and actually does the assignment.

Here's one of the definitions and its location:

```
« 261.63 .2 BEEP "C1" + » 26
```

You can see that the *definition*, everything between « and », is a program. The *location* is 26, which is the second row, sixth key. (It's actually the NXT key.) After this definition has been assigned and the user keyboard is active, when you press NXT, the following mini-program is executed:

Program Instructions	**Comments**
`«`	
`261.63 .2 BEEP`	Sounds 261.63-Hertz tone for 0.2 second.
`"C1" +`	Adds the substring C1 to the string in level 1.
`»`	

The definition program places 261.63 in stack level 2 and .2 in level 1. Then it uses these as arguments for BEEP, causing the calculator to emit a 261.63-Hertz tone (that's middle C on a real piano) for 0.2 second.

After sounding its note, the program places the letters "C1" in level 1. If this is the first key pressed, the null string, "", is in level 2 now. (If this isn't the first key pressed, the string in level 2 contains a list of the notes sounded.) The + operator adds the letters "C1" to the null string (or list of notes). Thus, there's always a "running total" of notes visible in level 1 of the stack.

The complete *Keylist*, with all the definitions and locations, is shown below.

Program Instructions	**Comments**
{	Begins the list.
« 261.63 .2 BEEP	Sounds tone for middle C.
"C1" + »	Adds note to string in level 1.
26	Assigns middle C to NXT key.
« 293.66 .2 BEEP	
"D" + »	
36	Assigns D to row 3, key 6.
« 329.63 .2 BEEP	
"E" + »	
46	Assigns E to row 4, key 6.
« 349.23 .2 BEEP	
"F" + »	
55	Assigns F to row 5, key 5.
« 392 .2 BEEP	
"G" + »	
65	Assigns G to row 6, key 5.
« 440 .2 BEEP	
"A" + »	
75	Assigns A to row 7, key 5.
« 493.88 .2 BEEP	
"B" + »	
85	Assigns B to row 8, key 5.

Program Instructions	**Comments**
`« 523.25 .2 BEEP`	
`"C2" + »`	
`95`	Assigns C above middle C to row 9, key 5.
`}`	End of list.

To save this list as *Keylist*:

Keystrokes	**Comments**
ENTER	Puts the list in level 1.
' Keylist STO	Stores it.

When *Piano* calls *Dokeys*, the STOKEYS command uses this list to assign the specified programs to the appropriate keys. Then you simply rotate the calculator 90 degrees counterclockwise, and you've got a piano!

For one octave, we need to go from middle C to the next-higher C. To differentiate between these two, we call one of these notes "C1" and the next "C2." You might also want to make other octaves available.

For the sake of simplicity, we've used "hard-coded" values of 0.2 for the duration of the beeps. This produces sounds that are rather slow (approximately an *andante* tempo). You can experiment to find other values, or you might even create another key that allows different tempos (that is, different values for the duration of the BEEPs) to be passed to *Keylist*, then used as a local variable.

When you run *Piano*, all keys except those specified by STOKEYS and ASN are "locked out." They have no effect. What this means is that the only way you can exit is by pressing EXIT, which calls the *Exit* subprogram. Upon exit, the list of notes you played is left in level 1 of the stack, for you to examine or save.

Exit—Restoring the Calculator Keyboard

Exit is the subprogram that's executed when you press the EXIT menu key. This subprogram isn't very long, but it's extremely important. Without it, your calculator will continue to function only as a piano—never again as a calculator. Not even ATTN will help!

Program Instructions	Comments
«	
-62 CF	Clears system flag −62 to kill User mode.
0 DELKEYS	Clears user-mode key assignments.
0 MENU	Gets back to normal VAR menu.
»	

Keystrokes	Comments
ENTER	
' Exit STO	Stores this program as *Exit*.

The line -62 CF clears system flag −62, switching the calculator from User mode back to normal operation. The next instruction, 0 DELKEYS, clears all user keys and reassigns all those useless keys deactivated by S DELKEYS.

The 0 MENU instruction switches back from the temporary menu to the ordinary menu of variables again. This step isn't really necessary, but it gives you some "visual feedback" when you exit, and presents you with PIANO key, ready for another run.

Playing the Piano

To run the *Piano* program, press PIANO.

Program Prompt or Display	Your Action
	PIANO
PIANO Turn the HP 48 a quarter-turn counter- clockwise, then play the 8 top keys. Any key to continue.	

Now you just follow the instructions. Rotate the HP 48 a quarter-turn counterclockwise, and start creating some beautiful music. As you play the notes, they appear in the display:

```
"C1EGGGGGC2G"
```

When you're done with your composition, press EXIT to convert your HP 48 back to a calculator again.

7

General Graphics and Animation

Even without programming, the HP 48 calculator is capable of creating pretty amazing charts and graphics on its tiny screen. You can plug in some data, press a key, and presto, you've got a bar chart, a plot of a complicated function, even a polar plot. In fact, the calculator's high-level plotting functions will take care of most *mathematical* plotting.

If you care to do a little graphics programming yourself, you can extend the pictorial powers of the HP 48 even further. You can create a full range of graphics effects, from a simple circle to a racing rocket ship.

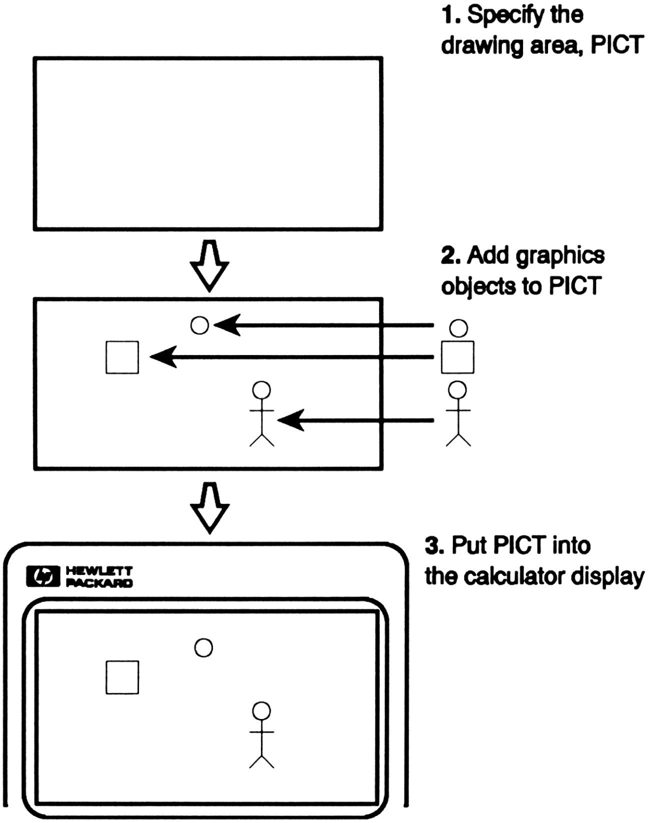
HP 48 Graphics Fundamentals

Plotting or programming graphics on the HP 48 isn't difficult, but it takes some practice. Moreover, you have to get used to dealing with the reserved PICT variable, graphics objects, and the stack and display.

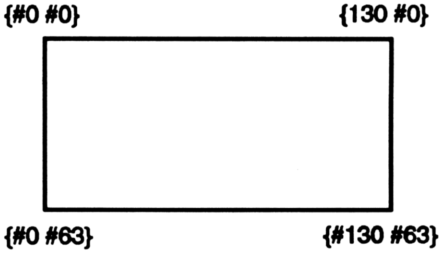
Let's start with PICT. As an artist, you use PICT as your "canvas." You can think of it as a drawing area that's separate from the rest of the calculator. Although you can specify the size of this area, and "draw" on it in a program, nothing is actually visible until you place PICT into the calculator display.

Three Steps to Seeing Your Pictures

Creating graphics and making them visible from a program is really a three-step process: You specify the size of PICT, add graphics objects, then display PICT.


1. Specify the drawing area, PICT
2. Add graphics objects to PICT
3. Put PICT into the calculator display
HEWLETT PACKARD

1. **Specify PICT**. You begin by specifying the size of PICT. From the keyboard, you can do this with the HP 48's RESET operation, which erases PICT and restores it to the default size of 131 by 64 pixels.



RESET, however, isn't programmable. So in a program, you can use PDIM, as shown in the following example:

```
«
# 130d
# 63d
PDIM
»
```

PDIM (*PICT dimension*) specifies the size of PICT. In this case, using the two binary numbers shown sets PICT to its default size—that is, 0 to 130 horizontal units, and 0 to 63 vertical units.

Another way to specify the size of PICT is with XRNG and YRNG.

2. **Put a graphics object in PICT**. You place a graphics object into PICT in any of several ways:

   - Draw the object using LINE, BOX, ARC, and similar commands. These commands actually "draw" right on PICT.

     or

   - Put both PICT and the graphics object on the stack, then combine them with GOR, GXOR, or REPL.

3. **Put PICT into the calculator display**. The →LCD or PVIEW command will allow you to actually see PICT in the display.

The Circs Directory

You're going to see how to draw some circles. First, create a directory for those circle programs:

['] CIRCS
[↰] [MEMORY] CRDIR
[VAR]
CIRCS

Circ1

The *Circ1* program shown below illustrates very simply the procedure for drawing on and displaying PICT.

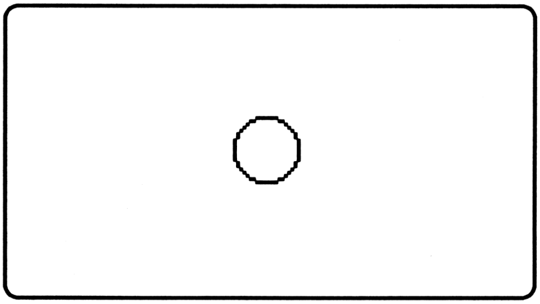
Program Instructions	Comments
«	
ERASE	Erases PICT.
# 130d # 63d PDIM	Sets default pixel dimensions for PICT.
{ # 65d # 32d }	Center of arc is center of PICT.
# 10d	Radius of arc is 10 pixels.
0	Starting angle for arc.
360	Ending angle for arc.
ARC	Draws 360-degree arc (a circle).
PICT	Puts PICT on the stack.
RCL	Recalls *contents* of PICT to stack.
→LCD	Displays graphics object (that is, PICT) from level 1.
7 FREEZE	Freezes display until next key press.
»	

To save this program as *Circ1*:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Circl STO	Stores the program as *Circ1*.

When you run *Circ1*, it produces a graphics display with a circle in the middle.

Press CIRC1 to begin the program.



Circ1 draws a circle on the screen

The program begins with ERASE, which clears any previous graphics from PICT. Then it places a pair of binary numbers, `# 130d` and `# 63d`, on the stack. PDIM uses these numbers to set the maximum dimensions for PICT; thus, PICT is a grid that's 0 to 130 pixels wide by 0 to 63 pixels tall.

Now *Circ1* uses the ARC command to draw a circle. ARC requires the following arguments on the stack.

Stack Level	Contents
Level 4:	center of arc
Level 3:	radius of arc
Level 2:	beginning angle of arc
Level 1:	ending angle of arc

With these quantities on the stack, ARC draws a counterclockwise arc in PICT. Since we've specified the beginning angle as 0 degrees and the ending angle as 360 degrees, we get a complete circle.

ARC isn't the only command you can use to "draw" in PICT; there are also LINE, TLINE, BOX, PIXON, and PIXOFF. These commands are terrific for creating charts and graphs in PICT. (As you'll see, there are easier ways to draw more complicated shapes.)

The next command in *Circ1* is `PICT`, which puts PICT on the stack. In fact, PICT is just a big graphics object; you can place it on the stack, store another graphics object in that name, or purge PICT. The important thing to remember is that when you draw a shape in PICT, that shape is part of PICT—it's not a *separate* graphics object. (Not yet, anyway.)

With the object PICT on the stack, the program *Circ1* executes RCL, which brings the contents of PICT into the stack. Then the →LCD command puts those contents into the calculator's display for viewing. So that the display doesn't flit by too quickly, it's frozen by `7 FREEZE`, which freezes the entire display until the next key press.

Designing Your Own Graphics Grid

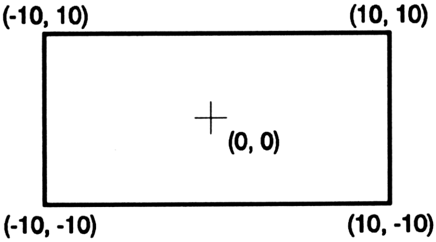
You can think of # `130d` and # `63d` as *absolute* pixel coordinates. They represent the actual number of pixels on the graphics display, and you can't really change them.

But you don't have to use those HP 48 pixels as graphics coordinates. Instead, you can set up PICT in your own units—that is, *user* units—to cover any range you want. You can change PICT into a grid that's 20 by 20, or that goes from 0 to 720 in the horizontal direction and −1 to +1 along the vertical axis. It all depends on how you set up the grid with PDIM or with XRNG and YRNG.

Because you can design PICT as a grid that suits your application, user units are a lot easier to manipulate than pixels. Moreover, user units are specified as complex numbers—which for graphics are nothing more complicated than the two-point (x,y) form so familiar to students of the Cartesian coordinate system.

Circ2

Let's go back to our circle again. Trying to figure out the exact center of a 130 × 63 grid isn't easy. It's much simpler to set up, say, a 20 × 20 grid with (0,0) in the center.



The *Circ2* program shown below is a modified version of *Circ1*. It accomplishes the same thing—drawing a circle in PICT and placing it in the center of the display—but *Circ2* shows how to replace those difficult-to-remember absolute pixel coordinates with user units.

Program Instructions	Comments
«	
ERASE	Erases PICT.
(-10,-10)	
(10,10)	
PDIM	Sets dimensions of PICT to be a 20 × 20 grid.
(0,0)	Center of arc is at point (0,0) on PICT.
1	Radius of arc is 1 unit.
0	Starting angle for arc.
360	Ending angle for arc.
ARC	Draws 360-degree arc (a circle).
PICT	Puts PICT on the stack.
RCL	Recalls *contents* of PICT to stack.
→LCD	Displays graphics object (that is, PICT) from level 1.
7 FREEZE	Freezes display until next key press.
»	

To save *Circ2*:

Keystrokes	Comments
ENTER	
' Circ2 STO	Stores the program.

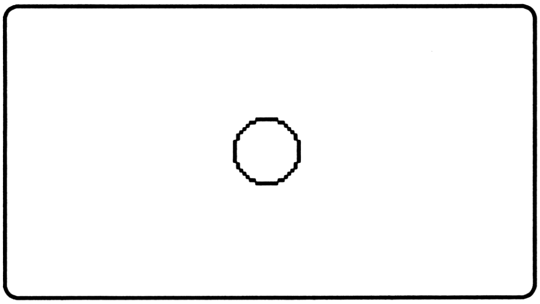
When you press CIRC2 to run this program, the coordinates (−10, −10) and (10, 10) are placed in levels 2 and 1 of the stack. Then PDIM is executed. With two complex numbers as arguments (instead of two binary numbers), PDIM turns the PICT drawing surface into the grid you've specified. The value in level 2, which is (−10, −10), is used by PDIM to specify the *lower left* coordinates of

PICT. The value in level 1, (10, 10), gives the *upper right* coordinates.

With two complex numbers as arguments as shown here, PDIM does something else, too: It creates the variable PPAR in the current directory. PPAR (which stands for *PLOT parameters*) is where the HP 48 keeps information about its graphics environment. You don't need to worry about PPAR, although if you prefer, you can make changes to this variable instead of using commands such as PDIM. Just remember that if PPAR suddenly appears in a directory, it's a good bet that user units are afoot.

With PICT specified as a 20 × 20 grid, we need a different center and radius for the ARC command that draws the circle. Since the PICT grid was specified in user units, we can utilize those same units as arguments for the ARC command.

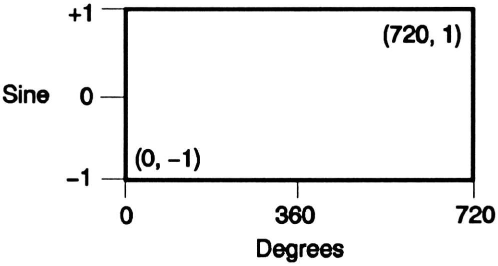
In *Circ2*, we specify the center of the arc with a complex number, (0, 0), instead of with a list of two binary numbers. This puts the center of the circle in the center of PICT—which is also the center of the calculator's display when PICT is shown. The radius of the arc is now 1 user unit. To draw a circle, ARC begins at 0 degrees and goes through 360 degrees, just as in *Circ1*.



Circ2 creates a circle out of user units

Plotting a Sine Curve

The sine function is always in the range between +1 and −1. To plot the sines of all angles from 0 through 720 degrees, you can use a graphics area that is laid out with 0 to 720 units in the horizontal (x) direction and −1 to +1 unit in the vertical (y) direction.



The *Sine1* program turns the HP 48's PICT area into this grid, and draws a sine curve on it.

The Sine1 Directory

To create a directory for the Sine1 program and subprograms, then get into that directory:

['] SINE1
[↰] [MEMORY] CRDIR
[VAR]
SINE1

The SINE1 directory will hold all programs and subprograms for this sine-drawing program.

The Main Sine1 Program

The main *Sine1* program consists of calls to three subprograms.

Program Instructions	Comments
«	
GraphArea	Calls subprogram to set PICT graphics grid.
AddSine	Calls subprogram to add sine curve to PICT.
ShowGraph	Calls subprogram to show PICT in display.
»	

To save the program:

Keystrokes	Comments
ENTER	
' Sine1 STO	Stores the program.

GraphArea—Setting the PICT Size

The first subprogram called by *Sine1* is *GraphArea*, which initializes the HP 48 and turns PICT into the grid that's needed:

Program Instructions	Comments
«	
DEG	Sets degrees mode.
ERASE	Clears any previous graphics from PICT.
0 720 XRNG	Specifies x (horizontal) range of 0 to 720 units.
-1 1 YRNG	Specifies y (vertical) range of −1 to 1 unit.
»	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' GraphArea STO	Stores the program.

In *GraphArea*, we haven't used PDIM to set the dimensions of the PICT graphics area. Instead, we've called on the XRNG and YRNG commands.

The subprogram begins by setting degrees mode with the DEG command. After ERASE clears any leftover graphics from PICT, the program places the value 0 in level 2 of the stack, and 720 in level 1. XRNG takes these two values off the stack and uses them as the "from" and "to" values, respectively. Executing XRNG with these values in the stack sets the range of x-values—that is, the horizontal range—of PICT to be 0 to 720.

Then the program places −1 and 1 on the stack, and executes the YRNG command. Like XRNG, the YRNG command takes two arguments off the stack and uses them to specify the "from" and "to" range. Thus, the y-range (the horizontal axis) of PICT is set to be from −1 to 1.

AddSine—Plotting a Curve on PICT

The next subprogram called by *Sine1* is *AddSine*, and it's here that the real work of calculating and drawing the sine curve is performed. *AddSine* plots each calculated sine as a single point on PICT, creating what appears as a curve. Although *AddSine* plots the curve on PICT, you don't see it yet.

Program Instructions	Comments
«	
0 720	Sets up beginning and ending of FOR loop.
FOR a	Starts loop, creating local variable *a*.
a	Puts *a* on the stack.
a SIN	Puts sine of *a* on the stack.
R→C	Converts two quantities to (x,y) form.
PIXON	Puts a dot at point (x,y).
5 STEP	Increases *a* by 5 and goes through loop again.
»	

To save the program:

Keystrokes	Comments
ENTER	
' AddSine STO	Stores the program.

AddSine begins by placing the numbers 0 and 720 on the stack. Then the FOR a command begins the loop clause of the FOR ... STEP loop. The loop counter *a* (it represents the angle) goes from 0 degrees to 720 degrees, in steps of 5 degrees.

Each time through the loop, the program places the current value of *a* on the stack, then puts *a* on the stack again and calculates its sine. At this point, two quantities are on the stack: the current value of *a* and the sine of *a*.

We're going to use PIXON to actually plot each sine on PICT. Now, PIXON will turn on the specified pixel or point in PICT, but it needs as its argument the coordinates of that point, in the current units. Because in this program everything is in user units, we need to convert the two quantities now on the stack to a single complex number in (x,y) form.

Thus, before PIXON is executed, the program executes R→C. This command changes the two quantities into a complex number. For instance, suppose this is the third time through the loop. Thus *a* is 10, and the sine of *a* is 0.1736. With these two quantities on the stack, executing R→C produces the complex number (10, 0.1736), ready for PIXON.

PIXON actually places the point into PICT. With a complex-number argument, PIXON draws a point in PICT at the (x, y) coordinates specified by the number. The third time through the loop, then, PIXON places a point at 10 along x-axis (the horizontal) and at +0.1736 along the y-axis (the vertical). One point is plotted each time through the loop.

Because we've specified `5 STEP`, this subprogram takes a while to run. To speed it up, you can use a larger STEP value, such as `10 STEP`, but you'll get fewer plotted points. Also, notice that although the program plots the sines of every fifth angle from 0 through 720 degrees, you don't see the plot yet.

ShowGraph—Displaying PICT

PICT now contains the plotted sine curve. But PICT is off in its own never-never land right now, not in the display. To let you *see* the plot, *ShowGraph* brings PICT into the display.

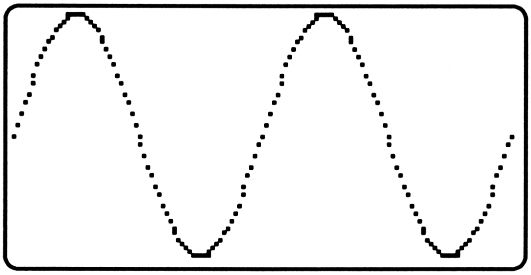
Program Instructions	**Comments**
`«`	
`{ } PVIEW`	Brings PICT into the calculator display.
`»`	

Keystrokes	**Comments**
ENTER	
' ShowGraph STO	

PVIEW brings PICT into the calculator display. If its argument is an empty list, as in this case, PICT is shown centered in the display, with scrolling mode activated.

Running Sine1

To run the program, just press the SINE1 menu key. There's a long pause while the sine curve is drawn on PICT. You don't see anything happening; it's all off-screen. When the sine curve has been completely drawn on PICT, you finally see it displayed.



Sine curve drawn by Sine1

Press ATTN to get back to the normal display.

Add Some Motion

What about movement, or watching something as it's plotted or drawn? One way to provide movement or animation is to re-show PICT every time it changes. That is, your program draws a point or object on PICT, then shows PICT in the calculator display. Then a point is added or the object moved, and PICT is displayed again. By using a loop to cycle through this procedure very quickly, you can see a line drawn on the screen, or follow the movements of a cartoon character. The *Sine2* program uses this technique to display the sine curve as it's drawn.

The Sine2 Directory

To begin, create a directory for the Sine2 program and subprograms:

['] SINE2
[↰] [MEMORY] CRDIR
[VAR]
SINE2

This SINE2 directory is going to be the repository for the main *Sine2* program and its subprograms.

The Main Sine2 Program

Program *Sine2* is even simpler than *Sine1*. It calls just two subprograms.

Program Instructions	**Comments**
«	
GraphArea	Same graphics area used in *Sine1*.
AddS2	Subprogram to show drawing of sine curve.
»	

To save the program:

Keystrokes	**Comments**
[ENTER]	Puts program on the stack.
['] Sine2 [STO]	Saves the *Sine2* program.

GraphArea is the very same subprogram used in *Sine1*. It sets degrees mode, erases PICT, and sets its plotting area to be from 0 to 720 along the x-axis, and from −1 to +1 on the y-axis.

The subprogram *AddS2* adds the sine plot to PICT, and displays it after each change, creating a “dynamic” drawing that happens before your eyes.

Modified Sine2—Featuring Directories and Paths

The main *Sine2* program brings up an interesting little problem: *GraphArea*. You used it in *Sine1*, and now you want to use it in *Sine2* as well. But if you've practiced good structured programming techniques, those main programs are in separate directories—SINE1 and SINE2, respectively.

How can you use a subprogram in this directory that you've created in another directory? Well, you could easily copy *GraphArea* to the SINE2 directory, using the techniques explained in chapter 2 of this book. But there's another way—adding path information with the PATH command. The following example illustrates how the *Sine2* program can go to another directory, execute a program there, and return to the current directory.

Program Instructions	**Comments**
`«`	
`PATH`	Gets copy of current directory path (for example, HOME MYGRAPHICS SINE2) and places it in the stack.
`SINE1 GraphArea`	Goes to directory SINE1 and executes the *GraphArea* subprogram. (*GraphArea* leaves nothing on the stack.)
`EVAL`	Changes back to the SINE2 directory.
`AddS2`	Executes subprogram to show drawing of sine curve.
`»`	

This modified version of *Sine2* begins with a PATH command. PATH puts a copy of the complete current path (for example, HOME MYGRAPHICS SINE2) on the stack. Then *Sine2* goes looking for the *GraphArea* subprogram.

Now remember that when a program tells the HP 48 to execute an object, the calculator searches first in its current directory, then in the directory above the current one, then in the directory above that, and so on. So unless *GraphArea* is in the same directory, or one above *Sine2*, the program won't find it. Thus, in this modified version, we add the directory, SINE1, in which *GraphArea* resides.

Now, *GraphArea* leaves nothing on the stack. However, after executing *GraphArea*, the calculator is set to the SINE1 directory. So we use the EVAL command to evaluate the path that's currently on the stack. This path (HOME MYGRAPHICS SINE2), when evaluated, changes the HP 48 back to the SINE2 directory again. Now it can execute the next subprogram, AddS2, and proceed normally.

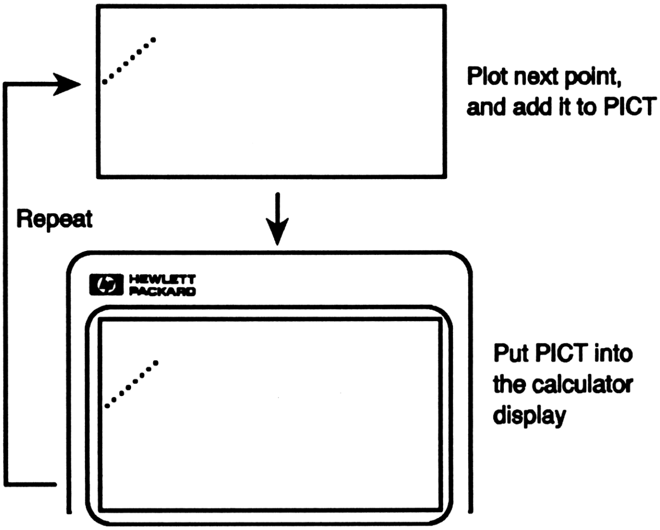
AddS2—Sequential PICT Displays

The *AddS2* subprogram is similar to *AddSine*, but it calls another subprogram, *ShoS2*, to display PICT every time a new point is drawn.

Program Instructions	**Comments**
«	
0 720 FOR a	Begins FOR ... STEP loop, with *a* from 0 to 720.
a	Puts *a* on stack.
a SIN	Puts *a* on stack and calculates its sine.
R→C	Converts *a* and sine of *a* to complex number (x,y) form.
PIXON	Draws a dot at point (*a*, sine *a*).
ShoS2	Calls subprogram to put PICT in calculator display.
5 STEP	
»	

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' AddS2 STO	Stores this program as *AddS2*.

AddS2 is virtually the same FOR … STEP loop as that found in the earlier *AddSine* subprogram—with one important difference. After plotting a point on PICT, *AddS2* calls the subprogram *ShoS2* to display PICT. This happens every time through the loop, so you see a continuously updated graphics screen. You watch as each dot is drawn, creating the familiar sine curve before your eyes. What *appears* to be a single graphics display, with dots added to it as the line is drawn, is actually a whole series of PICT displays, one after another.


Plot next point, and add it to PICT
Repeat
HEWLETT PACKARD
Put PICT into the calculator display

ShoS2—Placing PICT in a Corner of the Display

This subprogram is similar to an earlier program, *ShowGraph*. It displays PICT. Instead of an empty list as the argument for PVIEW, however, *ShoS2* supplies coordinates that put PICT at the upper left-hand corner of the display.

Program Instructions	Comments
`«`	
`(0,1) PVIEW`	Puts PICT at upper left corner of display.
`»`	

Keystrokes	**Comments**
ENTER	
' ShoS2 STO	Saves program as *ShoS2*.

Because *ShoS2* supplies the coordinates (0, 1) as the argument for PVIEW (instead of the empty list supplied by *ShowGraph*), PICT appears in the display only briefly. In fact, if you press the SHOS2 key, you'll get a lightning-fast look at PICT before the normal stack display returns.

In *AddS2*'s loop, though, *ShoS2* is called many times each second. You see several different images of PICT, one after another, until the loop has been executed—and the sine curve drawn for you.

Running Sine2

To run *Sine2* with all its subprograms, press the menu key marked SINE2. You'll see the same sine curve as you did with *Sine1*, but this time you get to view the drawing of every dot that makes up the curve—as it happens.

Animating Objects

You've seen how to handle PICT, and how to manipulate PICT's pixels to generate curves. Now we'll look at some techniques for adding *graphics objects* to PICT, and for moving those objects around.

The Flight Directory

Before you start learning about animation, make a directory to hold the programs and graphics objects you'll create. Here's how to make a directory and name it FLIGHT:

' FLIGHT
⬅ MEMORY CRDIR
VAR
FLIGH

The FLIGHT directory will hold all programs and subprograms for these animation experiments. Any graphics objects and programs that you save will now be safely tucked away in this directory.

Graphics Objects

One of the most fascinating features of the HP 48 calculator is the way it lets you work with graphics objects. A graphics object might be something as simple as a circle or a box, or it could be a face, a figure, or a fiery rocket. The graphics objects you create are limited only by your imagination and skill.

You create graphics objects using several different techniques. For instance, the HP 48 has its PIXON, LINE, BOX, and ARC commands for drawing on the PICT graphics area. Once something created with any of these tools appears in PICT, it can be saved as a graphics object and used again and again.

But that's not all. The HP 48 has an added feature: You can actually make "freehand" drawings, save them as graphics objects, and use them again in your programs.

Freehand Drawing of a Graphics Object

Here's how to use your freehand drawing skills to create a graphics object, one that you can then animate and manipulate in your programs.

1. Show the graphics screen. Use the following keys:
   a. [↰] [PLOT] PLOTR gets you to the PLOTR menu.
   b. ERASE erases anything already on PICT.
   c. [◀] (or [↰] GRAPH) places PICT in the display for viewing or editing.
2. Make your drawing. You can draw using [▲], [▼], [◀], and [▶] to move the cursor; and DOT+ , DOT- , LINE , TLINE, BOX , CIRCL, and MARK to add lines and shapes.

The cursor is the cross (+) you see in the main part of the screen. It shows the location where the next graphics action is going to happen.

DOT+ makes that cursor act like a pencil. Press DOT+ to toggle dots on, and you'll see a box next to the + sign in the *menu* display that shows this function is on. Now, as you move the cursor around the screen, you're drawing.

DOT- makes the cursor an eraser. To erase, toggle DOT- on and go back over the part you want to erase.

To erase a large area:

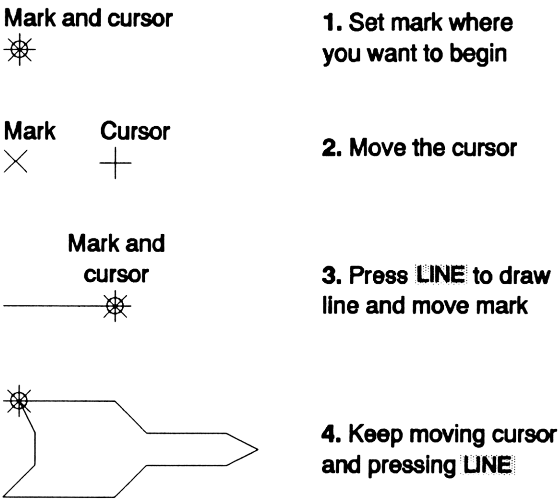
1. Move the cursor to the upper left of the area you want to erase.
2. Press MARK .
3. Move the cursor to the lower right of the area to erase.
4. Press DEL.

This leaves a small × where the mark was set.

LINE is perhaps your most useful drawing tool. LINE draws a line between the mark and the cursor, and moves the mark to the cursor. It's especially useful for creating diagonals.

Here's how to draw a figure with a lot of straight lines—like, say, a rocket ship:

1. Leave DOT+ off and set the cursor where you want to begin drawing. Then press MARK .


Mark and cursor
1. Set mark where you want to begin
Mark
Cursor
2. Move the cursor
Mark and cursor
3. Press LINE to draw line and move mark
4. Keep moving cursor and pressing LINE

2. Move the cursor to the first corner. Nothing is drawn yet.

3. Press LINE . You get a line from the mark to the cursor, and the mark moves.

4. Move the cursor and press LINE again. Keep following this procedure until your figure is drawn. Naturally, you can use DOT+ to fill in, or DOT- to erase, as you go.

TLINE toggles pixels on and off on the line between the mark and the cursor, so you can use it to erase entire lines. TLINE doesn't move the mark. Also, if the pixels between the cursor and mark are off, or clear, TLINE toggles them on.

For circles and boxes, use CIRCL and BOX .

Although MARK seems to leave a cross on the screen, the cross isn't part of the drawing, and disappears when you save a graphics object.

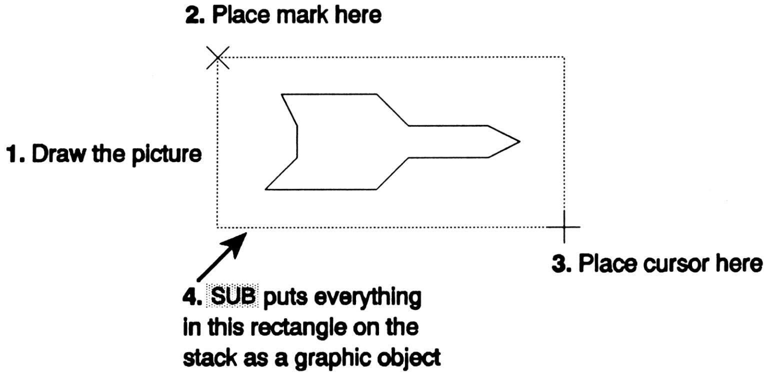
Saving Your Picture

Once you've created a picture, you probably want to save it. The easiest way is to convert it to a graphics object. You can then put this object on the stack, in a list, or anywhere into PICT (that is, anywhere in the graphics area) that you want. Here's the procedure:

1. Draw a picture using TLINE, DOT+ , DOT- , along with ▲, ▼, ◀, ▶, and the other drawing keys.

2. Move the cursor to the drawing's top left corner and press MARK .

3. Move the cursor to the lower right corner, so an imaginary box surrounds everything you want to save.


2. Place mark here
1. Draw the picture
3. Place cursor here
4. SUB puts everything in this rectangle on the stack as a graphic object

4. Press SUB . This puts the object on the stack.

5. Press ATTN to get back to the stack. You can see the size of the graphics object.

6. Key in 'Rocket' (or any other name), then press the STO key.

Now your drawing is stored as a graphics object. You can view it manually or use it in a program.

Viewing Your Drawing

You may want to check your graphics object manually, before using it in a program. Here's how:

1. Press VAR ROCK (or the name of the variable under which you've saved the drawing). This puts it on the stack.

2. Press ◀ to get to the graphics display and its menu keys.

3. Use the cursor keys to set the cursor where you want to put the top left corner of the object.

4. Press REPL . This puts the object into the displayed PICT.

REPL , of course, "uses up" the object from the stack. To put several graphics objects on the screen, you first put several in the stack, one after another. Then press ◀ to get the graphics environment, and press REPL . Move the cursor and press REPL again, repeating the process until all your rockets (or other objects) are used up.

Note to our readers: We're going to use the rocket to show how you can do animation on the HP 48. To follow along, you'll need something in the variable called *Rocket*. You can try your hand at drawing your own rocket, as explained here. But any old object will do. In fact, if you want a rocket in a hurry, you can just use the word "Rocket" instead of drawing it. The following keystrokes show how to turn text into a graphics object.

Keystrokes	Comments
" Rocket	
ENTER	Puts the word "Rocket" on the stack.
1 PRG DISPL	
NXT NXT →GRO	Turns that text into a graphics object, with letters sized small.
' Rocket STO	Saves the object as the variable *Rocket*.

If you use this procedure instead of actually drawing a rocket, you'll see the *word* "Rocket" flying past on your screen.

Animating a Drawing

Once you've created a graphics object, by whatever means, the next step is to animate it, make it move or jump around in the display. Here's one way to do it:

1. Determine the screen grid—that is, the size of PICT.
2. Show PICT using PVIEW.
3. Place the object in PICT. Use GOR or GXOR to combine the object with PICT, with its upper left corner at one set of coordinates.
4. Erase the object by using GXOR. If you GXOR the same object onto itself, the result is that the object is erased.
5. Keep repeating steps 3 and 4 with new coordinates each time. The result will be an appearance of movement.

The Main Flight Program

The *Flight* program gives a rudimentary procedure for making your rocket fly across the screen from left to right. It calls three subprograms, and has no overall effect on the stack.

Program Instructions	Comments
«	
Screen	Sets up the desired screen grid.
Showpict	Places PICT into the display.
Floop	Adds an object and makes it fly across the screen.
»	

To save the main *Flight* program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Flight STO	Stores the program.

Screen—Specifying a Flight Grid

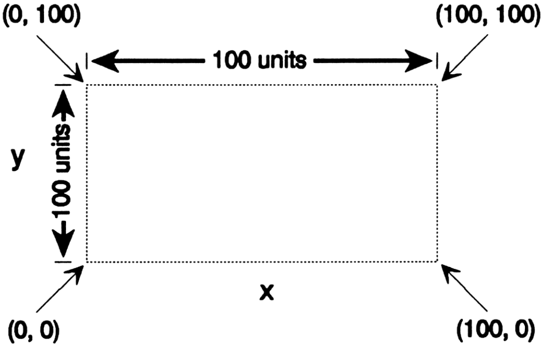
We begin with the *Screen* subprogram, which sets up a screen grid that's 0 to 100 in both the horizontal and vertical directions. This subprogram leaves the stack unchanged.

Program Instructions	Comments
«	
0 100 XRNG	Specifies horizontal axis from 0 to 100.
0 100 YRNG	Sets vertical axis from 0 to 100.
ERASE	Erases PICT.
CLLCD	Clears the HP 48's display.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Screen STO	Stores the program.

The statements `0 100 XRNG` and `0 100 YRNG` set up a PICT grid that's 100 × 100 units. The point (0, 0) is in the lower left corner and (100, 100) is in the upper right. We ERASE the graphics area, and clear the LCD screen with `CLLCD`.



Now we're ready to display PICT.

Showpict—Displaying PICT

The *Showpict* subprogram places PICT in the display.

Program Instructions	Comments
`«`	
`(0,100) PVIEW`	Sets 100 × 100 grid.
`»`	

To save this *Showpict* subprogram:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Showpict STO	Stores the program.

In order for you to see all of PICT, the PVIEW (*PICT view*) statement needs to know where to put PICT's upper left-hand corner. If we're using pixel units, PVIEW takes as its argument a list containing a pair of binary numbers like this:

```
{#0d #0d} PVIEW
```

Since we have chosen user units, though, PVIEW accepts a number that's in the commonly used (x,y) point form. (The HP 48 knows these as complex numbers.) We use `(0, 100) PVIEW` because it will place PICT's upper left corner 0 units along the x-axis, and 100 units up on the y-axis—that is in the upper left-hand corner of the display, right where we want it.

Floop—Simulating Movement Across the Screen

We've specified our screen grid size, and set the program to place PICT in the display. Now it's time for action. The Floop (*flight loop*) subprogram makes your rocket fly across the screen.

Program Instructions	Comments
`«`	
`0 100 FOR i`	Begins FOR ... STEP loop.
`  PICT`	Places PICT on stack.
`  i 50 R→C`	Puts $(i, 50)$ on stack.
`  Rocket`	Puts graphics object on stack.
`  GXOR`	Puts object in PICT at specified coordinates.
`  PICT`	Does the same thing again.
`  i 50 R→C`	
`  Rocket`	
`  GXOR`	This time, GXOR erases the object.
`5 STEP`	Ends FOR ... STEP loop.
`»`	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Floop STO	Stores the program.

To begin, we set up a FOR ... STEP loop. The statement `0 100 FOR i` is the beginning of the loop. It says "execute a loop that begins with an *i* of zero and goes up to an *i* of 100."

Everything between `FOR i` and `5 STEP` makes up the body—that is the loop clause. These instructions are all executed each time through the loop.

The key to "dynamically" placing a graphics object on PICT (and seeing it) is GOR or GXOR. Both operate much the same way. GOR and GXOR require the stack to be set up as shown below.

Stack Level	Contents
Level 3:	PICT
Level 2:	coordinates in PICT where graphics object will go
Level 1:	graphics object

Both of these commands use the three quantities shown, putting the graphics object into PICT at the location of the coordinates. (Actually, the upper left corner of the object is placed at the specified location in PICT.) PICT is left on the stack.

There's a subtle difference between GOR and GXOR. The GOR command draws the object in PICT no matter what. GXOR, however, toggles the PICT pixels underneath it. If they're light, the GXOR'ed object turns them dark—drawing the object. But if they're dark, the GXOR'ed object turns them light again. Simply put, if you GXOR an object onto itself, that object is erased.

To get ready for GXOR, within the loop, we begin by placing PICT on the stack. This is followed by two numbers. The first number is the current value of *i*—which is 0 the first time the loop is executed, 5 the second time, 10 the third time, and so on. This value will be used as the x-coordinate.

The second number onto the stack is 50. We use this as the y-coordinate. Although x changes, y is always the same; so whatever object we're placing in PICT move horizontally, but always keeps the same vertical position.

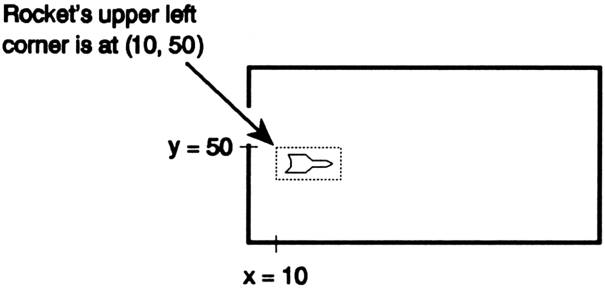
Now we have PICT in level 3, and those two numbers in levels 2 and 1 of the stack. But we need the numbers to be coordinates in a form GOR or GXOR can use. Since we're dealing in user units, that means a complex number—so we convert the number with R→C.

The R→C command takes two numbers off the stack and converts them to a complex number. Suppose this is the third time through the loop. The two numbers on the stack are 10 (in level 2) and 50 (in level 1). Executing R→C produces (10,50), which are the coordinates we need in the form we need them for GOR or GXOR.

Finally we put the graphics object—the Rocket—on the stack. Now the three stack levels are set up for GOR or GXOR. If this is the third time through the loop, the stack is set up this way:

Stack Level	Contents
Level 3:	PICT
Level 2:	(10,50)
Level 1:	Rocket

When we execute GOR or GXOR (it doesn't matter which), the rocket is drawn in PICT, with its upper left-hand corner at the specified coordinates.



By drawing and erasing the rocket many times, each time at a new position, we simulate its flight across the screen.

Running the Flight Program

To run the program, press the FLIGH menu key. You'll see the rocket travel once across the screen.

Note that this is one place where you can't simulate the *Flight* program by pressing the SCREE menu key followed by SHOW, followed by FLOOP. In order for PICT to be used "dynamically" like this, it has to appear after PVIEW, and before program control returns to the keyboard.

A Better Flight—Flit2

The *Flight* program works fine—as far as it goes. Now let's look at *Flit2*, which is a modified version of *Flight*. However, it has a few more bells and whistles.

The Flit2 Directory

If you want to keep the *Flit2* programs separate from your other graphics applications, create a separate directory.

['] FLIT2
[↰] [MEMORY] CRDIR
[VAR]
FLIT2

Flit2 uses some of the same subprograms as *Flight*. If you elect to use a separate directory, you can proceed as you did with *Sine2*: either copy the programs *Screen* and *Showpict* from FLIGHT; or else place the current path on the stack with the PATH command, specify the path to the needed subprograms in FLIGHT, then use EVAL to evaluate the current path and get back to the FLIT2 directory.

The Main Flit2 Program

Flit2 is very similar to *Flight*. It calls *Screen* to set up PICT as a 100 × 100 grid, then calls *Showpict* to place PICT in the upper left corner of the display, ready for viewing. After that, though, we make some changes. We execute the *Headline* subprogram, then call *Flop2* as the main loop.

Program Instructions	**Comments**
`«`	
`Screen`	Declares PICT as 100 × 100 screen grid.
`Showpict`	Displays PICT.
`Headline`	Displays a title on the screen.
`Flop2`	Main flight loop for *Flit2*.
`»`	

To save the main *Flit2* program:

Keystrokes	Comments
ENTER	
' Flit2 STO	Saves the program as *Flit2*.

Headline—Putting Text on the Graphics Screen

You're not limited in the number of graphics objects you can put into PICT. For instance, the *Headline* subprogram adds a written word to the top of the display, so as the rocket whizzes past, you also get a textual clue to what's going on.

After setting the screen grid and displaying PICT the *Flit2* program next calls *Headline*. This subprogram places the word ROCKETING in about the center of the display, and leaves it there.

Program Instructions	Comments
«	
PICT	PICT goes in level 3, ready for →GROB.
(30,90)	Coordinates where level 1 object will be put in PICT.
"ROCKETING"	Word to be placed in PICT.
3 →GROB	Converts "ROCKETING" to graphics object.
GOR	Puts the word “ROCKETING” into PICT at coordinates (30,90).
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Headline STO	Stores the program.

This tiny subprogram illustrates the use of →GROB (*to graphics object*). →GROB takes an object from level 2 and a number from level 1, and converts the object into a graphics object—that is, something you can put into PICT and its display.

In this case, the object is a text string, "ROCKETING". The number just before →GROB specifies the character size; it can be from 0 to 3. Here, the 3 signifies that the text string will be large letters. (A 2 would be medium-sized, and a 1 would give small letters. 0 is the same as 3 for text strings.)

Once "ROCKETING" has been converted to a graphics object, the stack is all set up for GOR. (You can see this if you execute the subprogram using the PRG CTRL DBUG keys, followed by SST.)

Stack Level	Contents
Level 3:	PICT
Level 2:	(30,90)
Level 1:	Graphic 54 × 10 (graphics object)

With these quantities on the stack, we execute GOR. The command puts the word ROCKETING into PICT at the specified coordinates—that is, with its top left-hand corner 30 units along the horizontal (x) axis, and 90 units up along the vertical (y) axis. This locates it above the rocket's path, in approximately the center of the display.

Notice that we're inserting the word ROCKETING into PICT before we call the animation loop in *Flop2*. In fact, using subprograms is an excellent way to add many graphics objects to a display.

Flop2

We've rewritten the animation loop so that now it uses local variables, and is executed continuously until you press a key.

Program Instructions	Comments
`«`	
`Do`	Starts DO loop.
`0 100 50 5`	Puts appropriate numbers on stack.
`→ xb xe y s`	Declares local variables.
`«`	Begins defining procedure (a program) for local variables.
`xb xe FOR i`	FOR *xb* to *xe* . . .
`PICT`	Puts PICT in level 3.
`i y R→C`	Converts these coordinates to (*i*, *y*) form; put in level 2.
`Rocket`	Puts graphics object *Rocket* in level 1.
`GXOR`	Places *Rocket* into PICT.
`PICT`	
`i y R→C`	
`Rocket`	
`GXOR`	GXORs again to turn off rocket display.
`s STEP`	STEPs by *s* units
`»`	
`UNTIL KEY END`	
`DROP`	
`»`	

To save this program:

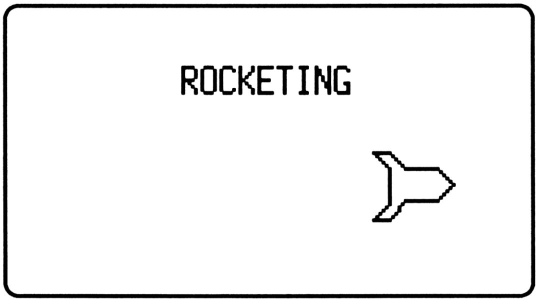
ENTER
' Flop2 STO

Running the Flit2 Program

With the main *Flit2* program and all subprograms in one directory of the calculator, just hit the FLIT2 menu key to run it.

Keystrokes	Comments
FLIT2	Press to begin program

The rocket speeds across the screen, highlighted by the word "ROCKETING" above. When you've had your fill of this animated display, press any key to stop the program after this complete passage of the rocket. Or hit ATTN to end the program immediately.



The rocketing rocket as captured by Flit2

8

Fun and Games

Small size, a phantasmagoric array of functions, and a sophisticated graphics capability make your HP 48 calculator ideal for creating games. You can slay dragons, rescue knights or damsels in distress, blast deep-space aliens from the skies, and more. All it takes is the right software.

Although complex, full-featured games are beyond the scope of this book, this chapter illustrates some of the fundamental techniques for creating games. Among other things, this section features a standalone random number generator that can easily be adapted to other uses.

Random Number Generator

One thing that makes computer games so intriguing is their unpredictability. When you play a really good video game, for instance, you never know where the next alien will land, or where the next turn in the road is going to take you. Chances are this lack of predictability is ensured by a *random number generator* within the game.

The program shown in this section is a general random number generator that can be "scaled" to any number. For instance, if you're creating a card game, you might want the random number to be between 1 and 52, to account for all 52 cards in a deck. If you're rolling a single die, you'd want the random number to be between 1 and 6. A lottery program might require six random numbers between 1 and 100. The random number generator can produce all these numbers, and a lot more besides.

Keyboard Example

First, let's look at how we can create random numbers on the HP 48. We begin with the RAND function. RAND produces a pseudo-random number between 0 and 1. For instance, look what happens if you press RAND several times in a row:

Keystrokes	**Display**	**Comments**
MTH PROB RAND	0.9683	First random number.
RAND	0.5162	Second random number.
RAND	0.5493	Third random number.

The numbers on your calculator will be different—after all, they're random, aren't they? However, they'll definitely be between 0 and 1. The biggest task of the random number generator is to adjust or scale these values so they fall in the correct range—whether 1 to 6, 1 to 52, 20 to 50, and so on.

The Random Directory

To create a directory for the random number program and subprograms, then get into that directory:

' RANDOM
↰ MEMORY CRDIR
VAR
RAND

The RANDOM directory will hold all programs and subprograms for generating random numbers. Any objects (including programs) that you enter and save will now be placed in this directory.

The Main Random Program

We use structured programming to create the main program, which is mostly a series of calls to various subprograms. Since we need to get two values, however, we set up the appropriate screen display message for each value, then call a generic *GetVal* subprogram.

The program prompts the user to enter a bottom limit and a top limit for the random number. For instance, if you want to throw a single die, you'd enter "1" for the bottom limit, and "6" for the top limit. The random number generated by the program will then be adjusted so it's somewhere in that range. If you wanted to simulate drawing a single playing card from a deck, you'd enter "1" for the bottom limit and "52" for the top. *Random* takes nothing from the stack, and leaves a tagged random number there. In the course of generating a random number, the main *Random* program places a string on the stack for use by one of its subprograms.

Arguments	Results
1:	1: tagged random number

Program Instructions	**Comments**
`«`	
`Init`	Calls initialization routine.
`Title`	Shows a message to the user.
`"Enter the top limit"`	User instructions for upper value.
`GetVal`	Gets a value from the user.
`"Enter the bottom limit"`	User instructions for lower value.
`GetVal`	Gets another value from the user.
`Create`	Creates a random number.
`Adjust`	Scales the random number between upper and lower values.
`Tagit`	Adds label to the random number.
`»`	

To save this main program as *Random*:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Random STO	Stores the program as *Random*.

The main *Random* program calls *Init* to perform some rudimentary initialization, then runs *Title* to introduce what's going to happen next. Then *Random* places a string onto stack level 1. This string is present when *GetVal* is called, and it's used by *GetVal* to explain which limit to enter.

Once both limits have been entered, *Random* calls the subprogram *Create*, which actually creates the random number. Then it calls *Adjust*, which scales the random number to the specified range. Finally, *Random* calls *Tagit* to add a suitable tag to the number.

Init—Displaying Whole Numbers Only

The *Init* subprogram does nothing more than set the calculator to show each number to zero decimal places:

Program Instructions	**Comments**
«	
0 FIX	From now on, numbers are shown with no decimal places.
»	

Keystrokes	**Comments**
ENTER	
' Init STO	Stores this short subprogram.

It's true that *Init* is almost too simple now, with a single instruction perhaps undeserving of a subprogram of its own. But this illustrates the benefits of structured programming. If you need further initialization in the future (such as setting a system or user flag, or displaying a menu), it's much easier to add to a simple subprogram like this than to wade through lines and lines of code in a long program, trying to find out where initialization begins.

Title—Displaying Until Continue

Next comes the *Title* subprogram, which tells the user what's coming up. As with nearly all the title or message subprograms in this book, you can omit *Title* and all references to it without changing the actual program operation. *Title* has no overall effect on the stack.

Program Instructions	Comments
`«`	

`"RANDOM NUMBER`
`GENERATOR`
`You enter the top and`
`bottom of the range.`
`You get a random`
`number in that range.`
`CONT to continue."`
	Message displayed by DISP.

`CLLCD`	Clears the calculator's entire display.
`1 DISP`	Shows the message, beginning at line 1 of the display.
`7 FREEZE`	Freezes the display, showing the message.
`HALT`	Halts the program until the user presses CONT.
`»`	

Use the following keystrokes to store this subprogram:

Keystrokes	Comments
ENTER	
' Title STO	Stores the program as *Title*.

Title begins by placing a long string on the stack. If you want your entire message to be displayed at once, without scrolling, use the endline (↱ ↵) to break the lines exactly where they're shown on this listing.)

We execute CLLCD to clear the calculator's LCD. Then we place a 1 in level 1 of the stack. Using the message string from level 2 and the 1 in level 1, DISP displays the message beginning at line 1 of the display—that is, at the top.

The next instruction, 7 FREEZE, freezes the display so all that's visible is the message. Because we've used 7 as the argument, you don't even see the status line or the menu keys.

Finally, *Title* has a HALT instruction. This stops program execution and returns control to the keyboard; to continue, you have to press ↰ CONT. There are other ways of continuing here—for instance, using 0 WAIT instead of HALT would suspend execution only until the next key press.

GetVal—Waiting for User Input

Up to this point, all the subprograms have been "neutral." They expect nothing from the stack when they're called, and they return nothing to the stack when finished. *GetVal*, however, is different. *GetVal* expects a string to be in level 1 of the stack. It uses this string to display a user instruction as it waits for input.

Arguments	Results
1: text string of user instructions	1: number (top or bottom limit)

Program Instructions	Comments
«	
":Limit:"	Second part of two-string argument for INPUT.
INPUT	Prompts and wait for input from the user.
OBJ→	Converts user input into numeric object.
»	

To save this as *GetVal*:

Keystrokes	Comments
ENTER	Puts program on the stack.
' GetVal STO	Watch the capitalization!

INPUT, remember, prompts for data input. It requires a prompt string in level 2, and a command-line string in level 1. In the case of *GetVal*, the prompt string is placed on the stack by the main program, *Random*. The command-line string is ":Limit:", and it's added at the beginning of the *GetVal* subprogram.

Thus, the first time through the *GetVal* program, the following quantities are on the stack when INPUT is executed:

Stack Level	Contents
Level 2:	"Enter the top limit"
Level 1:	":Limit:"

Thus, INPUT exhibits the prompt string from level 2 in the middle portion of the display, and the command-line string in the command line. The cursor is poised at the command line, ready for you to enter a number for the upper limit.

At this point in the running program, you enter a value for the upper limit, then press ENTER. The OBJ→ instruction turns your keyboard input into an object—that is, a number—and places it on the stack. Thus, *GetVal* expects a string on the stack when it's called, and returns a number when it's done.

GetVal is called once by *Random*, resulting in the upper limit being placed on the stack. Then the calling program places "Enter the bottom limit" on the stack and calls *GetVal* again. And once again *GetVal* returns a number, this time the lower limit.

Create—Generating a Random Number

With the top and bottom limits loaded on the stack, *Random* now calls the *Create* subprogram. *Create* generates a random number between 0 and 1, and places it on the stack (below the upper and lower limits).

Arguments	Results
3:	3: *top limit*
2: *top limit*	2: *bottom limit*
1: *bottom limit*	1: decimal random number

Program Instructions	Comments
«	
RAND	Generates a random number.
»	

To save it:

Keystrokes	Comments
ENTER	Puts this one-line program on the stack.
' Create STO	Stores the program as *Create*.

As you can see, *Create* now consists of just one instruction: RAND. This produces a pseudo-random number between 0 and 1 and places it on the stack.

Adjust—Defining Local Variables without a Program

The *Adjust* subprogram takes three quantities from the stack, the top and bottom limit and the decimal random number. It returns a scaled random integer that is no higher than the top limit, and no lower than the bottom limit.

Arguments	Results
`3:` top limit	`3:`
`2:` bottom limit	`2:`
`1:` decimal random number	`1:` scaled random number

Program Instructions	Comments
`« → t b r`	Makes local variables *t*, *b*, *r*.
`   'b+(t-b+1)*r'`	Algebraic object scales random number.
`   IP`	Uses only integer portion.
`»`	

Keystrokes	Comments
ENTER	Puts program on the stack.
' Adjust STO	Stores it as *Adjust*.

This program takes three numbers off the stack, and turns them into local variables *t*, *b*, and *r* (for top, bottom, and random, respectively). As soon as the three local variables are created, they're used immediately in their *defining procedure*.

In many cases the defining procedure is another program. Here, though, it's an algebraic object. Putting the entire formula `b+(t-b+1)*r` between tick marks means that those local variables aren't placed one by one on the stack. Instead, they're evaluated together, producing a random number that's scaled in the correct range.

Let's see how this formula works. Suppose we're playing a stock market game, and the stock price must be somewhere between 20 and 50. Thus, when the defining procedure is run, t is 50 and b is 20.

Random number r, remember, is between 0 and 1. For purposes of this illustration, we'll say it could be as low as 0.00001 or as high as 0.99999. Let's take 0.99999 first. With this value as the random number, the algebraic object evaluates like this: $20 + (50 - 20 + 1) * 0.99999 = 50.99969$.

Remember, the HP 48 does everything according to a strict order of evaluation—that is, according to rules of precedence. It does the subtraction and addition in parentheses first to get 31, then multiplies by 0.99999, then adds the value 20. The result is greater than 50, but by just a smidgin—and we'll take care of that soon enough.

What about the opposite pole: suppose r is around 0.00001? In this case, $20 + (50 - 20 + 1) * 0.00001 = 20.00031$. The answer is just a little more than 20.

At this point, we are guaranteed to have a random number that's in the range of just a little more than 20 to just a little more than 50. (It could be anywhere in between, too.) Now *Adjust* uses IP to extract just the *integer portion* of this value and put it on the stack. (The fractional part of the number—all those numbers to the right of the decimal point—is thrown away.) And voila! We have a random number that's in the range from 20, 21, 22 ... up to 49 or 50.

This subprogram illustrates one of the benefits of local variables: they make your programs more understandable. We make a simple, easy-to-read formula out of the variables, then evaluate that formula to produce the random result.

Tagit—Labeling the Result

This is another of those nice-but-not-really-necessary fillips we've added to *Random*. When *Tagit* is called, the random number is on the stack. *Tagit* merely adds an identifying tag to the number.

Arguments	Results
1: scaled random number	1: tagged, scaled random number

Program Instructions	Comments
«	
"Random no"	Tag for number.
→TAG	Adds tag to the number.
»	

Keystrokes	Comments
ENTER	
' Tagit STO	Stores the *Tagit* program.

Thanks to *Tagit*, the *Random* program produces a tagged random number.

Running Random

To run the *Random* program and generate a random number from 1 to 6, just press the key for RAND :

Program Prompt or Display	Your Action
	RAND

```
RANDOM NUMBER
GENERATOR
You enter the top and
bottom of the range.
You get a random
number in that range.
CONT to continue.
```

There are the user instructions. Now use the CONT key to continue.

Program Prompt or Display	Your Action
	↰ CONT
`Enter the top limit`	
`:Limit:`	6 ENTER
`Enter the bottom limit:`	
`:Limit:`	1 ENTER
`Random no: 4.`	

Your random number, of course, may be different from the one we got. The `Random no:` tag doesn't affect the number itself, and is simply lost if you use the number. So you can apply the random numbers generated by this program in other programs, if you choose.

Rolling Dice

Dice are at the heart of many popular games—and it's been that way for thousands of years. These small cubes, made of ivory, bone, or wood, were used during the heyday of Greece and Rome, and have even been found in ancient Egyptian tombs.

The dice program lets you "roll" a pair of six-sided dice, then see the result on the HP 48's liquid crystal display. The program also remembers your first roll and your latest roll, and displays them on demand.

The *Dice* program makes use of a random number generator and actually shows the dice on the graphics screen. It supplies a menu of choices for you, too.

The Dice Directory

This program has a lot of subprograms associated with it, so it's best to create a separate directory, called DICE. Here's how to do it:

' DICE
⬅ MEMORY CRDIR
VAR
DICE

Now you're in the DICE directory, and can begin typing the main program and subprograms.

The Main Dice Program—A Temporary Menu

The main *Dice* program accomplishes three tasks:

- Initializes the calculator.
- Displays a menu of user keys.
- Displays user instructions.

With menu keys, you can roll the dice, review your first roll, review your latest roll, begin a new game, or exit. After any choice (except exit), you're brought back to the menu and user instructions again. *Dice* has no overall effect on the HP 48 stack.

Program Instructions	**Comments**
`«`	
`Init`	Initializes HP 48.
`{`	Begins temporary menus
`{ "Roll"`	Label for menu key.
`«`	Begins key definition (a program of subprogram calls).
`Interim`	Displays message while rolling.
`Rollnum`	Determines which of two dice is being rolled.

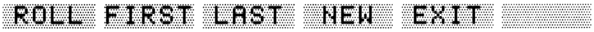
Program Instructions	Comments
`Ranum`	Generates random number, 1 through 6.
`Number`	Draws a die.
`Rollnum`	
`Ranum`	
`Number`	Draws a die again.
`Saveit`	Saves this roll (and first roll, if applicable).
`Showit`	Displays the pair of dice and waits for ATTN.
`Title`	Displays user instructions.
`»`	Ends key definition.
`}`	Ends list for first menu key.
`{ "FIRST"`	Label for second menu key.
`«`	Begins key definition.
`Getfirst`	Gets the first roll.
`Showit`	Displays the first roll.
`»`	Ends key definition.
`}`	Ends list for second menu key.
`{ "LAST"`	Key label for third menu key
`«`	Begins key definition.
`Getlast`	Gets the latest roll.
`Showit`	Displays that latest roll.
`»`	Ends key definition.
`}`	Ends list for third menu key.

Program Instructions	**Comments**
`{ "NEW"`	Label for fourth menu key.
`Dice`	Initializes program, shows menu and user instructions.
`}`	Ends list for fourth menu key.
`Exit`	Label *and* action for fifth menu key.
`}`	Ends temporary menu list.
`TMENU`	Creates temporary menu, using list.
`Title`	Displays user instructions.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Dice STO	Stores the main *Dice* program.

Dice calls the *Init* subprogram to prepare the calculator and specify graphics display range. Then *Dice* places a long list—it's actually a "list of lists" in the stack, and TMENU uses this list to create a temporary menu. The temporary menu looks like this:



The nice thing about TMENU, from a programming point of view, is that it creates a menu, but doesn't "lock up" the calculator waiting for you to press a key. That is, none of the actions actually occur, nor are any of the subprograms executed (other than *Init* and *Title*), until you press one of the five defined menu keys.

TMENU uses as its argument a list that's in this format:

`{ {"label1"` *object1* `} {"label2"` *object2* `}... }`

An outer set of curly brackets defines TMENU's entire list. Within this outer set of brackets are several inner sets of curly brackets—that is, several "inner lists." Each inner list contains the label and the definition (an object) for one menu key. The key's label is a string, enclosed in quotation marks. The key's definition, which is what happens when you press that key, is usually a program.

If you press ROLL, the HP 48 executes the object that is the definition of that key—that is, the program of subprogram calls. All the programs in that first inner list are called, in the order shown. *Interim* displays a message that stays on the screen until the actual dice are displayed. While that's happening, the *Rollnum* subprogram determines whether this is the first or second die, and sets up display coordinates accordingly. *Ranum* generates a random number from 1 through 6, and *Number* actually draws the die and the required number of dots on PICT, the HP 48's "canvas." *Rollnum*, *Ranum*, and *Number* are called twice, resulting in a pair of dice being drawn.

The next program, *Saveit*, saves a "picture" of this roll of the dice; and if this is the first roll, that picture is also saved in another location.

With the dice drawn in PICT and copies safely stored, *Showit* is called to overwrite the interim message and actually display the dice on the HP 48's LCD. The display persists until you press ATTN. Then TMENU's temporary menu is displayed again, and *Title* is called to redisplay detailed user instructions.

Notice that the a definition for a menu key used by TMENU must be a single object—often a name or a program. Thus, when using one key to call several programs like this, you must place their names between program brackets in the list. If you're only calling one program (look at the lists for NEW and EXIT) you can use the program name alone.

If you press the FIRST or LAST menu key, *Dice* executes *Getfirst* or *Getlast* to put a copy of your first roll or your latest roll, respectively, into PICT. Then *Showit* is called to display PICT, and you can verify what your roll was.

Pressing NEW allows *Dice* to call itself recursively. This erases the copies of the first and latest rolls, preparing the HP 48 for a new game.

Finally, take a close look at the "list" for the fifth menu action—exit. This shows another technique for simplifying and saving space in your menu displays. Notice that there's no separate set of curly brackets, and no separate program call. In this case, the word "Exit" is both the program *name* and the menu key *label*. You see EXIT on the menu line. But the program executed if you hit this key is *Exit* (which shows the ordinary VAR menu and stack display again). You don't need the curly brackets here, because the label and its called subprogram are spelled exactly the same.

Init—Creating Blank Pictures

Init prepares the calculator to roll dice. The *Init* subprogram puts blank copies of PICT in two variables, *First* and *Last*. It also sets the X-Y display range, and clears two user flags for keeping track of things later on. *Init* doesn't affect the stack.

Program Instructions	Comments
`«`	
`ERASE`	Erases graphics area, PICT.
`PICT RCL`	Recalls copy of erased PICT to the stack.
`DUP`	Copies erased PICT to stack level 2.
`'First' STO`	Stores copy of erased PICT in variable *First*.
`'Last' STO`	Stores copy of erased PICT in *Last*.
`1 131 XRNG`	Sets horizontal display range.
`1 64 YRNG`	Sets vertical display range
`11 CF`	Clears flag to keep track of die 1 or die 2.
`22 CF`	Clears flag to mark first roll.
`»`	

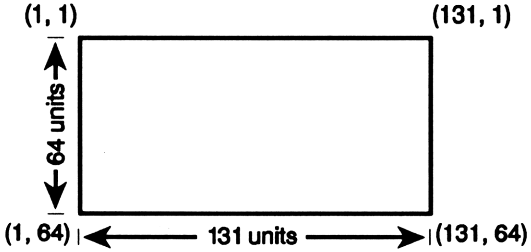
To save the *Init* program:

Keystrokes	Comments
ENTER	
' Init STO	Stores the program.

Init begins by using ERASE to clear the entire graphics "canvas", known as PICT. Then PICT RCL puts this graphics object into the stack, and DUP duplicates it. Thus, two copies of the empty PICT are on the stack.

Next, 'First' STO takes one of those blank graphics objects and stores it in a variable called *First*. Then a copy is also stored in the variable *Last*.

Although we'll be employing user units in *Dice*, we're going to set them to be the same number of pixels as are on the screen. This ensures the optimum in resolution. Thus, the horizontal display range is set to be 1 to 131 units by XRNG; and the vertical display range is set for 1 to 64 units by YRNG. This results in a display area that's in user units, but is the same size as the pixel-unit display.



Finally, *Init* clears a pair of user flags. The numbers for system flags are preceded by – signs, but user flags are not. The numbers for these flags are purely arbitrary, and were selected to make them easy to remember.

Flag 11 will keep track of whether this displayed die is the first or second one. It really determines whether *Number* draws this die on the left or right side of the display area. Flag 22 will be cleared if this is the first roll of the dice, and set for all other rolls. The flag's condition will tell us whether to store this roll in *First*.

Title—Detailed User Instructions

After *Init* has initialized the program and TMENU has displayed the temporary menu of key choices, *Dice* executes the *Title* subprogram. *Title* displays detailed instructions that explain more fully what each menu key does. *Title* has no overall effect on the stack.

Program Instructions	**Comments**
`«`	
`CLLCD`	Clears liquid crystal display

`"ROLLING DICE`
`ROLL-To roll dice`
`FIRST-Review 1st roll`
`LAST-Review last roll`
`NEW-New game`
`EXIT-To quit`
`[ATTN] after dice"`
	Seven-line display string for use by DISP.

`1 DISP`	Displays string beginning on line 1.
`3 FREEZE`	Freezes everything but menu keys.
`»`	

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Title STO	Saves the program as *Title*.

Title begins by clearing any previous messages from the HP 48's LCD, then places a long text string into the stack. (If you want all your text display to be visible at once, be sure to place endline characters, ↱ ↵, at the end of each line.) The text string occupies seven lines—which is the maximum number of lines that your HP 48 can display at once.

DISP needs two quantities on the stack: a string in level 2, and a number from 1 to 7 in level 1. The level 1 number tells DISP where to display that text. In this case, DISP displays these user instructions beginning at the top (line 1) of the display.

To keep the display from disappearing as quickly as it appears, we use 3 FREEZE. This command freezes the status line and main part of the display (the stack area) but permits the menu-key area to be updated. So you see only the user-instruction display and the temporary menu keys.

Interim—Providing an Action Message

Because of internal calculations and graphics requirements, the HP 48 takes several seconds from the time you press ROLL until you see the pair of dice on the screen. *Interim* displays a message during this entire time. It doesn't affect the stack.

Program Instructions	**Comments**
«	
ERASE	Erases PICT.
CLLCD	Erases liquid crystal display.
"Rolling..."	Text for display by DISP
4 DISP	Displays text on line 4 of LCD.
3 FREEZE	Freezes status area and stack area until next key press *or* next display.
»	

To save this *Interim* program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Interim STO	Stores the program.

Interim begins by clearing the graphics area with ERASE. This command doesn't affect the calculator's liquid crystal display (which is what you're viewing at the moment). Instead, it erases PICT, which is the HP 48's "canvas" for drawing pictures. Later on, we'll draw something on PICT and put PICT into the display for viewing—but for now, we use ERASE to get rid of any dice that may already be drawn on PICT.

To clear any previous messages or graphics from the *display* we use CLLCD. This command erases the HP 48's liquid crystal display and prepares it for new information.

Now *Interim* displays its message, "Rolling...". This status string is put into the stack, followed by the number 4. The DISP command takes these two quantities and displays the text right smack in the center row (row 4 of 7) of the display.

Because other actions occur right after *Interim* has run, we need FREEZE here. 3 FREEZE keeps the message on the screen while the rest of the subprograms are running. FREEZE, remember, freezes the display until the next key press or the next display. All the while the HP 48 is running the subprograms *Rollnum*, *Ranum*, *Number*, *Rollnum*, *Ranum*, *Number*, you see the message "Rolling..." in the display. Although a zillion things are happening in the background, you see only the message.

Rollnum—Testing a Flag and Making a Decision

This subprogram determines which of the two dice is drawn in PICT. It expects nothing from the stack, but after running, it leaves an x- and y-value on the stack for use by later subprograms. It also calls *Die*, to draw a die's outline on the left or the right side of PICT.

Arguments	Results
2:	2: x-coordinate
1:	1: y-coordinate

Program Instructions	Comments
«	
IF 11 FC ?	Tests user flag 11.
THEN	If flag 11 is clear ...
10	Places coordinates 10
6	and 6 on the stack.
11 SF	Sets flag 11.
ELSE	If flag 11 is set ...
70	Places coordinates 70
6	and 6 on the stack.
11 CF	Clears flag 11.
END	End of IF ... THEN ... ELSE.
Die	Calls subprogram to draw one die.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Rollnum STO	Stores *Rollnum*.

The subprogram begins by examining user flag 11. If this is the first time the subprogram is called, flag 11 is clear (it was cleared by *Init*), and the subprogram places the numbers 10 and 6 on the stack. These are the upper left coordinates of the left "die."

Next, the program sets flag 11, so the next time *Rollnum* is called, it places the coordinates 70 and 6 on the stack. These coordinates will be used as starting points to draw the die, and for the individual dots for each die in the PICT graphics area. *Rollnum* calls *Die* to draw one half of a pair of dice. The die is drawn on the left or right side depending on the coordinates in the stack. Nothing is actually displayed yet, though.

After drawing the outline of the die, the *Rollnum* subprogram leaves the numbers on the stack. These are beginning x- and y- coordinates for the dots that will later be drawn. The coordinates mark the upper left-hand corner of the current die.

What happens if flag 11 happens to be set when *Rollnum* is called? In this case, the statement `IF 11 FC?` is false, so everything between `ELSE` and `END` is executed. The program puts coordinates 70 and 6 on the stack for later use by *Die* and the dot-drawing program. Then *Rollnum* clears flag 11.

In successive calls, *Rollnum* toggles back and forth:

- 1st call to *Rollnum*: draws left die, sets flag 11.
- 2nd call to *Rollnum*: draws right die, clears flag 11.
- 3rd call to *Rollnum*: same as first call.
- 4th call to *Rollnum*: same as second call.

Since *Rollnum* is called twice for each "roll" of the dice, it draws both dice in the PICT graphics area.

Die—Drawing a Box

Die draws a die in PICT; where the die is drawn (on the left or right side) depends on the coordinates passed from *Rollnum*. *Die* expects two numbers on the stack, and leaves the same two numbers there when finished.

Arguments	Results
`2:` starting x-coordinate	`2:` starting x-coordinate
`1:` starting y-coordinate	`1:` starting y-coordinate

Program Instructions	Comments
`«`	
`→ x y`	Creates local variables *x* and *y*.
`«`	Starts defining procedure for local variables.
`x y`	Places *x* and *y* on stack.
`R→C`	Converts to complex number; upper left BOX coordinates.
`'x + 50' EVAL`	Adds 50 to *x*.
`'y + 50' EVAL`	Adds 50 to *y*.
`R→C`	Converts to complex number; lower right BOX coordinates.
`BOX`	Draws box.
`x y`	Leaves *x* and *y* on stack for next subprogram.
`»`	
`»`	

Keystrokes	Comments
ENTER	
' Die STO	Stores the program.

The heart of *Die* is the BOX command, which draws a box on the PICT graphics area. But BOX requires as its argument two sets of coordinates, in complex-number form: for example (10,6) or (60,56).

This is a perfect spot to use local variables. *Die* takes the x- and y-coordinates passed from *Rollnum*, and converts them to local variables *x* and *y*. Then the defining procedure (a program) for the local variables begins.

Executing `→ x y` uses up the two quantities from the stack. So the first thing we do in the defining procedure is put them back. Then the `R→C` command converts two quantities on the stack to a complex number. For instance, suppose the stack has x- and y-coordinates on it like the following:

Stack Level	Contents
Level 2:	10
Level 1:	6

After the R→C command, the stack looks like this:

Stack Level	Contents
Level 1:	(10,6)

That takes care of the upper left coordinates needed by BOX. Now we'll generate the lower right coordinates.

When we draw a die—a box—we want it to be 50 units by 50 units in size. So we add 50 to *x*, then add 50 to *y*. Using the algebraic form, as we have here (for example, 'x + 50' EVAL) makes the process easier to understand. But we could have saved a few bytes by using stack operations instead. (For example, x 50 +.)

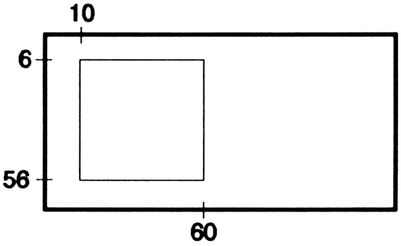
After execution of 'x + 50' EVAL and 'y + 50' EVAL, the two lower right coordinates are on the stack. Another R→C command turns these into a complex number so everything is set up for BOX. If the left die is to be drawn, the stack looks like this:

Stack Level	Contents
Level 2:	(10,6)
Level 1:	(60,56)

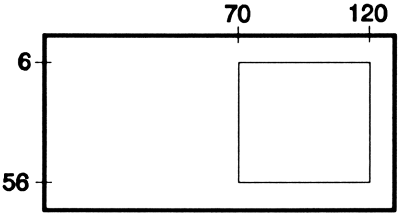
If we're drawing the right die, 70 and 6 have been passed from *Rollnum*, so the stack winds up looking like the following example.

Stack Level	Contents
Level 2:	(70,6)
Level 1:	(120,56)

Now BOX takes those two sets of coordinates and draws a box (the outline of a die) on PICT. Nothing is actually displayed yet. The location of the box is determined by the coordinates—and those have been determined by whether this is the first or second time *Rollnum* has been called. The effect for two calls to *Rollnum* is to draw both the left and right dice on PICT.



Left die (First run of Rollnum)



Right die (Second run of Rollnum)

One more thing remains to be done. *Die* has consumed the x- and y-coordinates passed by *Rollnum*. But there are other subprograms out there waiting to use these coordinates, too. So we execute x y within the defining procedure to place those quantities on the stack before exiting.

Ranum—Producing a Random Integer

Ranum, the next program called by the ROLL menu choice, produces a random number. It removes nothing from the stack, and leaves an integer in the range from 1 through 6.

Arguments	Results
3:	3: *starting x-coordinate*
2: *starting x-coordinate*	2: *starting y-coordinate*
1: *starting y-coordinate*	1: random integer (1-6)

Program Instructions	Comments
«	
RAND	Generates random number.
→ r	Makes number a local variable.
«	Begins defining procedure for local variable *r*.
'6*r+1' EVAL	Scales the number to the range 1-6.
IP	Uses only the integer portion.
»	
»	

To save the *Ranum* program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Ranum STO	Stores the program.

This subprogram begins with RAND, which generates a random number that's between zero and one. Then the subprogram converts the random number to the local variable *r*.

The defining procedure for *r* is a program; it must begin right after the → r statement.

Now think about what values are needed to simulate a single die. The die has one spot on one surface, two on another, and so on. The greatest number of spots on one surface is six.

This knowledge tells us that to simulate the throw of one die, we need a random number in the range from one through six. The algebraic object '6*r+1' scales the number to fit the desired range.

After scaling, the random number is a decimal; it could be 3.1567, or it could be as low as 1.0001 ... , or as high as 6.9999.... We don't need the decimal part of the number, though, so we use the IP command to return the integer portion (throwing away the decimal part). Thus 3.1567 becomes simply 3.

Number—Selecting a Drawing

Number, the next subprogram called, actually draws dots on the die in PICT. The number of dots is determined by the random integer passed from *Ranum*; and the location of the dots (left die or right die) depends on the x- and y-coordinate received by *Number*. This subprogram, then, removes a random number and two coordinates from the stack. It returns nothing to the stack.

Arguments	Results
3: starting x-coordinate	3:
2: starting y-coordinate	2:
1: random integer (1-6)	1:

Program Instructions	Comments
`«`	
`→ n`	Makes the random integer a local variable, *n*.
`«`	Begins defining procedure.
`CASE n 1 ==`	If $n = 1$
`THEN D1`	then draws one dot.
`END`	
`n 2 ==`	If $n = 2$
`THEN D2`	then draws two dots.
`END`	
`n 3 ==`	If $n = 3$
`THEN D1 D2`	then draws one dot and two dots.
`END`	
`n 4 ==`	If $n = 4$
`THEN D4`	then draws four dots.
`END`	
`n 5 ==`	If $n = 5$
`THEN D1 D4`	then draws one dot and four dots.
`END`	
`n 6 ==`	If $n = 6$
`THEN D2 D4`	then draws two dots and four dots.
`END`	
`END`	
`»`	Ends defining procedure.
`DROP2`	Gets rid of starting x- and y-coordinates.
`»`	

Keystrokes	Comments
ENTER	Puts program on the stack.
' Number STO	Stores the program as *Number*.

Number converts the random integer to a local variable called *n*. Within the defining procedure, *n* is used in a CASE ... END structure. Between the CASE and final END statements, *n* is compared to 1, to 2, to 3, and so on. When the test is true, everything between THEN and its END is executed—and these are calls to subprograms *D1*, *D2*, and so on, to draw the requisite number of dots on PICT.

Say the random number passed from *Ranum* is three. Thus, *n* is three. The case statement `n 3 ==` means "Does 3 equal *n*?" Since the answer is yes, `THEN D1 D2 END` is executed. *D1* draws one dot on the current die, while *D2* draws two dots—for a total of three.

At the beginning of *Number*, the starting x- and y-coordinates are in levels 1 and 2 of the stack. We'll show how *D1*, *D2*, and *D4* use these coordinates to draw dots in a moment. For now you should know that although they're going to use the coordinates, *D1*, *D2*, and *D4* have no effect on the stack. So we need to execute a `DROP2` command upon exit, to clear the two coordinates from the stack. This comes under the heading of "good housekeeping."

We could have had a separate drawing routine for each number of dots needed: *D1* for one dot, *D2* for two dots, and so on—right up to *D6*. But we don't need that many routines. In fact, we need only *D1*, *D2*, and *D4*. To generate three dots, we draw one dot with *D1*, then two dots with *D2*. To draw five dots, we call *D1* followed by *D4*, and so on.

Now let's see how and where these dots are actually drawn.

D1—Placing a Dot in the Center

D1 calls *Dot* to draw a single dot in the exact center of the current (left or right) die. It requires the starting x- and y-coordinates to be on the stack, using them to calculate where on the graphics area to place the dot. In case more dots will be drawn, *D1* returns the starting x- and y-coordinates, so it has no overall effect on the stack.

Arguments	Results
2: starting x-coordinate	2: starting x-coordinate
1: starting y-coordinate	1: starting y-coordinate

Program Instructions	Comments
«	
→ x y	Creates local variables x and *y*.
«	Begins defining procedure for local variables.
x 25 +	Adds 25 to x-coordinate.
y 25 +	Adds 25 to y-coordinate.
Dot	Calls *Dot* to draw the dot.
x y	Puts original x- and y-coordinate back on the stack.
»	Ends defining procedure.
»	

To save the program:

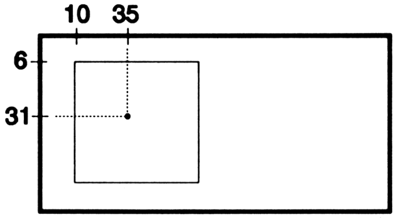
Keystrokes	Comments
[ENTER]	
['] D1 [STO]	Stores the program.

D1 begins by using the starting x-coordinate (either 10 or 70) and the starting y-coordinate (6) to create local variables *x* and *y*. Within the defining procedure, 25 is added to *x* and 25 to *y*, producing a pair of coordinates that represent the exact center of the current die. Then *Dot* is called to draw a dot there.

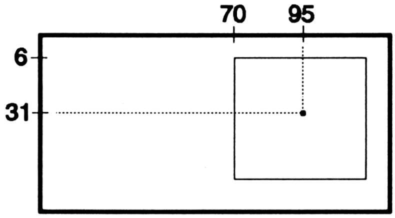
If this happens to be the left die, remember, its outline that was drawn on PICT by *Die* extends from an upper left corner at (10,6) to a lower right corner at (60,56). *D1* uses an *x* of 10 and a *y* of 6, adds

25 to each, and puts the coordinates 35 and 31 on the stack. *Dot* is then called, and uses these coordinates to draw a dot at that location.

If this is the right die, *x* and *y* are 70 and 6, respectively. *D1* adds 25 to each, and calls *Dot* to draw a dot at (95,31)—which is the exact center of the right die.



One dot on the left die



One dot on the right die

D2—Placing Two Dots

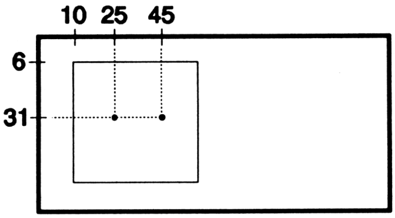
D2 is similar to *D1*, except that it draws two dots on the current die. Like *D1*, this subprogram expects a starting x- and y-coordinate to be on the stack, and it returns these coordinates to the stack upon exit.

Arguments	Results
2: starting x-coordinate	2: starting x-coordinate
1: starting y-coordinate	1: starting y-coordinate

Program Instructions	Comments
`«`	
`→ x y`	Creates local variables *x* and *y*.
`  «`	Begins defining procedure for variables.
`  x 15 +`	Adds 15 to x-coordinate.
`  y 25 +`	Adds 25 to y-coordinate.
`  Dot`	Draws a dot at the new coordinates.
`  x 35 +`	Adds 35 to x-coordinate.
`  y 25 +`	Adds 25 to y-coordinate.
`  Dot`	Draws a dot here, too.
`  x y`	Returns the original *x* and *y* to the stack.
`  »`	Ends defining procedure.
`»`	

Keystrokes	Comments
ENTER	Puts program on the stack.
' D2 STO	Stores the *D2* program.

D2 draws a pair of dots, one on each side of the center of the current die.



Two dots on the left die

D4—Placing Four Dots

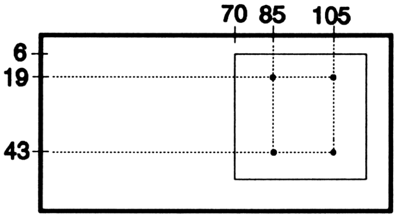
D4 calls *Dot* four times, to draw four dots on the current die.

Arguments	Results
2: starting x-coordinate	2: starting x-coordinate
1: starting y-coordinate	1: starting y-coordinate

Program Instructions	Comments
«	
→ x y	Creates local variables *x* and *y*.
«	Begins defining procedure.
x 15 +	Adds 15 to *x*.
y 13 +	Adds 13 to *y*.
Dot	Places a dot at the new coordinates.
x 35 +	Adds 35 to *x*.
y 13 +	Adds 13 to *y*.
Dot	Draws a dot there, too.
x 15 +	Adds 15 to *x*.
y 37 +	and 37 to *y*.
Dot	Draws another dot.
x 35 +	Adds 35 to *x*.
y 37 +	Adds 37 to *y*.
Dot	Draws the fourth dot.
x y	Puts *x* and *y* back on the stack.
»	Ends defining procedure.
»	

Keystrokes	Comments
ENTER	
' D4 STO	Stores the program *D4*.

D4 adds four dots to the current die. The dots are left and right of the die's center and above and below it, too.



Four dots on the right die

None of the dots drawn by *D1*, *D2*, or *D4* overlaps another dot, and no dots are erased from PICT yet. Thus, these three subprograms, *D1*, *D2* and *D4*, let you draw any combination of one to six dots on either die.

Dot—Using ARC to Draw

The *Dot* subprogram actually does the drawing of a single dot on PICT, at the coordinates passed from *D1*, *D2*, or *D4*. It takes two values from the stack and consumes them, leaving nothing on the stack when finished.

Arguments	Results
2: x-coordinate	2:
1: y-coordinate	1:

Program Instructions	**Comments**
«	
R→C	Converts starting coordinates to complex number.
2	Radius for arc.
0	Starting angle for arc.
360	Ending angle for arc.
ARC	Draws the arc (a circle) from 0 to 360 degrees.
»	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Dot STO	Stores the program.

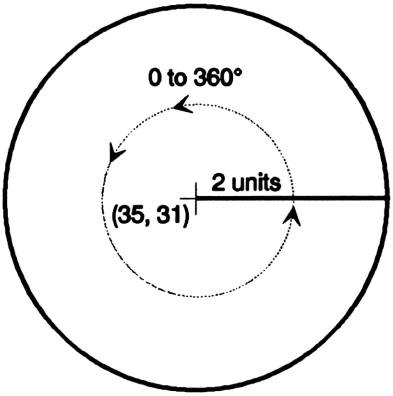
Dot uses the ARC command to draw a tiny circle (a dot) in PICT. ARC draws an arc centered at coordinates (x,y) in level 4, with a given radius (level 3), counterclockwise from one angle (level 2) to another (level 1).

When passed by *D1*, *D2*, or *D4*, the x- and y-coordinates are separate: one is in level 2 and one is in level 1. The first command in *Dot* is R→C, which converts these numbers to (x,y) form. The rest of the values entered by *Dot* prepare for the execution of ARC.

For example, if *Dot* has just been called by *D1*, and this is the left die, the coordinates passed to *Dot* are 35 (in level 2) and 31 (in level 1). Just before the execution of ARC, the stack is set up with all the necessary quantities.

Stack Level	Contents
Level 4:	(35,31)
Level 3:	2
Level 2:	0
Level 1:	360

When ARC is executed, it draws an arc on PICT, with its center at coordinates (35,31). The radius is 2 units (rather tiny), and the arc goes from 0 degrees to 360 degrees—in other words, it's a circle.



At this point, the ROLL choice from the *Dice* menu has executed the sequence *Rollnum Ranum Number*, followed by another *Rollnum Ranum Number*. Two dice have been drawn on PICT, and a random number of dots placed in each die. But all the user sees is `Rolling....` Now we need to save a copy of PICT, with its dice, as the current roll. If this is the first roll, we'll save it twice.

Saveit—Preserving PICT

The *Saveit* subprogram stores a copy of PICT in a variable called *Last*. If this is the first roll, *Saveit* also puts a copy in *First*. The *Saveit* subprogram doesn't affect the stack.

Program Instructions	Comments
`«`	
`IF 22 FC? THEN`	If user flag 22 is clear, then:
`  PICT`	Places the PICT name on the stack.
`  RCL`	Recalls the contents of PICT (a graphics object).
`  'First'`	Places the name `'First'` on the stack.
`  STO`	Stores the contents of PICT in *First*.
`END`	Ends IF clause.
`PICT`	
`RCL`	Recalls PICT's contents again.
`'Last'`	
`STO`	Stores contents of PICT in *Last*.
`»`	

Keystrokes	Comments
ENTER	
' Saveit STO	Stores the program.

Saveit begins by testing user flag 22. This is the flag we've designated to keep track of the roll. If flag 22 is clear, it means this is the first roll, so we want to preserve the dice for later examination. We could simply save the number (1 or 3 or 6) in a variable. But we'll get a little fancy and save the entire graphics screen.

The `PICT` command puts the name of the HP 48's graphics area on the stack. This is just like a variable name. Then `RCL` recalls the contents of PICT—that is, a graphics object containing the dice that have been drawn (plus anything else that happens to be drawn on PICT). The stack shows the size of PICT:

```
1: Graphic 131 × 64
```

Now we store this graphics object in a variable, just as we'd store a number or any other object. It's stored in a variable called *First*.

Similarly the PICT graphics object is also stored in *Last*. This lets you keep track of your latest roll. Unlike *First*, the contents of *Last* are updated with every roll of the dice.

Showit—Displaying PICT

Showit displays the dice where you can see them. It brings PICT into the display and waits for you to press ATTN before continuing. *Showit* has no overall effect on the stack.

Program Instructions	**Comments**
`«`	
`{ }`	Empty list for PVIEW.
`PVIEW`	Shows PICT centered in display.
`7 FREEZE`	Freezes the display until the next key press.
`»`	

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Showit STO	Stores the program.

Showit places an empty list on the stack and executes PVIEW. With this particular argument, PVIEW shows PICT centered in the HP 48's display, with scrolling mode activated. The current display is visible until you press ATTN.

The `7 FREEZE` command freezes the display. You don't really need this when *Showit* is called as a result of pressing ROLL. However, *Showit* is also called by temporary menu keys FIRST and LAST, to display the contents of *First* and *Last*, respectively. And in this case, you need `7 FREEZE` to prevent a return to the stack display after *Showit*.

Getfirst—Replacing PICT

Most of the work in the *Dice* program happens when you press the `ROLL` menu key. However there are other menu choices, too. One of these choices is `FIRST`, which runs the subprograms *Getfirst* and *Showit* to redisplay your first roll.

Getfirst recalls the contents of the *First* variable into the stack. (The contents are a graphics object that is exactly the same size as PICT.) Then it stores this graphics object in PICT, ready for later display by *Showit*. *Getfirst* has no overall effect on the stack.

Program Instructions	**Comments**
`«`	
`First`	Places the contents of *First* (a graphics object) on the stack.
`PICT`	Places the name PICT on the stack.
`STO`	Stores the contents of *First* in PICT.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	
' Getfirst STO	Stores the program.

In this subprogram, `First` is executed first. *First*, remember, is a variable that contains a graphics object: the PICT display after your first roll. After `First` is executed, the PICT-sized graphics object is in level 1.

Stack Level	Contents
Level 1:	Graphic 131 × 64

Now *Getfirst* puts the name PICT on the stack. This special reserved variable isn't executed; only the name is recalled.

Stack Level	Contents
Level 2:	Graphic 131 × 64
Level 1:	PICT

When the next command, STO, is executed, the object in level 2 is stored in the name that's in level 1. Voila! A copy of your first roll has overwritten PICT, and is now ready for display by *Showit*.

Getlast—Another Replacement for PICT

Getlast, which is called when you press the LAST menu key, is similar to *Getfirst*; except that *Getlast* puts the contents of the variable *Last* (a graphics object containing your latest roll) into PICT. Like *Getfirst*, the *Getlast* subprogram has no effect on the stack.

Program Instructions	Comments
«	
Last	Recalls contents of *Last* (a graphics object).
PICT	Places the name PICT or the stack.
STO	Stores the contents of *Last* in PICT.
»	

Keystrokes	Comments
ENTER	
' Getlast STO	Stores the program.

When you press LAST, *Getlast* is executed, followed by *Showit*. The result? You see a redisplay of your latest roll of the dice.

Exit—Getting the VAR Menu Back

Pressing the EXIT menu key calls *Exit*. This subprogram simply brings back the original VAR menu; it has no overall effect on the stack.

Program Instructions	Comments
«	
2 MENU	Displays the VAR menu again.
»	

To save this program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Exit STO	Stores the *Exit* program.

All that *Exit* really does is provide some visual feedback that you are done running *Dice*. *Exit* places 2 in the stack, then executes MENU—which simply replaces *Dice*'s temporary menu with the VAR menu. *Exit* doesn't erase PICT or *Last* or *First*, so these all maintain their contents until you run *Dice* again.

Running the Dice Program

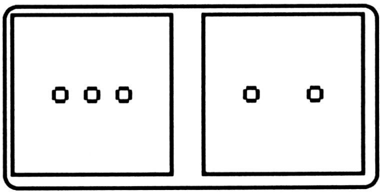
At last, it's time to run the *Dice* program. The following is one simple run; your own displays, of course, may be different.

When you press DICE, you see the following menu display:

ROLL FIRST LAST NEW EXIT

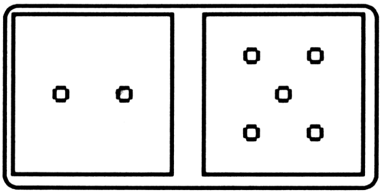
Program Prompt or Display	Your Action
ROLLING DICE	
ROLL-To roll dice	
FIRST-Review 1st roll	
LAST-Review last roll	
NEW-New game	
EXIT-To quit	
[ATTN] after dice	ROLL
Rolling...	

Your first roll is a 5:



Program Prompt or Display	Your Action
	ATTN
ROLLING DICE	
ROLL-To roll dice	
FIRST-Review 1st roll	
LAST-Review last roll	
NEW-New game	
EXIT-To quit	
[ATTN] after dice	ROLL
Rolling...	

Your second roll is a 7:



To see the first roll again, press FIRST; or to see the latest roll, press LAST . You can also start a new game (with NEW) or exit from the program (with EXIT).

9

Health and General Interest

You probably don't think of a scientific calculator like the HP 48 as being much good outside the laboratory—for instance, in the realm of health, or travel, or other areas. But with the right kind of programs, this amazing little machine can be an invaluable aid in your quest for fitness, enlightenment, or just plain fun. With its small size, the HP 48 is the perfect companion wherever you go.

Reflex Tester

How fast are your reflexes? The program shown below will tell you. Each time you see the word "GO" flash on the calculator screen, you press the ENTER key once as quickly as you can; the HP 48 automatically measures how long it took for you to respond. For greater accuracy, you go through five tests, then see your average response time on the screen.

This program shows off several features of the HP 48, but the most important is the use of the random number function, RAND, to make sure you can't anticipate the prompt. You'll find a detailed explanation of how RAND is used to produce random numbers, and how those numbers are scaled, in the random number generator in chapter 8.

The Reflex Directory

To create a directory for the *Reflex* program and subprograms, then get into that directory:

' REFLEX
↰ MEMORY CRDIR
VAR
REFLE

The REFLEX directory will hold all programs and subprograms for the reflex tester. Any objects you create, including programs and statistical data such as ΣDAT, will now be placed in this directory.

The Main Reflex Program

The main program, *Reflex*, like most main programs in this book, is *almost* nothing but calls to subprograms. Since there will be five trials, though, one portion of the program is executed five times, by means of a START ... NEXT loop. Although a number of the subprograms pass values to one another, your HP 48's stack is the same when *Reflex* ends as it was when the program began.

```
Program Instructions           Comments

«

Init                           Initialization routine.

Title                          Displays user instructions.

1 5 START                      Begins START ... NEXT loop.

  Randstart                    Waits for random length of time.

  Starttime                    Prompts for input, begins timing.

  Stoptime                     Stops timing when key is pressed.

  Computime                    Computes reflex (start-to-stop) time.

  Storetime                    Stores start-to-stop time.

NEXT                           Executes loop five times.

Showresult                     Displays average reflex time.

»
```

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Reflex STO	Stores the program.

Reflex runs from beginning to end, with no need for menu choices. By using a START ... NEXT loop (instead of FOR ... NEXT), you don't need to create a local variable within the loop.

Init

The *Init* subprogram sets the HP 48 to display numbers to four decimal places, and it clears the statistical data from the current directory. *Init* leaves the stack unchanged.

Program Instructions	Comments
«	
4 FIX	Sets display to four decimal places.
CLΣ	Clears reserved variable ΣDAT.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Init STO	Stores the program.

Although the display is set to four decimal places by the 4 FIX command, your calculator always uses the full value of any number internally.

The STAT functions are going to be used later to keep track of reflex times, so we use CLΣ to make sure no data remains left over from previous runs of the program. CLΣ clears only the ΣDAT variable in the *current* directory; so if you use good structured programming practice and keep all *Reflex* programs in their own unique directory (perhaps called REFLEX), you'll have no worries about purging statistical data in other directories.

Title—Waiting for Any Key

The *Title* subprogram is where the title and user instructions appear. You see a full screen explaining what comes next. The title is displayed until you press any key (except ATTN); then the test begins. *Title* doesn't affect the stack overall.

```
Program Instructions              Comments

«

"REFLEX TESTER
This tests your                   Message displayed by DISP.
response time. When               Be sure to place newline
you see GO, press                 characters where the lines end
[ENTER]. There are five           in this string.
tests. Press any key
now to go for it."

1 DISP                            Displays message beginning on
                                  LCD row 1.

7 FREEZE                          Freezes entire display until
                                  next key press.

0 WAIT                            Waits for the press of any key.

DROP                              Throws away key address
                                  returned by WAIT.

»
```

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Title STO	Stores the program.

Title begins by using DISP to display a long message of user instructions. The DISP command takes as its arguments a string from level 2 of the stack, and a number from level 1. The string is what is displayed—exactly as you key it in. Use newline characters (↱ ↵) to end the lines as they're shown, or you'll wind up with a long string that extends past the edge of the display screen. The number that's in stack level 1 when DISP is executed determines where the first character of the string will be placed. `4 DISP` would start the string at the left side of row 4 (the middle row) of the calculator's liquid crystal display. Since we've specified 1 DISP, the message begins on row 1. Thus, we'll see all seven rows of the message in the display.

FREEZE locks the display until the next key press. Without FREEZE, you'd see the title for a fleeting moment, then you'd see the stack display again as the HP 48 updated its display. `7 FREEZE` freezes the entire display—including the status area, stack area (that is, the central main area), and the menu area.

Finally, we want to give time to read this display, so we add a `0 WAIT` command. WAIT pauses execution for a specified time: 1 `WAIT` for one second, `2 WAIT` for two seconds, and so on. With an argument of 0, though, WAIT suspends execution until the next key press. Thus, `0 WAIT` holds up the execution of *Reflex*, displaying the message until you press any key. (The keystroke doesn't do anything—in effect, it's "swallowed up.")

For our purposes here, `0 WAIT` has an unfortunate side effect—it returns the address of the key that was pressed. So we execute a DROP command to rid the stack of that number.

Randstart—Waiting a Random Time

The START ... NEXT loop is executed five times. Each time through the loop, *Randstart* pauses execution for a period from 0 to 5 seconds. *Randstart* has no overall effect on the stack—unless you try to cheat, when it leaves in level 1 the telltale address of the key you pressed.

Program Instructions	Comments
«	
DO	Begins DO ... UNTIL loop.
"Get ready..."	String for DISP.
CLLCD	Clears HP 48's LCD.
4 DISP	Displays string on LCD row 4.
7 FREEZE	Freezes display until updated.
RAND	Generates random number between 0 and 1.
6 *	Scales number to the range 0 to 6.
WAIT	Pauses execution for 0 to 6 seconds.
UNTIL KEY 0 ==	
END	
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Randstart STO	Stores the program.

Randstart is a DO ... UNTIL loop that *should* be executed only once. (More about the loop in a minute.) *Randstart* begins by placing a short message string, "Get ready...", in level 1 of the stack. The next command CLLCD, clears the display, but has no effect on the stack. Thus, when 4 DISP is executed, it displays the message on the middle row (row 4) of the display. 7 FREEZE freezes the entire display until it's updated—in this case by another DISP command in the succeeding program, *Startime*.

WAIT pauses execution for the number of seconds specified in level 1 of the stack. Thus, 3 WAIT pauses execution for three seconds, 5.2 WAIT for five and two-tenths seconds, and so on. We don't want the waiting period between "Get ready..." and "GO" to be predictable, so we use a random number as the argument for WAIT.

RAND puts a random number on the stack; the number is somewhere between 0 and 1. It could be, say 0.00001 or it might be 0.99999. This number is scaled—it's multiplied by 6, which gives a random number between 0 and 6. (It's actually from, say, 0.00001 second to 5.9999 seconds.)

WAIT uses this random number as its argument, causing the program to pause and display the message Get ready... for a random length of time before continuing to the next subprogram.

The next subprogram, *Startime*, is going to start timing, then display the word GO and use 0 WAIT to time how long you take to respond. But what if (you sly devil) you press a key while Get ready... is on the screen? That 0 WAIT waits for a key press; and a key press is recorded by the HP 48 even if you press a key while you're supposed to be waiting in *Randstart*. Why, you could even cheat by pressing a key in the middle of *Randstart* so that when *Startime* is executed, a key has already been pressed. *Startime*'s 0 WAIT instruction would see the key press, and think you pressed the key *after* seeing GO, not while Get ready... was displayed. The result: your reflexes would seem faster than is humanly possible.

That's the purpose for the DO ... UNTIL loop. If you do as you're supposed to and wait until you see GO to press a key, the loop clause is executed just once. Within the test clause (after UNTIL), KEY returns a 0 if no key was pressed. Then the test 0 == asks "Is 0 equal to what's in level 1 of the stack?" Since the answer is "yes," the UNTIL test is true, and execution falls out of the loop and continues.

If you press a key while *Randstart*'s loop is being executed, though, KEY returns the key's address to level 2, and the number 1 to level 1. Now the test 0 == is false, so the loop is executed again, and the key's address left on the stack. A good way to think of it is that the loop is executed UNTIL no key is pressed.

Startime—"Starting" the Timer

Startime "starts the time" by placing the value of TICKS in the stack. It flashes the word GO and waits for you to press any key. *Startime* expects nothing from the stack, and leaves a binary integer there; the binary integer is the current number of clock ticks. *Startime* also leaves the key address of the key you pressed.

Arguments	Results
2:	2: beginning time
1:	1: key address

Program Instructions	Comments
«	
CLLCD	Clears the HP 48 liquid crystal display.
TICKS	Places current value of clock on stack.
"GO"	String for use by DISP.
4 DISP	Displays GO on row 4 of 7-row LCD.
7 FREEZE	Freezes entire display.
0 WAIT	Waits for press of any key before continuing.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Startime STO	Stores the program.

CLLCD clears the previous message from the calculator's liquid crystal display. Then TICKS places the current number of clock ticks on the stack. The HP 48's clock, you see, never goes backward—it keeps adding to the accumulated number of ticks at the rate of 8192 every second.

In a sense then, we aren't "starting" anything here—we're merely recording the value of clock ticks when you first see the word GO.

To display the word GO, a string containing that word is placed on the stack, after the number of tick marks. Then 4 DISP consumes the string, displaying it on row 4 (the middle row) of the LCD. 7 FREEZE freezes the entire display so it's not updated yet, and 0 WAIT pauses execution until a key—any key—is pressed.

Now comes the actual reflex test. As soon as you see GO, you press a key. *Startime* ends, leaving the clock ticks on the stack, and execution continues to the next program, *Stoptime*. In addition, 0 WAIT leaves the key address on the stack, too.

Stoptime—Determining Time Between Events

Stoptime takes the number of ticks from the stack, records the new number of ticks, and subtracts them. This gives the elapsed time (in clock ticks) between the start of timing (when you see the GO prompt) and the end of timing (when your speedy fingers finally make their response). It takes the key address and the beginning time from the stack, and leaves the elapsed time there.

Arguments	Results
2: beginning time	2:
1: key address	1: elapsed time

Program Instructions	Comments
«	
TICKS	Puts ending time on the stack.
SWAP	Puts key address in level 1, ending time in level 2.
DROP	Throws away key address.
SWAP	Puts ending time in level 2, beginning time in level 1.
-	Subtracts.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Stoptime STO	Stores the program.

The first thing we do is use TICKS to place the ending time on the stack. We wait to throw away the key address from the previous subprogram until *after* recording the ending time.

To throw away the key address, we SWAP the ending time and the key address, leaving the stack as shown here:

Stack Level	Contents
Level 3:	beginning time
Level 2:	ending time
Level 1:	key address

DROP gets rid of the key address then SWAP puts ending time *above* beginning time. Finally, the – arithmetic operator subtracts, leaving elapsed time on the stack.

Computime—Converting Clock Ticks to Real Time

The *Computime* subprogram converts elapsed time from clock ticks to seconds. It also subtracts a calibration factor to account for the time taken by the calculator for internal processing of a key press. *Computime* takes a binary integer from the stack and returns a number representing time in seconds.

Arguments	Results
1: elapsed time (binary)	1: elapsed time (hh.mmss)

Program Instructions	Comments
`«`	
`B→R`	Converts binary to real number.
`8192 /`	Divides by ticks/second.
`Calfactor -`	Subtracts calibration factor.
`»`	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Computime STO	Stores the program.

The number passed from *Stoptime* is the elapsed time, but it's not really in a form we can use. It's a binary integer, something like `#3278d`. So the first thing we do is convert it to a real number using the `B→R` command.

The resulting number, while more useful, is still in clock ticks. There are 8192 clock ticks for every second, so the number 3278 represents just 0.4 second. Thus, we divide by 8192 to give us the time in seconds.

That would be our answer, if we weren't careful. It turns out, however, that the HP 48's internal processing time—primarily, the time it takes between pressing a key and recording the keystroke—is fairly substantial (especially when weighed against your own quicksilver-like reflexes). So we subtract a calibration factor, which is held in the variable *Calfactor*. After subtracting *Calfactor*, the resulting elapsed time, in seconds, is left on the stack.

Calfactor—An Experimentally Determined Value

Calfactor, the number you subtract to get the true elapsed time for your response, will vary somewhat from calculator to calculator. For now, just use the value I obtained for my calculator, which is 0.3418 seconds. Use these keystrokes to store *Calfactor*:

.3418 ENTER
' Calfactor STO

You can determine the *Calfactor* time for your own HP 48 by using the *Cal1* program. You'll find *Cal1* a few pages down the road, at the end of this program description.

Storetime—Using the Statistical Matrix in a Program

The last subprogram called each time through *Reflex*'s START ... NEXT loop is *Storetime*. This subprogram takes the elapsed time from the stack and stores it in this directory's statistical matrix. It takes one number from the stack, and leaves nothing on the stack.

Arguments	Results
1: elapsed time	1:

Program Instructions	**Comments**
«	
Σ+	Stores this elapsed time in ΣDAT.
»	

Keystrokes	Comments
ENTER	
' Storetime STO	Stores the program *Storetime*.

You could, if you wanted, create a separate list or array to hold the five elapsed times. But why bother? Using Σ+ adds this new elapsed time to the statistical matrix, ΣDAT. You'll thus be able to use the statistics functions of the calculator to view a bar plot of your individual reflex times, or to do other kinds of analyses.

The reserved ΣDAT variable for each directory is unique to that directory. Even though you're creating and using ΣDAT here, any ΣDATs you may have in other directories remain pure and inviolate.

Storetime is the last subprogram called by the loop. Each time *Storetime* is called, it uses Σ+ to add the latest reflex time to the other times stored in the current statistical matrix. When five times have been stored, execution falls out of the START ... NEXT loop in *Reflex*, and *Showresult* is called.

Showresult—Displaying a Result in a String

Showresult the final subprogram called by *Reflex*, calculates the average reflex time, combines it with an explanatory string, and displays a detailed explanation of reflex time. *Showresult* uses the values stored in ΣDAT; it takes nothing from the stack, and leaves nothing there either.

Program Instructions	Comments
«	
"Your average reflex time in seconds was"	Beginning of display string.
MEAN	Calculates average reflex time.
→STR	Converts average to a string.
+	Combine the two strings.
CLLCD	Clears HP 48's display screen.
1 DISP	Displays string beginning on row 1.
7 FREEZE	
»	

To save this subprogram:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Showresult STO	Stores the program.

The subprogram *Showresult* begins by placing a long string on the stack. (Unless you want the sentence to run past the edges of the HP 48's display, be sure to add line breaks where they're shown in the listing.)

Next the subprogram uses MEAN to calculate the average reflex time. The MEAN command doesn't require anything to be on the stack. Instead, it uses information that's stored in the current statistical matrix, which in our case is ΣDAT.

At this point we have a partial sentence (a string) in stack level 2, and the average reflex time (a real number) in level 1. We want to combine these into a single string. So we use →STR to convert the real number to a string, then use + to add the two strings together. (The string from level 1—the reflex time—is appended to the end of the string from level 2.)

CLLCD clears the liquid crystal display of any information, and 1 DISP displays the string with its first character beginning at the left of row 1 of the HP 48's screen. 7 FREEZE freezes the entire display until the next key press, to let you see your results.

Running the Reflex Program

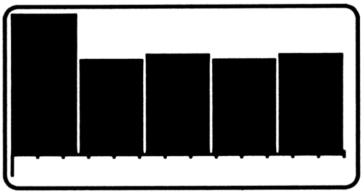
To run the *Reflex* program, just press REFLE, then follow the prompts.

Program Prompt or Display	**Your Action**
	REFLE
REFLEX TESTER This tests your response time. When you see GO, press ENTER. There are five tests. Press any key now to go for it.	
Get ready...	
GO	[ENTER]
Get ready...	
GO	[ENTER]

You answer the GO prompt five times. Then you see something like the following display:

```
Your average reflex
time in seconds
was 0.2784
```

Because the reflex times are saved in ΣDAT, you can use the HP 48's STAT keys ([↰] [STAT]) to view the result of each of the five tests, or to display a bar plot (BARPL) of the reflex times.



This bar plot shows your first trial was the slowest, while your second and fourth were the fastest.

Cal1—Determining the Calfactor

This program, called *Cal1*, isn't really part of the *Reflex* program. However, it returns a calibration value you can store in *Calfactor*.

Arguments	Results
1:	1: calibration factor

Program Instructions	Comments
«	
"Press a key now"	String for display by DISP.
CLLCD	Clears LCD.
1 DISP	Displays the string.
3 WAIT	Waits three seconds.
Startime	Begins timing.
Stoptime	Stops timing.
B→R	Converts result to real number.
8192 /	Converts to seconds.
»	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Cal1 STO	Stores the program.

The *Cal1* program actually ***encourages*** you to cheat. It prompts for you to press a key, and waits for three seconds. Then it calls *Startime* and *Stoptime*, just as they're called by the loop in *Reflex*.

There's one big difference between *Reflex* and *Cal1*, though. *Reflex*, remember, uses *Randstart* to tell you to "Get ready...", and to prevent your pressing a key until *Startime* is called. In *Cal1*, though, you ***can*** press a key during the 3-second WAIT period after the "Press a key now" message. That key press is remembered by the HP 48.

If you press a key during the three-second interval, it means that when *Startime* is called, ***a key has already been pressed***. The 0 WAIT command, which is waiting for a key press, immediately allows execution to proceed, with no pause between GO and the *Stoptime* call. The result is that only the amount of time consumed internally by *Startime* and *Stoptime* is recorded—with no molasses-slow user involved.

The remainder of *Cal1* is just a duplication of part of *Computime*, converting the time to seconds.

Press CAL1 to run *Cal1* and store the value in *Calfactor*:

Program Prompt or Display	**Your Action**
	CAL1
Press a key now	ENTER
GO	
.3418	

You may want to run *Cal1* several times, then use an average value, My values were always around .3416 to .3420. Interestingly, the calibration time is affected by the number of items on the stack, so execute ↰ CLR between runs.

Autogetem, Your Automatic Assistant

Consider the problem of a scientist who needs to perform a complicated series of tasks, or record a value (say, temperature), at specified intervals. Even if an ordinary alarm goes off, the harried researcher can lose valuable time trying to remember the exact keystrokes to execute, or the location to place the data.

The *Autogetem* program not only sets off an alarm, but it also executes a program that automatically prompts for data. Here, the prompt is for daily input of temperature, allowing you to record the temperature at the same time each day. However, you could have it do almost anything:

- Prompt for blood pressure three times a day.
- Awaken you and ask for your waking pulse rate.
- Determine which television show you're watching at a specified time.

Autogetem turns the HP 48 into an automatic data-gathering device. Think of the research possibilities: legions of *Autogetem*-equipped HP 48 users around the world, all of whom enter the same type of information at the same time.

No matter what you're doing, the *Autogetem* program interrupts to request its daily dose of data. If you happen to be editing another program at the time, the alarm returns you to the stack, then prompts for input. If the calculator is off, *Autogetem* turns it back on at the appointed hour. And what if, heaven forbid, you happen not to have your HP 48 close at hand when the alarm sounds? No problem—when you return and switch on the calculator, there's the *Autogetem* prompt waiting hopefully for your input.

As written, *Autogetem* has some other features, too. It can display the average value of all the data, or switch you to the calculator's statistics menu if you want to plot a bar chart or do further manipulations.

The Autogetem Directory

To create a directory for the Autogetem program and subprograms, then get into that directory:

['] AUTOGETEM
[⬅] [MEMORY] CRDIR
[VAR]
AUTO

Use the AUTOGETEM directory to hold all the programs and subprograms for this application.

The Main Autogetem Program

The main program calls three subprograms in order:

Program Instructions	Comments
«	
Init	Initializes the calculator.
Message	Displays a preliminary message.
Mainmenu	Displays main menu keys and instructions.
»	

The *Init* subprogram handles initialization chores, of course. *Message* is simply a preliminary message to users explaining the program. *Mainmenu* shows the main menu, allowing the user to choose among five options.

To save this three-line program as *Autogetem*:

Keystrokes	Comments
[ENTER]	
['] Autogetem [STO]	Stores the program as *Autogetem*.

Init—Ensuring an Audible Alarm

The initialization program, *Init*, takes nothing from the stack and leaves nothing on it.

Program Instructions	**Comments**
`«`	
`-40 SF`	Displays ticking clock at all times.
`-41 SF`	Sets for 24-hour format.
`-56 CF`	Enables sound for BEEP command.
`CLΣ`	Clears current statistical matrix.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Init STO	Stores the program.

Init prepares the calculator for subsequent operations by setting or clearing certain system flags. Flag −40 determines whether or not the ticking clock is shown in the display at all times; when it's set, you see the clock, and when it's cleared, you see the clock only when the TIME menu is selected. We won't be using the TIME menu; but it's a nice gesture to put the clock in the display along with the alarm. So we'll set flag −40.

Flag −41 determines the clock format: 12-hour (flag clear) or 24-hour (flag set). If you're keying in an alarm time using the keyboard and the TIME menu, you can use the 12/24 and A/PM keys. However, these particular functions aren't programmable, so we'll stick to 24-hour format and set flag −41.

We're going to sound an alarm to make sure our users do the right thing—at the right time. One way to sound an audible alarm is simply to make use of the calculator's alarm beep. However, we're going to specify our own alarm using the BEEP command, so we set flag −56; this enables sound for the BEEP.

The last thing accomplished by *Init* is to clear the current statistical matrix—that is, to clear any statistical data that happens to be in this directory. When you do statistical functions, a reserved variable called ΣDAT is created. If you've placed all the *Autogetem* programs in their own separate directory (and you should), the ΣDAT for this directory is different from, and unaffected by, statistical operations in other directories. Thus, CLΣ purges ΣDAT data from this directory, but doesn't affect any ΣDATs that may be in other parts of the calculator.

Message—An Introductory Display

The *Message* program also expects nothing from the stack and leaves nothing on when it's through. *Message* displays a general program title; the user reads the title, then presses any key to continue.

Program Instructions	**Comments**
`«`	

`"AUTOGETEM`
`This program`
`automatically prompts`
`daily for input.`
`Press any key to`
`continue."`
	Message displayed by DISP.

`CLLCD`	Clears the liquid crystal display.
`1 DISP`	Shows message, beginning on row 1 of the display.
`7 FREEZE`	Freezes display; doesn't update until a key press.
`0 WAIT`	Waits for the press of any key.
`DROP`	Removes the results of `0 WAIT` from the stack.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Message STO	Stores the program.

The *Message* subprogram begins by placing a long string on the stack. If you want your message to look like the one shown here, make sure you end your lines by pressing ↱ ↵ at the end of each line.

CLLCD (*clear LCD*) then erases everything from the display, including menu labels, stack display, and the status area. The message string is still on the stack, though, so executing `1 DISP` places the message into the LCD, with the first line of the message on the top row (row 1) of the 7-row display.

If all we did was display the message, you'd have to have pretty good eyes to read it—it would flit past in a heartbeat, to be replaced by the stack display as the program continued. So we use `7 FREEZE` to immobilize the display until the next key press. We use 7 as the argument because we don't want to update anything yet—not the status area, not the stack display, not even the menu labels. With an argument of 7, FREEZE freezes the entire display.

Finally, we want the user to be able to linger over every pearl-like word of our message, so we place the command `0 WAIT` after FREEZE. With an argument of 0, WAIT causes the program to be suspended, again until the next key press. Thus, the next time the user presses a key, the display is updated *and* execution continues.

The command `0 WAIT` has a side effect, though—it places a number on the stack. The number represents the keyboard address of the key that was pressed: 51 (row 5, key 1) for ENTER, 26 for NXT, and so on. Here, we don't need the key address, so we execute a DROP command to rid the stack of this value before exiting the subprogram.

Mainmenu—A Menu Without a Loop

The *Mainmenu* program displays the main user menu of choices for *Autogetem*. It shows a temporary menu of key labels at the bottom of the display, and it also displays user instructions for those keys. *Mainmenu* takes nothing from the stack, and it leaves nothing for the next subprogram.

Program Instructions	**Comments**
`«`	
`Menulist`	Calls list that is argument for TMENU.
`TMENU`	Uses list to display temporary menu.

`"MAIN MENU:`
`SETUP-Daily time`
`LIST-Dates, temps`
`AVGT-Average temp`
`RESET-All temps`
`SHOW-Stat menu"`
	Text string for display by DISP.

`CLLCD`	Clears calculator display.
`1 DISP`	Displays long text string beginning on row 1.
`3 FREEZE`	Freezes status area and main area of display.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	
' Mainmenu STO	Stores the program.

The program begins by calling *Menulist*, which is actually a list containing all the information needed by the TMENU command. Then the program executes TMENU (*temporary menu*). TMENU uses the list to create the menu labels you see across the bottom of the display, and to determine what happens when you press any of the menu keys. You see the following labels:

SETUP LIST AVGT RESET SHOW

Naturally, the information in *Menulist* must be in a specific format for TMENU to work correctly. (We'll discuss the actual contents of *Menulist* in a moment.)

After executing TMENU to create the menu labels, *Mainmenu* then goes about displaying some instructions on how to use those labels. First, a long string is placed on the stack. When you enter the string, be sure to put a line break (↱ ↵) at the end of every line.

With the string on the stack, *Mainmenu* now performs a CLLCD (*clear LCD*) to erase anything from the display. Next, 1 DISP is executed; this puts the string from level 1 of the stack into the display where you can read it. The display begins on row 1 of the calculator's viewing area.

The last command in this subprogram is 3 FREEZE, which freezes a portion of the display until the next key press. Using 3 as the argument for FREEZE means that the display's status area and stack display (that is, the central main part of the display) are frozen, but the display of menu keys is not. This is necessary because TMENU doesn't actually make the menu "switch" until its program ends; which means that using 7 FREEZE here would freeze the display, including the *old* set of menu labels.

Notice that we don't need to provide a loop or WAIT command at the end of *Mainmenu*. That's because the main part of the program is completely finished executing at this point. From now on, you make your choices from the menu key display—and your list of options is from the *Menulist* displayed by TMENU.

If you're entering and testing this subprogram by itself, remember that you'll need to supply the *Menulist*, too. Also, *Mainmenu* leaves the labels for the temporary menu in the display when it's finished running. To get back to labels for your variables, you can press VAR.

Menulist—The TMENU List of Lists

Menulist is the argument for the TMENU command found in *Mainmenu*. *Menulist* isn't a subprogram; it's really a "list of lists," with each inner list containing the information for one menu key.

Program Instructions	**Comments**
`{`	Begins *Menulist*.
`  { "Setup" Controltime }`	First menu label and object.
`  { "List" Templist }`	Second menu label and object.
`  { "Avgt" Avgtemp }`	Third menu label and object.
`  Reset`	Combined label/object for fourth menu key.
`  Show`	Combined label/object for fifth menu key.
`}`	End of list.

Saving the list is just like saving a program:

Keystrokes	**Comments**
ENTER	Puts list on the stack.
' Menulist STO	Stores list as *Menulist*.

Menulist, remember, is used by the TMENU command to build a temporary menu. TMENU normally expects as its argument a list in this form:

`{ {"label1"` *object1* `} {"label2"` *object2* `}...}`

In the case of *Menulist*, we have defined five labels; these are the words that appear on the menu keys at the bottom of the display. Each label is followed by an object, which happens to be a subprogram. If you press the key marked SETUP, it runs the program named *Controltime*. If you press AVGT, the *Avgtemp* program is run.

To save space, you can combine the label and the object in the list, as in the labels for the fourth and fifth menu keys. For instance, “Reset,” the word assigned to the fourth menu key, is actually a program name. So inserting it by itself into the TMENU argument list, without its own set of separate curly brackets, causes this word to do double duty. You see the word RESET as the label for the fourth menu key. And if you press this key, the subprogram *Reset* is run. Similarly, “Show” is both a label and an object for the fifth menu key.

Now let’s take a closer look at the subprograms summoned by those different menu keys.

Controltime—Setting Up the Automatic Alarm

If you press the SETUP key, the *Controltime* program is run. *Controltime* prompts for the daily alarm time, and uses the STOALARM command to place the alarm information in the calculator’s catalog of system alarms. *Controltime* expects nothing from the stack, and leaves nothing on it.

Program Instructions	**Comments**
`«`	
`{ }`	Begins list for use by STOALARM.
`DATE`	Gets today’s date.
`+`	Adds date to list.

`"TIME: Input daily`
`time as HH.MMSS.`
`Use 24-hour format."`
	Message displayed by INPUT command.

`""`	Ensures nothing is on command line for INPUT.
`INPUT`	Displays message, waits for input of time.
`OBJ→`	Converts time to a number.
`+`	Adds time to list.

Program Instructions	**Comments**
'Getemp'	Name of the program that will be executed when the alarm sounds.
+	Adds the *name* 'Getemp' to the list.
707788800	Repeat interval (24 hours) in clock ticks.
+	Adds repeat interval to the list.
STOALARM	Uses list to add alarm to the HP 48's alarm list.
DROP	Removes alarm index from stack.
»	

Keystrokes	**Comments**
ENTER	
' Controltime STO	Stores the program.

Controltime uses the STOALARM command to set up a daily alarm at the time you choose. Before proceeding, let's review how STOALARM works.

STOALARM lets you specify an alarm date; a date and time; a date, time, and action; or date, time, action, and repeat interval. We're going to use this last option, so STOALARM needs as its argument a list that looks like this:

{ *date time action repeat* }

The *date* is the beginning date of the alarm, in the HP 48's format of *mm.ddyyyy*. The *time* has to be in the *hh.mmss* format; this is the time each day when the alarm will occur.

When the alarm goes off, the *action* is executed. The action is what makes the HP 48 such a good assistant. It can be a string that's placed on the stack, or it can even be a complete program.

The action is executed no matter what else the calculator is doing. Even if the calculator happens to be turned off, the alarm turns it on so the action can be executed.

The repeat interval is specified in clock ticks. One clock tick is 1/8192 second, which means that some common repeat intervals you might want to use are:

For This Alarm Interval:	Use This for the Repeat Parameter:
1 minute	491,520
1 hour	24,491,200
1 day	707,788,800

Now let's return to the *Controltime* program. It begins by creating an empty list. Then it gets the date, using the DATE command. If the date is, say, September 20, 1991, the following quantities are now on the stack:

Stack Level	Contents
Level 2:	{ }
Level 1:	9.201991

The next instruction, +, combines these two. The date is added to the list, and the list left on the stack:

Stack Level	Contents
Level 2:	
Level 1:	{9.201991}

Next, the program gets the daily alarm time. The prompt string is placed on the stack followed by an empty string. INPUT then uses these two strings to prompt (and wait for) you to enter the time in 24-hour format. When you enter the number and press (ENTER), the program resumes. The OBJ→ command changes your input to a real number, and + adds it to the list. If you specified a daily alarm at 2:00 every afternoon, the list now looks like this:

```
{9.201991 14.0000}
```

Now comes the action. We want to execute a program, which is named *Getemp*. We add this name to the list by specifying `'Getemp'` `+`. When the program name is placed in the list, the tick marks disappear, and we're left with:

```
{9.201991 14.0000 Getemp}
```

The final component of our list is the repeat interval. We know it's going to be one day (which is 707,788,800 ticks of the clock). So we add that number to the list, too:

```
{9.201991 14.0000 Getemp 707788800}
```

The list is now set up for the calculator's STOALARM command. When STOALARM with this list as the argument, it uses the list to set an alarm that:

- Begins today.
- Goes off at 2:00 p.m.
- Automatically executes the program *Getemp*.
- Resets itself to go off again tomorrow at 2:00 p.m.

STOALARM leaves the index number of the alarm on the stack. For the sake of simplicity, we won't use it here. However, you could easily store this index number, then use it each time to delete the previous alarm before storing the latest one. Since we don't use the index number, we'll drop the stack before ending the program.

Getemp—Adding to a List

Every time the alarm goes off, it calls this *Getemp* subprogram. *Getemp* beeps, then prompts for the user to enter the current temperature. It tags this temperature with the date, and adds the tagged value to a list of all temperatures. *Getemp* expects an alarm number on the stack, and leaves nothing on the stack when it's done.

Arguments	Results
1: alarm index number	1:

Program Instructions	Comments
«	
DROP	Throws away alarm index number.
440 1 BEEP	Beeps at 440 Hertz for one second.
"TEMPERATURE INPUT	INPUT displays this prompt string.
"	
Datetag	Calls subprogram to place date in stack as a tag.
INPUT	Waits for input from the keyboard.
OBJ→	Converts tagged keyboard input to a tagged number.
Templist	Brings contents of temperature list into stack.
+	Adds current temperature to list.
'Templist'	
STO	Stores updated list as Templist.
»	

Now save the program as *Getemp*.

Keystrokes	Comments
ENTER	Puts program on the stack.
' Getemp STO	Stores the program.

The HP 48 alarm places the alarm's index number on the stack before calling this program. So the first thing we have to do is drop to the stack to get rid of the index number. (Notice that if you're testing *Getemp* by running it using its menu key, you'll need to provide something on the stack before you execute this subprogram.)

Getemp then beeps and prompts for input of temperature. INPUT, you remember, needs two arguments on the stack. The first is the prompt string, a message that appears at the top of the display. The second is a message appearing on the command line. If you begin and end this message with colon (:), the message acts as a tag for the keyboard input that follows.

Getemp calls another subprogram, *Datetag*, to provide this tag. *Datetag* merely gets the current date and turns it into a string, with colons. For instance, if the date is April 29, *Datetag* returns ":4.29:" to the stack. (More about how *Datetag* does this in a moment.)

Now INPUT has what it needs—two strings on the stack. When the INPUT command is executed the program halts, and you see a display like the following:

```
TEMPERATURE INPUT

:4.29:
```

The cursor is positioned to the right of :4.29:, ready for your temperature input.

Now you press number keys for the temperature. When you hit ENTER, the program continues. INPUT combines the date and the temperature into a tagged object, and OBJ→ turns that tagged object into something you can use in addition, subtraction, even statistics.

With the tagged temperature now on the stack, *Getemp* brings the contents of *Templist* into the stack. (Executing Templist, with no single quotation marks, evaluates the object—that is, brings in the list contents. Executing 'Templist', however, puts just the name on the stack.)

We now have two quantities on the stack. For example:

Stack Level	Contents
Level 2:	:4.29: 75
Level 1:	{:4.28: 56 :4.27: 43 :4.26: 61}

With a single date-tagged temperature in level 2 and a list of them in level 1, we execute + to combine them. The result in level 1 is:

```
{:4.29: 75 :4.28: 56 :4.27: 43 :4.26: 61}
```

When you add quantities, level 1 is always appended to level 2. So the list is appended to the current temperature, with the result that the latest temperature is always first.

Now all we need to do is save the updated list as *Templist* again. For this, the program places the name `'Templist'` on the stack, then executes STO. It's as if you had done the same thing yourself.

Datetag—Saving the Display Mode

Datetag is called by *Getemp*, and puts on the stack a string containing the current date surrounded by colons. *Datetag* expects nothing from the stack, and leaves the string there when it's done.

Arguments	Results
1:	1: ":today's date:"

Program Instructions	**Comments**
`«`	
`RCLF`	Gets the flag status (including display mode.)
`2 FIX`	Temporarily sets display mode to two decimal places.
`":"`	Creates string containing first colon.
`DATE`	Gets today's date.
`+`	Adds date to string.
`": "`	Creates string containing second colon.
`+`	Adds second string to first, leaving a single string.
`SWAP`	Brings flag status into the stack.
`STOF`	Restores original flag status and display mode.
`»`	

To save *Datetag*:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Datetag STO	Stores the program.

Datetag uses the FIX command to set the number of displayed decimal places to two. (You'll see why in a minute.) Since this subprogram is called as part of an alarm action, you might be in the middle of another application—an application that requires, say, 8 SCI display mode. So we want to save the current display mode when we enter *Datetag*, and restore it when we exit.

The RCLF and STOF commands are how we save the display mode. RCLF brings a list containing the status of all flags, including the display mode flag, into the stack. Moreover, RCLF also recalls the current number of displayed digits—for example, 8 SCI, 6 FIX, or whatever. (Status, including that of display mode and number of digits, is actually all lumped together in a pair of binary integers.) After the main portion of *Datetag* does its duty, the STOF command restores all flags to their original status. Most important for us, STOF also restores the display mode and number of displayed digits.

Datetag uses the HP 48's ability to add strings—and to add objects to strings—to create the date tag needed by the INPUT command in *Getemp*. The value retrieved by DATE (say 4.29) doesn't need to be a string; it becomes part of the string when it's added to ":", creating ":4.29".

Notice how the FIX mode affects the date tag. For instance, if you specified 6 FIX, the tag would include the year: ":4.291991:". But 4 FIX would give ":4.2920", which might be confusing. We use 2 FIX because it gives us useful date information, yet results in a short, concise tag.

Templist—Starting a List

Templist is a list containing date-tagged temperatures. As you'll see, a portion of *Autogetem*'s main menu lets you clear all temperature data. To be certain that the program runs the first time, though, you should create the variable *Templist*, containing an empty list, before firing up *Autogetem*:

Keystrokes	**Comments**
⬅ { }	Creates empty list.
ENTER	
' Templist	
STO	Saves empty list as *Templist*

Templist is used not only by the SETUP menu key, but also by all other menu selections as well. For instance, when you press LIST the contents of *Templist* are brought into the display. There you can use ▼ and the other cursor control keys to view all data and temperatures. Press ATTN when you're done.

One interesting feature of this tagged list is that you'll see the temperatures in the current display format: 0 FIX (no decimal places), 2 FIX (two decimal places), or whatever. The tag, though, is frozen at two decimal places—or whatever display mode is set by *Datetag*.

Avgtemp—Using Stat Commands Programmatically

Avgtemp is called when you press the AVGT key from the main menu. It uses Σ+ to place the contents of the temperature list into the current directory's ΣDAT variable, then calculates the average. *Avgtemp* expects nothing from the stack, and leaves the tagged average temperature there.

Arguments	Results
1:	1: tagged average temperature

Program Instructions	Comments
«	
CLΣ	Clears this directory's statistical data.
Templist	Gets contents of the list of temperatures.
OBJ→	Separates list into its elements.
1 SWAP	Puts 1 in level 2, number of list elements in level 1.
FOR i	Starts FOR ... NEXT loop, from 1 to number of list elements.
Σ+	Adds next element to statistical data.
NEXT	End of FOR ... NEXT loop.
MEAN	Calculates average temperature.
"Average"	
→TAG	Labels the average temperature.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Avgtemp STO	Stores the program.

First, the program clears any previous data from this directory's ΣDAT reserved variable. Then, when this subprogram "executes" *Templist*, the contents of the temperature list—all those date-tagged values—are brought into level 1 of the stack. Next, the OBJ→ command separates the list into its elements, returning the number of elements to level 1, and the first temperature to level 2, the second temperature to level 3, and so on.

We want to use a FOR ... NEXT loop to add up all the temperatures, so the program next places a 1 in level 1 and executes SWAP. The stack is now set like this:

Stack Level	Contents
Level 4:	Second temperature
Level 3:	First temperature
Level 2:	1
Level 1:	Number of temperatures

When executed, FOR goes "from" the number in level 2 "to" the number in level 1. It executes everything down to NEXT, then does the loop again and again. The only command executed each time through the loop is Σ+, which adds the current temperature to the statistical data for all the temperatures. (It's all saved in this directory's reserved ΣDAT variable).

To find the average temperature now, you could press ⬅ STAT NXT MEAN. But we'll place this command in the program, too, so that *Avgtemp* gives us the average (or mean) temperature.

With the average temperature on the stack, the program next inputs the string "Average". The →TAG command tags the average temperature for easy identification, and leaves the tagged value on the stack, as shown in the following example.

```
Average: 61.22
```

Even though the average temperature is tagged, you can add, subtract, multiply, or divide this quantity, just as you can any other number.

Reset—Clearing a List

The *Reset* subprogram clears all data from *Templist*, the list of temperatures. It expects nothing from the stack, and adds nothing when it exits.

Program Instructions	Comments
«	
{ }	Puts empty list on stack.
'Templist'	Places variable name on stack.
STO	Stores empty list in *Templist*.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Reset STO	Stores the program.

To reset and begin storing a new set of temperatures, we simply store an empty list in the variable *Templist*.

If you want to check and clear alarms, the easiest way is to go right to the HP 48's alarm catalog by pressing ← TIME CAT. Catalogs are different from menus—you can't get to them from a program. But although you can't get to this CAT or its menu keys from a program, you could easily toss in some user instructions and a DISP command to tell users how to do it.

This is not to say you can't clear an alarm programmatically, either. On the contrary, you could:

1. Save the alarm index in a variable when you create the alarm.
2. Bring the alarm index into level 1 of the stack and execute DEL ALARM to delete the alarm without affecting other alarms.

Show—Displaying a Built-In Menu

Pressing *Autogetem*'s SHOW menu key executes the *Show* subprogram. This does nothing more than bring up page 3 of the HP 48's statistics menu, so you can easily create a bar chart showing the data in ΣDAT. It leaves the stack unchanged.

Program Instructions	Comments
«	
40.03	Number for page 3 of STAT menu.
MENU	Displays that menu.
Avgtemp	Calls the *Avgtemp* program.
DROP	Removes the result of *Avgtemp* from the stack.
»	

To save the program:

Keystrokes	Comments
['] Show [ENTER]	Puts program on the stack.
['] [STO]	Stores the program.

The MENU command lets you display any of the HP 48's built-in menus. MENU takes as its argument a number of the form *xx.yy*, where *xx* is the menu number and *yy* is the page.

The argument for MENU is 40.03. The 40 refers to the STAT menu, while 03 means page 3 of this menu. Thus, executing 40.03 MENU in a program has the same effect as pressing [⬅] [STAT] [NXT] [NXT]: You see page 3 of the STAT menu. You can then press BARPL, or use another STAT key to analyze the temperature data.

There's a potential problem with trying to execute any of the statistics commands, even from the keyboard. What if *Avgtemp* hasn't been executed yet, and there's no ΣDAT reserved variable? You'll get an error if you try to execute a statistics function like BARPL on a nonexistent ΣDAT.

For this reason, *Show* calls *Avgtemp*, which, as you know, *creates* ΣDAT when it calculates the average temperature. *Avgtemp* also leaves the average temperature on the stack, though; so *Show*'s DROP command gets rid of that value, and leaves the stack just as it was before *Show* was called.

After performing statistics operations, the user can press VAR to get back to the menu of variables, then hit AUTO to see *Autogetem*'s main menu again.

Running the Autogetem Program

Here's how to set up *Autogetem* so it prompts for temperature input every day at 6:00 am:

Program Prompt or Display	**Your Action**
	AUTO
AUTOGETEM This program automatically prompts daily for input Press any key to continue	ENTER (or any key)
MAIN MENU: SETUP-Daily time LIST-Dates, temps AVGT-Average temp RESET-All temps SHOW-Stat menu	

You see the following menu key display at the bottom of the main menu:

SETUP LIST AVGT RESET SHOW

Program Prompt or Display	**Your Action**
	SETUP
TIME: Input daily time as HH.MMSS. Use 24-hour format.	6.0000 ENTER

Now the alarm is set. Every morning at 6 am, you'll hear a one-second beep. When you peer at the HP 48, you'll see a display showing today's date. The cursor will be on the command line, waiting patiently for you to input the temperature:

Program Prompt or Display	Your Action
TEMPERATURE INPUT	65 (ENTER)
:5.04:	

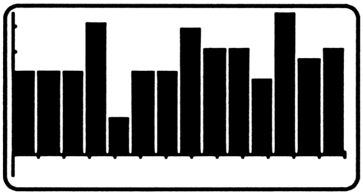
After, let's say, a couple of weeks of this, you want to see how those temperatures look. You can bring the list of temperatures into the display with LIST .

Program Prompt or Display	Your Action
	AUTO , then any key to get to main menu
MAIN MENU: SETUP-Daily time LIST-Dates, temps AVGT-Average temp RESET-All temps SHOW-Stat menu	LIST
1: {:5.04: 65.00 :5.03: 96.00 :5.02: 50.00...	

The temperatures are all there, tagged with their dates of entry. To examine the complete list, use (▼) to scroll through the list. (ATTN) gets you back to the main menu.

The other menu keys perform their own functions.

Keystrokes	Display	Comments
AVGT	`1: Average: 64.15`	Display of average temperature.
SHOW		Brings up the page 3 of the STAT menu.
BARPL		Produces a bar plot of the recorded temperatures, as shown below.



Press RESET to reset the temperature list back to an empty list again. When you reset the application, you're all ready for a new set of temperatures.

Even after RESET, the alarm will continue to prompt you every day. Moreover, every alarm you create with SETUP will repeat itself daily, unless you explicitly purge it with the TIME menu:

Keystrokes	Comments
↰ TIME CAT	Shows catalog of alarms.
↰ PURGE	Purges the current alarm.

Autogetem and Your Directories

One caveat: When the alarm goes off, it looks for *Getemp* (and *Datetag* and *Templist*). It looks first in whatever is the current directory at the time. If *Getemp* isn't there the next directory up is searched, then the next, until HOME is reached. If *Getemp* still isn't found, the calculator simply puts the name `'Getemp'` onto the stack.

To ensure that *Getemp*, *Datetag*, and *Templist* are found, you could put copies of them in your HOME directory. But the easiest way to handle this situation is to add path information to the STOALARM list in *Controltime*. For instance, instead of 'Getemp' in the list, you could use a program, like this:

Program Instructions	**Comments**
«	
HOME	Goes to HOME directory.
HEALTH	Goes to HEALTH subdirectory.
AUTOGETEM	Goes to AUTOGETEM subdirectory.
Getemp	Runs *Getemp* program.
»	

Just place this entire program, brackets and all, in the list created in *Controltime*. Use it in place of the 'Getemp' instruction that's now there. (And of course, substitute your own path for HOME HEALTH AUTOGETEM.)

Race Timer

The final program in this book converts the HP 48 to a race timer, one that can store the numbers and record the times of hundreds of competitors. Actually, "converts" isn't the right word—because even while it's handling the timing chores, your HP 48 is available for other uses as well.

The application is called *Racers*, and it relies heavily on one of the HP 48's most impressive features: the ability to display menus—and menus within menus—while waiting passively for the press of a key. In fact, except when you're actually performing a task like entering a competitor's number or viewing the latest list of times, *Racers* isn't really "running" at all. It just looks that way.

Racers has a main menu and several "sub-menus". One sub-menu starts a new race. Another handles the timing chores—you identify runners by the numbers attached to their jerseys, then press ENTER to record each runner's number as he or she crosses the finish line. There's still another sub-menu that lets you examine the times of all runners who have finished.

Racers isn't for sprints, but it will handle a 10K or a marathon pretty well. In fact, it's good for any situation where you need to keep a record of people, things, or tasks and their comparative times.

The Racers Directory

First, create a directory for the *Racers* program and subprograms, then get into that directory:

```
' RACERS
↰ MEMORY CRDIR
VAR
RACER
```

The RACERS directory takes care of the *Racers* program and all its subprograms, as well as the list of competitors and their times.

The Main Racers Program

The main *Racers* program is deceptively simple. It consists of an opening message and a menu. It doesn't affect the stack.

Program Instructions	**Comments**
«	
`Openmsg`	Pauses to display opening message.
`Menu`	Displays main menu keys and user instructions.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Racers STO	Stores the program.

Openmsg

Openmsg displays an opening message that's seen any time you press the RACER menu key. It waits for you to press the ENTER key before it lets you continue. If you press any key except ENTER, you see the same opening message again. *Openmsg* also sets the display mode to four decimal places, so that you can view times in the HP 48's format of *hh.mmss*.

Program Instructions	Comments
«	
"RACERS This is a race timer for recording and storing the times of many competitors. Press ENTER to see the menu."	Message for display by DISP. Add a newline character after each line.
CLLCD	Clears HP 48's LCD.
1 DISP	Displays message beginning on line 1 of LCD.
7 FREEZE	Freezes entire display to show message.

Program Instructions	Comments
`0 WAIT`	Waits for press of any key.
`IF`	Begins IF clause; checks key address left by WAIT.
`  51.1 ≠`	If key pressed wasn't ENTER, executes the THEN action.
`THEN`	
`  Openmsg`	Calls itself again unless key was ENTER.
`END`	
`4 FIX`	Sets display mode to four decimal places.
`»`	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Openmsg STO	Stores the program.

Menu

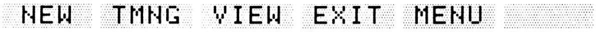
Menu is the main menu for *Racers*. You see this main menu after the opening message displayed by *Openmsg*, and you also see it after exiting from "sub-menus" displayed by other subprograms. In a sense, *Menu* is like a starting point to which you always return.

Program Instructions	Comments
«	
{	Begins key list for use by TMENU.
{ "NEW" New }	Label and object for first menu key.
{ "TMNG" Timing }	Second menu key's label and object.
{ "VIEW" View }	Third menu key's label and object.
{ "EXIT" Exit }	Fourth menu key's label and object.
{ "MENU" Menu }	Fifth menu key's label and object.
}	Ends key list.
TMENU	Displays menu keys specified in list.
Mmsg	Displays main message.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Menu STO	Stores the program.

When executed, the TMENU command uses as its argument the "list of lists" that's placed on the stack by *Menu*, creating the following menu keys across the bottom of the display:



For clarity, we've provided just one object for each of the main menu keys displayed by TMENU. However, each of these objects is actually a full-fledged "sub-application" in its own right. And what exactly are those sub-applications? As it happens, they're explained in *Mmsg*, the subprogram that's run by *Menu*. Read on!

Mmsg

When the *Menu* subprogram is executed, it displays the main menu keys. It also calls *Mmsg*, the main message subprogram, to explain more fully how to use those keys.

Program Instructions	Comments
«	
"MAIN MENU: NEW-Start new race TMNG-At finish line VIEW-All competitors and times EXIT-While timing MENU-Shows this menu"	Message for display by DISP.
CLLCD	Clears HP 48's LCD.
1 DISP	Displays message beginning on line 1 of LCD.
3 FREEZE	
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Mmsg STO	Stores the program.

Mmsg explains what all those menu keys mean:

- NEW is for starting a new race. Before you can start, it warns that all existing data and times will be lost if you continue. When you press ENTER to continue, you can then start the timer at the same time you fire a gun to begin the race.
- TMNG gets to the timing menu. From this menu, you can input a competitor's number, then record the time when that competitor crosses the finish line.

- VIEW gets to the view menu. From here, you can see the numbers and times for all competitors who have finished.
- EXIT lets you exit from the *Racers* application without affecting the timing. Thus, even while the race goes on, you can go out and use the calculator for other purposes. Then you can come back to the main menu, go to the timing menu, and pick up where you left off.
- MENU displays the main menu. It's a way to get user instructions and the main menu keys back on the screen.

New

New is executed when you press the NEW key. It displays a warning, and allows you to abort by pressing any key except ENTER. If you do press ENTER to continue, *New* runs *Init*, to initialize the calculator, then calls *Smsg* and *Smenu* to provide a start message and a start menu.

Program Instructions	**Comments**
`«`	

`"WARNING:`
`All race data will be`
`LOST if you start a`
`new race! Press ENTER`
`for a new race, or any`
`other key to keep`
`existing race times."`
	Message for display by DISP. Add a newline character after each line.

`CLLCD`	Clears HP 48's LCD.
`1 DISP`	Displays message; it starts on row 1 of the calculator display.
`-1 WAIT`	Waits for key press while displaying current menu.

Program Instructions	Comments
`IF`	Begins IF clause; checks key address left by WAIT.
`  51.1 ==`	If key pressed was ENTER, executes everything between THEN and ELSE.
`THEN`	
`  Init`	Initializes calculator.
`  Smsg`	Displays user instructions for starting the race.
`  Smenu`	Displays start menu.
`ELSE`	
`  Menu`	Returns to main menu.
`END`	
`»`	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' New STO	Stores the program.

Init

If in *New* you choose to continue and begin a new race, *Init* saves an empty list in the variable *Clist*. This list is where the competitor numbers and times will be stored.

Program Instructions	Comments
«	
{ }	
'Clist'	
STO	
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Init STO	Stores the program.

Smsg

After running *Init*, the *New* subprogram calls *Smsg* and *Smenu*. They work hand in hand: *Smsg* displays user instructions, while *Smenu* puts the necessary keys on the display screen. *Smsg* uses 3 FREEZE after displaying its message, allowing the new menu keys to be added to the display by *Smenu*.

Program Instructions	Comments
«	
"Press START at the instant to start timing."	Message for display by DISP.
CLLCD	Clears HP 48's LCD.
3 DISP	Displays message beginning on row 3 of LCD.
3 FREEZE	Freezes display, but allows updating of menu area.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Smsg STO	Stores the program.

Smenu

Smenu provides a temporary menu with just two keys: If you press START, you run the *Start* subprogram, then return to the *Menu* subprogram. Pressing the other option, MENU, gets you back to the main menu without running *Start* to reset the start time.

Program Instructions	Comments
`«`	
`{`	Begins key list for use by TMENU.
`  { "START"`	Label for first menu key.
`  «`	Begins object (a program) for first menu key.
`  Start`	Begins race timer.
`  Menu`	Returns to main menu.
`  »`	
`  }`	Ends key list.
`  { "MENU" Menu }`	Label and object for second menu key.
`}`	
`TMENU`	Displays menu keys specified in list.
`»`	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Smenu STO	Stores the program.

Start

This subprogram is run when you press the START menu key displayed by *Smenu*. It stores the current number of clock ticks in a variable. This variable, called *Begintime*, actually records the starting time of the race; by subtracting it from each competitor's time later, we'll come up with that runner's elapsed time.

Program Instructions	Comments
«	
`TICKS`	Returns current system time as binary integer.
`'Begintime'`	Places name in stack.
`STO`	Stores current system time in variable *Begintime*.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Start STO	Stores the program.

Well, that takes care of starting the race. Now let's see what happens at the finish line.

Timing

The second option in the main menu key list is TMNG. When you press this key, you run the *Timing* program; this program displays the timing menu (using TMENU again) and calls *Tmsg* to display the user instructions for timing.

```
Program Instructions          Comments

«
{                             Begins key list for use by TMENU.
  { "TIME"                    Label for first menu key.
  «                           Begins object (a program) executed
                              by first key.
  Getnum                      Gets a competitor's number.
  Crossline                   Prepares to record time for that
                              competitor.
  Getime                      Records competitor's raw time at the
                              finish line.
  Calctime                    Calculates competitor's elapsed time.
  Storem                      Stores competitor's time in Clist.
  Timing                      Returns to timing menu again.
  »                           End of first key's object.
  }                           End of first key list.
  { }                         Blank key.
  { }                         Blank key.
  { }                         Blank key.
  { }                         Blank key.
  { "MENU" Menu }             Sixth menu key returns to main
                              menu.
}                             Ends key list.
```

Program Instructions	Comments
TMENU	Displays menu keys specified in list.
Tmsg	Displays user instructions for timing menu.
»	

Keystrokes	Comments
ENTER	Puts program on the stack.
' Timing STO	Stores the program.

We'll see the individual subprograms for handling timing in a moment. First, let's take a fast look at *Tmsg*.

Tmsg

When *Timing* is run, it shows the timing menu keys at the bottom of the display, and it also calls *Tmsg* to provide more detailed instructions on how to handle the finish-line action. *Tmsg* is a straightforward use of DISP and FREEZE to show a message; notice that we've used `3 FREEZE` to allow the menu list to be updated while the rest of the display is frozen.

Program Instructions	Comments
«	
"TIMING MENU: Choose one: TIME to time next competitor. MENU to return to main menu."	Message for display by DISP. Add a newline character after each line.
CLLCD	Clears HP 48's LCD.
1 DISP	Displays message beginning on line 1 of LCD.
3 FREEZE	
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Tmsg STO	Stores the program.

Getnum

OK, you started the race, and the runners are off. Now you go to the finish line and wait for the first runner to come into sight. While you're waiting, you press TMNG to display the timing menu.

When you see a competitor, you press TIME. This key calls several subprograms, the first of which is *Getnum*. The *Getnum* subprogram prompts you to enter a competitor's number, and it puts that number on the stack. The number is identified by the word "No.."

Arguments	Results
1:	1: competitor's number

Program Instructions	Comments
«	
"Put next competitor's number in display. Then press ENTER."	Prompt string for INPUT.
"No. "	Command-line string for INPUT.
INPUT	Displays prompt and command-line messages, waits for keyboard input.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Getnum STO	Stores the program.

If the runner's number is 765, you enter that number at the prompt. The quantity placed on the stack, then, looks like this:

```
No. 765
```

Notice that although *Getnum* asks for a runner's number, it accepts virtually any kind of input. You could even enter a runner's name, like "Fred." Of course, the "number" then would read `No. Fred`.

Crossline

The next subprogram called when you press TIME is *Crossline*. It prompts you to press ENTER when the runner crosses the finish line, then waits for a press of that key. The prompt is individualized; that is, it shows the runner's number. If you hit any key except ENTER, input is aborted and execution returns to the *Timing* subprogram.

```
Program Instructions              Comments

«
→ n                               Creates local variable n.
  «                               Begins defining procedure for
                                  local variable.
  "Hit ENTER when                 Begins text string. (Add
                                  newline character.)
  "                               Ends text string.
  n +                             Adds n to text string.
  " crosses                       Begins second text string.
  the finish line.
                                  Blank line in second text
                                  string.
  Any other key cancels."         Ends second text string.
  +                               Adds text strings to form
                                  prompt message.
  CLLCD                           Clears HP 48's LCD.
  1 DISP                          Displays message beginning on
                                  line 1 of LCD.
  7 FREEZE                        Freezes entire display to show
                                  message.
  0 WAIT                          Waits for press of any key.
  IF                              Begins IF clause; checks key
                                  address left by WAIT.
    51.1 ≠                        If key pressed wasn't ENTER,
                                  executes the THEN action.
  THEN
    Timing                        Calls Timing subprogram if
                                  key wasn't ENTER.
  END                             Ends IF ... THEN structure.
  n                               Puts number n on the stack.
  »
»
```

To save the program:

Keystrokes	**Comments**
`ENTER`	Puts program on the stack.
`'` Crossline `STO`	Stores the program.

If you press `ENTER` to record the time, execution falls through to the next subprogram (*Getime*). In this case, *Crossline* leaves the competitor's number on the stack. The number isn't left on the stack, however, if you abort by pressing any key except `ENTER`.

Getime

Getime records the competitor's raw time. It does this by placing the current value of TICKS on the stack, then brings in the value of *Begintime*, which was the time at the start of the race. Finally, *Getime* subtracts the beginning time from the current time to get the elapsed time for this competitor. The time is raw—that is, it's a binary integer in units of 1/8192 second.

Arguments	Results
2:	2: *competitor's number*
1: *competitor's number*	1: time in clock ticks (binary integer)

Program Instructions	**Comments**
`«`	
`TICKS`	Records current system time in clock ticks.
`Begintime`	Puts beginning time on the stack.
`-`	Subtracts beginning time from current time.
`»`	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Getime STO	Stores the program.

Now the stack has two quantities on it: the competitor's number and that competitor's time in clock ticks.

Calctime

Calctime converts the competitor's time in clock ticks to elapsed time in hours, minutes, and seconds. The converted time is in the HP 48's *hh.mmss* format.

Arguments	Results
2: *competitor's number*	2: *competitor's number*
1: time in clock ticks (binary integer)	1: time as hh.mmss

Program Instructions	Comments
«	
B→R	Converts binary integer to number.
8192 /	Converts clock ticks to seconds.
3600 /	Converts seconds to decimal hours.
→HMS	Converts decimal hours to hours, minutes, seconds (*hh.mmss*) form.
»	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Calctime STO	Stores the program.

Now we're ready to combine and store the competitor's number and time.

Storem

Storem uses SWAP to reverse the order of the competitor's number and time on the stack. Then it combines them with →TAG, producing a time that's tagged with the runner's number. Finally, *Storem* adds the runner's tagged time to the competitor list, *Clist*.

Arguments	Results
2: competitor's number	2:
1: time as hh.mmss	1:

Program Instructions	**Comments**
`«`	
`SWAP`	Exchanges competitor's number and time on the stack.
`→TAG`	Tags time with competitor's number.
`Clist`	Brings competitor list into the stack.
`+`	Adds the list to the tagged time in the stack.
`'Clist' STO`	Saves the updated list.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Storem STO	Stores the program.

That takes care of the timing for one competitor. The timing menu appears again and again, allowing you to enter runner's numbers and times as they cross the finish line.

Notice that the information for the latest competitor is placed first in the list, bumping subsequent information down. Thus, the runner with the fastest time is at the list's tail end. (This will be important later on, when we show the runners and their times.)

Now what about displaying those times?

View

For displaying race information, you use MENU to get back to the main menu, then press VIEW. This executes the *View* subprogram, which displays the view menu and user instructions.

Program Instructions	**Comments**
«	
Vmenu	Displays view menu.
Vmsg	Displays user instructions for the view menu.
»	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' View STO	Stores the program.

Vmenu

The *Vmenu* subprogram uses TMENU to create a temporary menu, one with three active keys and three blank ones.

Program Instructions	Comments
`«`	
`{`	Begins key list for use by TMENU.
`  { "SHOW"`	Label for first menu key.
`  «`	Begins object (a program) for first menu key.
`  Vfirst`	Begins display string showing latest finisher.
`  Vtimes`	Adds all finishers to the string and displays them.
`  »`	Ends object for first menu key.
`  }`	Ends first menu key.
`  { }`	Second menu key is blank.
`  { "CLIST" Clist }`	Label and object for third menu key; brings *Clist* into stack.
`  { }`	Blank key.
`  { }`	Blank key.
`  { "MENU" Menu }`	Sixth menu key returns to main menu.
`}`	Ends key list.
`TMENU`	Displays menu keys specified in list.
`»`	

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Vmenu STO	Stores the program.

Vmenu, then, provides a display of three menu keys, with the other three keys blank.

SHOW | | CLIST | | | MENU

Here's what those view menu keys mean:

- SHOW calls several subprograms to show you competitor numbers and times. You see up to seven competitors at once, and they cycle through the display automatically. At the end, you're left with the top seven finishers in the display.
- CLIST brings the entire competitor list (numbers and times) into the stack, where you can examine it.
- MENU, of course, returns to the main menu.

Vmsg

Vmsg supplies user instructions for the view menu keys displayed by *Vmenu*. It uses FREEZE with an argument of 3; this allows updating of the menu list while freezing the rest of the display to show the text.

Program Instructions	Comments
«	
"VIEW MENU: SHOW-Autodisplay all times CLIST-Examine times MENU-Main menu"	Message for display by DISP.
CLLCD	Clears HP 48's LCD.
1 DISP	Displays message beginning on line 1 of LCD.
3 FREEZE	Freezes all of display except menu key area.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Vmsg STO	Stores the program.

Vfirst

If you press the SHOW key for autodisplay of competitors and times, *Vfirst* is executed. This subprogram sets up the stack for the automatic display of all competitors. *Vfirst* leaves on the stack a string that contains the latest competitor followed by an endline character.

Arguments	Results
1:	1: latest competitor (string)

Program Instructions	Comments
«	
`Clist`	Brings competitor list into stack.
`1 GET`	Gets first competitor.
`→STR`	Converts to string.
`"`	Begins second string. (Add endline.)
`"`	Ends second string.
`+`	Adds the two strings.
`DUP`	Duplicates the resulting string.
`CLLCD`	Clears HP 48's LCD.
`1 DISP`	Displays string on row 1 of the calculator display.
`1 WAIT`	Waits one second.
»	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Vfirst STO	Stores the program.

Vfirst places the tagged time of the last finisher (first element of the list) on the stack and converts it to a string. Then *Vfirst* creates a second string, consisting of an endline character surrounded by quotation marks, and adds it to the first. After duplicating the string (for passing to the next subprogram), *Vfirst* displays the string in row 1 of the calculator's LCD, then pauses for one second.

Vtimes

Vtimes brings the rest of the competitor times into the display for viewing, updating the LCD every second. Each new time is added on the top, and subsequent times are bumped down. At the end, you're left with a display of the top seven competitors—that is, the first seven to cross the finish line. *Vtimes* requires a string on the stack, and leaves nothing when it's done.

Arguments	Results
1: latest competitor (string)	1:

Program Instructions	Comments
`«`	
`Clist`	Brings *Clist* into stack.
`SIZE`	Puts size of *Clist* (how many competitors) on stack.
`→last`	Converts size to local variable *last*.
`  «`	Begins defining procedure for local variable.
`  2 last FOR num`	Begins FOR loop from 2 to *last*.
`    Clist`	Brings *Clist* into stack.
`    num GET`	Gets the next competitor from the list.
`    →STR`	Converts to string.
`    "`	Begins another string. (Add endline.)
`    "`	Ends string.
`    +`	Adds the two strings.
`    SWAP`	Exchanges first competitor's string and the new string.
`    +`	Adds the two strings to form string that includes latest competitor.

Program Instructions	Comments
`DUP`	Duplicates resulting string.
`CLLCD`	Clears HP 48's LCD.
`1 DISP`	Displays complete string, beginning on row 1 of the calculator display.
`1 WAIT`	Pauses one second.
`NEXT`	Does the FOR loop again.
`DROP`	Throws away final string from stack.
`7 FREEZE`	Freezes entire display to show message.
`»`	Ends defining procedure for local variable.
`»`	

To save the program:

Keystrokes	Comments
ENTER	Puts program on the stack.
' Vtimes STO	Stores the program.

When *Vtimes* is called, the first element of *Clist* is already on the stack. So the FOR ... NEXT loop in this subprogram goes from 2 to the maximum number of elements. Each time through the loop, GET gets a copy of the current competitor's information, then →STR converts it to a string.

We add another string containing an endline character (to put the next competitor on a separate row), then SWAP the two strings. Next we add the strings together; this places the latest element from the list at the top of this string. Finally, we display the resulting string for one second, beginning on row 1 of the calculator's liquid-crystal display screen.

In effect, we start by displaying the slowest runner first, then the next-slowest, and so on. Each faster runner is added in at the top of the display, and the rest bumped down. Thus, at the end, we have a display of the top seven runners and their times.

Exit

Well, what do you think *Exit* does? If you said "Exit from the program," you're right. Actually, *Exit* does nothing more than return the calculator to the normal VAR menu.

Program Instructions | **Comments**

```
«
2 MENU
»
```

To save the program:

Keystrokes	**Comments**
ENTER	Puts program on the stack.
' Exit STO	Stores the program.

In truth, you can start a race, then use EXIT to leave. When you press RACER to come back, you eventually get back to the main menu again—and the timer is still "running." You can press TMNG for timing or VIEW to see race results just as if you'd never left.

Running the Racers Program

It's time to start the Calculator Classic, a popular marathon race that attracts HP 48'ers from all over the world.

Program Prompt or Display	Your Action
	RACER
RACERS This is a race timer for recording and storing the times of many competitors. Press ENTER to see the menu.	ENTER
MAIN MENU: NEW-Start new race TMNG-At finish line VIEW-All competitors and times EXIT-While timing MENU-Shows this menu	

Naturally, we want to begin by starting a new race.

Program Prompt or Display	Your Action
	NEW
WARNING: All race data will be LOST if you start a new race! Press ENTER for a new race, or any other key to keep existing race times.	ENTER
Press START at the instant to start timing.	START

Now you see the main menu keys again, with their accompanying user instructions. After a couple of hours, you see the first finisher in the distance, so you switch to the timing menu.

Program Prompt or Display	**Your Action**
MAIN MENU: NEW-Start new race TMNG-At finish line VIEW-All competitors and times EXIT-While timing MENU-Shows this menu	TMNG
TIMING MENU: Choose one: TIME to time next competitor. MENU to return to main menu.	TIME

As the competitor comes closer, you see it's runner number 1139. You enter that number.

Program Prompt or Display	**Your Action**
Put next competitor's number in display. Then press ENTER.	
No.	1139 ENTER

The application includes the runner's number in its next prompt. You wait until runner 1139 reaches the finish, then hit the ENTER key.

Program Prompt or Display **Your Action**

```
Hit ENTER when
No. 1139 crosses
the finish line.

Any other key cancels.          ENTER


TIMING MENU:

Choose one:
TIME to time next
competitor.
MENU to return to
main menu.
```

You proceed in this fashion, pressing the TIME menu key, followed by the competitor's number, then hitting ENTER the instant that runner crosses the finish line.

At the end of the race—or even in the middle—you can go to the view menu to see results.

Program Prompt or Display **Your Action**

```
                                MENU

MAIN MENU:
NEW-Start new race
TMNG-At finish line
VIEW-All competitors
and times
EXIT-While timing
MENU-Shows this menu            VIEW
```

This gets you into the view menu, where you can view an automatic display of times or examine the competitor list.

Program Prompt or Display	Your Action
VIEW MENU: SHOW-Autodisplay all times CLIST-Examine times MENU-Main menu	SHOW

The information about competitors and their times is cycled through the display, with the faster runner always being added to row 1, until you're left with the top seven runners frozen on the screen:

```
:No. 1139: 2.0355
:No. 61: 2.1237
:No. Fred: 2.1340
:No. 652: 2.2117
:No. 1486: 2.2135
:No. 87: 2.2702
:No. 341: 2.3418
```

Looks like runner number 1139 came in at 2 hours, 3 minutes, and 55 seconds—a new world record. And even old Fred will qualify for the Olympics with that time!

Where to Go from Here

This programs in this book are just the beginning. Moreover, programming is just one facet of the HP 48's remarkable set of capabilities. As you modify these programs or write solutions for your own applications, try to take advantage of the calculator's complete set of tools; don't be afraid to experiment with several ways to do things.

Above all, have fun programming your HP 48!

Index

Save Time with the Companion Disk for *HP 48 Programming Examples*

If you have a personal computer, there's no need to type in code listings for *HP 48 Programming Examples*. All application programs and subprograms in the book are available on a floppy disk, and can be transferred to the HP 48 using the Serial Interface Kit (available from Hewlett-Packard).

Requirements:
- Macintosh® personal computer

or
- MS-DOS IBM PC® or compatible, with floppy disk drive and serial port
- Serial Interface Kit for the HP 48
- File transfer software

Use the coupon below to order.

Please send me:

______ (quantity) 5 1/4", 360K IBM PC-compatible disk(s) to accompany *HP 48 Programming Examples* at $19.95 each. ISBN: 0-201-56359-2

______ (quantity) 3 1/2", 720K IBM PC-compatible disk(s) to accompany *HP 48 Programming Examples* at $19.95 each. ISBN: 0-201-56360-6

______ (quantity) 3 1/2", Macintosh-compatible disk(s) to accompany *HP 48 Programming Examples* at $19.95 each. ISBN: 0-201-56361-4

______ Check enclosed (include your state sales tax; Addison-Wesley will pay postage and handling).

______ Charge my Visa card # ______
Expiration date ______
______ Charge my MasterCard # ______
Expiration date ______
Four digits above your name ______
______ Charge my American Express # ______
Expiration date ______

Your signature: ______

Name: ______
Title & company if applicable: ______
Address ______
City ______ State ______ Zip ______

Mail to: Addison-Wesley Publishing Company, Inc.
Order Department, 1 Jacob Way, Reading, MA 01867

HP48 Programming Examples

D. R. Mackenroth

Anyone who owns a Hewlett-Packard HP 48 calculator will find this book invaluable. No other book on the HP 48 has as many real world examples or such thorough step-by-step explanations of HP 48 programs.

Whether you are an engineer, student, business person, mathematician, or scientist, you will appreciate the wide variety of usable programs. Programs are drawn from the diverse areas of business, statistics, calculus, and graphics. Each program begins with simple problem statements and is followed by a sample run to show exactly how it works.

This highly accessible book does not assume any programming expertise. Each of the more than 150 usable example programs is accompanied by detailed explanations so anyone with an HP 48 can easily understand them. *HP 48 Programming Examples* features:

- Structured examples that can be used for real work immediately.
- Lots of fun programs for games, sound effects, piano, notepads, graphics; and step-by-step procedures for entering, running, and saving the programs.
- Complete program listings and detailed discussions that allow experienced programmers to easily modify programs.

D. R. Mackenroth is an experienced programmer and the author of several successful programming and electronics books. He is a former Hewlett-Packard Learning Products Engineer and wrote a wide variety of owners' and service manuals for HP calculators.

Other books of interest:

Monday/Robinson, *Using Your HP 95LX: Practical Examples and Applications*

Printed on recycled paper
Cover design by Joyce C. Weston

Corporate & Professional Publishing Group
Addison-Wesley Publishing Company, Inc.

90000>
9 780201 563252
ISBN 0-201-56325-8